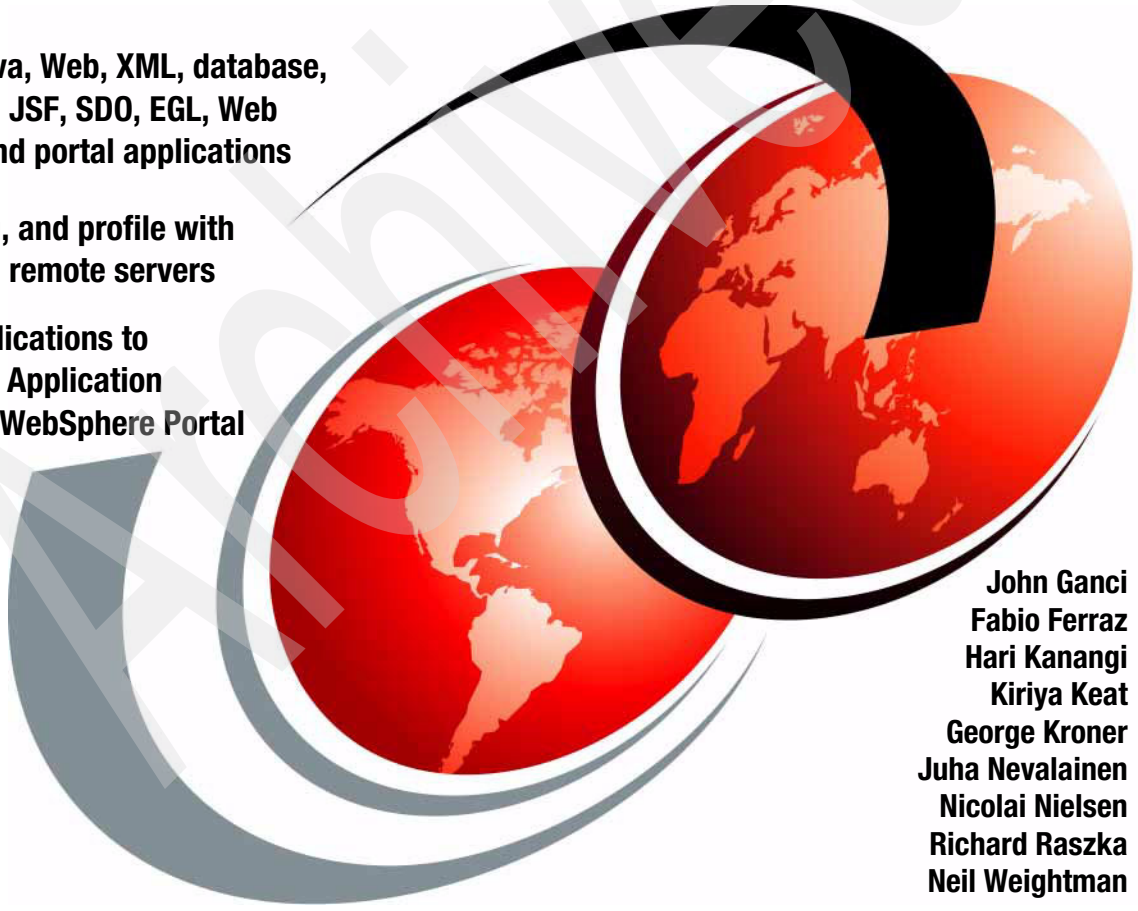
Rational. software

IBM

Rational Application Developer V6 Programming Guide

- Develop Java, Web, XML, database, EJB, Struts, JSF, SDO, EGL, Web Services, and portal applications
- Test, debug, and profile with built-in and remote servers
- Deploy applications to WebSphere Application Server and WebSphere Portal



John Ganci
Fabio Ferraz
Hari Kanangi
Kiriya Keat
George Kroner
Juha Nevalainen
Nicolai Nielsen
Richard Raszka
Neil Weightman

Archived

Redbooks

ibm.com/redbooks

Archived

International Technical Support Organization

Rational Application Developer V6 Programming Guide

June 2005

Archived

SG24-6449-00

Note: Before using this information and the product it supports, read the information in “Notices” on page xxi.

Archived

First Edition (June 2005)

This edition applies to IBM Rational Application Developer V6.0 and IBM WebSphere Application Server V6.0.

© Copyright International Business Machines Corporation 2005. All rights reserved.
Note to U.S. Government Users Restricted Rights -- Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

Contents

Notices xxi
Trademarks xxii

Preface xxiii
The team that wrote this redbook xxiii
Become a published author xxvi
Comments welcome xxvii
Summary of changes xxvii
June 2005, First Edition xxvii

Part 1. Introduction to Rational Application Developer 1

Chapter 1. Introduction 3
1.1 Introduction and concepts 4
1.1.1 IBM Rational Software Development Platform 4
1.1.2 Version 6 terminology 7
1.1.3 Application development challenges 7
1.1.4 Key themes of Version 6 8
1.2 Product packaging 9
1.2.1 Rational Developer supported platforms and databases 9
1.2.2 Rational Application Developer V6 product packaging 10
1.2.3 Rational Web Developer V6 product packaging 11
1.3 Product features 12
1.3.1 Summary of new features in Version 6 14
1.3.2 Specification versions 18
1.3.3 Eclipse and IBM Rational Software Development Platform 19
1.3.4 Test server environments 22
1.3.5 Licensing and installation 23
1.3.6 Migration and coexistence 26
1.3.7 Tools 28
1.4 Sample code 28

Chapter 2. Programming technologies 31
2.1 Desktop applications 32
2.1.1 Simple desktop applications 32
2.1.2 Database access 34
2.1.3 Graphical user interfaces 35
2.1.4 Extensible Markup Language (XML) 38
2.2 Static Web sites 39

© Copyright IBM Corp. 2005. All rights reserved.

2.2.1 Hypertext Transfer Protocol (HTTP) 40
2.2.2 HyperText Markup Language (HTML) 42
2.3 Dynamic Web applications 43
2.3.1 Simple Web applications 44
2.3.2 Struts 51
2.3.3 JavaServer Faces (JSF) and Service Data Objects (SDO) 52
2.3.4 Portal applications 55
2.4 Enterprise JavaBeans 57
2.4.1 Different types of EJBs 58
2.4.2 Other EJB features 60
2.4.3 Requirements for the development environment 61
2.5 J2EE Application Clients 62
2.5.1 Application Programming Interfaces (APIs) 63
2.5.2 Security 63
2.5.3 Naming 63
2.5.4 Deployment 64
2.5.5 Requirements for the development environment 65
2.6 Web Services 66
2.6.1 Web Services in J2EE V1.4 67
2.7 Messaging systems 70
2.7.1 Java Message Service (JMS) 71
2.7.2 Message-driven EJBs (MDBs) 72
2.7.3 Requirements for the development environment 72

Chapter 3. Workbench setup and preferences 75
3.1 Workbench basics 76
3.1.1 Workspace basics 78
3.1.2 Rational Application Developer log files 83
3.2 Preferences 84
3.2.1 Automatic builds 85
3.2.2 Clean build (manual) 86
3.2.3 Capabilities 86
3.2.4 File associations 90
3.2.5 Local history 92
3.2.6 Perspectives preferences 94
3.2.7 Internet preferences 95
3.3 Java development preferences 98
3.3.1 Java classpath variables 98
3.3.2 Appearance of Java elements 100
3.3.3 Code style and formatting 101
3.3.4 Compiler options 115
3.3.5 Java editor settings 117
3.3.6 Installed JREs 119

Redbooks

3.3.7 Templates 121
3.3.8 Code review 124

Chapter 4. Perspectives, views, and editors 131
4.1 Integrated development environment (IDE) 132
4.1.1 Rational Application Developer online help 132
4.1.2 Perspectives 136
4.1.3 Views 137
4.1.4 Editors 137
4.1.5 Perspective layout 137
4.1.6 Switching perspectives 138
4.1.7 Specifying the default perspective 140
4.1.8 Organizing and customizing perspectives 140
4.2 Available perspectives 143
4.2.1 CVS Repository Exploring perspective 143
4.2.2 Data perspective 145
4.2.3 Debug perspective 146
4.2.4 Generic Log Adapter perspective 148
4.2.5 J2EE perspective 149
4.2.6 Java perspective 151
4.2.7 Java Browsing perspective 153
4.2.8 Java Type Hierarchy perspective 155
4.2.9 Plug-in Development perspective 157
4.2.10 Profiling and Logging perspective 158
4.2.11 Resource perspective 159
4.2.12 Team Synchronizing perspective 160
4.2.13 Test perspective 161
4.2.14 Web perspective 162
4.2.15 Progress view 166
4.3 Rational Product Updater 168

Chapter 5. Projects 169
5.1 J2EE architecture 170
5.1.1 EAR files 170
5.1.2 WAR files 170
5.1.3 EJB JAR files 171
5.1.4 J2EE Application Client JAR files 171
5.1.5 RAR files 171
5.2 Projects and folders 171
5.3 Rational Application Developer projects 173
5.3.1 Enterprise Application project 175
5.3.2 J2EE Application Client project 175
5.3.3 Dynamic Web Project 175

Archived

5.3.4 Static Web Project . . . 176
5.3.5 EJB project . . . 176
5.3.6 Connector project . . . 176
5.3.7 Java project . . . 177
5.3.8 Simple project . . . 177
5.3.9 Server project . . . 177
5.3.10 Component test project . . . 177
5.3.11 Checkout projects from CVS . . . 177
5.4 Creating a new project . . . 178
5.5 Project properties . . . 180
5.6 Rational Application Developer samples . . . 181
5.6.1 The samples gallery . . . 183

Part 2. Develop applications . . . 187

Chapter 6. RUP and UML . . . 189
6.1 Overview . . . 190
6.2 Rational Unified Process (RUP) . . . 190
6.2.1 Process Advisor . . . 191
6.2.2 Process Browser . . . 192
6.2.3 Setting process preferences . . . 193
6.3 Visualize applications with UML . . . 194
6.3.1 Unified Modeling Language (UML) . . . 195
6.3.2 Browse diagram . . . 197
6.3.3 Topic Diagram . . . 199
6.3.4 Static Method Sequence Diagram . . . 203
6.3.5 Class Diagram . . . 205
6.3.6 Sequence Diagram . . . 213
6.3.7 J2EE visualization . . . 219
6.4 More information on UML . . . 220

Chapter 7. Develop Java applications . . . 221
7.1 Java perspective overview . . . 222
7.1.1 Package Explorer view . . . 223
7.1.2 Call Hierarchy view . . . 223
7.1.3 Type Hierarch view . . . 224
7.1.4 Problems view . . . 225
7.1.5 Declaration view . . . 225
7.1.6 Code review . . . 226
7.1.7 Outline view . . . 230
7.1.8 Diagram Navigator view . . . 230
7.2 Develop the Java Bank application . . . 231
7.2.1 Java Bank application overview . . . 231
7.2.2 Create a Java Project . . . 235

Archived

7.2.3 Create a class diagram 243
7.2.4 Create Java packages 246
7.2.5 Create a Java interface 248
7.2.6 Create Java classes 249
7.2.7 Create the Java attributes and accessor methods 254
7.2.8 Add method declarations to an interface 258
7.2.9 Add Java methods and constructors 262
7.2.10 Define relationships (extends, implements, association) 267
7.2.11 Implement the methods for each class 270
7.2.12 Run the Java Bank application 286
7.3 Additional features used for Java applications 286
7.3.1 Locating compile errors in your code 287
7.3.2 Running your programs 290
7.3.3 Debug your programs 292
7.3.4 Java Scrapbook 293
7.3.5 Pluggable Java Runtime Environment (JRE) 296
7.3.6 Add a JAR file to the classpath 297
7.3.7 Export the Java code to a JAR file 299
7.3.8 Run the Java application external to Application Developer 301
7.3.9 Import a Java JAR file into a project 301
7.3.10 Utility Java Projects 302
7.3.11 Javadoc 303
7.4 Java editor and Rapid Application Development 311
7.4.1 Navigate through the code 311
7.4.2 Source folding 314
7.4.3 Type hierarchy 315
7.4.4 Smart Insert 316
7.4.5 Mark occurrences 316
7.4.6 Word skipping 317
7.4.7 Smart compilation 318
7.4.8 Java search 318
7.4.9 Working sets 319
7.4.10 Quick Assist (Quick Fix) 320
7.4.11 Code Assist (content) 320
7.4.12 Import generation 322
7.4.13 Generate getters and setters 324
7.4.14 Override/implement methods 325
7.4.15 Adding constructors 326
7.4.16 Refactoring 328

Chapter 8. Develop Java database applications 333
8.1 Introduction to Java database programming 334
8.1.1 JDBC overview 334

Redbooks

8.1.2 Data source versus direct connection 335
8.1.3 XMI and DDL. 335
8.1.4 Rational Application Developer database features 336
8.2 Preparing for the sample 337
8.2.1 Import the BankDB sample project 337
8.2.2 Set up the BANK sample database. 338
8.3 Data perspective 339
8.3.1 Data Definition view 340
8.3.2 Database Explorer view 341
8.3.3 DB Output view 342
8.3.4 Navigator view. 343
8.4 Create databases and tables from scripts. 343
8.4.1 Create a database. 344
8.4.2 Create a database connection 347
8.4.3 Create the database tables from scripts 349
8.4.4 Populate database tables with data 352
8.5 Create and work with database objects 354
8.5.1 Create a database. 355
8.5.2 Create a database connection 356
8.5.3 Create a schema. 357
8.5.4 Create a table 358
8.5.5 Generate a DDL file 361
8.5.6 Deploy DDL from the workspace to a database 362
8.5.7 Copy database objects from a DDL file to a workspace 362
8.5.8 Generate DDL and XSD files for database objects. 365
8.6 UML visualization 370
8.6.1 Class diagrams 370
8.6.2 Information engineering (IE) diagrams 374
8.6.3 IDEF1X (Integrated Definition Extended) diagrams 375
8.7 Create SQL statements. 376
8.7.1 Using the SQL Statement wizard 376
8.7.2 Using the SQL Query Builder 384
8.8 Access a database from a Java application 389
8.8.1 Prepare for the sample 389
8.8.2 Access the database using the DriverManager 390
8.8.3 Access using a data source 393
8.9 Java stored procedures. 394
8.9.1 Prepare for the sample 395
8.9.2 Create a Java stored procedure 398
8.9.3 Build a stored procedure (deploy to database) 405
8.9.4 Java DriverManager access to a Java stored procedure 408
8.9.5 JavaBean access to Java stored procedure 409

Chapter 9. Develop GUI applications 415
9.1 Introduction to the Visual Editor 416
9.2 Prepare for the sample 416
9.2.1 Create the project for the sample 417
9.2.2 Add JDBC driver for Cloudscape to project 417
9.2.3 Set up the sample database 418
9.2.4 Import the model classes for the sample 419
9.3 Launching the Visual Editor 419
9.3.1 Create a visual class 420
9.3.2 Open an existing class with the Visual Editor 422
9.4 Visual Editor overview 423
9.4.1 Visual Editor layout 423
9.4.2 Customizing the appearance of the Visual Editor 424
9.5 Work with the Visual Editor 427
9.5.1 Resize a JavaBean component 427
9.5.2 Code synchronization 427
9.5.3 Changing the properties of a component 428
9.5.4 Add JavaBeans to a visual class 428
9.5.5 Work with the Properties view 432
9.5.6 Testing the appearance of the GUI 433
9.5.7 Add event handling to GUI 434
9.5.8 Verify the Java GUI application 435
9.5.9 Run the sample GUI as a Java application 436
9.5.10 Automatically add event handling 437
9.5.11 Visual Editor binding 438

Chapter 10. Develop XML applications 443
10.1 XML overview and technologies 444
10.1.1 XML and XML processor 444
10.1.2 DTD and XML schema 445
10.1.3 XSL and XSLT 446
10.1.4 XML namespaces 446
10.1.5 XPath 447
10.2 Rational Application Developer XML tools 447
10.2.1 Create a project for XML sample 448
10.2.2 Work with DTD files 449
10.2.3 Work with XML schema files 458
10.2.4 Work with XML files 475
10.2.5 Work with XSL files 482
10.2.6 Transform an XML file 491
10.2.7 Java code generation 496
10.3 Where to find more information 497

Archived

Chapter 11. Develop Web applications using JSPs and servlets 499
11.1 Introduction to Web applications 500
11.1.1 Concepts and technologies 500
11.1.2 Model-view-controller (MVC) pattern 503
11.2 Web development tooling 505
11.2.1 Web perspective and views 506
11.2.2 Web Projects 507
11.2.3 Web Site Designer 508
11.2.4 Page Designer 509
11.2.5 Page templates 510
11.2.6 CSS Designer 511
11.2.7 Javascript Editor 511
11.2.8 WebArt Designer 511
11.2.9 AnimatedGif Designer 511
11.2.10 File creation wizards 512
11.3 Prepare for the sample 513
11.3.1 ITSO Bank Web application overview 514
11.3.2 Create a Web Project 517
11.3.3 Web Project directory structure 522
11.3.4 Import the ITSO Bank model 524
11.4 Define the site navigation and appearance 524
11.4.1 Launch the Web Site Designer 525
11.4.2 Create a new page template 526
11.4.3 Customize a page template 531
11.4.4 Customize a style sheet 535
11.4.5 Create the Web site navigation and pages 538
11.4.6 Verify the site navigation and page templates 542
11.5 Develop the static Web resources 544
11.5.1 Create the index.html page content (text, links) 544
11.5.2 Create the rates.html page content (tables) 546
11.5.3 Create the insurance.html page content (list) 547
11.5.4 Create the redbank.html page content (forms) 548
11.6 Develop the dynamic Web resources 549
11.6.1 Creating model classes 550
11.6.2 Working with servlets 576
11.6.3 Working with JSPs 595
11.7 Test the application 610
11.7.1 Prerequisites to run sample Web application 611
11.7.2 Run the sample Web application 611
11.7.3 Verify the sample Web application 611

Chapter 12. Develop Web applications using Struts 615
12.1 Introduction to Struts 616

12.1.1 Model-view-controller (MVC) pattern with Struts 616
12.1.2 Rational Application Developer support for Struts 619
12.2 Prepare for the sample application 620
12.2.1 ITSO Bank Struts Web application overview 620
12.2.2 Create a Dynamic Web Project with Struts support 622
12.2.3 Add JDBC driver for Cloudscape to project 628
12.2.4 Set up the sample database 629
12.2.5 Configure the data source 630
12.3 Develop a Web application using Struts 632
12.3.1 Create the Struts components 632
12.3.2 Realize the Struts components 640
12.3.3 Modify ApplicationResources.properties 652
12.3.4 Struts validation framework 653
12.3.5 Page Designer and the Struts tag library 655
12.3.6 Using the Struts configuration file editor 659
12.4 Import and run the Struts sample application 665
12.4.1 Import the Struts Bank Web application sample 665
12.4.2 Prepare the application and sample database 666
12.4.3 Run the Struts Bank Web application sample 666

Chapter 13. Develop Web applications using JSF and SDO 673
13.1 Introduction to JSF and SDO 674
13.1.1 JavaServer Faces (JSF) overview 674
13.1.2 Service Data Objects (SDO) 678
13.2 Prepare for the sample 678
13.2.1 Create a Dynamic Web Project 679
13.2.2 Set up the sample database 681
13.2.3 Configure the data source via the enhanced EAR 681
13.3 Develop a Web application using JSF and SDO 684
13.3.1 Create a page template 684
13.3.2 Useful views for editing page template files 687
13.3.3 Customize the page template 695
13.3.4 Create JSF resources using the Web Diagram tool 700
13.3.5 Edit a JSF page 715
13.3.6 Completing the SDO example 730
13.4 Run the sample Web application 746
13.4.1 Prerequisites to run sample Web application 746
13.4.2 Run the sample Web application 747
13.4.3 Verify the sample Web application 747

Chapter 14. Develop Web applications using EGL 751
14.1 Introduction to EGL 752
14.1.1 Programming paradigms 752

14.1.2 IBM Enterprise Generation Language . . . 753
14.1.3 IBM EGL and Rational brand software . . . 758
14.1.4 IBM EGL feature enhancements . . . 759
14.1.5 Where to find more information on EGL . . . 761
14.2 IBM EGL tooling in Rational Developer products . . . 761
14.2.1 EGL preferences . . . 761
14.2.2 EGL perspective and views . . . 762
14.2.3 EGL projects . . . 763
14.2.4 EGL wizards . . . 763
14.2.5 EGL migration . . . 764
14.2.6 EGL debug support . . . 765
14.2.7 EGL Web application components . . . 765
14.3 Prepare for the sample application . . . 767
14.3.1 Install the EGL component of Rational Application Developer . . . 768
14.3.2 Enable the EGL development capability . . . 771
14.3.3 Install DB2 Universal Database . . . 773
14.3.4 Create an EGL Web Project . . . 773
14.3.5 Set up the sample database . . . 777
14.3.6 Configure EGL preferences for SQL database connection . . . 779
14.3.7 Configure the data source . . . 780
14.3.8 Configure the DB2 JDBC class path environment variables . . . 783
14.4 Develop the Web application using EGL . . . 784
14.4.1 Create the EGL data parts . . . 785
14.4.2 Create and customize a page template . . . 799
14.4.3 Create the Faces JSPs using the Web Diagram tool . . . 802
14.4.4 Add EGL components to the Faces JSPs . . . 806
14.5 Import and run the sample Web application . . . 816
14.5.1 Import the EGL Web application sample . . . 816
14.5.2 Prerequisites . . . 816
14.5.3 Generate Java from EGL source . . . 817
14.5.4 Run the sample EGL Web application . . . 818
14.6 Considerations for exporting an EGL project . . . 820
14.6.1 Reduce the file size of the Project Interchange file . . . 821
14.6.2 Manually adding the runtime libraries after migration . . . 822
14.6.3 Export WAR/EAR with source . . . 823

Chapter 15. Develop Web applications using EJBs . . . 827
15.1 Introduction to Enterprise JavaBeans . . . 828
15.1.1 What is new . . . 828
15.1.2 Enterprise JavaBeans overview . . . 828
15.1.3 EJB server . . . 831
15.1.4 EJB container . . . 832
15.1.5 EJB components . . . 834

15.2 RedBank sample application overview 840
15.3 Prepare for the sample 844
15.3.1 Required software 844
15.3.2 Create and configure the EJB projects 844
15.3.3 Create an EJB project 845
15.3.4 Configure the EJB projects 849
15.3.5 Import BankBasicWeb Project 853
15.3.6 Set up the sample database 854
15.3.7 Configure the data source 856
15.4 Develop an EJB application 858
15.4.1 Create the entity beans 859
15.4.2 Create the entity relationships 872
15.4.3 Customize the entity beans and add business logic 880
15.4.4 Creating custom finders 890
15.4.5 Object-relational mapping 892
15.4.6 Implement the session facade 901
15.5 Testing EJB with the Universal Test Client 915
15.6 Adapting the Web application 919

Chapter 16. Develop J2EE application clients 925
16.1 Introduction to J2EE application clients 926
16.2 Overview of the sample application 928
16.3 Preparing for the sample application 931
16.3.1 Import the base enterprise application sample 932
16.3.2 Set up the sample database 933
16.3.3 Configure the data source 934
16.3.4 Test the imported code 936
16.4 Develop the J2EE application client 936
16.4.1 Create the J2EE application client projects 937
16.4.2 Configure the J2EE application client projects 938
16.4.3 Import the graphical user interface and control classes 939
16.4.4 Create the BankDesktopController class 940
16.4.5 Complete the BankDesktopController class 942
16.4.6 Register the BankDesktopController class as the Main class 946
16.5 Test the J2EE application client 946
16.6 Package the application client project 949

Chapter 17. Develop Web Services applications 951
17.1 Introduction to Web Services 952
17.1.1 Service-oriented architecture (SOA) 952
17.1.2 Web Services as an SOA implementation 953
17.1.3 Related Web Services standards 955
17.2 Web Services tools in Application Developer 957

Archived

17.2.1 Creating a Web Service from existing resources 957
17.2.2 Creating a skeleton Web Service 958
17.2.3 Client development 958
17.2.4 Testing tools for Web Services 959
17.3 Preparing for the samples 959
17.3.1 Import the sample code. 959
17.3.2 Enable the Web Services Development capability 960
17.3.3 Set up the sample back-end database 961
17.3.4 Add Cloudscape JDBC driver (JAR) to the project 962
17.3.5 Define a server to test the application. 963
17.3.6 Test the application 963
17.4 Create a Web Service from a JavaBean 964
17.4.1 Create a Web Service using the Web Service wizard. 964
17.4.2 Resources generated by the Web Services wizard 968
17.4.3 Test the Web Service using the Web Services Explorer. 971
17.4.4 Generate and test the client proxy 973
17.4.5 Monitor the Web Service using the TCP/IP Monitor 976
17.5 Create a Web Service from an EJB 980
17.6 Web Services security. 980
17.7 Publish a Web Service using UDDI. 982

Chapter 18. Develop portal applications. 985
18.1 Introduction to portals 986
18.1.1 Portal concepts and definitions 986
18.1.2 IBM WebSphere Portal 989
18.1.3 IBM Rational Application Developer 989
18.2 Developing applications for WebSphere Portal. 992
18.2.1 Portal samples and tutorials 992
18.2.2 Development strategy 993
18.2.3 Portal tools for developing portals. 996
18.2.4 Portal tools for developing portlets 1002
18.2.5 Portal tools for testing and debugging portlets 1016
18.2.6 Portal tools for deploying and managing portlets 1020
18.2.7 Enterprise Application Integration Portal Tools. 1022
18.2.8 Coexistence and migration of tools and applications 1023
18.3 Portal development scenario. 1025
18.3.1 Prepare for the sample 1025
18.3.2 Create a portal project. 1026
18.3.3 Add and modify a portal page 1027
18.3.4 Create and modify two portlets 1030
18.3.5 Add portlets to a portal page. 1033
18.3.6 Run the project in the test environment 1037

Part 3. Test and debug applications 1041

Chapter 19. Servers and server configuration 1043
19.1 Introduction to server configuration 1044
19.1.1 Supported test server environments 1045
19.1.2 Local vs. remote test environments 1046
19.1.3 Commands to manage test servers 1046
19.2 Configure a WebSphere V6 Test Environment 1046
19.2.1 Understanding WebSphere Application Server V6.0 profiles . . . 1047
19.2.2 WebSphere Application Server V6 installation 1050
19.2.3 WebSphere Application Server V6 profile creation 1051
19.2.4 Define a new server in Rational Application Developer 1057
19.2.5 Verify the server 1061
19.2.6 Customize a server in Rational Application Developer 1062
19.3 Add a project to a server 1064
19.3.1 Considerations for adding a project to a server 1064
19.3.2 Add a project to a server 1065
19.4 Remove a project from a server 1066
19.4.1 Remove a project via Rational Application Developer 1066
19.4.2 Remove a project via WebSphere Administrative Console 1067
19.5 Publish application changes 1068
19.6 Configure application and server resources 1069
19.6.1 Configure application resources 1069
19.6.2 Configure server resources 1078
19.6.3 Configure messaging resources 1079
19.6.4 Configure security 1079
19.7 TCP/IP Monitor 1079

Chapter 20. JUnit and component testing 1081
20.1 Introduction to application testing 1082
20.1.1 Test concepts 1082
20.1.2 Benefits of unit and component testing 1085
20.1.3 Eclipse Hyades 1086
20.2 JUnit testing 1087
20.2.1 JUnit fundamentals 1087
20.2.2 Prepare for the sample 1089
20.2.3 Create the JUnit test case 1089
20.2.4 Run the JUnit test case 1098
20.3 Automated component testing 1102
20.3.1 Prepare for the sample 1102
20.3.2 Create a test project 1103
20.3.3 Create a Java component test 1103
20.3.4 Complete the component test code 1107

Archive

20.3.5 Run the component test 1110
20.4 Web application testing 1112
20.4.1 Preparing for the sample 1113
20.4.2 Create a Java project 1113
20.4.3 Create (record) a test 1113
20.4.4 Edit the test 1115
20.4.5 Generate an executable test 1115
20.4.6 Create a deployment definition 1116
20.4.7 Run the test 1117
20.4.8 Analyze the test results 1118

Chapter 21. Debug local and remote applications 1121
21.1 Introduction to the debug tooling 1122
21.1.1 Summary of new Version 6 features 1122
21.1.2 Supported languages and environments 1124
21.1.3 General functionality 1124
21.1.4 Drop-to-frame 1126
21.1.5 View Management 1127
21.1.6 XSLT debugger 1128
21.2 Prepare for the sample 1131
21.3 Debug a Web application on a local server 1132
21.3.1 Set breakpoints in a servlet 1132
21.3.2 Set breakpoints in a JSP 1135
21.3.3 Start the application for debugging 1136
21.3.4 Run the application in the debugger 1136
21.3.5 Debug view with stack frames 1141
21.3.6 Debug functions 1141
21.3.7 Breakpoints view 1142
21.3.8 Watch variables 1142
21.3.9 Inspect variables 1143
21.3.10 Evaluate an expression 1143
21.3.11 Debug a JSP 1145
21.4 Debug a Web application on a remote server 1145
21.4.1 Export the BankBasicWeb project to a WAR file 1145
21.4.2 Deploy the BankBasicWeb.war 1146
21.4.3 Install the IBM Rational Agent Controller 1147
21.4.4 Configure debug on remote WebSphere Application Server . . . 1147
21.4.5 Attach to the remote server in Rational Application Developer . . 1148
21.4.6 Debug the application on the remote server 1151

Part 4. Deploy and profile applications 1153

Chapter 22. Build applications with Ant 1155
22.1 Introduction to Ant 1156

22.1.1 Ant build files . . . 1156
22.1.2 Ant tasks . . . 1157
22.2 New features . . . 1157
22.2.1 Code Assist . . . 1158
22.2.2 Code snippets . . . 1159
22.2.3 Format an Ant script . . . 1163
22.2.4 Define format of an Ant script . . . 1164
22.2.5 Problem view . . . 1166
22.3 Build a simple Java application . . . 1167
22.3.1 Prepare for the sample . . . 1168
22.3.2 Create a build file . . . 1169
22.3.3 Project definition . . . 1171
22.3.4 Global properties . . . 1171
22.3.5 Build targets . . . 1171
22.3.6 Run Ant . . . 1173
22.3.7 Ant Log Console . . . 1176
22.3.8 Rerun Ant . . . 1177
22.3.9 Forced build . . . 1177
22.3.10 Classpath problem . . . 1178
22.3.11 Run the sample application to verify the Ant build . . . 1178
22.4 Build a J2EE application . . . 1178
22.4.1 J2EE application deployment packaging . . . 1179
22.4.2 Prepare for the sample . . . 1179
22.4.3 Create the build script . . . 1180
22.4.4 Run the Ant J2EE application build . . . 1183
22.5 Run Ant outside of Application Developer . . . 1185
22.5.1 Prepare for the headless build . . . 1186
22.5.2 Run the headless Ant build script . . . 1187

Chapter 23. Deploy enterprise applications . . . 1189
23.1 Introduction to application deployment . . . 1190
23.1.1 Common deployment considerations . . . 1190
23.1.2 J2EE application components and deployment modules . . . 1191
23.1.3 Java and WebSphere class loader . . . 1191
23.1.4 Deployment descriptors . . . 1196
23.1.5 WebSphere deployment architecture . . . 1199
23.2 Prepare for the sample . . . 1212
23.2.1 Review the deployment scenarios . . . 1213
23.2.2 Install prerequisite software . . . 1213
23.2.3 Import the sample application Project Interchange file . . . 1214
23.2.4 Set up the sample database . . . 1215
23.3 Package the application for deployment . . . 1218
23.3.1 Packaging recommendations . . . 1218

Archived

23.3.2 Generate the EJB to RDB mapping 1218
23.3.3 Customize the deployment descriptors 1220
23.3.4 Remove the Enhanced EAR datasource 1221
23.3.5 Generate the deploy code 1222
23.3.6 Export the EAR 1222
23.4 Deploy the enterprise application 1224
23.4.1 Configure the data source in WebSphere Application Server . . . 1225
23.4.2 Deploy the EAR 1229
23.5 Verify the application 1230

Chapter 24. Profile applications 1237
24.1 Introduction to profiling 1238
24.1.1 Profiling features 1238
24.1.2 Profiling architecture 1242
24.1.3 Profiling and Logging perspective 1244
24.1.4 Profiling sets 1245
24.2 Prepare for the profiling sample 1246
24.2.1 Prerequisites hardware and software 1247
24.2.2 Enable the Profiling and Logging capability 1247
24.2.3 Import the sample project interchange file 1248
24.2.4 Publish and run sample application 1249
24.3 Profile the sample application 1249
24.3.1 Start server in profile mode 1249
24.3.2 Collect profile information 1253
24.3.3 Analysis of code coverage information 1253

Part 5. Team development 1255

Chapter 25. Rational ClearCase integration 1257
25.1 Introduction to IBM Rational ClearCase 1258
25.1.1 IBM Rational Application Developer ClearCase overview 1258
25.1.2 IBM Rational ClearCase terminology 1259
25.1.3 IBM Rational ClearCase LT installation 1260
25.1.4 IBM Rational Application Developer integration for ClearCase . . 1260
25.2 Integration scenario overview 1263
25.3 ClearCase setup for a new project 1264
25.3.1 Enable Team capability in preferences 1264
25.3.2 Create new ClearCase project 1265
25.3.3 Join a ClearCase project 1268
25.3.4 Create a Web project 1274
25.3.5 Add a project to ClearCase source control 1274
25.4 Development scenario 1277
25.4.1 Developer 1 adds a servlet 1277
25.4.2 Developer 1 delivers work to the integration stream 1279

Archived

25.4.3 Developer 1 makes a baseline 1282
25.4.4 Developer 2 joins the project. 1284
25.4.5 Developer 2 imports projects into Application Developer 1287
25.4.6 Developer 2 modifies the servlet. 1289
25.4.7 Developer 2 delivers work to the integration stream 1292
25.4.8 Developer 1 modifies the servlet. 1293
25.4.9 Developer 1 delivers new work to the integration stream 1294

Chapter 26. CVS integration 1299
26.1 Introduction to CVS 1300
26.1.1 CVS features. 1300
26.1.2 New V6 features for team development 1301
26.2 CVSNT Server implementation 1301
26.2.1 CVS Server installation 1302
26.2.2 CVS Server repository configuration. 1303
26.2.3 Create the Windows users and groups used by CVS 1306
26.2.4 Verify the CVSNT installation 1307
26.2.5 Create CVS users 1308
26.3 CVS client configuration for Application Developer. 1309
26.3.1 Configure CVS Team Capabilities 1309
26.3.2 Access the CVS Repository 1310
26.4 Configure CVS in Rational Application Developer 1312
26.4.1 Configure Rational Application Developer CVS preferences . . . 1312
26.5 Development scenario. 1321
26.5.1 Create and share the project (step 1 - cvsuser1) 1322
26.5.2 Add a shared project to the workspace (step 2 - cvsuser2) 1327
26.5.3 Modifying the Servlet (step 2 - cvsuser1) 1332
26.5.4 Synchronize with repository (step 3 - cvsuser1) 1333
26.5.5 Parallel development (step 4 - cvsuser1 and cvsuser2) 1335
26.5.6 Versioning (step 5- cvsuser1) 1342
26.6 CVS resource history 1343
26.7 Comparisons in CVS. 1344
26.7.1 Comparing workspace file with repository. 1345
26.7.2 Comparing two revisions in repository 1347
26.8 Annotations in CVS 1348
26.9 Branches in CVS. 1350
26.9.1 Branching 1350
26.9.2 Merging 1354
26.9.3 Refreshing server-defined branches. 1357
26.10 Work with patches. 1360
26.11 Disconnecting a project. 1360
26.12 Synchronize perspective 1361
26.12.1 Custom configuration of resource synchronization 1362

Archived

26.12.2 Schedule synchronization 1367

Part 6. Appendixes 1369

Appendix A. IBM product installation and configuration tips 1371
IBM Rational Application Developer V6 installation 1372
Rational Application Developer installation 1372
WebSphere Portal V5.0 Test Environment installation 1376
WebSphere Portal V5.1 Test Environment installation 1377
Rational Application Developer Product Updater - Interim Fix 0004 1380
IBM Rational Agent Controller V6 installation 1382
IBM Rational ClearCase LT installation 1385
IBM DB2 Universal Database V8.2 installation 1387
IBM WebSphere Application Server V6 installation 1387
WebSphere Application Server messaging configuration 1389
Configure the service bus 1390
Configure the bus members 1390
Configure the destinations 1390
Verify the messaging engine startup 1391
Configure JMS connection queue factory 1391
Configure the destination JMS queue 1392
Configuration of a JMS activation specification 1392

Appendix B. Additional material 1395
Locating the Web material 1396
System requirements for downloading the Web material 1396
Unpack the 6449code.zip 1396
Description of sample code 1396
Import sample code from a Project Interchange file 1398

Related publications 1401
IBM Redbooks 1401
Other publications 1402
Online resources 1402
How to get IBM Redbooks 1404
Help from IBM 1404

Index 1405

Archived

Notices

This information was developed for products and services offered in the U.S.A.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:
IBM Director of Licensing, IBM Corporation, North Castle Drive Armonk, NY 10504-1785 U.S.A.

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:
This information contains sample application programs in source language, which illustrates programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. You may copy, modify, and distribute these sample programs in any form without payment to IBM for the purposes of developing, using, marketing, or distributing application programs conforming to IBM's application programming interfaces.

© Copyright IBM Corp. 2005. All rights reserved.

Trademarks

The following terms are trademarks of the International Business Machines Corporation in the United States, other countries, or both:

AIX®
Balance®
ClearCase MultiSite®
ClearCase®
Cloudscape™
DB2 Universal Database™
DB2®
developerWorks®
Informix®
IBM®
ibm.com®
IMS™
iSeries™
Lotus Notes®
Lotus®
Notes®
OS/390®
OS/400®
Rational Rose®
Rational Unified Process®
Rational®
Redbooks™
Redbooks (logo) ™
RUP®
S/390®
ThinkCentre™
VisualAge®
VisualGen®
WebSphere®
Workplace™
Workplace Web Content Management™
XDE™
z/OS®
zSeries®

The following terms are trademarks of other companies:

Java and all Java-based trademarks and logos are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

Intel, Intel Inside (logos), MMX, and Pentium are trademarks of Intel Corporation in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Linux is a trademark of Linus Torvalds in the United States, other countries, or both.

Other company, product, and service names may be trademarks or service marks of others.

Archive

Preface

IBM® Rational® Application Developer V6.0 is the full function Eclipse 3.0 based development platform for developing Java™ 2 Platform Standard Edition (J2SE) and Java 2 Platform Enterprise Edition (J2EE) applications with a focus on applications to be deployed to IBM WebSphere® Application Server and IBM WebSphere Portal. Rational Application Developer provides integrated development tools for all development roles, including Web developers, Java developers, business analysts, architects, and enterprise programmers.

This IBM Redbook is a programming guide that highlights the features and tooling included with IBM Rational Application Developer V6.0. Many of the chapters provide working examples that demonstrate how to use the tooling to develop applications as well as achieve the benefits of visual and rapid Web development.

This book consists of six parts:

- Introduction to Rational Application Developer
- Develop applications
- Test and debug applications
- Deploy and profile applications
- Team development
- Appendixes

The team that wrote this redbook

This redbook was produced by a team of specialists from around the world working at the International Technical Support Organization, Raleigh Center.

Archive

© Copyright IBM Corp. 2005. All rights reserved.

Figure 1 IBM Redbook team (top, l-r: Neil Weightman, Fabio Ferraz, Richard Raszka, Juha Nevalainen; bottom, l-r: Hari Kanangi, John Ganci, Nicolai Nielsen, Kiriya Keat)

John Ganci is a Consulting IT Specialist at the IBM ITSO, Raleigh Center, USA. John has 15 years of experience in application design, development, testing, and consulting. His areas of expertise include e-business integration, WebSphere Application Server, e-commerce, portals, pervasive computing, technical team leadership, Linux®, and J2EE programming.

Daniel Farrell is a Certified Specialist in the IBM Software Group USA. He has a Masters degree in Computer Science from Regis University and is currently pursuing a PhD from Clemson in Adult Education and Human Resource Studies. He has 20 years of experience working with Informix® and related database technologies, and more recently has been focusing on IBM EGL and supporting tooling.

Fabio Ferraz is the Chief Consultant for Advus Corp, São Paulo, Brazil. He has 12 years of experience in the IT field, with eight of those dedicated to e-business.

Ed Gondek is a Product Manager in the USA with IBM Software Group - Rational products division. He has 25 years of Information Systems and software management experience with expertise in 4GLs, Web-based application design, Web analytics, business intelligence tooling, and software product delivery.

Hari Kanangi is a Senior Technical Consultant for Stratus Solutions, Inc. He has nine years of industry experience. Over the past six years, Hari has focused on WebSphere, J2EE, and Java-based technologies. His expertise includes architecting, designing, developing, and deploying J2EE-based solutions. He is a Sun-certified Java Programmer, architect, and a WebSphere-certified specialist.

Kiriya Keat is an IT Specialist in Australia with IBM Global Services. He has five years of experience in the Web development field. His areas of expertise include e-business integration, application architecture and development, technical team leadership, and WebSphere solutions.

George Kroner is a Co-op IT Specialist at the IBM ITSO Center in Raleigh, North Carolina. He is currently pursuing a Bachelor of Science degree in Information Sciences and Technology at Pennsylvania State University. His interests include Web applications, wireless/mobile computing, and intelligent interfaces.

Juha Nevalainen is an Consulting IT Specialist for Rational Technical Sales with IBM Finland. His areas of expertise include Rational brand software products, WebSphere Commerce, e-business, and systems integration.

Nicolai Nielsen is an Advisory IT Specialist with IBM Global Services, Denmark. He has nine years of experience in the fields of consulting, application development, and systems integration. Nicolai holds a degree in engineering from the Technical University of Denmark. He has written extensively about WebSphere Commerce, J2EE application development, and has worked on several WebSphere Commerce projects over the last three years.

Richard Raszka is a Senior IT Specialist in IBM Global Services, Application Management Services, Australia. He has 19 years of IT experience in the fields of application development, systems integration (particularly of e-business/e-commerce solutions), project planning, compiler development, configuration management, graphics and CAD/CAM development, simulation, and agent-oriented artificial intelligence. He holds degrees in Mechanical Engineering, Mathematical and Computer Science from the University of Adelaide and a Masters of Technology Management from Monash University in Melbourne. His areas of expertise include end-to-end systems integration, application design including patterns, application architecture, and development,

technical team leadership, e-business solutions, WebSphere Application Server, UNIX®, J2EE architecture, and Java development.

Neil Weightman is a Technical Instructor in the United Kingdom. He has 15 years of experience in the software development field. He holds a degree in Physics from Imperial College, University of London, and a masters in Scientific Computing from South Bank University, London. His areas of expertise include J2EE component development, Web services development, and best practices.

Thanks to the following people for their contributions to this project:

- Ueli Wahli, Ian Brown, Fabio Ferraz, Maik Schumacher, Henrik Sjostrand who wrote the previous redbook edition, *WebSphere Studio Application Developer Version 5 Programming Guide*, SG24-6957
- Special thanks to Eric Erpenbach from the IBM WebSphere Technology and Training organization for providing early training materials on IBM Rational Application Developer V6.0, which was a tremendous help in ramping up on the new features for many topics found in this book
- Beverly DeWitt, Rational Application Developer Product Manager, IBM Toronto, Canada
- Geni Hutton, Rational Application Developer Product Manager, IBM Toronto, Canada
- Todd Britton, Manager for IBM Rational Application Developer Tools Development, IBM Raleigh, USA
- Kate Price, Rational Application Developer Information Development, IBM Toronto, Canada
- Leigh Davidson, Rational Application Developer Information Development Editor, IBM Toronto, Canada
- Tim Deboer, IBM WebSphere Tools, IBM Toronto, Canada
- Prem Lall, IT Specialist for IBM Global Services, IBM Gaithersburg, USA
- Chris Feldhacker, Principal Financial Group

Become a published author

Join us for a two- to six-week residency program! Help write an IBM Redbook dealing with specific products or solutions, while getting hands-on experience with leading-edge technologies. You'll team with IBM technical professionals, Business Partners and/or customers.

Archive

Your efforts will help increase product acceptance and customer satisfaction. As a bonus, you'll develop a network of contacts in IBM development labs, and increase your productivity and marketability.

Find out more about the residency program, browse the residency index, and apply online at:

ibm.com/redbooks/residencies.html

Comments welcome

Your comments are important to us!

We want our Redbooks™ to be as helpful as possible. Send us your comments about this or other Redbooks in one of the following ways:

- Use the online **Contact us** review redbook form found at:

  ibm.com/redbooks

- Send your comments in an email to:

  redbook@us.ibm.com

- Mail your comments to:

  IBM Corporation, International Technical Support Organization
  Dept. HZ8 Building 662
  P.O. Box 12195
  Research Triangle Park, NC 27709-2195

Summary of changes

This section describes the technical changes made in this edition of the book and in previous editions. This edition may also include minor corrections and editorial changes that are not identified.

Summary of Changes
for SG24-6449-00
for the Rational Application Developer V6 Programming Guide

June 2005, First Edition

This book is a major rewrite of *WebSphere Studio Application Developer Version 5 Programming Guide*, SG24-6957. The previous book was based on Application Developer Version 5; this book is based on Version 6.

Archived

This revision reflects the addition, deletion, or modification of new and changed information described below.

New information

- Broaden appeal
  - Improves user assistance and ease of learning
  - Progressive disclosure of tools
- Raise productivity
  - Rapid Web and Portal application development
  - Rapid deployment to WebSphere Application Server V6.0
- Extended integration
  - Integration of Rational technology to enhance the application development lifecycle and raise productivity
  - Enhance portal as a first-class deployment target
- Maintain standards and middleware support
  - Java 2 Platform Enterprise Edition V1.4 specification compliance
  - Support for many new integrated test servers including WebSphere Application Server V6.0/V5.x, and WebSphere Portal V5.0.2.2
  - New and enhanced tooling (Web Services, EGL, Visual UML, portal, Struts, JSF, SDO, etc.).

Changed information

- General update of existing information to Version 6

Deleted information

- Developing Web applications with database access using DB beans. The preferred method of database access is Service Data Objects (SDO).

Archived

Part 1

Introduction to Rational Application Developer

© Copyright IBM Corp. 2005. All rights reserved.

Archived

1

Introduction

IBM Rational Application Developer V6.0 is the full function development platform for developing Java 2 Platform Standard Edition (J2SE) and Java 2 Platform Enterprise Edition (J2EE) applications with a focus on applications to be deployed to IBM WebSphere Application Server and IBM WebSphere Portal.

This chapter contains an introduction to the concepts, packaging, and features of IBM Rational Application Developer V6.0 and IBM Rational Web Developer V6.0 products.

The chapter is organized into the following sections:

- Introduction and concepts
- Product packaging
- Product features
- Sample code

Note: Although this chapter does contain information on both IBM Rational Application Developer V6.0 and IBM Rational Web Developer V6.0, the majority of the chapters and samples found in this book were only tested with IBM Rational Application Developer V6.0.

© Copyright IBM Corp. 2005. All rights reserved.

1.1 Introduction and concepts

This section provides an introduction to the Rational Software Development Platform, Eclipse, Rational Application Developer, and Rational Web Developer.

Rational products help your business or organization manage the entire software development process. Software modelers, architects, developers, and testers can use the same team-unifying Rational Software Development Platform tooling to be more efficient in exchanging assets, following common processes, managing change and requirements, maintaining status, and improving quality.

1.1.1 IBM Rational Software Development Platform

The IBM Rational Software Development Platform is not a single product, but rather an integrated set of products that share a common technology platform built on the Eclipse 3.0 framework in support of each phase of the development life cycle.

The IBM Rational Software Development Platform provides a team-based environment with capabilities that are optimized for the key roles of a development team including business analyst, architect, developer, tester, and deployment manager. It enables a high degree of team cohesion through shared access to common requirements, test results, software assets, and workflow and process guidance. Combined, these capabilities improve both individual and team productivity.

Figure 1-1 on page 5 provides perspective on how the Rational Application Developer and Rational Web Developer products fit within the IBM Rational Software Development Platform product set.

Archived

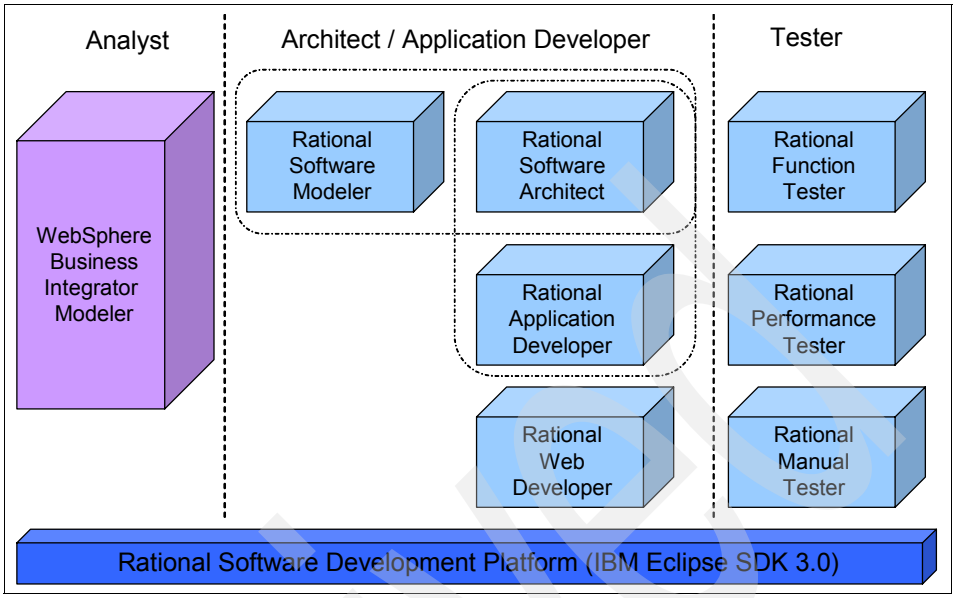


Figure 1-1 Rational Software Development Platform products

We have included a brief description of each of the products included in the IBM Rational Software Development Platform (see Figure 1-1) that share common tooling based on the IBM Eclipse SDK V3.0 (IBM-supported Eclipse 3.0):

- Rational Software Modeler

  The Software Modeler is a UML-based visual modeling and design tool for system analysts, software architects, and designers who need to clearly define and communicate their architectural specifications to stakeholders.

  This product was known in previous releases as Rational XDE™ Modeler and is targeted at development shops where the business analyst has a distinct role of architecture and design (no development).

- Rational Software Architect

  The Software Architect is a design and construction tool that leverages model-driven development with UML for creating well-architected applications, including those based on a Service Oriented Architecture (SOA). It unifies modeling, Java structural review, Web Services, J2SE, J2EE, database, XML, Web development, and process guidance for architects and senior developers creating applications in Java or C++.

  This product was known in previous releases as Rational Rose® and Rational XDE for Java. Software Architect includes architecture and design capability as well as full J2EE development functionality provided by Rational Application Developer. This product is targeted at development shops where

the architect has a strong architecture and design role, as well as application development. If architects only need the modeling functionality, they should use the Rational Software Modeler product.

- Rational Web Developer

  The IBM Rational Web Developer (or simply Web Developer) extends the capabilities of Eclipse 3.0 with visual tools for Web, Java, and rich client applications, and full support for XML, Web services, and Enterprise Generation Language.

  In previous releases this product was known as WebSphere Studio Site Developer. Rational Web Developer is packaged with IBM WebSphere Application Server Express V6.0.

- Rational Application Developer

  The IBM Rational Application Developer is a full suite of development, analysis, and deployment tools for rapidly implementing J2EE applications, Enterprise JavaBeans, portlets, and Web applications.

  In previous releases this product was known as WebSphere Studio Application Developer and is targeted at J2EE developers.

  Note: IBM Rational Application Developer V6.0 is the focus of this book.

- Rational Functional Tester

  The Rational Function Tester is an automated testing tool that tests Java, HTML, VB.NET, and Windows® applications. It provides the capability to record robust scripts that can be played back to validate new builds of an application.

- Rational Performance Tester

  The Rational Performance Tester is a multi-user system performance test product designed to test Web applications, and focuses on ease-of-use and scalability.

- WebSphere Business Integrator Modeler

  The WebSphere Business Integrator Modeler does not carry the Rational brand name, but is an important product of the Rational Software Development Platform. WebSphere Business Integrator Modeler targets the business analyst who models business processes. WebSphere Business Integrator Modeler can be used to generate Business Process Execution Language (BPEL) definitions to be deployed to WebSphere Business Integrator production environments. The WebSphere Business Integrator Modeler BPEL provides a more seamless move to production and eliminates the need to create Visio diagrams and then move to production.

1.1.2 Version 6 terminology

Table 1-1 provides a basic terminology comparison from Version 6 and Version 5 for reference purposes.

Table 1-1 Terminology

Version 6	Version 5
Rational Developer **Note**: Used to describe development products built on common Eclipse base	WebSphere Studio
Rational Application Developer (known as Application Developer)	WebSphere Studio Application Developer
Rational Web Developer (known as Web Developer)	WebSphere Studio Site Developer
IBM Eclipse SDK 3.0 **Note**: IBM branded and value-added version of Eclipse SDK 3.0	WebSphere Studio Workbench (IBM supported Eclipse 2.x)
Workbench	Workbench
IBM Rational Software Development Platform **Note**: Used to describe product set built on common Eclipse 3.0 platform	N/A

1.1.3 Application development challenges

To better grasp the business value that IBM Rational Application Developer V6.0 provides, it is important to understand the challenges businesses face in application development.

Table 1-2 highlights the key application development challenges as well as desired development tooling solutions.

Table 1-2 Application development challenges

Challenges	Solution tooling
Application development is complex, time consuming, and error prone.	Raise productivity by automating time consuming and error prone tasks.
Highly skilled developers are required and in short supply.	Improve code quality early in the development life cycle.

Archived

Challenges	Solution tooling
Learning curves are long.	Shorten learning curves by providing Rapid Application Development (RAD) tooling (visual layout and design, resusable components, code generators, etc.).

1.1.4 Key themes of Version 6

There are many very significant enhancements and features in IBM Rational Application Developer V6.0. We have listed the key themes of Version 6 tooling:

- Broaden appeal.
  - Improves user assistance and ease of learning
  - Progressive disclosure of tools
- Raise productivity.
  - Rapid Web and Portal application development
  - Rapid deployment to WebSphere Application Server and WebSphere Portal
- Extended integration.
  - Integration of Rational technology to enhance the application development life cycle and raise productivity
  - Enhances portal as a first-class deployment target
- Maintain standards and middleware support.
  - Java 2 Platform Enterprise Edition V1.4 specification compliance
  - Support for many new integrated test servers including WebSphere Application Server V6.0/V5.x and WebSphere Portal V5.0.2.2/5.1
  - New and enhanced tooling (Web Services, EGL, Visual UML, portal, etc.)
- Team unifying platform.

  As described in “IBM Rational Software Development Platform” on page 4, Rational Application Developer and Rational Web Developer are products included in the IBM Rational Software Development Platform product set that promote team development.

We provide more detail on the these new features in “Summary of new features in Version 6” on page 14, as well as throughout the chapters of this book.

Archived

1.2 Product packaging

This section highlights the product packaging for IBM Rational Web Developer V6.0 and IBM Rational Application Developer V6.0.

1.2.1 Rational Developer supported platforms and databases

This section describes the platforms and databases supported by the Rational Developer products.

Supported operating system platforms

IBM Rational Application Developer V6.0 supports the following operating systems:

- Microsoft® Windows:
  - Windows XP with Service Packs 1 and 2
  - Windows 2000 Professional with Service Packs 3 and 4
  - Windows 2000 Server with Service Packs 3 and 4
  - Windows 2000 Advanced Server with Service Packs 3 and 4
  - Windows Server 2003 Standard Edition
  - Windows Server 2003 Enterprise Edition
- Linux on Intel®:
  - Red Hat Enterprise Linux Workstation V3 (all service packs)
  - SuSE Linux Enterprise Server (SLES) V9 (all service packs)

The IBM Rational Agent Controller included with IBM Rational Application Developer V6.0 is supported on many platforms running WebSphere Application Server. For details refer to the *Installation Guide, IBM Rational Application Developer V6.0* product guide (install.html) found on the IBM Rational Application Developer V6 Setup CD 1.

Supported databases

IBM Rational Application Developer V6.0 supports the following database products:

- IBM Cloudscape™ V5.0
- IBM Cloudscape V5.1 (bundled with the WebSphere Application Server V6.0 Test Environment)
- IBM DB2 Universal Database V8.1
- IBM DB2 Universal Database V8.2
- IBM DB2 Universal Database Express V8.1
- IBM DB2 Universal Database Express V8.2
- IBM DB2 Universal Database for iSeries™ V4R5
- IBM DB2 Universal Database for iSeries V5R1

Archived

- IBM DB2 Universal Database for iSeries V5R2
- IBM DB2 Universal Database for iSeries V5R3
- IBM DB2 Universal Database for z/OS® and OS/390® V7
- IBM DB2 Universal Database for z/OS V8
- Informix Dynamic Server V7.3
- Informix Dynamic Server V9.2
- Informix Dynamic Server V9.3
- Informix Dynamic Server V9.4
- Microsoft SQL Server V7.0
- Microsoft SQL Server 2000
- Oracle8i V8.1.7
- Oracle9i
- Oracle10g
- Sybase Adaptive Server Enterprise V12
- Sybase Adaptive Server Enterprise V12.5

1.2.2 Rational Application Developer V6 product packaging

Table 1-3 lists the software CDs included with IBM Rational Application Developer V6.0.

Table 1-3 IBM Rational Application Developer V6.0 product packaging

CDs	Windows	Linux
IBM Rational Application Developer V6.0 - Core installation files (required)	X	X
IBM WebSphere Application Server V6.0 Integrated Test Environment	X	X
IBM WebSphere Application Server V5.0.2/V5.1 Integrated Test Environment	X	X
IBM Enterprise Generation Language (EGL)	X	X
IBM Rational Application Developer V6.0 - Language Pack	X	X
IBM WebSphere Portal V5.0.2.2 Integrated Test Environment **Note**: IBM WebSphere Portal V5.1 Test Environment is installed via its own installer on a separate CD packaged with Rational Application Developer.	X	N/A

Archived

CDs	Windows	Linux
IBM WebSphere Portal V5.1 Test Environment The following CDs are included for the WebSphere Portal V5.1 Test Environment (separate installer from Rational Application Developer): ► IBM WebSphere Portal V5.1 - Portal Install (Setup) CD ► IBM WebSphere Portal V5.1 - WebSphere Business Integrator Server Foundation (1-1) CD ► IBM WebSphere Portal V5.1 - WebSphere Business Integrator Server Foundation (1-2) CD ► IBM WebSphere Portal V5.1 - WebSphere Business Integrator Server Foundation WebSphere Application Server V5.1 Fixpack 1 (1-15) CD ► IBM WebSphere Portal V5.1 - Portal Server (2) CD ► IBM WebSphere Portal V5.1 - Lotus® Workplace™ Web Content Management (3) CD	X	X
IBM Rational Agent Controller **Note**: Support for many additional platforms	X	X
IBM WebSphere Application Server V5 - Embedded messaging client and server	X	X
IBM Rational ClearCase® LT **Note**: Web download	Web	Web
Crystal Enterprise V10 Professional Edition	X	X
Crystal Enterprise V10 Embedded Edition	X	na
IBM DB2 Universal Database V8.2, Express Edition	X	X
IBM WebSphere Application Server for Developers V6.0 (IBM HTTP Server, Web server plug-ins, DataDirect JDBC drivers, Appl. Clients)	X	X

1.2.3 Rational Web Developer V6 product packaging

Table 1-4 lists the software CDs included with IBM Rational Web Developer V6.0.

Table 1-4 IBM Rational Web Developer V6.0 product packaging

CD title	Windows	Linux
IBM Rational Web Developer V6.0 - Core installation files (required)	X	X
IBM WebSphere Application Server V6.0 Integrated Test Environment	X	X
IBM WebSphere Application Server V5.0.2 and V5.1 Integrated Test Environment	X	X
IBM Enterprise Generation Language (EGL)	X	X
IBM Rational Web Developer V6.0 - Language Pack	X	X

Archived

CD title	Windows	Linux
IBM Rational Agent Controller **Note**: Support for many additional platforms	X	X
IBM DB2 Universal Database V8.2, Express Edition	X	X
IBM WebSphere Application Server for Developers V6.0 (IBM HTTP Server, Web server plug-ins, DataDirect JDBC drivers, Appl. Clients)	X	X

1.3 Product features

This section provides a summary of the new features of IBM Rational Application Developer V6.0 and IBM Rational Web Developer V6.0. We will provide more detailed information on the new features throughout the chapters of this book.

Figure 1-2 on page 13 displays a summary of features found in the IBM Rational Developer V6.0 products. We have organized the description of the product features into the following topics:

- Summary of new features in Version 6
- Specification versions
- Eclipse and IBM Rational Software Development Platform
- Test server environments
- Licensing and installation
- Migration and coexistence
- Tools

Archived

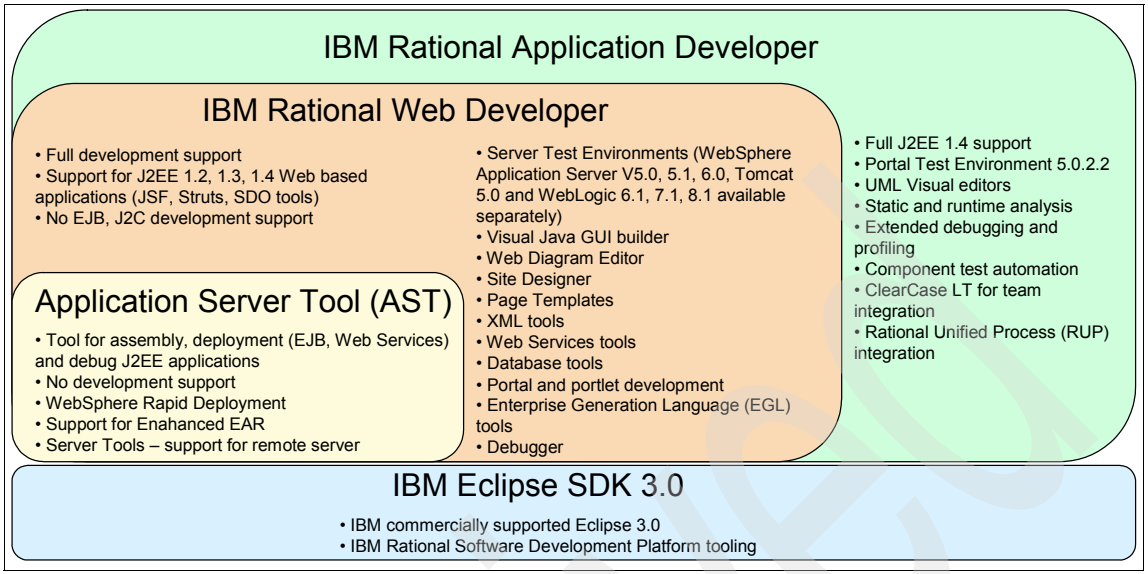


Figure 1-2 Tools and features summary

Figure 1-3 provides a summary of technologies supported by Rational Application Developer categorized by the applications component.

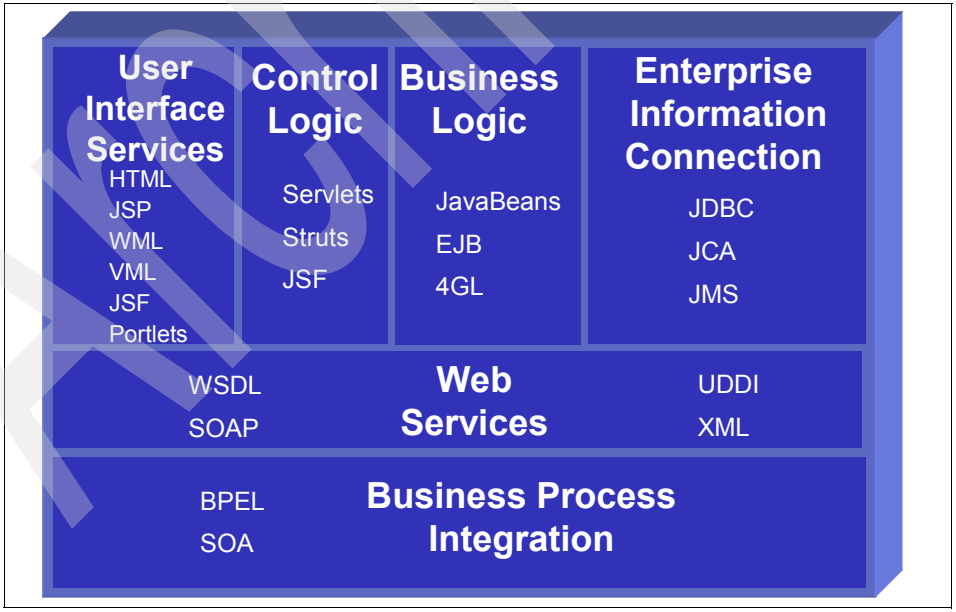


Figure 1-3 Supported technologies for developing applications

1.3.1 Summary of new features in Version 6

There are many new features in VERSION 6, many of which we highlight in detail in the remaining chapters of this book. The objective of this section is to summarize the new features in IBM Rational Application Developer V6.0:

- Specification versions: Full support for J2EE V1.4 and IBM WebSphere Application Server V6.0. See “Specification versions” on page 18 for more information on new features.
- Eclipse and IBM Rational Software Development Platform: Based on Eclipse 3.0. See “Eclipse and IBM Rational Software Development Platform” on page 19 for more information on new features.
- Test server environments:
  - Test environments included for WebSphere Application Server V6.0, V5.1, V5.0, and WebSphere Portal V5.0.2.2, V5.1.
  - Integration with IBM WebSphere Application Server V6.0 for deployment, testing, and administration is the same (test environment, separate install, and Network Deployment edition).

  See “Test server environments” on page 22 for more information on new features.
- Web Services:
  - Build and consume Web Services with JSR 101/109 support for implementing Web Services for EJBs and Java beans.
  - Support for setting conformance levels, including WS-I SSBP 1.0 (simple SOAP Basic Profile 1.0) and WS-I AP (Attachments Profile 1.0).
  - Integrated SOAP Monitor to view Web Services traffic via creation wizard.
  - Secure Web Services request/response capability using WS-Security (security values specified in the J2EE deployment descriptor).

  For more detailed information and a programming example, refer to Chapter 17, “Develop Web Services applications” on page 951.
- Portal application development:
  - Rational Application Developer V6.0 includes portal development tooling, as well as integrated test environments for WebSphere Portal V5.0.2 and V5.1.
  - Model-View-Control (MVC) architecture for portal applications, including support for JSF or Struts.
  - Portal Site Designer for WebSphere Portal V5.0.2.2 and V5.1 to customize the layout and navigation or portal pages.
  - Wired portlets used to link portlets.

Archived

- Portlets for business process tasks via integration with WebSphere Business Integrator (WBI).
- SAP and Seibel integration using JSF and SDO.

For more detailed information and a programming example, refer to Chapter 18, “Develop portal applications” on page 985.

- Application modeling with UML:
  - Model EJBs (top down or bottom up), Java classes, and database schemas visually.
  - New topic, browse, and sequence diagram views available for viewing class relationships and method interaction.
  - This feature is offered in Rational Application Developer and Rational Software Architect (not Rational Web Developer).

  For more detailed information and a programming example, refer to Chapter 6, “RUP and UML” on page 189.
- Rational Unified Process® integration:
  - Process Browser provides search capability for Rational Unified Process (RUP®) best practices and life cycle for development.
  - Process Advisor view displays RUP information specific to current task.
  - This feature is offered in Rational Application Developer and Rational Software Architect (not Rational Web Developer).

  For more detailed information and a programming example, refer to Chapter 6, “RUP and UML” on page 189.
- Annotated programming:
  - EJB and Web resources can be created with annotations following the proposed XDoclet standard (JSR 175).
  - A number of required development resources are reduced since code is generated at the time of deployment.
- Application code analysis: Application code can be reviewed for coding practices. The application code analysis tool can identify the problem and provide examples and a possible solution following best practice guidelines.
- Rapid Web Development:
  - Page template provides a common and consistent look, feel, and layout for Web applications. Page template fragments are now available with support for nesting page templates.
  - Web Site Designer provides group support for applying page templates and creating navigation bars.

Archived

- Drag and drop page design and construction.
- Struts includes enhancements for the Web Diagram Editor for visual Struts development and improved integration with portal.
- Point and click database access.
- Event-driven programming model.
- Automated code construction.
- Rich client construction.
- Robust Web services construction and consumption.
- XML integration and database access.

- JavaServer Faces (JSF):
  - Full support for JSF 1.0 (JSR 127), plus additional IBM components
  - Enhanced visual development of JSF with Web Diagram Editor, such as visual layout of different actions and navigation of JSF pages
  - Client components off new display formats for data

  For more detailed information and a programming example, refer to Chapter 13, “Develop Web applications using JSF and SDO” on page 673.
- Service Data Objects (SDO):
  - Service Data Objects (SDO) offers a common API for working with data across heterogeneous datastores.
  - Access to different datastores becomes transparent as access is through mediators.
  - Developers can focus on real problems rather than spend time learning low-level data access APIs.
  - Available for use in Web and EJB components.

  For more detailed information and a programming example, refer to Chapter 13, “Develop Web applications using JSF and SDO” on page 673.
- Crystal Reports integration:
  - New Report Designer included for designing Crystal Reports.
  - JSF components for Crystal Enterprise included for adding Crystal Reports to Web applications from Palette view.
  - Development and testing licenses included for Crystal Enterprise V10 Professional and Embedded Editions.
  - This feature is offered in Rational Application Developer and Rational Software Architect (not Rational Web Developer).

  For more detailed information refer to the product documentation.

Archived

- Enterprise Generation Language (EGL):
  - High-level programming language for developing business logic previously available in WebSphere Studio Application Developer V5 Enterprise Edition
  - Easy language for non-Java developers for building applications
  - Independent of implementation to hide the complexities of the technology (encourages Rapid Application Development)
  - EGL code generated to Java for runtime

  For more detailed information and a programming example, refer to Chapter 14, “Develop Web applications using EGL” on page 751
- Component testing:
  - Easy creation, execution, and maintenance of dynamic unit tests for J2EE components (Java classes, EJBs 1.1/2.0/2.1, Web services).
  - Test patterns available for defining complex unit test cases.
  - Test data can be stored in separate data pool table for flexible test case definition.
  - Based on JUnit framework and open source Hyades project.
  - This feature is offered in Rational Application Developer and Rational Software Architect (not Rational Web Developer).

  For more detailed information and examples, refer to Chapter 20, “JUnit and component testing” on page 1081.
- Profiling tools:
  - Enhanced Memory Analysis features help locate memory leaks in heap dumps
  - New Thread Analysis view for monitoring thread state for locks
  - New ProbeKit feature allows for easy profiling at key points in the application using byte-code instrumentation
  - Robust summary and detailed code and line execution statistics with Code Coverage feature

  For more detailed information and profiling examples, refer to Chapter 24, “Profile applications” on page 1237.
- Additional enhancements:
  - Database tools: Many enhancements around SQL Tools, SQLJ, and importing, exporting, and deploying stored procedures and user-defined functions.
  - XML: Enhancements to WSDL, XML Schema, and XPATH editors.

Archived

– XSLT: New debugging capabilities allow developers to debug XSLT called from Java and Java called from XSLT.

– WebSphere Programming Model Extensions (PMEs): Snippet and Deployment Descriptors included to enable applications. Note that PMEs are not included within the scope of this book.

1.3.2 Specification versions

This section highlights the specification versions found in IBM Rational Application Developer V6.0, which supports development for the Java 2 Platform Enterprise Edition V1.4. We have included WebSphere Studio Application Developer V5.0 for comparison purposes, which is based on the Java 2 Platform Enterprise Edition V1.3.

Table 1-5 includes a comparison of the J2EE specification versions, and Table 1-6 on page 19 includes a comparison of the WebSphere Application Server specification versions.

Table 1-5 J2EE specification versions

Specification	Rational Application Developer V6.0	WebSphere Studio Application Developer V5.0
IBM Java Runtime Environment (JRE)	1.4.2	1.3.1
JavaServer Page (JSP)	2.0	1.2
Java Servlet	2.4	2.3
Enterprise JavaBeans (EJB)	2.1	2.0
Java Message Service (JMS)	1.1	1.0
Java Transaction API (JTA)	1.0	1.0
JavaMail	1.3	1.2
Java Activation Framework (JAF)	1.0	1.0
Java API for XML Processing (JAXP)	1.2	1.0
J2EE Connector	1.5	1.0
Web Services	1.1	1.0
Java API for XML RPC (JAX-RPC)	1.1	N/A
SOAP with Attachments API for Java (SAAJ)	1.2	N/A

Specification	Rational Application Developer V6.0	WebSphere Studio Application Developer V5.0
Java Authentication and Authorization Service (JAAS)	1.2	1.0
Java API for XML Registries (JAXR)	1.0	N/A
J2EE Management API	1.0	N/A
Java Management Extensions (JMX)	1.2	N/A
J2EE Deployment API	1.1	N/A
Java Authorization Service Provider Contract for Containers (JAAC)	1.0	N/A

Table 1-6 WebSphere Application Server specification versions

Specification	Rational Application Developer V6.0	WebSphere Studio Application Developer V5.0
JavaServer Faces (JSF)	1.0 (JSR 127)	N/A **Note**: JSF 1.0 included in WSAD V5.1.2
Service Data Objects (SDO)	1.0	N/A **Note**: SDO 1.0 formerly named WDO included in WSAD V5.1.2
Struts	1.1	1.0.2 and 1.1 Beta 2 **Note**: Struts 1.0.2 and 1.1 included in WSAD V5.1.1 and V5.1.2

1.3.3 Eclipse and IBM Rational Software Development Platform

This section provides an overview of the Eclipse Project as well as how Eclipse relates to the IBM Rational Software Development Platform and IBM Rational Application Developer V6.0.

Eclipse Project

The Eclipse Project is an open source software development project devoted to creating a development platform and integrated tooling.

Archive

Figure 1-4 on page 20 depicts the high-level Eclipse Project architecture and shows the relationship of the following sub projects:

- Eclipse Platform
- Eclipse Java Development Tools (JDT)
- Eclipse Plug-in Development Environment (PDE)

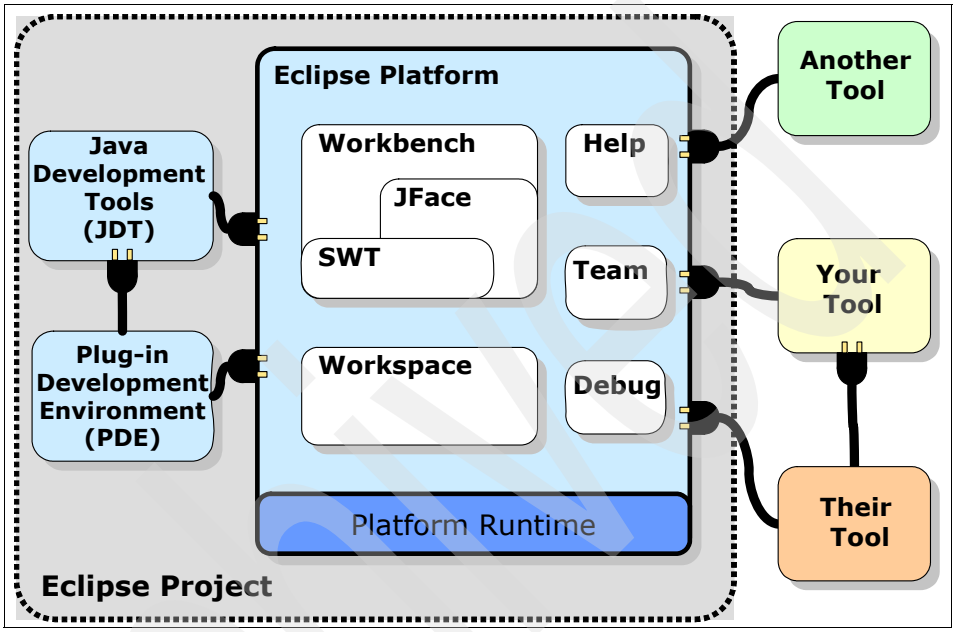


Figure 1-4 Eclipse Project overview

With a common public license that provides royalty free source code and world-wide redistribution rights, the Eclipse Platform provides tool developers with great flexibility and control over their software technology.

Industry leaders like IBM, Borland, Merant, QNX Software Systems, RedHat, SuSE, TogetherSoft, and WebGain formed the initial eclipse.org board of directors of the Eclipse open source project.

More detailed information on Eclipse can be found at:

`http://www.eclipse.org`

Eclipse Platform

The Eclipse Platform provides a framework and services that serve as a foundation for tools developers to integrate and extend the functionality of the Platform. The Platform includes a Workbench, concept of projects, user interface libraries (JFace, SWT), built-in help engine, and support for team development and debug. The Platform can be leveraged by a variety of software development

purposes including modeling and architecture, integrated development environment (Java, C/C++, Cobol, etc.), testing, etc.

Eclipse Java Development Tools (JDT)

The JDT provides the plug-ins for the Platform specifically for a Java-based integrated development environment, as well as the development of plug-ins for Eclipse. The JDT adds the concepts of Java projects, perspectives, views, editors, wizards, and refactoring tools to extend the Platform.

Eclipse Plug-in Development Environment (PDE)

The PDE provides the tools to facilitate the development of Eclipse plug-ins.

Eclipse Software Developer Kit (SDK)

The Eclipse SDK consists of the software created by the Eclipse Project (Platform, JDT, PDE), which can be licensed under the Eclipse Common Public License agreement, as well as other open source third-party software licensed separately.

Note: The Eclipse SDK does not include a Java Runtime Environment (JRE) and must be obtained separately and installed for Eclipse to run.

IBM Eclipse SDK V3.0

The IBM Eclipse SDK V3.0 is an IBM branded and value added version of the Eclipse SDK. The IBM Eclipse SDK V3.0 includes additional plug-ins and the IBM Java Runtime Environment (JRE) V1.4.2. The IBM Eclipse SDK V3.0 was formerly known as the WebSphere Studio Workbench.

The IBM Eclipse SDK V3.0 is highly desirable for many reasons:

- The IBM Eclipse SDK V3.0 offers the ability for loose or tight integration of tooling with the Platform as well as industry standard tools and repositories.
- Provides frameworks, services, and tools that enable tool builders to focus on tool building, not on building tool infrastructure. ISVs can develop new tools to take advantage of the infrastructure and integration points for seamless integration between the tools and the SDK.
- Provides flexibility for rapid support for new standards and technologies (for example, Web Services).
- The IBM Eclipse SDK provides a consistent way of representing and maintaining objects.
- The IBM Eclipse SDK V3.0 tools have been designed to support a roles-based development model in which the assets created in one tool are

are consistent for other tools (for example, XML created in one tool and used in another).

- The IBM Eclipse SDK also provides the support for the full life cycle, including test, debug, and deployment.

The following components are included in the IBM Eclipse SDK V3.0:

- Eclipse SDK 3.0
  - Eclipse Platform
  - Eclipse Java Development Tooling (JDT)
  - Eclipse Plug-in Development Environment (PDE)
- Eclipse Modeling Framework (EMF)
- Eclipse Hyades
- C/C++ Development Tooling (CDT)
- Graphical Editing Framework (GEF)
- XML Schema Infoset Model (XSD)
- UML 2.0 Metamodel Implementation (UML2)
- IBM Java Runtime Environment (JRE) V1.4.2

In summary, the IBM Eclipse SDK V3.0, which provides an open, portable, and universal tooling platform, serves as the base for the IBM Rational Software Development Platform common to many Rational products including IBM Rational Application Developer V6.0 and IBM Rational Web Developer V6.0.

1.3.4 Test server environments

IBM Rational Web Developer V6.0 and IBM Rational Application Developer V6.0 support a wide range of test server environments for running, testing, and debugging application code.

In IBM Rational Application Developer V6.0, the integration with the IBM WebSphere Application Server V6.0 for deployment, testing, and administration is the same as for the IBM WebSphere Application Server V6.0 (Test Environment, separate install, and the Network Deployment edition). In previous versions of WebSphere Studio, the configuration of the test environment was different than a separately installed WebSphere Application Server.

We have categorized the test server environments as follows:

- Integrated test servers

  Integrated test servers refers to the test servers included with the Rational Developer edition (see Table 1-7 on page 23).

- Test servers available separately

Test servers available separately refers to the test servers that are supported by Rational Developer edition but available separately from the Rational Developer products (see Table 1-8 on page 23).

Table 1-7 Integrated test servers

Install option	Integrated test server	Web Developer	Application Developer
IBM WebSphere Application Server V6.0			
	IBM WebSphere Application Server V6.0	X	X
IBM WebSphere Application Server V5.x			
	IBM WebSphere Application Server V5.1	X	X
	IBM WebSphere Application Server Express V5.1	X	X
	IBM WebSphere Application Server V5.0.2	X	X
	IBM WebSphere Application Server Express V5.0.2	X	X
IBM WebSphere Portal			
	IBM WebSphere Portal V5.0.2.2 **Note**: Available on Windows (not Linux)	N/A	X
	IBM WebSphere Portal V5.1	N/A	X

Table 1-8 Additional supported test servers available separately

Integrated Test Server	Web Developer	Application Developer
Tomcat V5.0	X	X
WebLogic V6.1	X	X
WebLogic V7.1	X	X
WebLogic V8.1	X	X

1.3.5 Licensing and installation

This section provides a summary for licensing, installation, product updates, and uninstallation for Rational Application Developer and Rational Web Developer.

Licensing

Both IBM Rational Application Developer V6.0 and IBM Rational Web Developer V6.0 include a license as part of the product. The license is registered

automatically when the product is installed. Separate license manager is not required.

The license information location is as follows:

- Windows

  ```
  C:\Documents and Settings\All Users\Application Data\IBM\LUM\nodelock
  ```

- Linux

  ```
  /opt/IBM/RSDP/6.0/
  ```

The license installation and registration results are stored in the following log file:

```
<IRAD_install_dir>\logs\license.log
```

If licensing fails (for example, due to a permission problem), it can be manually executed as follows:

```
<IRAD_install_dir>\setup\lum\rad\enroll rad-6.0-full.lic
```

Installation

This section provides a summary of the new installation architecture, key software and hardware requirements, as well as basic scenarios for installation.

All Version 6.0 products include the Rational Software Development Platform and plug-ins. If the Rational Software Development Platform is found during the installation; only the plug-ins for the new product are installed. Products that contain the functionality of an installed product as well as additional plug-ins will upgrade the installed product. Products with less functionality than the installed products will be blocked from installation.

For details on the supported platforms refer to “Supported operating system platforms” on page 9.

The system hardware requirements are as follows:

- Pentium® III 800 MHz or higher.
- 768 MB RAM minimum (1 GB RAM recommended).
- 1024x768 video resolution or higher.
- A common install will need 2 GB RAM with 4 GB or free hard disk space to install all components.
- The /tmp (Linux or c:\temp (Windows) should have at least 500 MB of free space.

The Application Developer/Web Developer installations can be started by running launchpad.exe. There are many possible scenarios for installation.

Archived

Figure 1-5 on page 25 displays the installation options for Rational Application Developer.

Figure 1-5 IBM Rational Application Developer V6.0 installation options

Updates

Once Rational Application Developer or Rational Web Developer have been installed, the *IBM Rational Product Updater* tool provides and interface to update the products. This replaces the Update/Install perspectives provided in WebSphere Studio. The product updates are retrieved from preconfigured sites or can be customized.

For details refer to “Rational Application Developer Product Updater - Interim Fix 0004” on page 1380.

Uninstall

Rational Application Developer and Rational Web Developer can be uninstalled interactively or from the command line silently.

On the Windows platform, Web Developer and Application Developer can be uninstalled using the Add/Remove Programs feature of Windows. Alternatively, they can be uninstalled from a command line silently as follows:

```
<IRAD_install_dir>\rad_prod\_uninst\uninstall.exe –silent
```

```
<IRWD_install_dir>\rwd_prod\_uninst\uninstall.exe –silent
```

On the Linux platform, Web Developer and Application Developer can be uninstalled as follows:

```
<IRAD_install_dir>/rad_prod/_uninst/uninstall.bin
```

Or silently as follows:

```
<IRAD_install_dir>/rad_prod/_uninst/uninstall.bin -silent
```

If other Rational products that share the Rational Software Development Platform are installed, these products will remain after uninstalling Rational Application Developer or Rational Web Developer.

After the uninstall on Windows, the following artifacts remain:

- <IRAD_INSTALL_DIR>\ still exists.
- <IRAD_INSTALL_DIR>\runtimes\base_v6\logs\uninstlog.txt contains uninstall results for the WebSphere V6.0 Test Environment.
- C:\Program Files\Common Files\Crystal Decisions and C:\Program Files\Common Files\Business Objects\ still exist.
- %USER_PROFILE%\WINDOWS\vpd.properties is clean.
- %USER_PROFILE%\.WASRegistry cleared of WebSphere V6.0 Test Environment entry.
- HKEY_LOCAL_MACHINE\SOFTWARE\IBM\License Use Runtime\ still exists.
- HKEY_LOCAL_MACHINE\SOFTWARE\Crystal Decisions\ still exists.
- License information remains in C:\Documents and Settings\All Users\Application Data\IBM\LUM\nodelock.
- Windows 2000 %USER_PROFILE%\WINDOWS\IBM\RAT60 still exists.
- Windows XP directories that still exist after uninstall.

  ```
  C:\WINDOWS\IBM\RAD60
  ```

- On Linux (SuSE 9.0) the following directory and some subdirectories remain:

  ```
  <IRAD_INSTALL_DIR>/..
  ```

1.3.6 Migration and coexistence

This section highlights the IBM Rational Application Developer V6.0 migration features, and coexistence and compatibility with WebSphere Studio.

Archived

Migration and coexistence

Migration includes the migration of WebSphere Studio as well as J2EE projects and code assets.

IBM Rational Application Developer V6.0 does include the ability to migrate IBM WebSphere Studio V5.1.x edition installations.

Workspaces created in WebSphere Studio can be used with Rational Application Developer or Rational Web Developer; however, once opened (migrated), they cannot be used again with WebSphere Studio.

Projects exported from WebSphere Studio, such as EARs, WARs, JARs, or with Project Interchange ZIP, will be migrated when imported into Rational Application Developer or Rational Web Developer.

Compatibility with WebSphere Studio V5.1.x

Rational Application Developer and Rational Web Developer include a compatibility option.

Projects may be shared with WebSphere Studio developers through a SCM, such as ClearCase or CVS, or Project Interchange Zip files.

Metadata and project structure are not updated to Rational Application Developer and Rational Web Developer format. A .compatibility file is added to projects and is used to track the timestamps of resources.

Compatibility support can be removed when finished developing in a mixed environment.

Migration considerations

Migration information available on specific components:

- Migrating JavaServer Faces resources in a Web Project
- Migrating JavaServer Faces resources with Faces Client Components
- WDO to SDO migration
- Portal Applications with and without JavaServer Faces
- Considerations around using the Debugger
- EGL reserved words

Previous installations of the IBM Agent Controller should be uninstalled before installing a new version.

Archived

1.3.7 Tools

IBM Rational Application Developer V6.0 includes a wide array of tooling to simplify or eliminate tedious and error-prone tasks, and provide the ability for Rapid Web Development. We have listed the key areas of tooling included with Rational Application Developer:

- Java development tools (JDT)
- Relational database tools
- XML tools
- Web development tools
- Struts tools
- JSF development tools
- SDO development tools
- Enterprise Generation Language tools
- EJB tools
- Portal tools
- Web Services tools
- Team collaboration tools
- Debugging tools
- Performance profiling and analysis tools
- Server configuration tools
- Testing tools
- Deployment tools
- Plug-in development tools

Note: Each of the chapters of this book provides a description of the tools related to the given topic and demonstrates how to use the Rational Application Developer tooling.

1.4 Sample code

The chapters are written so that you can follow along and create the code from scratch. In places where there is lots of typing involved we have provided snippets of code to cut and paste.

Alternatively, you can import the completed sample code from a Project Interchange file. For details on the sample code (download, unpack, description, import interchange file, create databases) refer to Appendix B, “Additional material” on page 1395.

Archived

Important: Interim Fix 0004

Much of this book was originally researched and written using IBM Rational Application Developer V6.0. During the course of completing this book, we upgraded the system to the most current Interim Fix level at the time (Interim Fix 0004). In some cases, our samples require that you use the Interim Fix level for the sample to run properly.

For more information refer to “Rational Application Developer Product Updater - Interim Fix 0004” on page 1380.

Archived

Archived

2

Programming technologies

This chapter describes a number of example application development scenarios, based on a simple banking application. Throughout these examples, we will review the Java and supporting technologies, as well as highlight the tooling provided by IBM Rational Application Developer V6.0, which can be used to facilitate implementing the programming technologies.

This chapter is organized into the following sections:

- Desktop applications
- Static Web sites
- Dynamic Web applications
- Enterprise JavaBeans
- J2EE Application Clients
- Web Services
- Messaging systems

Archived

© Copyright IBM Corp. 2005. All rights reserved.

2.1 Desktop applications

By *desktop applications* we mean applications in which the application runs on a single machine and the user interacts directly with the application using a user interface on the same machine.

When this idea is extended to include database access, some work may be performed by another process, possibly on another machine. Although this begins to move us into the client-server environment, the application is often only using the database as a service—the user interface, business logic, and control of flow are still contained within the desktop application. This contrasts with full client-server applications in which these elements are clearly separated and may be provided by different technologies running on different machines.

This type of application is the simplest type we will consider. Many of the technologies and tools involved in developing desktop applications, such as the Java editor and the XML tooling, are used widely throughout all aspects of Rational Application Developer.

The first scenario deals with a situation in which a bank requires an application to allow workers in a bank call center to be able to view and update customer account information. We will call this the Call Center Desktop.

2.1.1 Simple desktop applications

A starting point for the Call Center Desktop might be a simple stand-alone application designed to run on desktop computers.

Java 2 Platform Standard Edition (J2SE) provides all the elements necessary to develop such applications. It includes, among other elements, a complete object-oriented programming language specification, a wide range of useful classes to speed development, and a runtime environment in which programs can be executed. We will be dealing with J2SE V1.4.

The complete J2SE specification can be found at:

`http://java.sun.com/j2se/`

Java language

Java is a general purpose, object-oriented language. The basic language syntax is similar to C and C++, although there are significant differences. Java is a higher-level language than C or C++, in that the developer is presented with a more abstracted view of the underlying computer hardware and is not expected to take direct control of issues such as memory management. The compilation process for Java does not produce directly executable binaries, but rather an

intermediate bytecode, which may be executed directly by a virtual machine or further processed by a just-in-time compiler at runtime to produce platform-specific binary output.

Core Java APIs

J2SE V1.4 includes a wide-ranging library of useful classes that can greatly improve developer productivity. These are arranged in packages of classes, such as:

- Logging: Provides a simple mechanism for reporting events during program execution
- Locale support: Provides facilities for internationalizing applications
- Security: Provides mechanisms for authenticating users and determining which resources they are authorized to use
- Lang: Provides basic functionality that the Java language relies on, such as basic types and classes allowing access to features of the runtime environment
- Util: Provides a range of utility classes representing object collections, event handling, and time and date facilities
- Input/output: Provides facilities for stream input and output, serialization of objects, and file system access
- Networking: Allows programs to access network resources using TCP/IP

Java Virtual Machine

The Java Virtual Machine (JVM) is a runtime environment designed for executing compiled Java bytecode, contained in the .class files, which result from the compilation of Java source code. Several different types of JVM exist, ranging from simple interpreters to just-in-time compilers that dynamically translate bytecode instructions to platform-specific instructions as required.

Requirements for the development environment

The developer of the Call Center Desktop should have access to a development tool, providing a range of features to enhance developer productivity:

- A specialized code editor, providing syntax highlighting
- Assistance with completing code and correcting syntactical errors
- Facilities for visualizing the relationships between the classes in the application
- Assistance with documenting code

Archived

- Automatic code review functionality to ensure that code is being developed according to recognized best practices
- A simple way of testing applications

IBM Rational Application Developer V6.0 provides developers with an integrated development environment with these features.

2.1.2 Database access

It is very likely that the Call Center Desktop will need to access data residing in a relational database, such as IBM DB2® Universal Database™.

J2SE V1.4 includes several integration technologies:

- JDBC is the Java standard technology for accessing data stores.
- Java Remote Method Invocation (RMI) is the standard way of enabling remote access to objects within Java.
- Java Naming and Directory Interface (JNDI) is the standard Java interface for naming and directory services.
- Java IDL is the Java implementation of the Interface Definition Language (IDL) for the Common Object Request Broker Architecture (CORBA), allowing Java programs to access objects hosted on CORBA servers.

We will focus on the Java DataBase Connectivity (JDBC) technology in this section.

JDBC

J2SE V1.4 includes JDBC V3.0. In earlier versions of J2SE, the classes contained in the jaxa.sql package were known as the JDBC Core API, whereas those in the javax.sql package were known as the JDBC Standard Extension API (or Optional Package), but now the V1.4 JDBC includes both packages as standard. Since Java 2 Platform Enterprise Edition V1.4 (J2EE V1.4, which we will come to shortly) is based on J2SE V1.4, all these features are available when developing J2EE V1.4 applications as well.

More information on JDBC can be found at:

`http://java.sun.com/j2se/1.4.2/docs/guide/jdbc/`

Although JDBC supports a wide range of data store types, it is most commonly used for accessing relational databases using SQL. Classes and interfaces are provided to simplify database programming, such as:

- java.sql.DriverManager and javax.sql.DataSource can be used to obtain a connection to a database system.

Archived

- java.sql.Connection represents the connection that an application has to a database system.
- java.sql.Statement, PreparedStatement, and CallableStatement represent executable statements that can be used to update or query the database.
- java.sql.ResultSet represents the values returned from a statement that has queried the database.
- Various types such as java.sql.Date and java.sql.Blob are Java representations of SQL data types that do not have a directly equivalent primitive type in Java.

Requirements for the development environment

The development environment should provide access to all the facilities of JDBC V3.0. However, since JDBC V3.0 is an integral part of J2SE V1.4, this requirement has already been covered in 2.1.1, “Simple desktop applications” on page 32. In addition, the development environment should provide:

- A way of viewing information about the structure of an external database
- A mechanism for viewing sample contents of tables
- Facilities for importing structural information from a database server so that it can be used as part of the development process
- Wizards and editors allowing databases, tables, columns, relationships, and constraints to be created or modified
- A feature to allow databases created or modified in this way to be exported to an external database server
- A wizard to help create and test SQL statements

These features would allow developers to develop test databases and work with production databases as part of the overall development process. They could also be used by database administrators to manage database systems, although they may prefer to use dedicated tools provided by the vendor of their database systems.

IBM Rational Application Developer V6.0 includes these features.

2.1.3 Graphical user interfaces

A further enhancement of the Call Center Desktop would be to make the application easier to use by providing a graphical user interface (GUI).

Abstract Window Toolkit (AWT)

The Abstract Window Toolkit (AWT) is the original GUI toolkit for Java. It has been enhanced since it was originally introduced, but the basic structure remains the same. The AWT includes the following:

- A wide range of user interface components, represented by Java classes such as java.awt.Frame, Button, Label, Menu, and TextArea
- An event-handling model to deal with events such as button clicks, menu choices, and mouse operations
- Classes to deal with graphics and image processing
- Layout manager classes to help with positioning components in a GUI
- Support for drag-and-drop functionality in GUI applications

The AWT is implemented natively for each platform's JVM. AWT interfaces typically perform relatively quickly and have the same look-and-feel as the operating system, but the range of GUI components that can be used is limited to the lowest common denominator of operating system components and the look-and-feel cannot be changed.

More information on the AWT can be found at:

http://java.sun.com/j2se/1.4.2/docs/guide/awt/

Swing

Swing is a newer GUI component framework for Java. It provides Java implementations of the components in the AWT and adds a number of more sophisticated GUI components, such as tree views and list boxes. For the basic components, Swing implementations have the same name as the AWT component with a J prefix and a different package structure, for example, java.awt.Button becomes javax.swing.JButton in Swing.

Swing GUIs do not normally perform as quickly as AWT GUIs, but have a richer set of controls and have a pluggable look-and-feel.

More information on Swing can be found at:

http://java.sun.com/j2se/1.4.2/docs/guide/swing/

Standard Widget[1] Toolkit

The Standard Widget Toolkit (SWT) is the GUI toolkit provided as part of the Eclipse Project and used to build the Eclipse GUI itself. The SWT is written entirely in Java and uses the Java Native Interface (JNI) to pass the calls through

[1] In the context of windowing systems, a widget is a reusable interface component, such as a menu, scroll bar, button, text box or label.

to the operating system where possible. This is done to avoid the *lowest common denominator* problem. The SWT uses native calls where they are available and builds the component in Java where they are not.

In many respects, the SWT provides the best of both worlds (AWT and Swing):

- It has a rich, portable component model, like Swing.
- It has the same look-and-feel as the native operating system, like the AWT.
- GUIs built using the SWT perform well, like the AWT, since most of the components simply pass through to operative system components.

A disadvantage of the SWT is that, unlike the AWT and Swing, it is not a standard part of J2SE V1.4. Consequently, any application that uses the SWT has to be installed along with the SWT class libraries. However, the SWT, like the rest of the components that make up Eclipse, is open source and freely distributable under the terms of the Common Public License.

More information on the SWT can be found at:

`http://www.eclipse.org/swt/`

Java components providing a GUI

There are two types of Java components that might provide a GUI:

- Stand-alone Java applications: Launched in their own process (JVM). This category would include J2EE Application Clients, which we will come to later.
- Java applets: Normally run in a JVM provided by a Web browser or a Web browser plug-in.

An applet normally runs in a JVM with a very strict security model, by default. The applet is not allowed to access the file system of the machine on which it is running and can only make network connections back to the machine from which it was originally loaded. Consequently, applets are not normally suitable for applications that require access to databases, since this would require the database to reside on the same machine as the Web server. If the security restrictions are relaxed, as might be possible if the applet was being used only on a company intranet, this problem is not encountered.

An applet is downloaded on demand from the Web site that is hosting it. This gives an advantage in that the latest version is automatically downloaded each time it is requested, so distributing new versions is trivial. On the other hand, it also introduces disadvantages in that the applet will often be downloaded several times even if it has not changed, pointlessly using bandwidth, and the developer has little control over the environment in which the applet will run.

Archive

The requirements for the development environment

The development environment should provide a specialized editor that allows a developer to design GUIs using a variety of component frameworks (such as the AWT, Swing, or the SWT). The developer should be able to focus mainly on the visual aspects of the layout of the GUI, rather than the coding that lies behind it. Where necessary, the developer should be able to edit the generated code to add event-handling code and business logic calls. The editor should be dynamic, reflecting changes in the visual layout immediately in the generated code and changes in the code immediately in the visual display. The development environment should also provide facilities for testing visual components that make up a GUI, as well the entire GUI.

IBM Rational Application Developer V6.0 includes a visual editor for Java classes, which offers this functionality.

2.1.4 Extensible Markup Language (XML)

Communication between computer systems is often difficult because different systems use different data formats for storing data. XML has become a common way of resolving this problem.

It may be desirable for the Call Center Desktop application to be able to exchange data with other applications. For example, we may want to be able to export tabular data so that it can be read into a spreadsheet application to produce a chart, or we may want to be able to read information about a group of transactions that can then be carried out as part of an overnight batch operation.

A convenient technology for exchanging information between applications is XML. XML is a standard, simple, flexible way of exchanging data. The structure of the data is described in the XML document itself, and there are mechanisms for ensuring that the structure conforms to an agreed format (these are known as Document Type Definitions (DTDs) and XML Schemas (XSDs)).

XML is increasingly also being used to store configuration information for applications. For example, many aspects of J2EE V1.4 use XML for configuration files called *deployment descriptors*, and WebSphere Application Server V6 uses XML files for storing its configuration settings.

For more information on XML, see the home of XML - the World Wide Web Consortium (W3C) at:

`http://www.w3c.org/XML/`

Archived

Using XML in Java code

J2SE V1.4 includes the Java API for XML Processing (JAXP). JAXP contains several elements:

- A parser interface based on the Document Object Model (DOM) from the W3C, which builds a complete internal representation of the XML document
- The Simple API for XML Parsing (SAX), which allows the document to be parsed dynamically using an event-driven approach
- XSL Transformations (XSLT), which uses Extensible Stylesheet Language (XSL) to describe how to transform XML documents from one form into another

Since JAXP is a standard part of J2SE V1.4, all these features are available in any Java code running in a JVM.

Requirements for the development environment

In addition to allowing developers to write code to create and parse XML documents, the development environment should provide features that allow developers to create and edit XML documents and related resources. In particular:

- An XML editor that will check the XML document for well-formedness (conformance with the structural requirements of XML) and for consistency with a DTD or XML Schema
- Wizards for:
  - Creating XML documents from DTDs and XML Schemas
  - Creating DTDs and XML Schemas from XML documents
  - Converting between DTDs and XML Schemas
  - Generating JavaBeans to represent data stored in XML documents
  - Creating XSL
- An environment to test and debug XSL transformations

IBM Rational Application Developer V6.0 includes all these features.

2.2 Static Web sites

A static Web site is one in which the content viewed by users accessing the site using a Web browser is determined only by the contents of the file system on the Web server machine. Because the user's experience is determined only by the content of these files and not by any action of the user or any business logic running on the server machine, the site is described as static.

Archived

In most cases, the communication protocol used for interacting with static Web sites is the Hypertext Transfer Protocol (HTTP).

In the context of our sample scenario, the bank may wish to publish a static Web site in order to inform customers of bank services, such as branch locations and opening hours, and to inform potential customers of services provided by the bank, such as account interest rates. This kind of information can safely be provided statically, since it is the same for all visitors to the site and infrequently changes.

2.2.1 Hypertext Transfer Protocol (HTTP)

HTTP follows a request/response model. A client sends an HTTP request to the server providing information about the request method being used, the requested Uniform Resource Identifier (URI), the protocol version being used, various other header information and often other details, such as details from a form completed on the Web browser. The server responds by returning an HTTP response consisting of a status line, including a success or error code, and other header information followed by a the HyperText Markup Language (HTML) code for the static page requested by the client.

Full details of HTTP can be found at:

`http://www.w3c.org/Protocols/`

Information on HTML can be found at:

`http://www.w3c.org/MarkUp/`

Methods

HTTP 1.1 defines several request methods: GET, HEAD, POST, PUT, DELETE, OPTIONS, and TRACE. Of these, only GET and POST are commonly used in Web applications:

- GET requests are normally used in situations where the user has entered an address into the address or location field of a Web browser, used a bookmark or favorite stored by the browser, or followed a hyperlink within an HTML document.
- POST requests are normally used when the user has completed an HTML form displayed by the browser and has submitted the form for processing. This request type is most often used with dynamic Web applications, which include business logic for processing the values entered into the form.

Status codes

The status code returned by the server as the first line of the HTTP response indicates the outcome of the request. In the event of an error, this information

can be used by the client to inform the user of the problem. In some situations, such as redirection to another URI, the browser will act on the response without any interaction from the user. The classes of status code are:

- 1xx: Information - The request has been received and processing is continuing.
- 2xx: Success - The request has been correctly received and processed; an HTML page will normally accompany a 2xx status code as the body of the response.
- 3xx: Redirection - The request did not contain all the information required or the browser needs to take the user to another URI.
- 4xx: Client error - The request was incorrectly formed or could not be fulfilled.
- 5xx: Server error - Although the request was valid, the server failed to fulfill it.

The most common status code is `200 (OK)`, although `404 (Not Found)` is very commonly encountered. A complete list of status codes can be found at the W3C site mentioned above.

Cookies

Cookies are a general mechanism that server-side connections can use to both store and retrieve information on the client side of the connection. Cookies can contain any piece of textual information, within an overall size limit per cookie of 4 kilobytes. Cookies have the following attributes:

- Name: The name of the cookie.
- Value: The data that the server wants passed back to it when a browser requests another page.
- Domain: The address of the server that sent the cookie and that receives a copy of this cookie when the browser requests a file from that server. The domain may be set to equal the subdomain that contains the server so that multiple servers in the same subdomain receive the cookie from the browser.
- Path: Used to specify the subset of URLs in a domain for which the cookie is valid.
- Expires: Specifies a date string that defines the valid lifetime of that cookie.
- Secure: Specifies that the cookie is only sent if HTTP communication is taking place over a secure channel (known as HTTPS).

A cookie's life cycle proceeds as follows:

1. The user gets connected to a server that wants to record a cookie.
2. The server sends the name and the value of the cookie in the HTTP response.

Archived

3. The browser receives the cookie and stores it.
4. Every time the user sends a request for a URL at the designated domain, the browser sends any cookies for that domain that have not expired with the HTTP request.
5. Once the expiration date has been passed, the cookie crumbles.

Non-persistent cookies are created without an expiry date—they will only last for the duration of the user's browser session. Persistent cookies are set once and remain on the user's hard drive until the expiration date of the cookie.

Cookies are widely used in dynamic Web applications, which we address later in this chapter, for associating a user with server-side state information.

More information on cookies can be found at:

http://wp.netscape.com/newsref/std/cookie_spec.html

2.2.2 HyperText Markup Language (HTML)

HTML is a language for publishing hypertext on the Web. HTML uses tags to structure text into headings, paragraphs, lists, hypertext links, and so forth. Table 2-1 lists some of the basic HTML tags.

Table 2-1 Some basic HTML tags

Tag	Description
<html>	Tells the browser that the following text is marked up in HTML. The closing tag </html> is required and is the last tag in your document.
<head>	Defines information for the browser that may or may not be displayed to the user. Tags that belong in the <head> section are <title>, <meta>, <script>, and <style>. The closing tag </head> is required.
<title>	Displays the title of your Web page, and is usually displayed by the browser at the top of the browser pane. The closing tag </title> is required.
<body>	Defines the primary portion of the Web page. Attributes of the <body> tag enables setting of the background color for the Web pages, the text color, the link color, and the active and visited link colors. The closing tag </body> is required.

Cascading Style Sheets (CSS)

Although Web developers can use HTML tags to specify styling attributes, the best practice is to use a cascading style sheet (CSS). A CSS file defines a hierarchical set of style rules that the creator of an HTML (or XML) file uses in

Archived

order to control how that page is rendered in a browser or viewer, or how it is printed.

CSS allows for separation of presentation content of documents from the content of documents. A CSS file can be referenced by an entire Web site to provide continuity to titles, fonts, and colors.

Below is a rule for setting the H2 elements to the color red. Rules are made up of two parts: *Selector* and *declaration*. The selector (H2) is the link between the HTML document and the style sheet, and all HTML element types are possible selectors. The declaration has two parts: Property (color) and value (red):

```
H2 { color: red }
```

More information on CSS can be found at:

http://www.w3.org/Style/CSS/

Requirements for the development environment

The development environment should provide:

- An editor for HTML pages, providing WYSIWYG (*what you see is what you get*), HTML code, and preview (browser) views to assist HTML page designers
- A CSS editor
- A view showing the overall structure of a site as it is being designed
- A built-in Web server and browser to allow Web sites to be tested

IBM Rational Application Developer V6.0 provides all of these features.

2.3 Dynamic Web applications

By *Web applications* we mean applications that are accessed using HTTP (Hypertext Transfer Protocol), usually using a Web browser as the client-side user interface to the application. The flow of control logic, business logic, and generation of the Web pages for the Web browser are all handled by software running on a server machine. Many different technologies exist for developing this type of application, but we will focus on the Java technologies that are relevant in this area.

Since the technologies are based on Java, most of the features discussed in 2.1, “Desktop applications” on page 32, are relevant here as well (the GUI features are less significant). In this section we focus on the additional features required for developing Web applications.

Archive

In the context of our example banking application, thus far we have provided workers in the bank's call center with a desktop application to allow them to view and update account information and members of the Web browsing public with information about the bank and its services. We will now move into the Internet banking Web application. We want to extend the system to allow bank customers to access their account information online, such as balances and statements, and to perform some transactions, such as transferring money between accounts and paying bills.

2.3.1 Simple Web applications

The simplest way of providing Web-accessible applications using Java is to use Java Servlets and JavaServer Pages (JSPs). These technologies form part of the Java 2 Platform Enterprise Edition (J2EE), although they can also be implemented in systems that do not conform to the J2EE specification, such as Apache Jakarta Tomcat:

`http://jakarta.apache.org/tomcat/`

Information on these technologies (including specifications) can be found at the following locations:

- Servlets:

  `http://java.sun.com/products/servlet/`

- JSPs:

  `http://java.sun.com/products/jsp/`

In this book we discuss J2EE V1.4, since this is the version supported by IBM Rational Application Developer V6.0 and IBM WebSphere Application Server V6.0. J2EE V1.4 requires Servlet V2.4 and JSP V2.0. Full details of J2EE V1.4 can be found at:

`http://java.sun.com/j2ee/`

Servlets

A *servlet* is a Java class that is managed by server software known as a *Web container* (sometimes referred to as a *servlets container* or *servlets engine*). The purpose of a servlet is to read information from an HTTP request, perform some processing, and generate some dynamic content to be returned to the client in an HTTP response.

The Servlet Application Programming Interface (API) includes a class, javax.servlet.http.HttpServlet, which can be subclassed by a developer. The developer needs to override methods such as the following to handle different

Archived

types of HTTP requests (in these cases, POST and GET requests; other methods are also supported):

- public void doPost (HttpServletRequest request, HttpServletResponse response)
- public void doGet (HttpServletRequest request, HttpServletResponse response)

When a HTTP request is received by the Web container, it consults a configuration file, known as a *deployment descriptor,* to establish which servlets class corresponds to the URL provided. If the class is already loaded in the Web container and an instance has been created and initialized, the Web container invokes a standard method on the servlets class:

```
public void service (HttpServletRequest request, HttpServletResponse
response)
```

The service method, which is inherited from HttpServlet, examines the HTTP request type and delegates processing to the doPost or doGet method as appropriate. One of the responsibilities of the Web container is to package the HTTP request received from the client as an HttpServletRequest object and to create an HttpServletResponse object to represent the HTTP response that will ultimately be returned to the client.

Within the doPost or doGet method, the servlets developer can use the wide range of features available within Java, such as database access, messaging systems, connectors to other systems, or Enterprise JavaBeans.

If the servlet has not already been loaded, instantiated, and initialized, the Web container is responsible for carrying out these tasks. The initialization step is performed by executing the method:

```
public void init ()
```

And there is a corresponding method:

```
public void destroy ()
```

This is called when the servlet is being unloaded from the Web container.

Within the code for the doPost and doGet methods, the usual processing pattern is:

1. Read information from the request - This will often include reading cookie information and getting parameters that correspond to fields in an HTML form.
2. Check that the user is in the appropriate state to perform the requested action.

3. Delegate processing of the request to the appropriate type of business object.
4. Update the user's state information.
5. Dynamically generate the content to be returned to the client.

The last step could be carried out directly in the servlets code by writing HTML to a PrintWriter object obtained from the HttpServletResponse object:

```
PrintWriter out = response.getWriter();
out.println("<html><head><title>Page title</title></head>");
out.println("<body>The page content:");
// etc...
```

This approach is not recommended, because the embedding of HTML within the Java code means that HTML page design tools, such as those provided by Rational Application Developer, cannot be used. It also means that development roles cannot easily be separated—Java developers must maintain HTML code. The best practice is to use a dedicated display technology, such as JSP, covered next.

JavaServer Pages (JSPs)

JSPs provide a server-side scripting technology that enables Java code to be embedded within Web pages, so JSPs have the appearance of HTML pages with embedded Java code. When the page is executed, the Java code can generate dynamic content to appear in the resulting Web page. JSPs are compiled at runtime into servlets that execute to generate the resulting HTML. Subsequent calls to the same JSP simply execute the compiled servlet.

JSP scripting elements (some of which are shown in Table 2-2) are used to control the page compilation process, create and access objects, define methods, and manage the flow of control.

Table 2-2 Examples of JSP scripting elements

Element	Meaning
Directive	Instructions that are processed by the JSP engine when the page is compiled to a servlet <%@ ... %> or <jsp:directive.page ... />
Declaration	Allows variables and methods to be declared <%! ... %> or <jsp:declaration> ... </jsp:declaration>
Expression	Java expressions, which are evaluated, converted to a String and entered into the HTML <%= ... %> or <jsp:expression ... />
Scriptlet	Blocks of Java code embedded within a JSP <% ... %> or <jsp:scriptlet> ... </jsp:scriptlet>

Element	Meaning
Use Bean	Retrieves an object from a particular scope or creates an object and puts it into a specified scope `<jsp:useBean ... />`
Get Property	Calls a getter method on a bean, converts the result to a String, and places it in the output `<jsp:getProperty ... />`
Set Property	Calls a setter method on a bean `<jsp:setProperty ... />`
Include	Includes content from another page or resource `<jsp:include ... />`
Forward	Forwards the request processing to another URL `<jsp:forward ... />`

The JSP scripting elements can be extended, using a technology known as *tag extensions* (or *custom tags*), to allow the developer to make up new tags and associate them with code that can carry out a wide range of tasks in Java. Tag extensions are grouped in *tag libraries*, which we will discuss shortly.

Some of the standard JSP tags are only provided in an XML-compliant version, such as <jsp:useBean ... />. Others are available in both traditional form (for example, <%= ... %> for JSP expressions) or XML-compliant form (for example, <jsp:expression ... />). These XML-compliant versions have been introduced in order to allow JSPs to be validated using XML validators.

JSPs generate HTML output by default—the Multipurpose Internet Mail Extensions (MIME) type is text/html. It may be desirable to produce XML (text/xml) instead in some situations. For example, a developer may wish to produce XML output, which can then be converted to HTML for Web browsers, Wireless Markup Language (WML) for wireless devices, or VoiceXML for systems with a voice interface. Servlets can also produce XML output in this way—the content type being returned is set using a method on the HttpServletResponse object.

Tag libraries

Tag libraries are a standard way of packaging tag extensions for applications using JSPs.

Tag extensions address the problem that arises when a developer wishes to use non-trivial processing logic within a JSP. Java code can be embedded directly in the JSP using the standard tags described above. This mixture of HTML and Java makes it difficult to separate development responsibilities (the HTML/JSP

designer has to maintain the Java code) and makes it hard to use appropriate tools for the tasks in hand (a page design tool will not provide the same level of support for Java development as a Java development tool). This is essentially the reverse of the problem described when discussing servlets above. To address this problem, developers have documented the *View Helper* design pattern, as described in *Core J2EE Patterns: Best Practices and Design Strategies* by Crupi, et al. (the pattern catalog contained in this book is also available at http://java.sun.com/blueprints/corej2eepatterns/Patterns/). Tag extensions are the standard way of implementing View Helpers for JSPs.

Using tag extensions, a Java developer can create a class that implements some view-related logic. This class can be associated with a particular JSP tag using a tag library descriptor (TLD). The TLD can be included in a Web application, and the tag extensions defined within it can then be used in JSPs. The JSP designer can use these tags in exactly the same way as other (standard) JSP tags. The JSP specification includes classes that can be used as a basis for tag extensions and (new in JSP v2.0) a simplified mechanism for defining tag extensions that does not require detailed knowledge of Java.

Many convenient tags are provided in the JSP Standard Tag Library (JSTL), which actually includes several tag libraries:

- Core tags: Flow control (such as loops and conditional statements) and various general purpose actions.
- XML tags: Allow basic XML processing within a JSP.
- Formatting tags: Internationalized data formatting.
- SQL tags: Database access for querying and updating.
- Function tags: Various string handling functions.

Tag libraries are also available from other sources, such as those from the Jakarta Taglibs Project (http://jakarta.apache.org/taglibs/), and it is also possible to develop tag libraries yourself.

Expression Language

Expression Language (EL) was originally developed as part of the JSTL, but it is now a standard part of JSP (from V2.0). EL provides a standard way of writing expressions within a JSP using implicit variables, objects available in the various scopes within a JSP and standard operators. EL is defined within the JSP V2.0 specification.

Filters

Filters are objects that can transform a request or modify a response. They can process the request before it reaches a servlet, and/or process the response

leaving a servlet before it is finally returned to the client. A filter can examine a request before a servlet is called and can modify the request and response headers and data by providing a customized version of the request or response object that wraps the real request or response. The deployment descriptor for a Web application is used to configure specific filters for particular servlets or JSPs. Filters can also be linked together in chains.

Life cycle listeners

Life cycle events enable listener objects to be notified when servlet contexts and sessions are initialized and destroyed, as well as when attributes are added or removed from a context or session.

Any listener interested in observing the ServletContext life cycle can implement the ServletContextListener interface, which has two methods, contextInitialized (called when an application is first ready to serve requests) and contextDestroyed (called when an application is about to shut down).

A listener interested in observing the ServletContext attribute life cycle can implement the ServletContextAttributesListener interface, which has three methods, attributeAdded (called when an attribute is added to the ServletContext), attributeRemoved (called when an attribute is removed from the ServletContext), and attributeReplaced (called when an attribute is replaced by another attribute in the ServletContext).

Similar listener interfaces exist for HttpSession and ServletRequest objects:

- javax.servlet.http.HttpSessionListener: HttpSession life cycle events.
- javax.servlet.HttpSessionAttributeListener: Attributes events on an HttpSession.
- javax.servlet.HttpSessionActivationListener: Activation or passivation of an HttpSession.
- javax.servlet.HttpSessionBindingListener: Object binding on an HttpSession.
- javax.servlet.ServletRequestListener: Processing of a ServletRequest has begun.
- javax.servlet.ServletRequestAttributeListener: Attribute events on a ServletRequest.

Requirements for the development environment

The development environment should provide:

- Wizards for creating servlets, JSPs, Listeners, Filters, and Tag Extensions
- An editor for JSPs that enables the developer to use all the features of JSP in an intuitive way, focussing mainly on page design

Archived

- An editor for Web deployment descriptors allowing these components to be configured
- Validators to ensure that all the technologies are being used correctly
- A test environment that will allow dynamic Web applications to be tested and debugged

IBM Rational Application Developer V6.0 includes all these features.

Figure 2-1 shows the interaction between the Web components and a relational database, as well as the desktop application discussed in 2.1, “Desktop applications” on page 32.

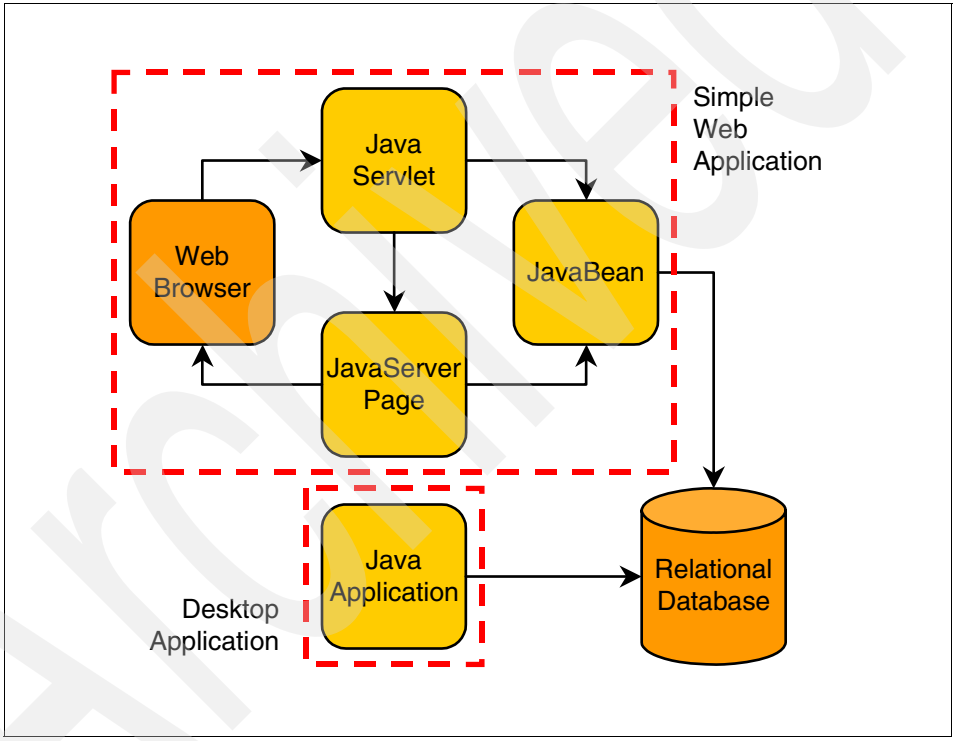


Figure 2-1 Simple Web application

Archived

2.3.2 Struts

The model-view-controller (MVC) architecture pattern is used widely in object-oriented systems as a way of dividing applications into sections with well-defined responsibilities:

- Model: Manages the application domain's concepts, both behavior and state. It responds to requests for information about its state and responds to instructions to change its state.
- View: Implements the rendering of the model, displaying the results of processing for the use, and manages user input.
- Controller: Receives user input events, determines which action is necessary, and delegates processing to the appropriate model objects.

In dynamic Web applications, the servlet normally fills the role of *controller*, the JSP fills the role of *view* and various components, and JavaBeans or Enterprise JavaBeans fill the role of *model*. The MVC pattern will be described in more detail in Chapter 12, "Develop Web applications using Struts" on page 615, as will Struts.

In the context of our banking scenario, this technology does not relate to any change in functionality from the user's point of view. The problem being addressed here is that, although many developers may wish to use the MVC pattern, J2EE V1.4 does not provide a standard way of implementing it. The developers of the bank's Internet banking application wish to design their application according to the MVC pattern, but do not want to have to build everything from the ground up.

Struts was introduced as a way of providing developers with an MVC framework for applications using the Java Web technologies—servlets and JSPs. Complete information on Struts is available at:

`http://struts.apache.org/`

Struts provides a Controller servlets, called ActionServlet, which acts as the entry point for any Struts application. When the ActionServlet receives a request, it uses the URL to determine the requested action and uses an ActionMapping object, created when the application starts up, based on information in an XML file called struts-config.xml. From this ActionMapping object, the Struts ActionServlet determines the Action-derived class that is expected to handle the request. The Action object is then invoked to perform the required processing. This Action object is provided by the developer using Struts to create a Web application and may use any convenient technology for processing the request. The Action object is the route into the model for the application. Once processing has been completed, the Action object can indicate what should happen next—the ActionServlet will use this information to select the appropriate

response agent (normally a JSP) to generate the dynamic content to be sent back to the user. The JSP represents the view for the application.

Struts provides other features, such as FormBeans, to represent data entered into HTML forms and JSP tag extensions to facilitate Struts JSP development.

Requirements for the development environment

Since Struts applications are also Web applications, all the functionality described in 2.3.1, “Simple Web applications” on page 44, is relevant in this context as well. In addition, the development environment should provide:

- Wizards to create:
  - A new Struts Action class and corresponding ActionMapping
  - A new ActionForm bean
  - A new Struts exception type
  - A new Struts module
- An editor to modify the struts-config.xml file
- A graphical editor to display and modify the relationship between Struts elements, such as Actions, ActionForms, and View components

In addition, the basic Web application tools should be Struts-aware. The wizard for creating Web applications should include a simple mechanism for adding Struts support, and the wizard for creating JSPs should offer to add the necessary Struts tag libraries.

IBM Rational Application Developer V6.0 provides all of these features.

Figure 2-1 on page 50 still represents the structure of a Web application using Struts. Although Struts provides us with a framework on which we can build our own applications, the technology is still the same as for basic Web applications.

2.3.3 JavaServer Faces (JSF) and Service Data Objects (SDO)

When we build a GUI using stand-alone Java applications, we can include event-handling code, so that when UI events take place they can be used immediately to perform business logic processing or update the UI. Users are familiar with this type of behavior in desktop applications, but the nature of Web applications has made this difficult to achieve using a browser-based interface; the user interface provided through HTML is limited, and the request-response style of HTTP does not naturally lead to flexible, event-driven user interfaces.

Many applications require access to data, and there is often a requirement to be able to represent this data in an object-oriented way within applications. Many tools and frameworks exist for mapping between data and objects (we will see

J2EE's standard system, CMP entity beans, later in this chapter), but often these are proprietary or excessively heavy weight systems.

In the Internet Banking Web Application we want to make the user interface richer, while still allowing us to use the MVC architecture described in 2.3.2, "Struts" on page 51. In addition, our developers want a simple, lightweight, object-oriented database access system, which will remove the need for direct JDBC coding.

JavaServer Faces (JSF)

JSF is a framework for developing Java Web applications. The JSF framework aims to unify techniques for solving a number of common problems in Web application design and development, such as:

- User interface development: JSF allows direct binding of user interface (UI) components to model data. It abstracts request processing into an event-driven model. Developers can use extensive libraries of prebuilt UI components that provide both basic and advanced Web functionality.
- Navigation: JSF introduces a layer of separation between business logic and the resulting UI pages; stand-alone flexible rules drive the flow of pages.
- Session and object management: JSF manages designated model data objects by handling their initialization, persistence over the request cycle, and cleanup.
- Validation and error feedback: JSF allows direct binding of reusable validators to UI components. The framework also provides a queue mechanism to simplify error and message feedback to the application user. These messages can be associated with specific UI components.
- Internationalization: JSF provides tools for internationalizing Web applications, supporting number, currency, time, and date formatting, and externalizing of UI strings. JSF is easily extended in a variety of ways to suit the requirements of your particular application. You can develop custom components, renderers, validators, and other JSF objects and register them with the JSF runtime.

The JSF specification can be found at:

http://www.jcp.org/en/jsr/detail?id=127

Service Data Objects (SDO)

SDO is a data programming architecture and API for the Java platform that unifies data programming across data source types; provides robust support for common application patterns; and enables applications, tools, and frameworks to more easily query, view, bind, update, and introspect data.

SDO was originally developed by IBM and BEA Systems and is now the subject of a Java specification request (JSR-235), but has not yet been standardized under this process.

SDOs are designed to simplify and unify the way in which applications handle data. Using SDO, application programmers can uniformly access and manipulate data from heterogeneous data sources, including relational databases, XML data sources, Web services, and enterprise information systems.

The SDO architecture consists of three major components:

- Data object: The data object is designed to be an easy way for a Java programmer to access, traverse, and update structured data. Data objects have a rich variety of strongly and loosely typed interfaces for querying and updating properties. This enables a simple programming model without sacrificing the dynamic model required by tools and frameworks. A data object may also be a composite of other data objects.
- Data graph: SDO is based on the concept of disconnected data graphs. A data graph is a collection of tree-structured or graph-structured data objects. Under the disconnected data graphs architecture, a client retrieves a data graph from a data source, mutates the data graph, and can then apply the data graph changes to the data source. The data graph also contains some metadata about the data object, including change summary and metadata information. The metadata API allows applications, tools, and frameworks to introspect the data model for a data graph, enabling applications to handle data from heterogeneous data sources in a uniform way.
- Data mediator: The task of connecting applications to data sources is performed by a data mediator. Client applications query a data mediator and get a data graph in response. Client applications send an updated data graph to a data mediator to have the updates applied to the original data source. This architecture allows applications to deal principally with data graphs and data objects, providing a layer of abstraction between the business data and the data source.

More information on JSF and SDO can be found in the IBM Redbook *WebSphere Studio V5.1.2 JavaServer Faces and Service Data Objects*, SG24-6361. This book covers the use of these technologies in WebSphere Studio rather than the Rational Software Development Platform, but the coverage of the technologies is still very useful.

Requirements for the development environment

The development environment should provide tooling to create and edit pages based on JSF, to modify the configuration files for JSF applications, and to test them. For SDO, the development environment should provide wizards to create

SDOs from an existing database (bottom-up mapping) and should make it easy to use the resulting objects in JSF and other applications.

IBM Rational Application Developer V6.0 includes these features.

Figure 2-2 shows how JSF and SDO can be used to create a flexible, powerful MVC-based Web application with simple database access.

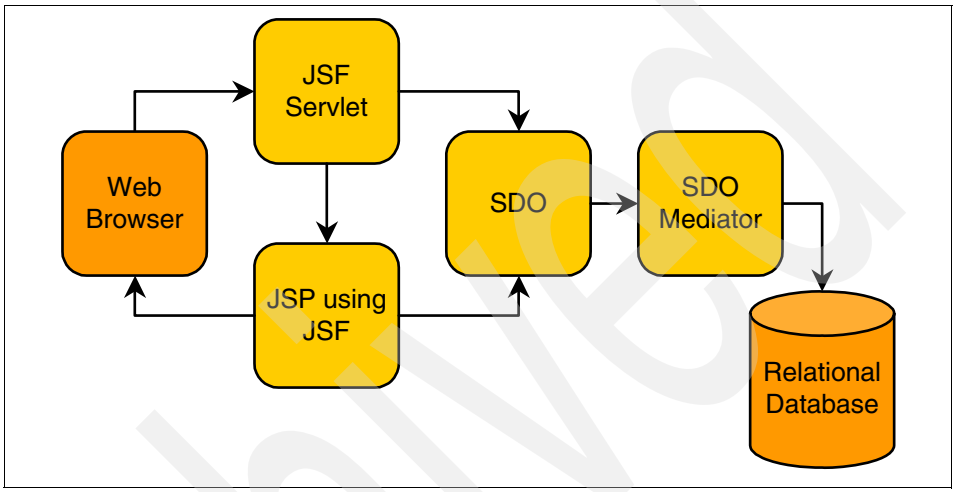


Figure 2-2 JSF and SDO

2.3.4 Portal applications

Portal applications have several important features:

- They can collect content from a variety of sources and present them to the user in a single unified format.
- The presentation can be personalized so that each user sees a view based on their own characteristics or role.
- The presentation can be customized by the user to fulfill their specific needs.
- They can provide collaboration tools, which allow teams to work in a virtual office.
- They can provide content to a range of devices, formatting and selecting the content appropriately according to the capabilities of the device.

In the context of our sample scenario, we can use a portal application to enhance the user experience. The Internet Banking Web Application can be integrated with the static Web content providing information about branches and bank services. If the customer has credit cards, mortgages, personal loans, savings

Archived

accounts, shares, insurance, or other products provided by the bank or business partners, these could also be seamlessly integrated into the same user interface, providing the customer with a convenient single point of entry to all these services. The content or these applications can be provided from a variety of sources, with the portal server application collecting the content and presenting it to the user. The user can customize the interface to display only the required components, and the content can be varied to allow the customer to connect using a Web browser, a personal digital assistant (PDA), or mobile phone.

Within the bank, the portal can also be used to provide convenient intranet facilities for employees. Sales staff can use a portal to receive information on the latest products and special offers, information from human resources, leads from colleagues, and so on.

IBM WebSphere Portal

WebSphere Portal runs within WebSphere Application Server, using the J2EE standard services and management capabilities of the server as the basis for portal services. WebSphere Portal provides its own deployment, configuration, administration, and communication features.

The WebSphere Portal Toolkit is provided as part of the Rational Application Developer as a complete development environment for developing portal applications. A wizard allows a developer to begin development of a new portlet application, generating a skeleton portlet application as a project in Rational Application Developer and required deployment descriptors. The Portlet Toolkit also provides debugging support for portal developers.

Java Portlet specification

The Java Portlet V1.0 specification (`http://jcp.org/en/jsr/detail?id=168`) has been developed to provide a standard for the development of Java portlets for portal applications. WebSphere Portal V5.1 supports the Java Portlet standard.

Requirements for the development environment

The development environment should provide wizards for creating portal applications and the associated components and configuration files, as well as editors for all these files. A test environment should be provided to allow portal applications to be executed and debugged.

IBM Rational Application Developer V6.0 includes the WebSphere Portal Toolkit and the WebSphere Portal V5.0.2.2 Integrated Test Environment.

Figure 2-3 on page 57 shows how portal applications fit in with other technologies mentioned in this chapter.

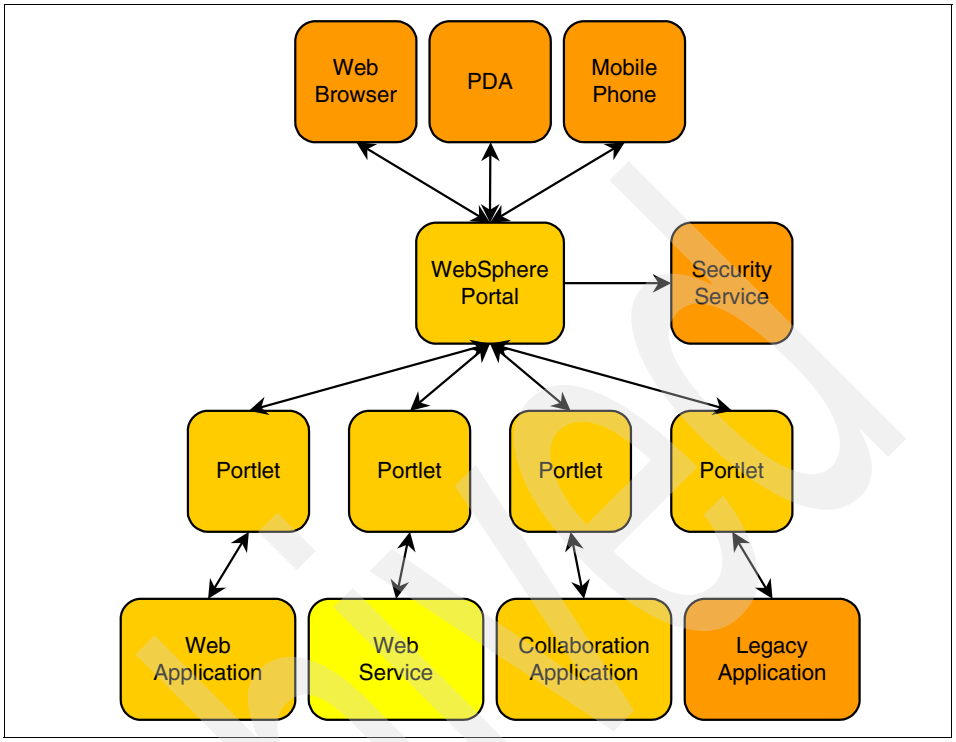


Figure 2-3 Portal applications

2.4 Enterprise JavaBeans

Now that the Internet Banking Web Application is up and running, more issues arise. Some of these relate to the services provided to customers and bank workers and some relate to the design, configuration, and functionality of the systems that perform the back-end processing for the application.

First, we want to provide the same business logic in a new application that will be used by administration staff working in the bank's offices. We would like to be able to reuse the code that has already been generated for the Internet banking Web application without introducing the overhead of having to maintain several copies of the same code. Integration of these business objects into a new application should be made as simple as possible.

Next, we want to reduce development time by using an object-relational mapping system that will keep an in-memory, object-oriented view of data with the relational database view automatically, and provide convenient mapping tools to set up the relationships between objects and data. This system should be

capable of dealing with distributed transactions, since the data might be located on several different databases around the bank's network.

Since we are planning to make business logic available to multiple applications simultaneously, we want a system that will manage such issues as multithreading, resource allocation, and security so that developers can focus on writing business logic code without having to worry about infrastructure matters such as these.

Finally, the bank has legacy systems, not written in Java, that we would like to be able to update to use the new functionality provided by these business objects. We would like to use a technology that will allow this type of interoperability between different platforms and languages.

We can get all this functionality by using Enterprise JavaBeans (EJBs) to provide our back-end business logic and access to data. Later, we will see how EJBs can also allow us to integrate messaging systems and Web services clients with our application logic.

2.4.1 Different types of EJBs

This section describes several types of EJBs including session, entity, and message driven beans.

Session EJBs

Session EJBs are task-oriented objects, which are invoked by an EJB client. They are non-persistent and will not survive an EJB container shutdown or crash. There are two types of session EJB: *Stateless* and *stateful*.

Session beans often act as the external face of the business logic provided by EJBs. The session facade pattern, described in many pattern catalogs including *Core J2EE Patterns: Best Practices and Design Strategies* by Crupi, et al., describes this idea. The client application that needs to access the business logic provided by some EJBs sees only the session beans. The low-level details of the persistence mechanism are hidden behind these session beans (the session bean layer is known as the session facade). As a result of this, the session beans that make up this layer are often closely associated with a particular application and may not be reusable between applications.

It is also possible to design reusable session beans, which might represent a common service that can be used by many applications.

Stateless session EJBs

Stateless session EJBs are the preferred type of session EJB, since they generally scale better than stateful session EJBs. Stateless beans are pooled by

the EJB container to handle multiple requests from multiple clients. In order to permit this pooling, stateless beans cannot contain any state information that is specific to a particular client. Because of this restriction, all instances of a stateless bean are equivalent, allowing the EJB container to assign an instance to any client.

Stateful session EJBs

Stateful session EJBs are useful when an EJB client needs to call several methods and store state information in the session bean between calls. Each stateful bean instance must be associated with exactly one client, so the container is unable to pool stateful bean instances.

Entity EJBs

Entity EJBs are designed to provide an object-oriented view of data. The data itself is stored by an external persistence mechanism, such as a relational database or other enterprise information system. Entity beans should normally be designed to be general purpose, not designed to work with one particular application—the application-specific logic should normally be placed in a layer of session EJBs that use entity beans to access data when required, as described above.

Once an entity bean has been created it can be found using a key or using some other search criteria. It persists until it is explicitly removed. Two main types of entity bean exist: Container-managed or bean-managed persistence.

Container-managed persistence (CMP)

The EJB container handles all database access required by the entity bean. The bean's code contains no database access (SQL) calls. As a result, the bean's code is not tied to a specific database. Because of this flexibility, even if you redeploy the same entity bean on different J2EE servers that use different databases, you do not have to modify or recompile the bean's code. The container must provide an object-relational mapping tool to allow a developer or deployer to describe how the attributes of an entity bean map onto columns in tables of a database.

Bean-managed persistence (BMP)

The developer handles all storage-specific access required by the entity bean. This allows the developer to use non-relational storage options and features of relational databases that are not supported by CMP entity beans, such as complex SQL and stored procedures.

Message-driven EJBs (MDBs)

MDBs are designed to receive and process messages.They can be accessed only by sending a message to the messaging server that the bean is configured

to listen to. MDBs are stateless and can be used to allow asynchronous communication between a client EJB logic via some type of messaging system. MDBs are normally configured to listen to Java Message Service (JMS) resources, although from EJB V2.1 other messaging systems may also be supported. MDBs are normally used as adapters to allow logic provided by session beans to be invoked via a messaging system; as such, they may be thought of as an asynchronous extension of the session facade concept described above, known as the message facade pattern. Message-driven beans can only be invoked in this way and therefore have no specific client interface.

2.4.2 Other EJB features

This section describes other EJB features not discussed previously.

Container-managed relationships (CMR)

From EJB V2.0, the EJB container is able to manage the relationships between entity beans. All the code required to manage the relationships is generated automatically as part of the deployment process.

- One-to-one relationships: A CMP entity bean is associated with a single instance of another CMP entity bean (for example, customer has one address).
- One-to-many relationships: A CMP entity bean is associated with multiple instances of another CMP entity bean (for example, account has many transactions).
- Many-to-many relationship: Multiple instances of a CMP entity bean are associated with multiple instances of another CMP entity bean (for example, customer has many accounts, account belongs to many customers).

The methods required to traverse the relationships are generated as part of the development and code generation mechanisms.

EJB query language (EJB QL)

EJB QL is used to specify queries for CMP entity beans. It is based on SQL and allows searches on the persistent attributes of an enterprise bean and allows container-managed relationships to be traversed as part of the query. EJB QL defines queries in terms of the declared structure of the EJBs themselves, not the underlying data store; as a result, the queries are independent of the bean's mapping to a persistent store.

An EJB query can be used to define a finder method or a select method of a CMP entity bean. Finder and select methods are specified in the bean's deployment descriptor using the <ejb-ql> element. Queries specified in the deployment descriptor are compiled into SQL during deployment.

An EJB QL query is a string that contains the following elements:

- A SELECT clause that specifies the enterprise beans or values to return
- A FROM clause that names the bean collections
- A WHERE clause that contains search predicates over the collections

Only SELECT statements can be created using EJB QL. EJB QL is like a cut-down version of SQL, using the abstract persistence schema defined by the EJBs rather than tables and columns. The language has been extended in EJB V2.1 compared with V2.0. A query can contain input parameters that correspond to the arguments of the finder or select method.

Local and remote interfaces

EJBs were originally designed around remote invocation using the Java Remote Method Invocation (RMI) mechanism, and later extended to support standard Common Object Request Broker Architecture (CORBA) transport for these calls using RMI over the Internet Inter-ORB Protocol (RMI-IIOP). Since many EJB clients make calls to EJBs that are in the same container, from EJB V2.0 onwards local interfaces can be used instead for calls when the client is in the same JVM as the target EJB. For EJB-to-EJB and Servlet-to-EJB calls, this avoids the expensive network call. A session or entity EJB can be defined as having a local and/or remote interface. MDBs do not have any interfaces, so this issue does not arise.

EJB Timer Service

The EJB Timer Service was introduced with EJB V2.1. A bean provider can choose to implement the javax.ejb.TimedObject interface, which requires the implementation of a single method, ejbTimeout. The bean creates a Timer object by using the TimerService object obtained from the bean's EJBContext. Once the Timer object has been created and configured, the bean will receive messages from the container according to the specified schedule; the container calls the ejbTimeout method at the appropriate interval.

2.4.3 Requirements for the development environment

The development environment should provide wizards for creating the various types of EJB, tools for mapping CMP entity beans to relational database systems and test facilities.

IBM Rational Application Developer V6.0 provides all these features.

Figure 2-4 shows how EJBs work with other technologies already discussed. The provision of remote interfaces to EJBs means that the method calls made by the JavaBean to the session EJB could take place across a network connection, allowing the application to be physically distributed across several machines.

Performance considerations might make this option less attractive, but it is a useful technology choice in some situations. As we will see later, using remote interfaces is sometimes the only available choice.

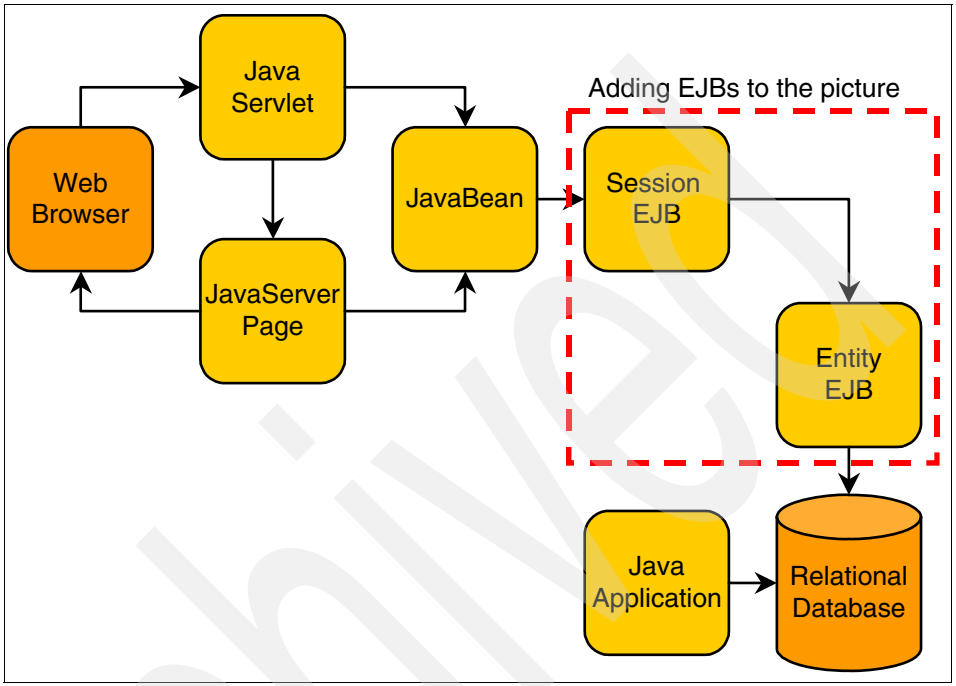


Figure 2-4 EJBs as part of an enterprise application

2.5 J2EE Application Clients

J2EE Application Clients are one of the four types of components defined in the J2EE specification—the others being EJBs, Web components (servletss and JSPs), and Java Applets. They are stand-alone Java applications which use resources provided by a J2EE application server, such as EJBs, data sources and JMS resources.

In the context of our banking sample application, we want to provide an application for bank workers who are responsible for creating accounts and reporting on the accounts held at the bank. Since a lot of the business logic for accessing the bank's database has now been developed using EJBs, we want to avoid duplicating this logic in our new application. Using a J2EE Application Client for this purpose will allow us to develop a convenient interface, possibly a GUI, while still allowing access to this EJB-based business logic. Even if we do not wish to use EJBs for business logic, a J2EE Application Client will allow us to

access data sources or JMS resources provided by the application server and will allow us to integrate with the security architecture of the server.

2.5.1 Application Programming Interfaces (APIs)

The J2EE specification (available from http://java.sun.com/j2ee/) requires the following APIs to be provided to J2EE Application Clients, in addition to those provided by a standard J2SE JVM:

- EJB V2.1 client-side APIs
- JMS V1.1
- JavaMail V1.3
- JavaBeans Activation Framework (JAF) V1.0
- Java API for XML Processing (JAXP) V1.2
- Web Services for J2EE V1.1
- Java API for XML-based Remote Procedure Call (JAX-RPC) V1.1
- SOAP with Attachments API for Java (SAAJ) V1.2
- Java API for XML Registries (JAXR) V1.0
- J2EE Management 1.0
- Java Management Extensions (JMX) V1.2

2.5.2 Security

The J2EE specification requires that the same authentication mechanisms should be made available for J2EE Application Clients as for other types of J2EE components. The authentication features are provided by the J2EE Application Client container, as they are in other containers within J2EE. A J2EE platform can allow the J2EE Application Client container to communicate with an application server to use its authentication services; WebSphere Application Server allows this.

2.5.3 Naming

The J2EE specification requires that J2EE Application Clients should have exactly the same naming features available as are provided for Web components and EJBs. J2EE Application Clients should be able to use the Java Naming and Directory Interface (JNDI) to look up objects using object references as well as real JNDI names. The reference concept allows a deployer to configure references that can be used as JNDI names in lookup code. The references are bound to real JNDI names at deployment time, so that if the real JNDI name is subsequently changed, the code does not need to be modified or

Archived

recompiled—only the binding needs to be updated. References can be defined for:

- EJBs - For J2EE Application Clients, only remote references, since the client cannot use local interfaces
- Resource manager connection factories
- Resource environment values
- Message destinations
- User transactions
- ORBs

Code to look up an EJB might look like this (this is somewhat simplified):

```
accountHome = (AccountHome) initialContext.lookup (
"java:comp/env/ejb/account" );
```

java:comp/env/ is a standard prefix used to identify references, and ejb/account would be bound at deployment time to the real JNDI name used for the Account bean.

2.5.4 Deployment

The J2EE specification only specified the packaging format for J2EE Application Clients, not how these should be deployed—this is left to the Platform provider. The packaging format is specified, based on the standard Java JAR format, and it allows the developer to specify which class contains the *main* method to be executed at run time.

J2EE application clients for the WebSphere Application Server platform run inside the *Application Client for WebSphere Application Server*. This is a product that is available for download from developerWorks®, as well the WebSphere Application Server installation CD.

Refer to the WebSphere Application Server Information Center for more information about installing and using the Application Client for WebSphere Application Server.

The Application Client for WebSphere Application Server provides a **launchClient** command, which sets up the correct environment for J2EE Application Clients and runs the main class.

Archived

2.5.5 Requirements for the development environment

In addition to the standard Java tooling, the development environment should provide a wizard for creating J2EE Application Clients, editors for the deployment descriptor for a J2EE Application Client module, and a mechanism for testing the J2EE Application Client.

IBM Rational Application Developer V6.0 provides these features.

Figure 2-5 on page 65 shows how J2EE Application Clients fit into the picture; since these applications can access other J2EE resources, we can now use the business logic contained in our session EJBs from a stand-alone client application. J2EE Application Clients run in their own JVM, normally on a different machine from the EJBs, so they can only communicate using remote interfaces.

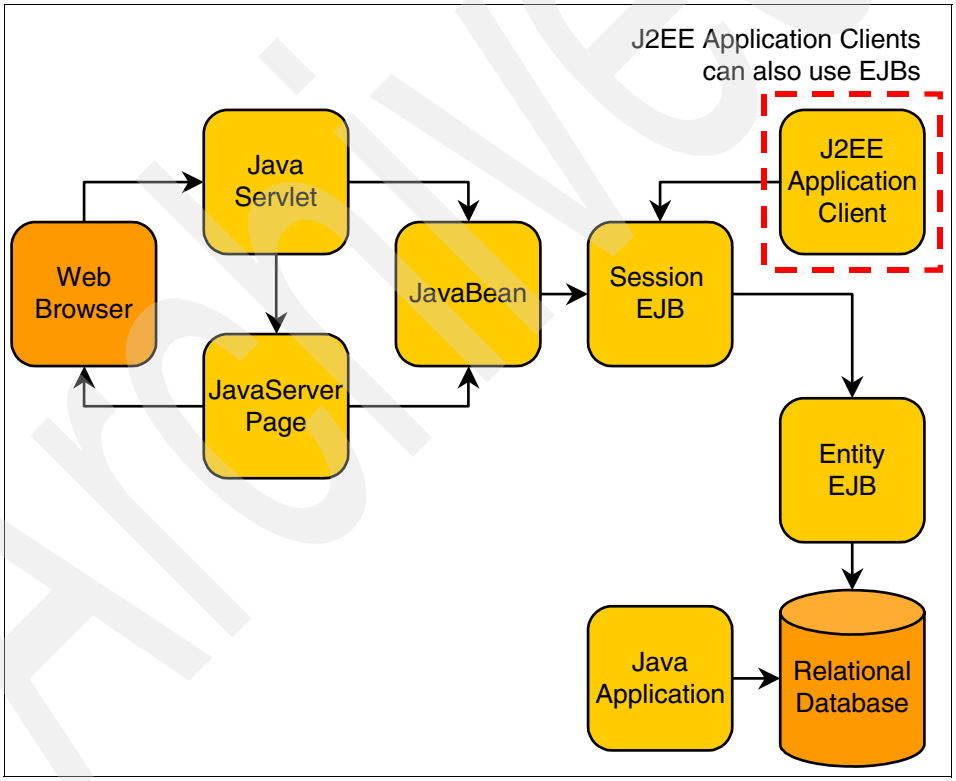


Figure 2-5 J2EE Application Clients

Archive

2.6 Web Services

The bank's computer system is now quite sophisticated, comprising:

- A database for storing the bank's data
- A Java application allowing bank employees to access the database
- A static Web site, providing information on the bank's branches, products, and services
- A Web application, providing Internet banking facilities for customers, with various technology options available
- An EJB back-end, providing:
  - Centralized access to the bank's business logic through session beans
  - Transactional, object-oriented access to data in the bank's database through entity beans
- A J2EE Application Client that can use the business logic in session beans

So far, everything is quite self-contained. Although clients can connect from the Web in order to use the Internet banking facilities, the business logic is all contained within the bank's systems, and even the Java application and J2EE Application Client are expected to be within the bank's private network.

The next step in developing our service is to enable mortgage agents, who search many mortgage providers to find the best deal for their customers, to access business logic provided by the bank to get the latest mortgage rates and repayment information. While we want to enable this, we do not want to compromise security, and we need to take into account that fact that the mortgage brokers may not be using systems based on Java at all.

The League of Agents for Mortgage Enquiries has published a description of services that its members might use to get this type of information. We want to conform to this description in order to allow the maximum number of agents to use our bank's systems.

We may also want to be able to share information with other banks; for example, we may wish to exchange information on funds transfers between banks. Standard mechanisms to perform these tasks have been provided by the relevant government body.

These issues are all related to interoperability, which is the domain addressed by Web services. Web services will allow us to enable all these different types of communication between systems. We will be able to use our existing business logic where applicable and develop new Web services easily where necessary.

Archived

2.6.1 Web Services in J2EE V1.4

Web Services provide a standard means of communication among different software applications. Because of the simple foundation technologies used in enabling Web services, it is very simple to a Web service regardless of the Platform, operating system, language, or technology used to implement it.

A *service provider* creates a Web service and publishes its interface and access information to a *service registry* (or *service broker*). A *service requestor* locates entries in the *service registry*, then binds to the *service provider* in order to invoke its Web service.

Web services use the following standards:

- SOAP: A XML-based protocol that defines the messaging between objects
- Web Services Description Language (WSDL): Describes Web services interfaces and access information
- Universal Description, Discovery, and Integration (UDDI): A standard interface for service registries, which allows an application to find organizations and services

The specifications for these technologies are available at:

- http://www.w3.org/TR/soap/
- http://www.w3.org/TR/wsdl/
- http://www.oasis-open.org/committees/uddi-spec/doc/tcspecs.html

Figure 2-6 shows how these technologies fit together.

Archived

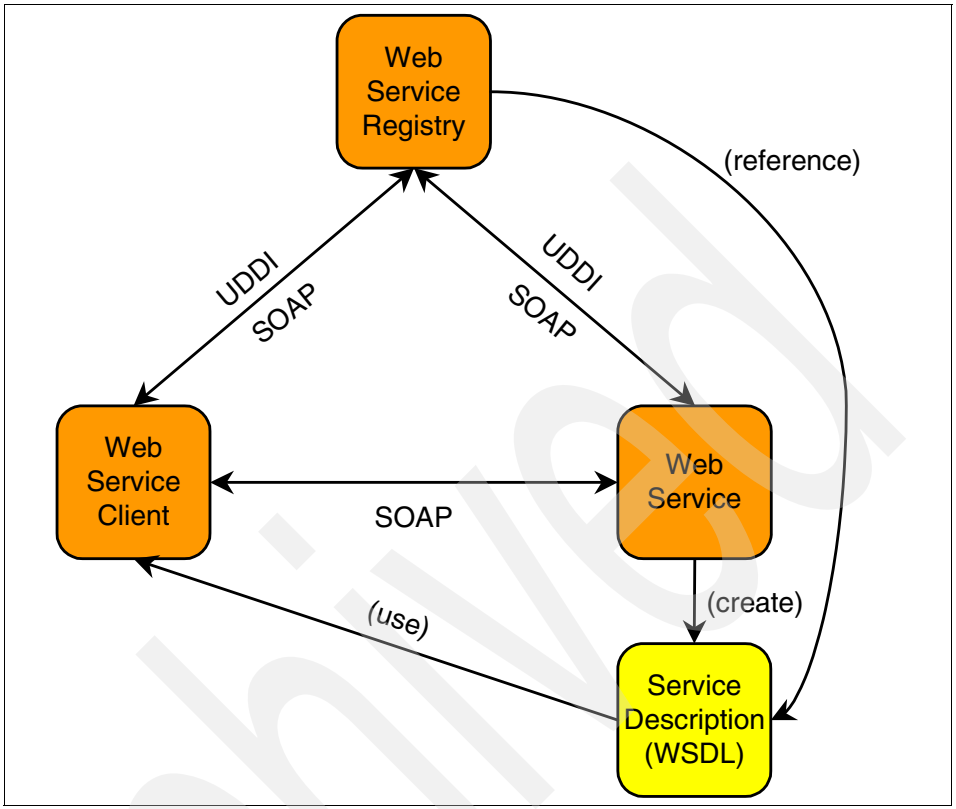


Figure 2-6 Web services foundation technologies

Web services are now included in the J2EE specification (from V1.4), so all J2EE application servers that support J2EE V1.4 have exactly the same basic level of support for Web services; some will also provide enhancements as well.

The key Web services-related Java technologies that are specified in the J2EE V1.4 specification are:

- Web Services for J2EE V1.1:

  `http://jcp.org/en/jsr/detail?id=921`

> **Note:** The URL quoted above for the Web Services for J2EE V1.1 specification leads to a page entitled `JSR 921: Implementing Enterprise Web Services 1.1`. This is the V1.1 maintenance release for Web Services for J2EE, which was originally developed under JSR 109.

Archived

- Java API for XML-based Remote Procedure Call (JAX-RPC) V1.1:

  http://java.sun.com/xml/jaxrpc/

The Web services support in J2EE also relies on the underlying XML support in the Platform, provided by JAXP and associated specifications.

Web Services for J2EE defines the programming and deployment model for Web services in J2EE. It includes details of the client and server programming models, handlers (a similar concept to servlets filters), deployment descriptors, container requirements, and security.

JAX-RPC defines the various APIs required to enable XML-based remote procedure calls in Java.

Since interoperability is a key goal in Web services, an open, industry organization known as the Web Services Interoperability Organization (WS-I, http://ws-i.org/) has been created to allow interested parties to work together to maximize the interoperability between Web services implementations. WSI-I has produced the following set of interoperability profiles:

- WS-I Basic Profile

  http://ws-i.org/Profiles/BasicProfile-1.0.html

- WS-I Simple SOAP Binding Profile

  http://ws-i.org/Profiles/SimpleSoapBindingProfile-1.0.html

- WS-I Attachments Profile

  http://ws-i.org/Profiles/AttachmentsProfile-1.0.html

The J2EE V1.4 specification allows for ordinary Java resources and stateless session EJBs to be exposed as Web services. The former, known as *JAX-RPC service endpoint implementations*, can be hosted within the J2EE application server's Web container, and the latter can be hosted within the EJB container.

Requirements for the development environment

The development environment should provide facilities for creating Web services from existing Java resources—both JAX-RPC service endpoint implementations and stateless session EJBs. As part of the creation process, the tools should also produce the required deployment descriptors and WSDL files. Editors should be provided for WSDL files and deployment descriptors.

The tooling should also allow skeleton Web services to be created from WSDL files and should provide assistance in developing Web services clients, based on information obtained from WSDL files.

A range of test facilities should be provided, allowing a developer to test Web services and clients as well as UDDI integration.

IBM Rational Application Developer V6.0 provides all this functionality.

Figure 2-7 shows how the Web services technologies fit into the overall programming model we have been discussing.

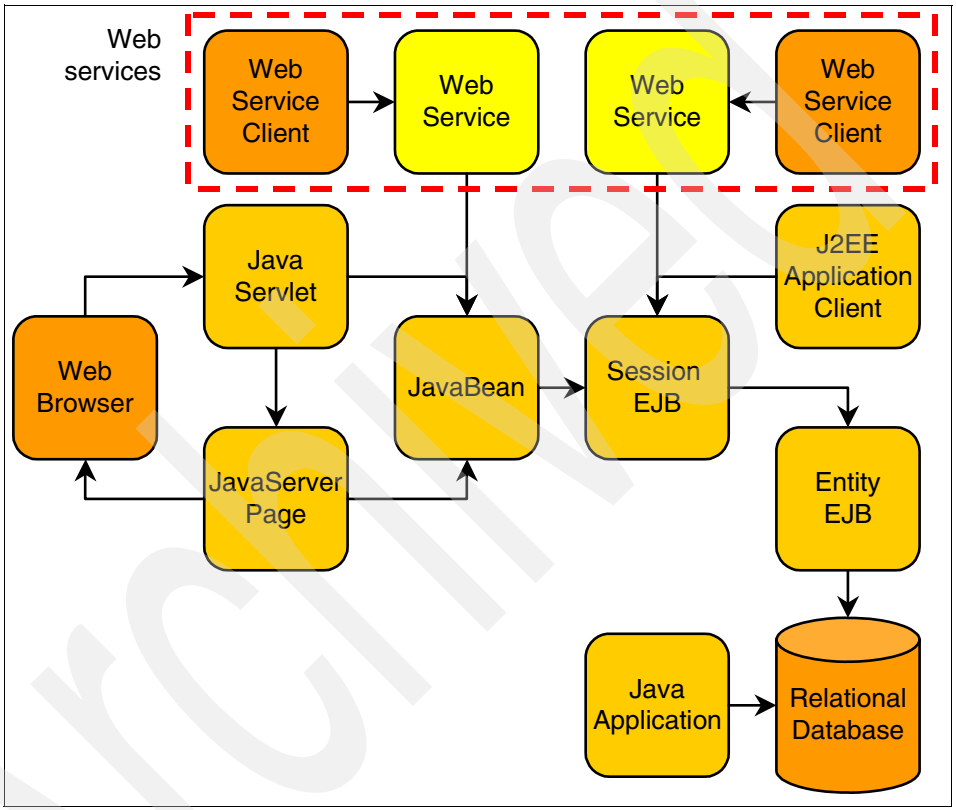


Figure 2-7 Web services

2.7 Messaging systems

Although Web services offer excellent opportunities for integrating disparate systems, they currently have some drawbacks:

- Security specifications are available for Web services, but they have not been fully implemented on all platforms at this stage.

Archive

- Web services do not provide guaranteed delivery, and there is no widely implemented, standard way of reporting the reliability of a Web service.
- There is no standard way of scaling Web services to the degree required by modern information technology systems, no way of distributing Web services transparently across servers, and no way of recovering from system failures.

The bank has numerous automatic teller machines (ATMs), each of which has a built-in computer providing user interface and communication support. The ATMs are designed to communicate with the bank's central computer systems using a secure, reliable, highly scalable messaging system. We would like to integrate the ATMs with our system so that transactions carried out at an ATM can be processed using the business logic we have already implemented. Ideally, we would also like to have the option of using EJBs to handle the messaging for us.

Many messaging systems exist that provide features like these. IBM's solution in this area is IBM WebSphere MQ, which is available on many platforms and provides application programming interfaces in several languages. From the point of view of our sample scenario, WebSphere MQ provides Java interfaces that we can use in our applications—in particular, we will consider the interface that conforms to the Java Message Service (JMS) specification. The idea of JMS is similar to that of JDBC—a standard interface providing a layer of abstraction for developers wishing to use messaging systems without being tied to a specific implementation.

2.7.1 Java Message Service (JMS)

JMS defines (among other things):

- A messaging model: The structure of a JMS message and an API for accessing the information contained within a message. The JMS interface is, javax.jms.Message, implemented by several concrete classes, such as javax.jms.TextMessage.
- Point-to-point (PTP) messaging: A queue-based messaging architecture, similar to a mailbox system. The JMS interface is javax.jms.Queue.
- Publish/subscribe (Pub/Sub) messaging: A topic-based messaging architecture, similar to a mailing list. Clients subscribe to a topic and then receive any messages that are sent to the topic. The JMS interface is javax.jms.Topic.

More information on JMS can be found at:

`http://java.sun.com/products/jms/`

2.7.2 Message-driven EJBs (MDBs)

MDBs were introduced into the EJB architecture at V2.0 and have been extended in EJB V2.1. MDBs are designed to consume incoming messages sent from a destination or endpoint system the MDB is configured to listen to. From the point of view of the message-producing client, it is impossible to tell how the message is being processed—whether by a stand-alone Java application, a MDB, or a message-consuming application implemented in some other language. This is one of the advantages of using messaging systems; the message-producing client is very well decoupled from the message consumer (similar to Web services in this respect).

From a development point of view, MDBs are the simplest type of EJB, since they do not have clients in the same sense as session and entity beans. The only way of invoking an MDB is to send a message to the endpoint or destination that the MDB is listening to. In EJB V2.0, MDBs only dealt with JMS messages, but in EJB V2.1 this is extended to other messaging systems. The development of an MDB is different depending on the messaging system being targeted, but most MDBs are still designed to consume messages via JMS, which requires the bean class to implement the javax.jms.MessageListener interface, as well as javax.ejb.MessageDrivenBean.

A common pattern in this area is the message facade pattern, as described in *EJB Design Patterns: Advanced Patterns, Processes and Idioms* by Marinescu. This book is available for download from:

`http://theserverside.com/articles/`

According to this pattern, the MDB simply acts as an adapter, receiving and parsing the message, then invoking the business logic to process the message using the session bean layer.

2.7.3 Requirements for the development environment

The development environment should provide a wizard to create MDBs and facilities for configuring the MDBs in a suitable test environment. The test environment should also include a JMS-compliant server (this is a J2EE V1.4 requirement anyway).

Testing MDBs is challenging, since they can only be invoked by sending a message to the messaging resource that the bean is configured to listen to. However, WebSphere Application Server V6.0, which is provided as a test environment within Rational Application Developer, includes an embedded JMS messaging system that can be used for testing purposes. A JMS client must be developed to create the test messages.

Figure 2-8 shows how messaging systems and MDBs fit into the application architecture.

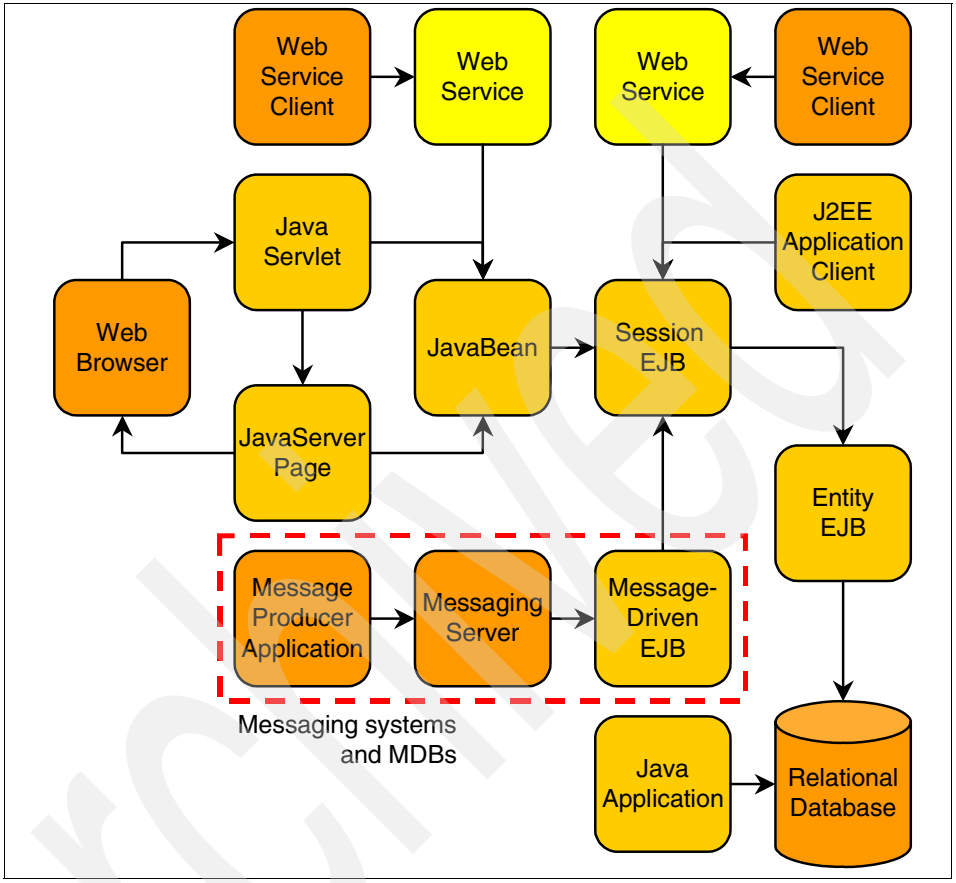


Figure 2-8 Messaging systems

Archive

Archived

3

Workbench setup and preferences

After installing IBM Rational Application Developer V6.0, the Workbench is configured with a simple default configuration to make it easier to navigate for new users. Developers are made aware of new features as needed for a given task via the progressive disclosure of tools concept (capabilities) provided by Eclipse 3.0. Alternatively, developers can configure the Workbench preferences for their needs manually at any time. This chapter describes the most commonly customized Rational Application Developer preferences.

The following topics are discussed in this chapter:

- Workbench basics
- Preferences
- Java development preferences

Note: For information on installing IBM Rational Application Developer V6.0 refer to Appendix A, “IBM product installation and configuration tips” on page 1371.

© Copyright IBM Corp. 2005. All rights reserved.

3.1 Workbench basics

When starting Rational Application Developer after the installation, you will see a single window with the Welcome screen, as seen in Figure 3-1. The Welcome page can be accessed subsequently by selecting **Help** → **Welcome** from the Workbench menu bar. The Welcome screen is provided to guide a new user of IBM Rational Application Developer V6.0 to the various aspects of the tool.

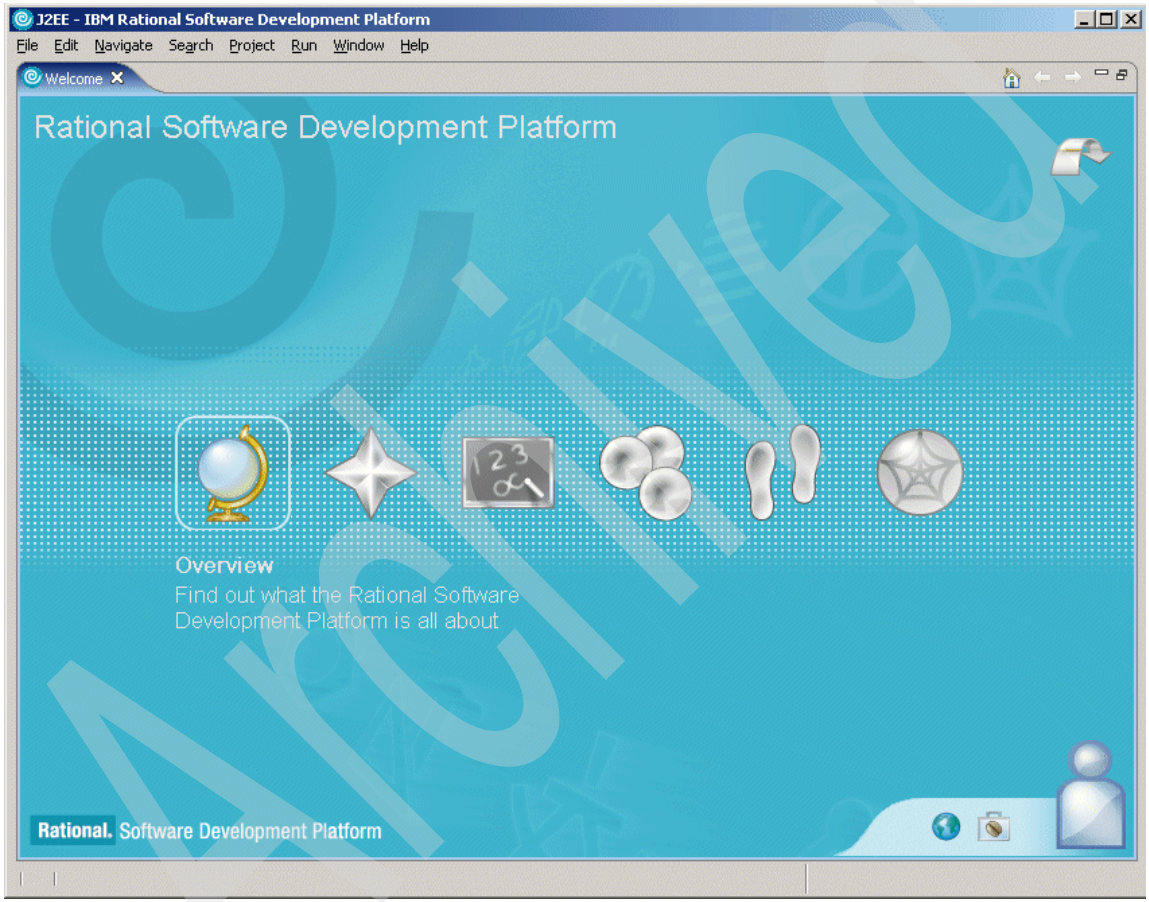


Figure 3-1 Rational Application Developer Workbench startup screen

The Welcome screen presents six icons, each including a description via hover help (move mouse over icon to display description). We provide a summary of each Welcome screen icon in Table 3-1 on page 77.

Table 3-1 Welcome screen assistance capabilities

Icon Image	Name	Description
	Overview	Provides an overview of the key functions in Rational Application Developer and a tutorial requiring Flash Player 6 r65 or later
	What's New	A description of the major new features and highlights of the product
	Tutorials	Tutorial screens to assist in a stepwise manner through some of the major points of Web development using Rational Application Developer
	Samples	Sample code for the user to begin working with "live" examples with minimal assistance
	First Steps	Instructions and a quick access wizards to focus on a *first time* user in developing Web-based applications
	Web Resources	URL links to understand the product and technology in the tooling

Users experienced with Rational Application Developer or the concepts that the product provides can close the Welcome screen by clicking the **X** for the view to close it down, or clicking the icon in the top right corner arrow. They will then be presented with the default perspective of the J2EE perspective. Each perspective in Rational Application Developer contains multiple views, such as the Project Explorer view, Snippets view, and others. More information regarding perspectives and views are provided in Chapter 4, "Perspectives, views, and editors" on page 131.

The far right of the window has a shortcut icon, highlighted by a circle in Figure 3-2 on page 78, that allows you to open available perspectives, and places them in the shortcut bar next to it. Once the icons are on the shortcut bar, the user is able to navigate between perspectives that are already open. The

name of the active perspective is shown in the title of the window, and its icon is in the shortcut bar on the right side as a pushed button.

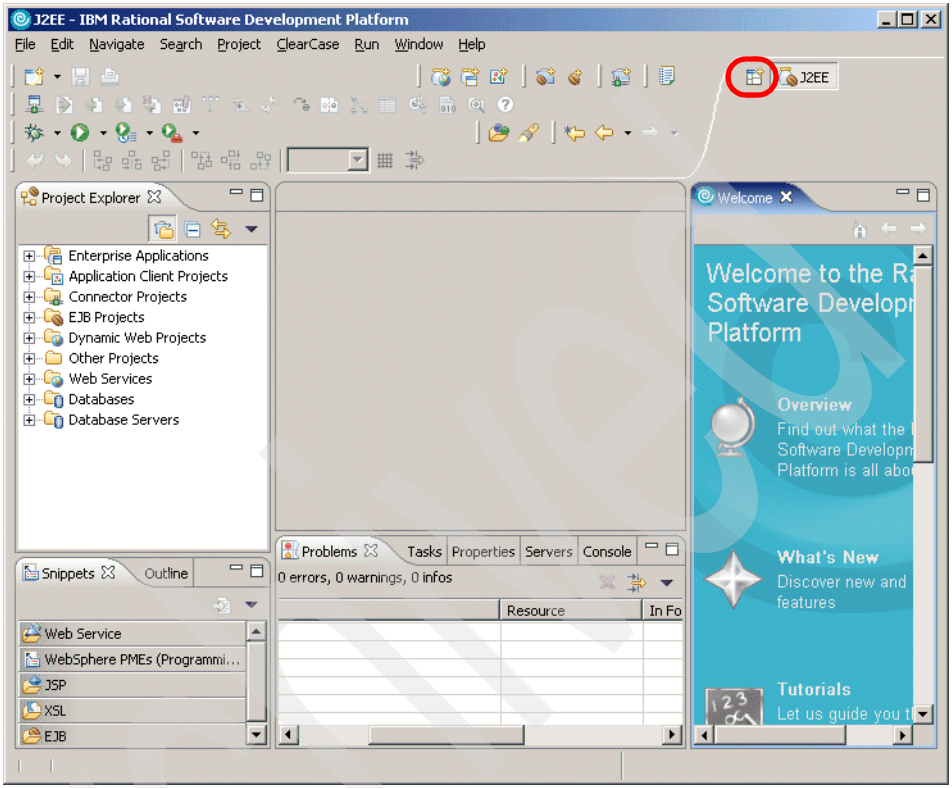


Figure 3-2 J2EE perspective in Rational Application Developer

The term *Workbench* refers to the desktop development environment. Each Workbench window of Rational Application Developer contains one or more perspectives. Perspectives contain views and editors and control what appears in certain menus and toolbars.

3.1.1 Workspace basics

When you start up Rational Application Developer you are prompted to provide a workspace to start up. On first startup, the path will look something like the following, depending on the installation path:

```
c:\Documents and Settings\<user>\IBM\rationalsdp6.0\workspace
```

Where <user> is the Windows user ID you have logged in as.

The Rational Application Developer workspace is a private work area created for the individual developer, and it holds the following information:

- Rational Application Developer environment metadata, such as configuration information and temporary files.
- A developer's copy of projects that they have created, which includes source code, project definition, or config files and generate files such as class files.

Resources that are modified and saved are reflected on the local file system. Users can have many workspaces on their local file system to contain different project work that they are working on or differing versions. Each of these workspaces may be configured differently, since they will have their own copy of metadata that will have configuration data for that workspace.

Important: Although workspace metadata stores configuration information, this does not mean that the metadata can be transferred between workspaces. In general, we do not recommend copying or using the metadata in one workspace, with a another one. The recommended approach is to create a new workspace and then configure it appropriately.

Rational Application Developer allows you to open more than one Workbench at a time. It opens another window into the same Workbench, allowing you to work in two differing perspectives. Changes that are made in one window will be reflected back to the other windows, and you are not permitted to work in more that one window at a time. That is, you cannot switch between windows while in the process of using a wizard in one window. Opening a new Workbench in another window is done by selecting **Window** → **New Window**, and a new Workbench with the same perspective will open in a new window.

The default workspace can be started on first startup of Rational Application Developer by specifying the workspace location on the local machine and selecting the check box **Use this as the default and do not ask again**, as shown in Figure 3-3 on page 80.

This will ensure on the next startup of Rational Application Developer that the workspace will automatically use the directory specified initially, and it will not prompt for the workspace in the future.

Note: See "Setting the workspace with a prompt dialog" on page 82 describing how to reestablish Rational Application Developer prompting for the workspace at startup.

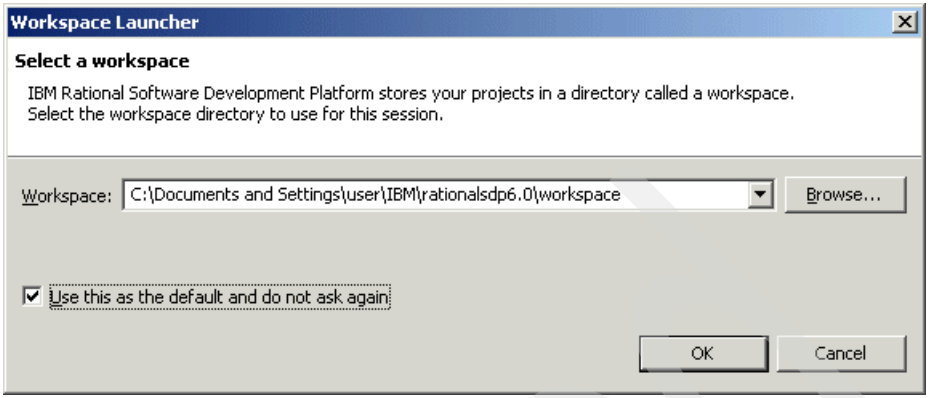


Figure 3-3 Setting the default workspace on startup

The other way to enforce the use of a particular workspace is by using the **-data <workspace>** command-line argument on Rational Application Developer, where <workspace> is a path name on the local machine where the workspace is located and should be a full path name to remove any ambiguity of location of the workspace.

Tip: On a machine where there are multiple workspaces used by the developer, a shortcut would be the recommended approach in setting up the starting workspace location. The target would be "<IRAD Install Dir>\rationalsdp.exe" -data <workspace>, and the startup directory <IRAD Install Dir>.

By using the -data arguement you can start a second instance of Rational Application Developer that uses a different workspace. For example, if your second instance should use the NewWorkspace workspace folder, you can launch Rational Application Developer with this command (assuming that the product has been installed in the default installation directory):

```
c:\Program Files\IBM\Rational\SDP\6.0\rationalsdp.exe -data c:\NewWorkspace
```

There are a number of arguments that you can add when launching the Rational Software Development Platform; some of these have been sourced from the online help and are shown in Figure 3-2 on page 81. More options can be found by searching for Running Eclipse in the online help.

Table 3-2 Startup parameters

Command	Description
-configuration configurationFileURL	The location for the Platform configuration file, expressed as a URL. The configuration file determines the location of the Platform, the set of available plug-ins, and the primary feature. Note that relative URLs are not allowed. The configuration file is written to this location when Application Developer is installed or updated.
-consolelog	Mirrors the Eclipse platform's error log to the console used to run Eclipse. Handy when combined with **-debug**.
-data <workspace directory>	Start Rational Application Developer with a specific workspace located in <workspace directory>.
-debug [optionsFileURL]	Puts the Platform in debug mode and loads the debug options from the file at the given URL, if specified. This file indicates which debug points are available for a plug-in and whether or not they are enabled. If a file path is not given, the Platform looks in the directory that Rational Application Developer was started from for a file called .options. Note that relative URLs are not allowed.
-refresh	Option for performing a global fresh of the workspace on startup to reconcile any changes made on the file system since the Platform was last run.
-showlocation	Option for displaying the location of the workspace in the window title bar.
-vm vmPath	This optional option allows you to set the location of Java Runtime Environment (JRE) to run Application Developer. Relative paths are interpreted relative to the directory that Eclipse was started from.
-vmargs -Xmx256M	For large-scale development you should modify your VM arguments to make more heap available. This example allows the Java heap to grow to 256 MB. This may not be enough for large projects.

Memory consideration

Use the -vmargs flag to set limits to the memory that is used by Application Developer. For example, with only 768 MB RAM you may be able to get better performance by limiting the memory:

```
-vmargs -Xmx150M
```

You can also modify VMArgs initialization parameters in the rationalsdp.ini file (in the installation directory):

```
VMArgs=-Xms64M -Xmx150M
```

These arguments significantly limit the memory utilization. Setting the -Xmx argument below 150M does begin to degrade performance.

Setting the workspace with a prompt dialog

The default behavior on installation is that Rational Application Developer will prompt for the workspace on startup. In the event that you have set the check box on this startup screen to not ask again (see Figure 3-3 on page 80), there is a procedure to re-initiate this, described as follows:

1. Select **Window** → **Preferences**.
2. On opening the dialog, expand the tree, as shown in Figure 3-4.

Archived

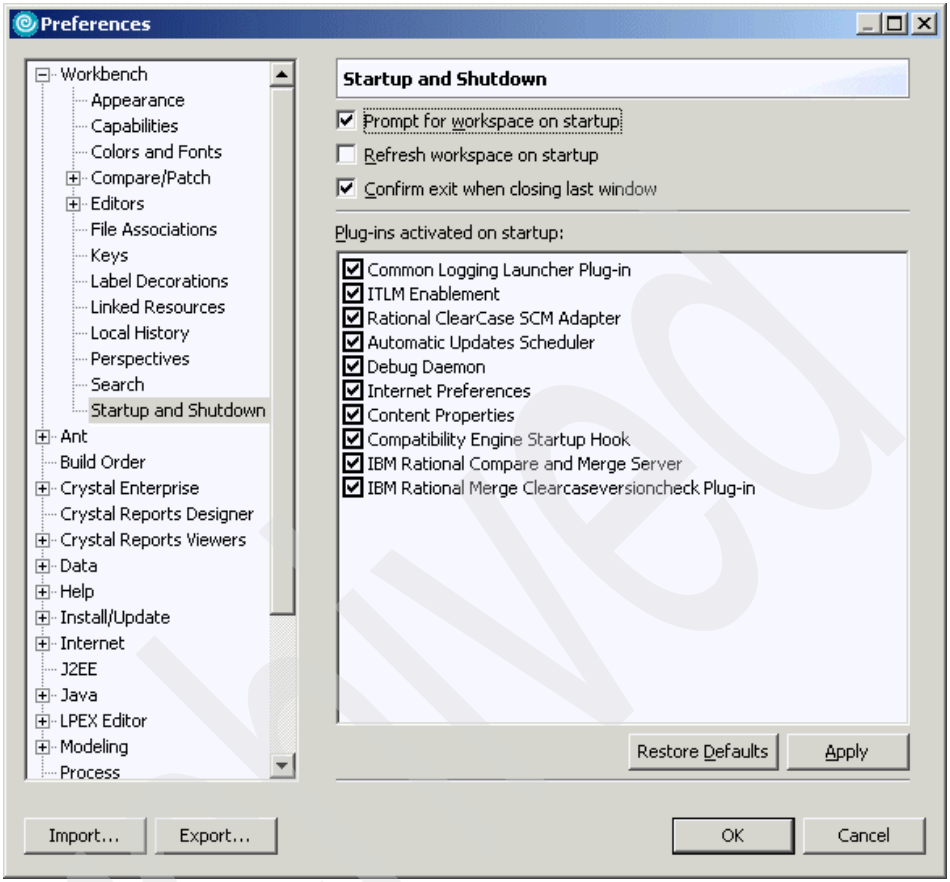


Figure 3-4 Setting the prompt dialog box for workspace selection on startup

3. Check the **Prompt for workspace on startup** checkbox, click **Apply**, and then click **OK**.

On the next startup of Rational Application Developer, the workspace prompt dialog will appear asking the user which workspace to load up.

3.1.2 Rational Application Developer log files

Rational Application Developer provides logging facilities for plug-in developers to log and trace important events, primarily expected or unexpected errors. Log files are a crucial part of the Application Developer problem determination process.

Archived

The primary reason to use log files is if you encounter unexpected program behavior. In some cases, an error message tells you explicitly to look at the error log.

There are two main log files in the .metadata directory of the workspace folder:

- .metadata/.plugins/com.ibm.pzn.resource.wizards/debug.log

  This log is used to catch errors in the event of an error during the use of a wizard.

- .log

  The .log file is used by the Rational Application Developer to capture errors, and any uncaught exceptions from plug-ins. The .log file is cumulative, as each new session of Application Developer appends its messages to the end of the .log file without deleting any previous messages. This enables you to see a history of past messages over multiple Application Developer sessions, each one starting with the `!SESSION` string.

Both log files are ASCII files and can be viewed with a text editor.

3.2 Preferences

The Rational Application Developer preferences can be modified by selecting **Window** → **Preferences** from the menu bar. Opening the preferences displays the dialog shown in Figure 3-5 on page 85.

In the left pane you can navigate through many entries. Each entry has its own preferences page, where you can change the initial options.

This section describes the most important options. Rational Application Developer online help contains a complete description of all options available in the preferences dialogs.

Tip: Each page of Application Developer's preferences dialog contains a Restore Defaults button (see Figure 3-5 on page 85). When you click this button, Application Developer restores the settings of the current dialog to its initial values.

Archived

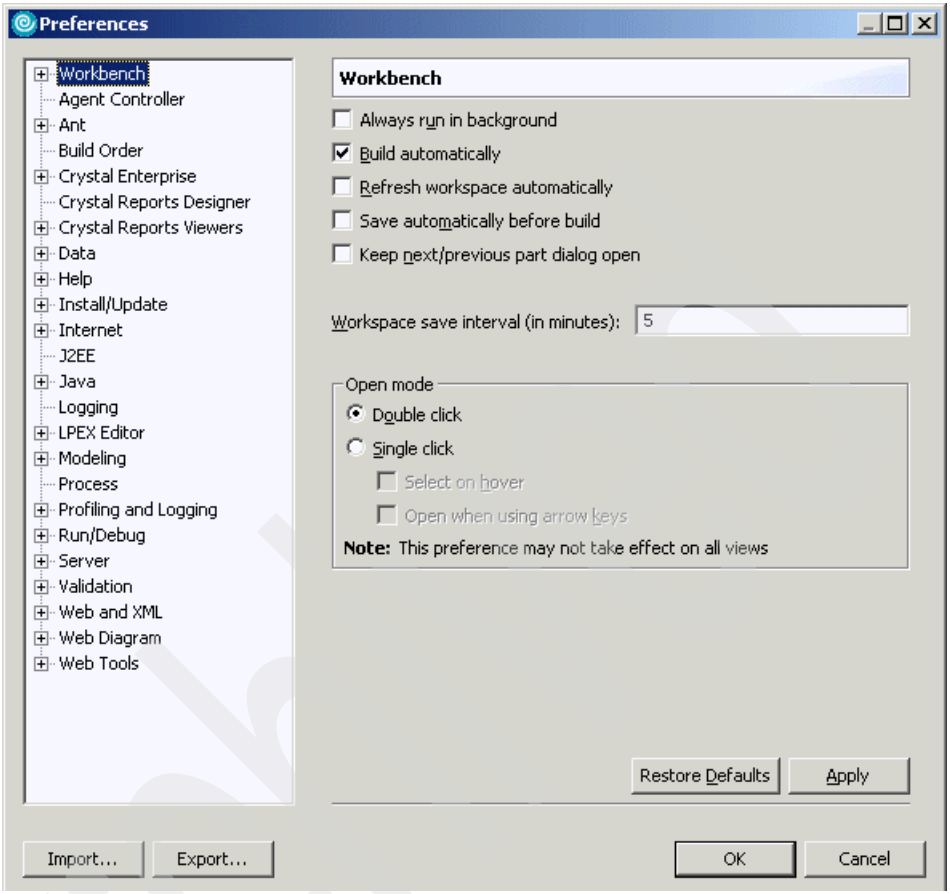


Figure 3-5 Workbench preferences

3.2.1 Automatic builds

Builds or a compilation of Java code in Rational Application Developer is done automatically whenever a resource has been modified and saved. If you require more control regarding builds, you can disable the auto-building feature. To perform a build you have to explicitly start it. This may be desirable in cases where you know that building is of no value until you finish a large set of changes.

If you want to turn off the automatic build feature, select **Windows** → **Preferences** → **Workbench** and uncheck **Build automatically** (see Figure 3-5 on page 85).

In this same dialog (see Figure 3-5 on page 85) you can specify whether you want unsaved resources to be saved before performing a manual build. Check **Save automatically before build** to enable this feature.

3.2.2 Clean build (manual)

Although the automatic build feature may be adequate for many developers, there are a couple of scenarios in which a developer may want to perform a build manually. First, some developers do not want to build automatically since it may slow down development. In this case the developer will need a method of building at the time of their choosing. Second, there are cases when you want to force a build of a project or all projects to resolve build errors and dependency issues. To address these types of issues, Application Developer provides the ability to perform a manual build, known as a *clean* build.

To perform a clean build, do the following:

1. Select the desired project in the Project Explorer view (or other view).
2. Select **Project** → **Clean**.

   Note: From the same menu you will notice that Build Automatically is enabled by default.

3. When the Clean dialog appears, select one of the following options and then click **OK**:
   - Clean all projects (This will perform a build of all projects.)
   - Clean selected projects: <project> (The project selected in step 1 will be displayed or you can use the Browse option to select a project.)

3.2.3 Capabilities

A new feature in Rational Application Developer is the ability to enable and disable capabilities in the tooling. The default setup does not enable a large proportion of the capabilities such as team support for CVS and Rational Clearcase, Web Services development, or XML development.

Capabilities can be enabled in a number of ways in Rational Application Developer. We will describe how to enable capabilities via the following mechanisms:

- Welcome screen
- Windows preferences
- Opening a perspective

Archived

Important: Capabilities that are not enabled will impact the context of a number of areas in Rational Application Developer. This includes the help system, windows preferences, opening perspectives, and the file new menus. Items will not be available, setable, or searchable. For example, the help system will not search through its help files if the capability is disabled.

Enable capability via Welcome screen

The Rational Application Developer Welcome screen provides an icon in the shape of a human figure in the bottom right-hand corner (see Figure 3-1 on page 76) used to enable roles. These assist in setting up available capabilities for the user of the tool through the following process.

The scenario that we will attempt is enable the team capability or role so that the developer can save their resources in a repository.

1. In the Welcome Screen move the mouse to the bottom right corner over the human figure. Single-click the figure.
2. Move the mouse until it is over the desired capability or role, and click the icon. For example, move the mouse over the Team capability or role so that it is highlighted (see Figure 3-6), and click the icon; this will enable the Team capability.

Figure 3-6 Enable Team capability or role in Welcome screen

Archived

Enable capability via Windows Preferences

Using the scenario of enabling the Team capability. The process to enable is as follows:

1. Select **Windows** → **Preferences**.
2. Select and expand the **Workbench** tree.
3. Click **Capabilities** and expand out the **Team** tree, as shown in Figure 3-7 on page 88.

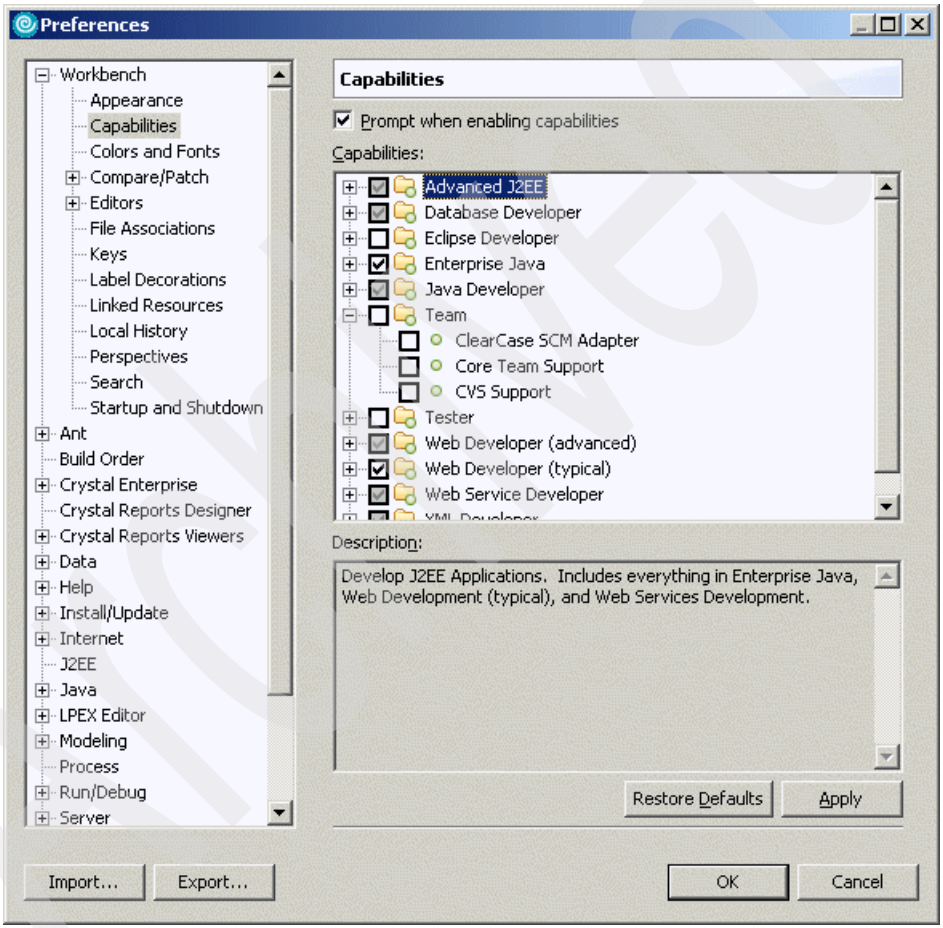


Figure 3-7 Setting the Team capability using Windows preferences

4. Check the **Team** check box, and this will set the check boxes for all components under this tree.
5. Click **Apply** and then click **OK**. This will enable the Team capability.

Enable a capability by opening a perspective

A capability can be enabled by opening the particular perspective required by the capability. Using the scenario of enabling the CVS Team capability, this can be achieved using the following:

1. In any perspective, click **Window** → **Open Perspective** → **Other**.
2. Check the **Show all** check box, select **CVS Repository Exploring** (see Figure 3-8 on page 89), and then click **OK**.

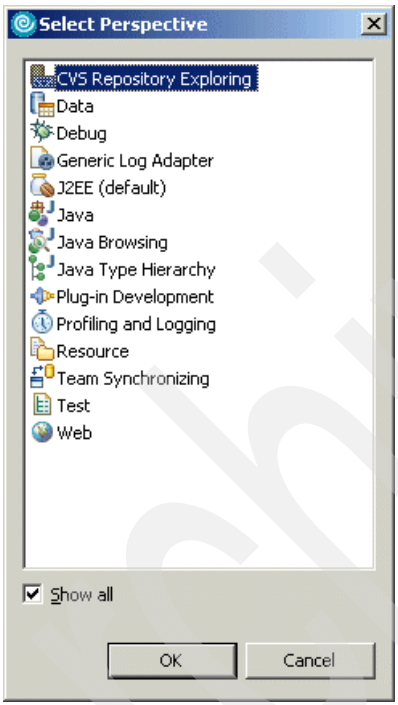


Figure 3-8 Enable capability by opening a perspective

3. A prompt will appear as shown in Figure 3-9; click **OK**, and optionally check the check box **Always enable capabilities and don't ask me again**.

Archived

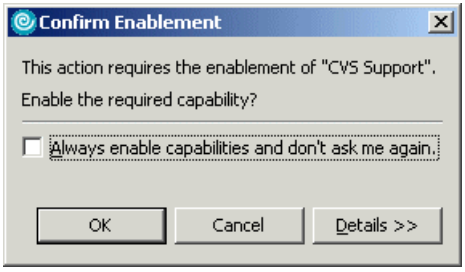


Figure 3-9 Confirm capability enablement

This will enable the CVS repository exploring capability and switch the user to the CVS repository exploring perspective.

3.2.4 File associations

The File Associations preferences page (Figure 3-10) enables you to add or remove file types recognized by the Workbench. You can also associate editors or external programs with file types in the file types list.

Archived

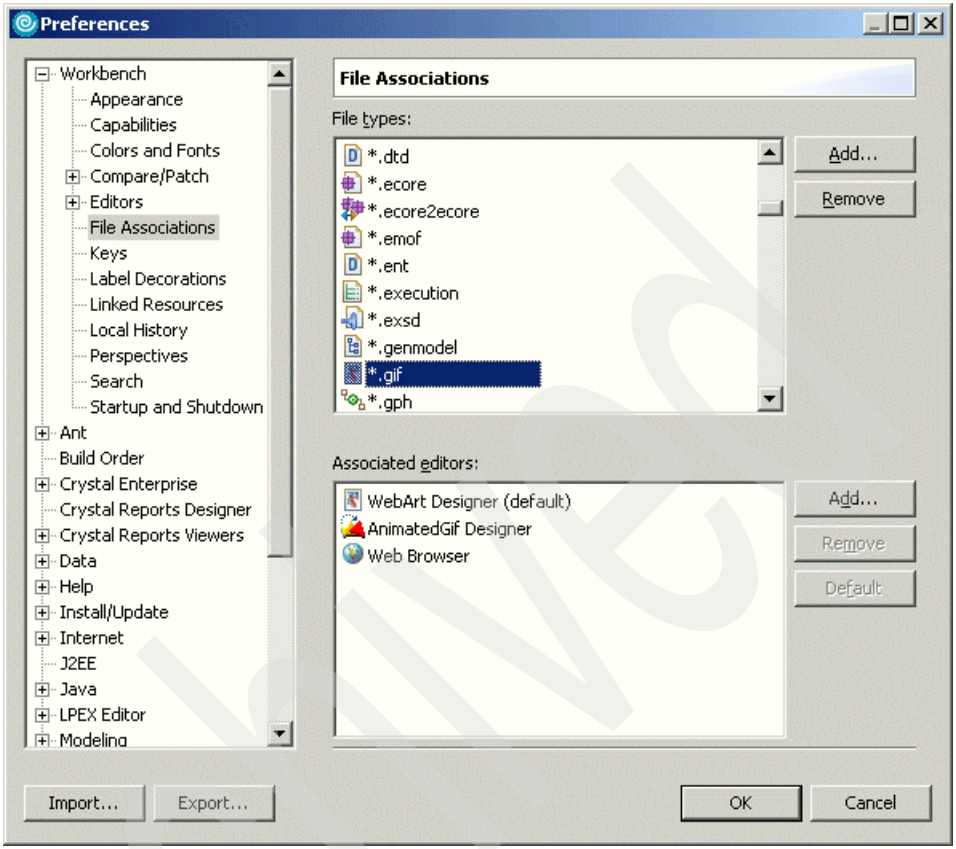


Figure 3-10 File associations preferences

The top right pane allows you to add and remove the file types. The bottom right pane allows you to add or remove the associated editors.

If you want to add a file association, do the following:

1. For an example, we will add the Internet Explorer as an additional program to open .gif files. Select ***.gif** from the file types list and click **Add** next to the associated editors pane.
2. A new dialog opens (Figure 3-11 on page 92) where you have to select the **External Programs** option, then click **Browse**.
3. Select **iexplore.exe** from the folder where Internet Explorer is installed (for example, C:\WINDOWS\ServicePackFiles\i386) and confirm the dialog with **Open**.

Archived

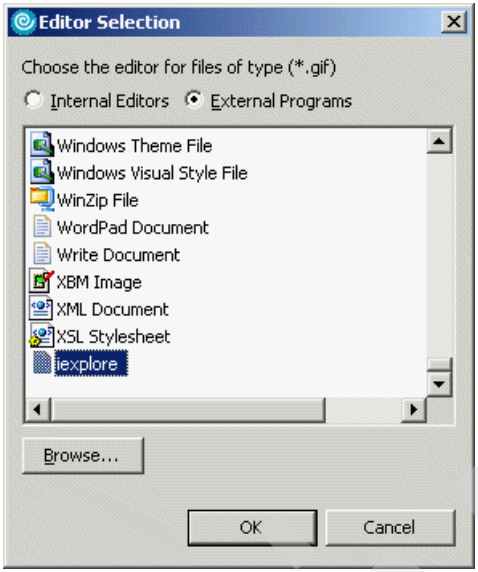


Figure 3-11 File association editor selection

4. Click **OK** to confirm the Editor Selection dialog, and you will notice that the program has been added to the editors list.

> **Note:** Optionally, you can set this program as the default program for this file type by clicking **Default**.

5. Now you can open the file by using the context menu on the file and selecting **Open With**, and selecting the appropriate program.

3.2.5 Local history

A local edit history of a file is maintained when you create or modify a file. A copy is saved each time you edit and save the file. This allows you to replace the current file with a previous edit or even restore a deleted file. You can also compare the content of all the local edits. Each edit in the local history is uniquely represented by the data and time the file has been saved.

> **Note:** Only files have local history. Projects and folders do not have a local history.

To compare a file with the local history, do the following:

1. This assumes you have a Java file. If you do not, add or create a file.

Archived

2. Select the file, right-click, and select **Compare With** → **Local History** from its context menu.

To replace a file with an edit from the local history, do the following:

1. This assumes you have a Java file. If you do not, add or create a file.
2. Select the file, right-click, and select **Replace With** → **Local History** from its context menu.
3. Select the desired file time stamp and then click **Replace**.

To configure local history settings, select **Window** → **Preferences** → **Workbench** → **Local History** to open its preferences page (see Figure 3-12).

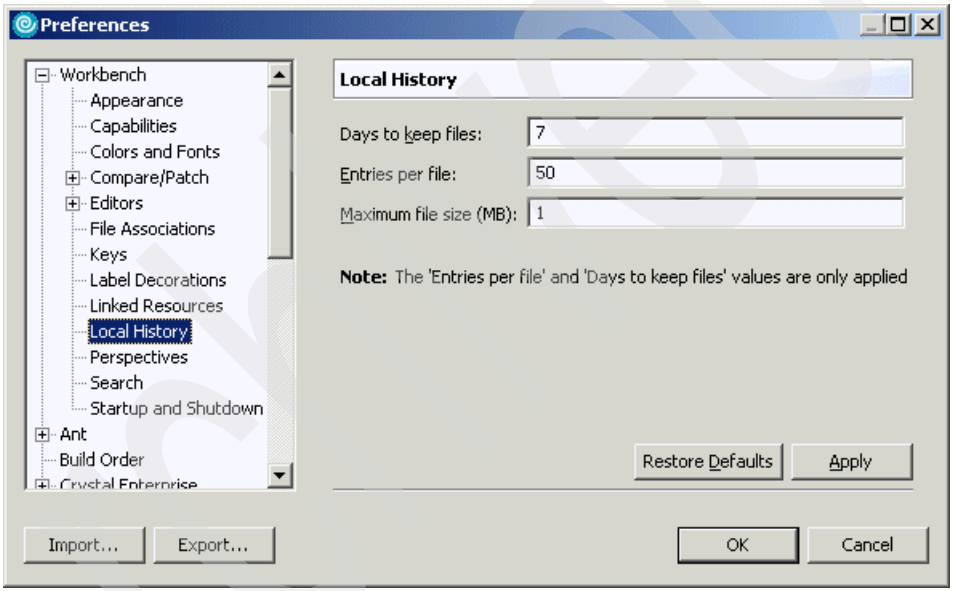


Figure 3-12 Local history preferences

Table 3-3 explains the options for the local history preferences.

Table 3-3 Local history settings

Option	Description
Days to keep files	Indicates for how many days you want to maintain changes in the local history. History state older than this value will be lost.
Entries per file	This option indicates how many history states per file you want to maintain in the local history. If you exceed this value, you will lose older history to make room for new history.

Option	Description
Maximum file size (MB)	Indicates the maximum size of individual states in the history store. If a file is over this size, it will not be stored.

3.2.6 Perspectives preferences

The Perspectives preferences page enables you to manage the various perspectives defined in the Workbench. To open the page, select **Window** → **Preferences** → **Workbench** → **Perspectives** (see Figure 3-13).

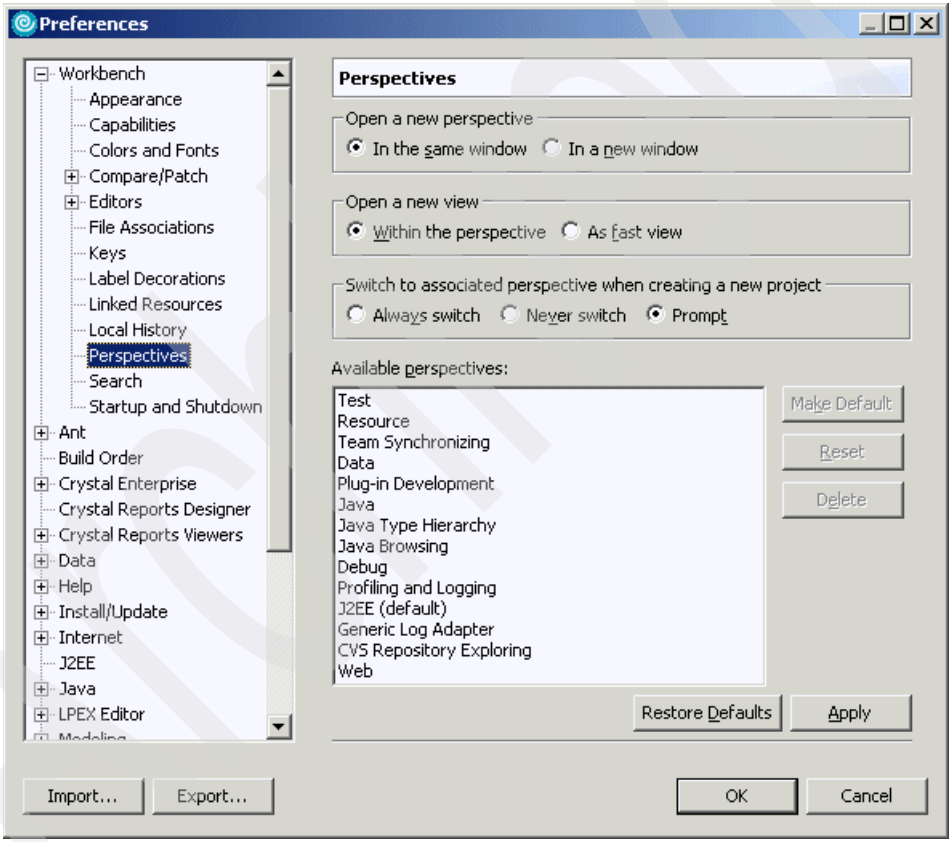


Figure 3-13 Perspectives preferences

Here you can change the following options:

- Open a new perspective in the same or in a new window.
- Open a new view within the perspective or as a fast view (docked to the side of the current perspective).

- The option to always switch, never switch, or prompt when a particular project is created to the appropriate perspective.

There is also a list with all available perspectives where you can select the default perspective. If you have added one or more customized perspectives you can delete them here if you want to.

3.2.7 Internet preferences

The Internet preferences in Rational Application Developer have four types of settings available to be configured:

- FTP
- Proxy settings
- TCP/IP monitor
- Web browser

Only proxy settings and Web browser settings will be covered, with the other two settings left for the reader to investigate using Rational Application Developer's extensive help facility.

Proxy settings

When using Rational Application Developer and working within an intranet, you may want to use a proxy server to get across the firewall to access the Internet.

To set the preferences for the HTTP proxy server within the Workbench to allow Internet access from Rational Application Developer, do the following:

1. From the Workbench select **Window** → **Preferences**.
2. Select **Internet** → **Proxy Settings** (see Figure 3-14 on page 96).
3. Check **Enable Proxy**, and enter the proxy host and port.

   There are additional optional settings for the use of SOCKS and enabling proxy authentication.
4. Click **Apply** and then **OK**.

Archived

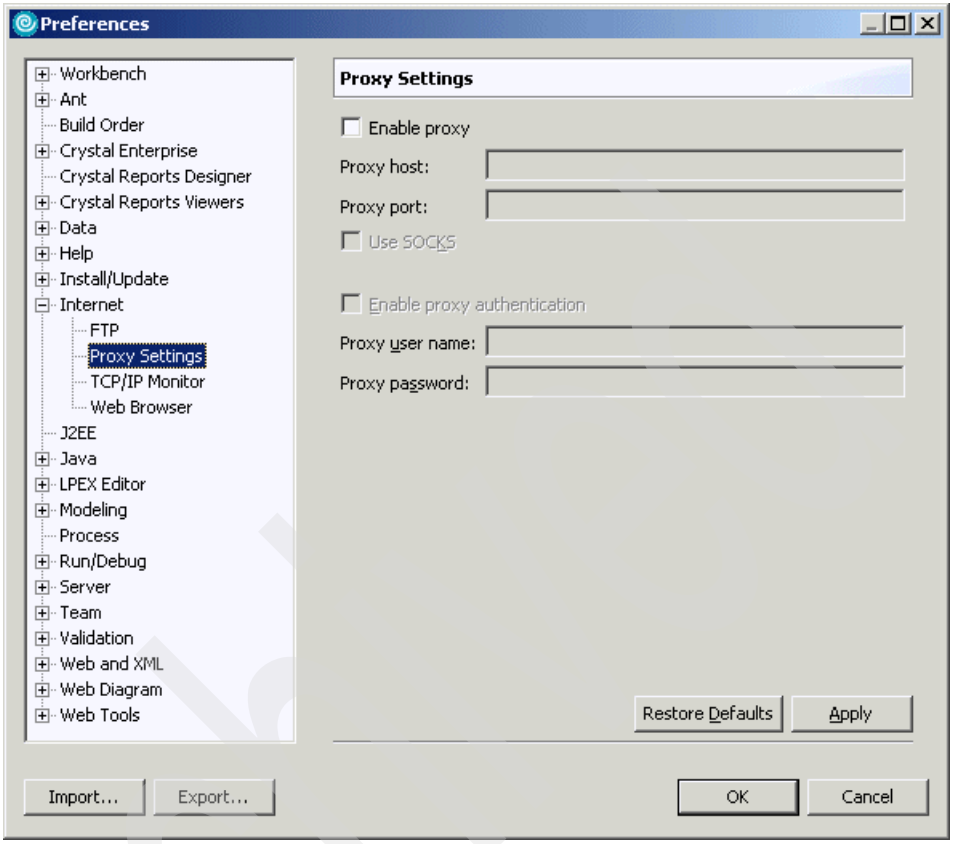


Figure 3-14 Internet proxy settings preferences

Web browser settings

The Web browser settings allow the user to select which Web browser will be the default browser used by Rational Application Developer for displaying Web information.

To change the Web browser settings, do the following:

1. From the Workbench select **Window** → **Preferences**.
2. Select **Internet** → **Web Browser** (see Figure 3-15 on page 97).

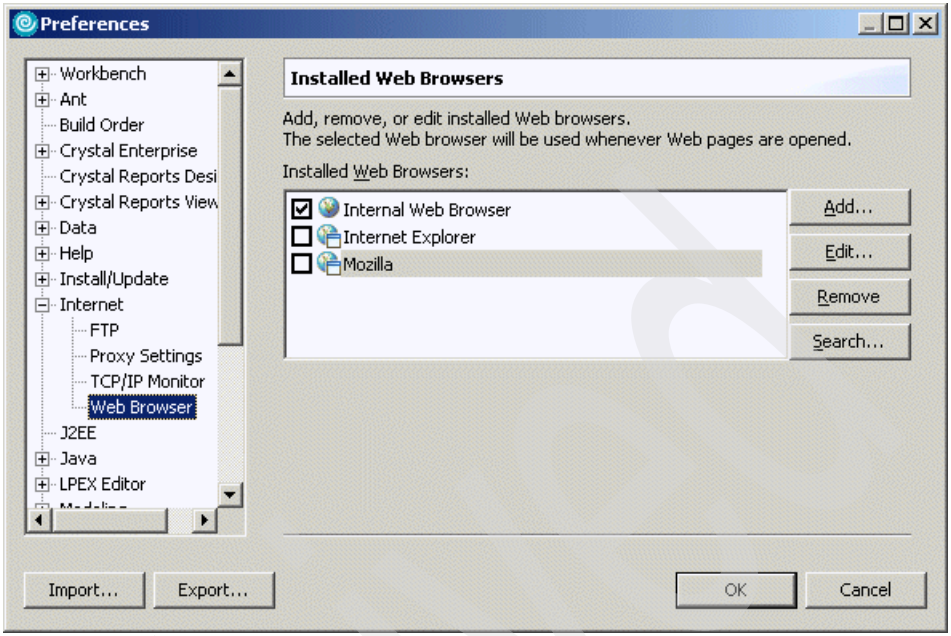


Figure 3-15 Default Web browser preferences

3. The default option is to use the internal Web browser. To change to another browser, select from the available list; otherwise you can click **Add** to add a new Web browser (see Figure 3-16).
4. The user would fill out a name supplied by the user in the Name field, add the location of the browser using the **Browse...** button into the Location field, and add any parameters required in the Parameters field. Click **OK**.

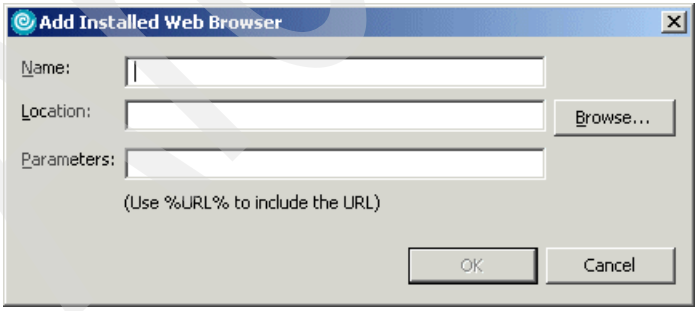


Figure 3-16 Adding a new browser to Rational Application Developer

3.3 Java development preferences

Rational Application Developer provides a number of coding preferences. Some of these are described in this section. Chapter 7, “Develop Java applications” on page 221, also provides information regarding these preferences and Java coding.

3.3.1 Java classpath variables

Rational Application Developer provides a number of default classpath variables that can be used in a Java build path to avoid a direct reference to the local file system in a project. This method ensures that the project only references classpaths using the variable names and not specific local file system directories or paths. This is a good programming methodology when developing within a team and using multiple projects using the same variable. This means that all team members need to do is set the variables required that are defined for the project, and this will be maintained in the workspace.

Tip: We recommend that you standardize the Rational Application Developer installation path for your development team. We found that many files within the projects have absolute paths based on the Rational Application Developer installation path, thus when you import projects from a team repository such as CVS or ClearCase you will get many errors even when using classpath variables.

Depending on the type of Java coding you plan to do, you may have to add variables pointing to other code libraries. For example, this can be driver classes to access relational databases or locally developed code that you would like to reuse in other projects.

Once you have created a Java project, you can add any of these variables to the project’s classpath. Chapter 7, “Develop Java applications” on page 221, provides more information on adding classpath variables to a Java project.

To view and change the default classpath variables:

1. From the Workbench select **Window** → **Preferences**.
2. Select **Java** → **Build Path** → **Classpath Variables** from the list.

   A list of the existing classpath variables is displayed, as shown in Figure 3-17 on page 99.

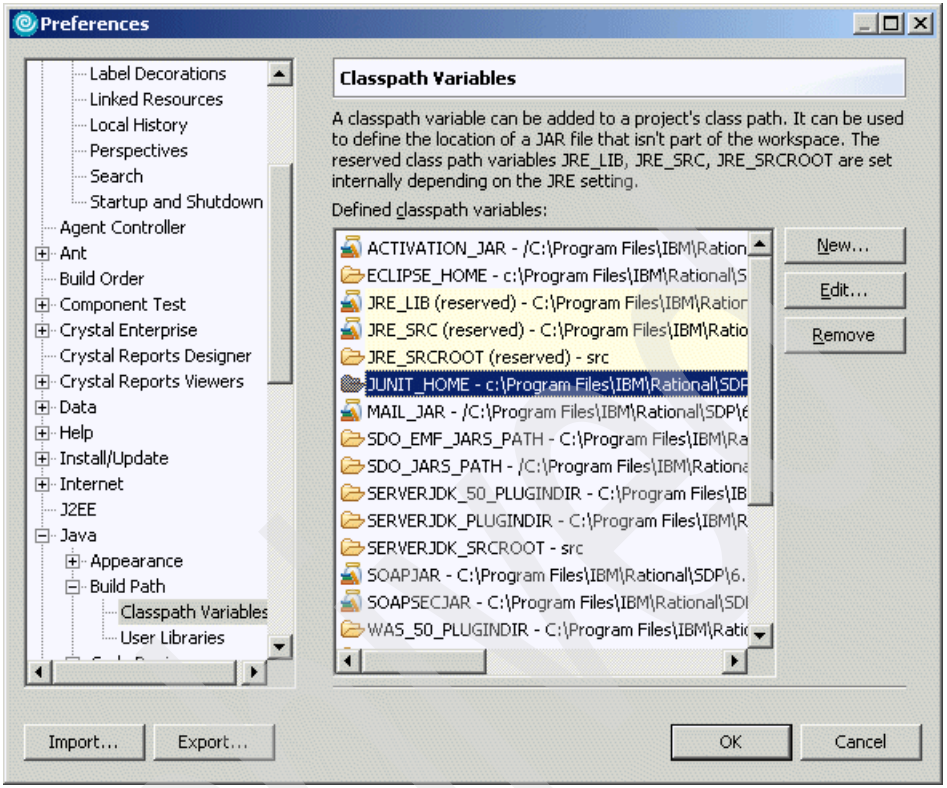


Figure 3-17 Classpath variables preferences

3. Creation, editing, or removing of variables can be performed on this screen. Click **New** to add a new variable. A dialog appears prompting for the variable to create. For example, for an application using DB2 you would enter the information as shown in Figure 3-18, and then click **OK**.

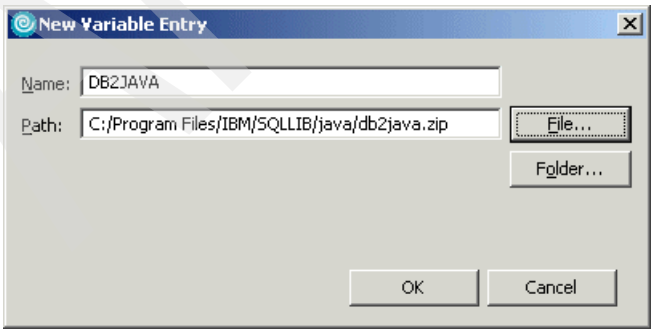


Figure 3-18 New Variable Entry dialog

3.3.2 Appearance of Java elements

The appearance and the settings of associated Java elements, such as methods, members, and their access types, can be adjusted for display in the Java viewers. Some of the views it impacts on are the package and outline explorer views. To adjust the appearance of Java elements in the viewers, the following would be performed:

1. From the Workbench select **Window** → **Preferences**.
2. Select **Java** → **Appearance** from the list.

   A window like Figure 3-19 on page 101 will be displayed with appearance check boxes as described in Table 3-4.

Table 3-4 Description of appearance settings for Java views

Appearance setting	Description
Show method return types	The views will display the return type of the methods for a class.
Show override indicators in outline and hierarchy	Displays an indicator for overridden and implemented methods in the Outline and Type Hierarchy views.
Show members in Package Explorer	Displays the members of the class and their scope such as private, private or protected, including others.
Compress package name segments	Compresses the name of the package based on a pattern supplied in the dialog below the check box.
Stack views vertically in the Java Browsing perspective	Displays the views in the Java Browsing perspective vertically rather than horizontally.

Archive

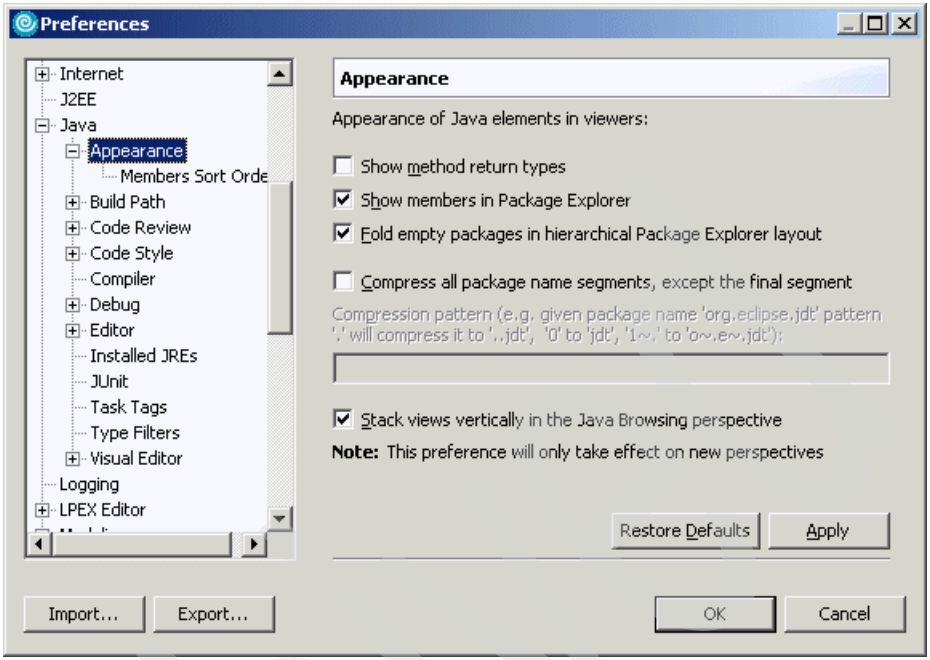


Figure 3-19 Java appearance settings

To change the appearance of the order of the members to be displayed in the Java viewers, do the following:

1. From the Workbench select **Window** → **Preferences**.
2. Select **Java** → **Appearance** → **Members Sort Order**.

   This preference allows you to display the members in the order you prefer as well as the order of scoping within that type of member.
3. When done selecting preferences, click **Apply** and then **OK**.

3.3.3 Code style and formatting

The Java editor in the Workbench can be configured to format code and coding style in conformance to personal preferences or team-defined standards. When setting up the Workbench you can decide what formatting style should be applied to the Java files created using the wizards, as well as how the Java editors operate to assist what has been defined.

Important: Working in a team environment requires a common understanding between all the members of the team of the style of coding and conventions such as class, member, and method name definitions. The coding standards need to be documented and agreed upon to ensure a consistent and standardized method of operation.

There are many books that can be found to provide guidance on Java coding standards. A good reference available for download is the *Writing Robust Java Code* white paper by Scott Ambler found at:

`http://www.ambysoft.com/javaCodingStandards.pdf`

Code style

The Java code style provides a facility to define the prefix and postfix style for member names, parameters, and local variables for use in your Java class files.

To demonstrate setting up a style, suppose we define a style in which the following style will be defined:

- Member attributes or fields will be prefixed by an `m`.
- Static attributes or fields will be prefixed by an `s`.
- Parameters of methods will be prefixed by a `p`.
- Local variables can have any name.
- Boolean getter methods will have a prefix of `is`.

Rational Application Developer would be able to assist in providing this style by performing the following:

1. From the Workbench, select **Windows** → **Preferences**.
2. Select **Java** → **Code Style** as shown in Figure 3-20 on page 103.

Archive

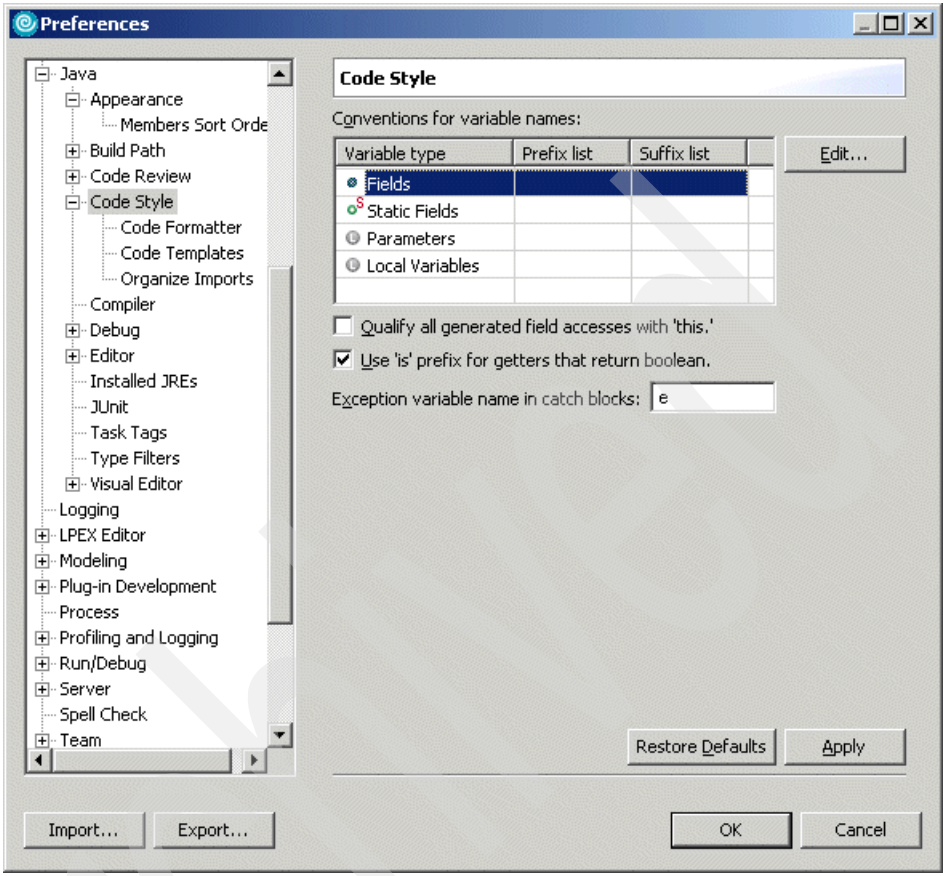


Figure 3-20 Code style preferences page

3. Select the **Fields** column and click **Edit**.
4. Fill in the information as shown in Figure 3-21 on page 104, as defined for the scenario, and click **OK**.

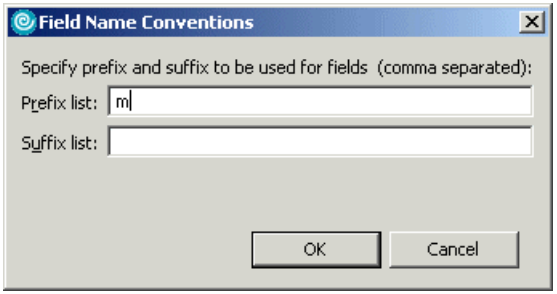


Figure 3-21 Filling in the prefix and suffix list for field name conventions

5. Repeat for all the conventions defined to produce the result shown in Figure 3-22, and click **OK**.

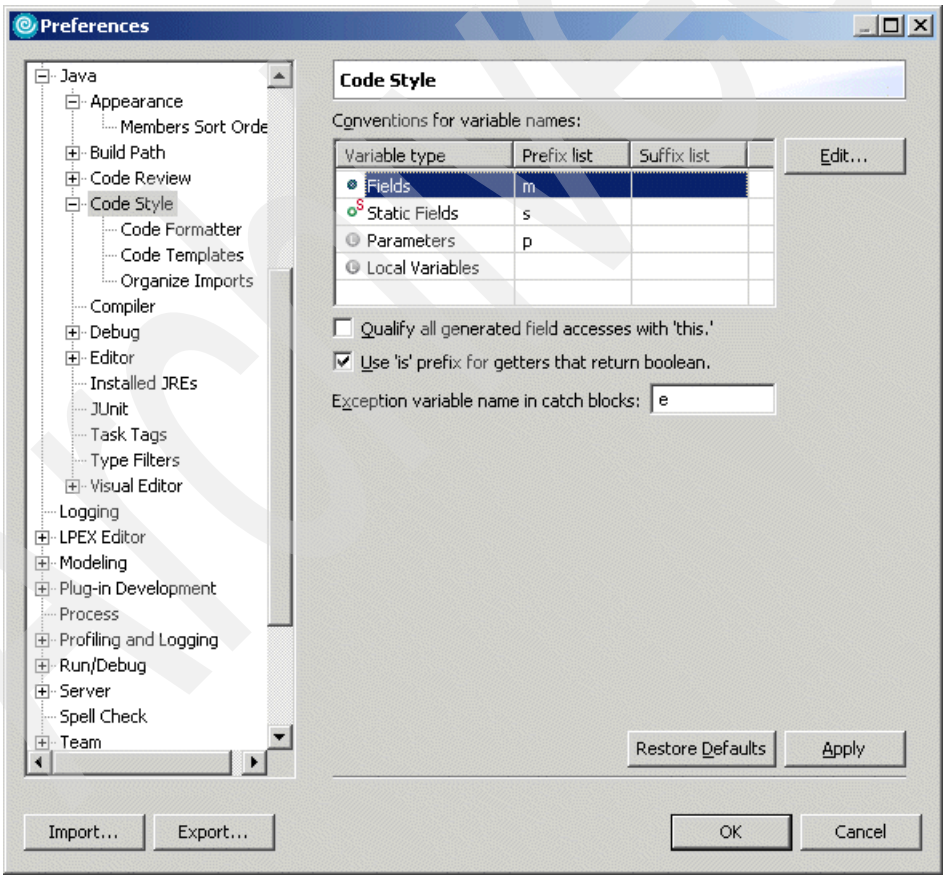


Figure 3-22 Completed code style definition preferences

These coding conventions will be used in the generation of setters and getters for Java classes. Whenever a prefix of `m` followed by a capital letter is found on an attribute, this would ignore the prefix and generate a getter and setter without the prefix. If the prefix is not found followed by a capitalized letter, then the setter and getter would be generated with the first letter capitalized followed by the rest of the name of the attribute. An example of the outcome of performing code generation of a getter is shown in Example 3-1 for some common examples of attributes.

Note: The capitalization of getters in IBM Rational Application Developer V6.0 is based on the way the attributes are named.

Example 3-1 Example snippet of code generation output for getters

```
private long mCounter;
 private String maddress;
 private float m_salary;
 private int zipcode;
 /**
  * @return Returns the counter.
  */
 public long getCounter() {
     return mCounter;
 }
 /**
  * @return Returns the maddress.
  */
 public String getMaddress() {
     return maddress;
 }
 /**
  * @return Returns the m_salary.
  */
 public float getM_salary() {
     return m_salary;
 }
 /**
  * @return Returns the zipcode.
  */
 public int getZipcode() {
     return zipcode;
 }
```

Code formatter

The code formatter preferences in Rational Application Developer are used to ensure that the format of the Java code meets the standard defined for a team of

developers working on code. There are two profiles built-in with IBM Rational Application Developer V6.0 with the option of creating new profiles specific for your project.

- Eclipse 2.1 (similar to WebSphere Studio V5)
- Java Conventions (default profile on startup for IBM Rational Application Developer V6.0)

The code formatting rules are enforced on code when a developer has typed in code using their style, and does not conform to the team-defined standard. When this is the case, after the code has been written and tested and preferably before check into the Team repository, it is recommended that it be formatted. The procedure to perform this operation would be as follows:

1. In the editor that the Java code is in, right-click in the window.
2. Select **Source** → **Format**, and this will use the rules that have been defined.

The code formatter preferences/definitions can be accessed via the following:

1. From the Workbench, select **Window** → **Preferences**.
2. Select **Java** → **Code Style** → **Code Formatter**.
3. To display the information about a profile, click **Show...**, or to create a new profile click **New...**, as shown in Example 3-23 on page 107.
4. Loading of a preexisting XML-defined format can be loaded via the Import button that was defined within IBM Rational Application Developer V6.0.

Archived

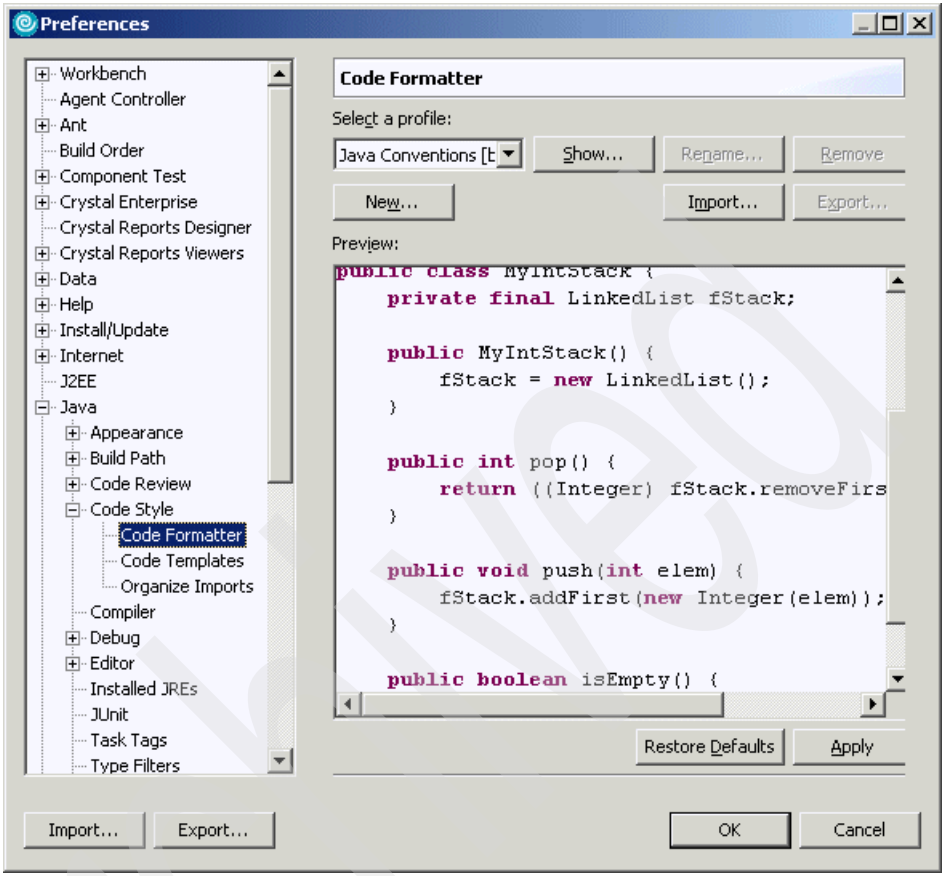


Figure 3-23 Code formatter profiles setup in IBM Rational Application Developer V6

User-defined profiles are established from one of the existing built-in profiles, which can be exported to the file system to share with team members. In Rational Application Developer V6.0, you can modify or create a new profile from either a built-in profile or import a custom profile from the file system. If the existing profile is modified then you will be prompted to create a new profile. Any profile that is created can be exported as an XML file that contains the standardized definitions required for your project. This can then be stored in a team repository and imported by each member of the team.

A profile consists of a number of sections that are provided as tab sections to standardize on the format and style that the Java code is written. Each of these tab sections are self-explanatory and provide a preview of the code after

selection of the required format. Each of these tabs is listed with a brief description below:

- Indentations

  Specifies the indentations that you wish on your Java code in the Workbench, as shown in Figure 3-24. The area it covers includes:

  – Tab spacing
  – Alignment of fields
  – Indentation of code

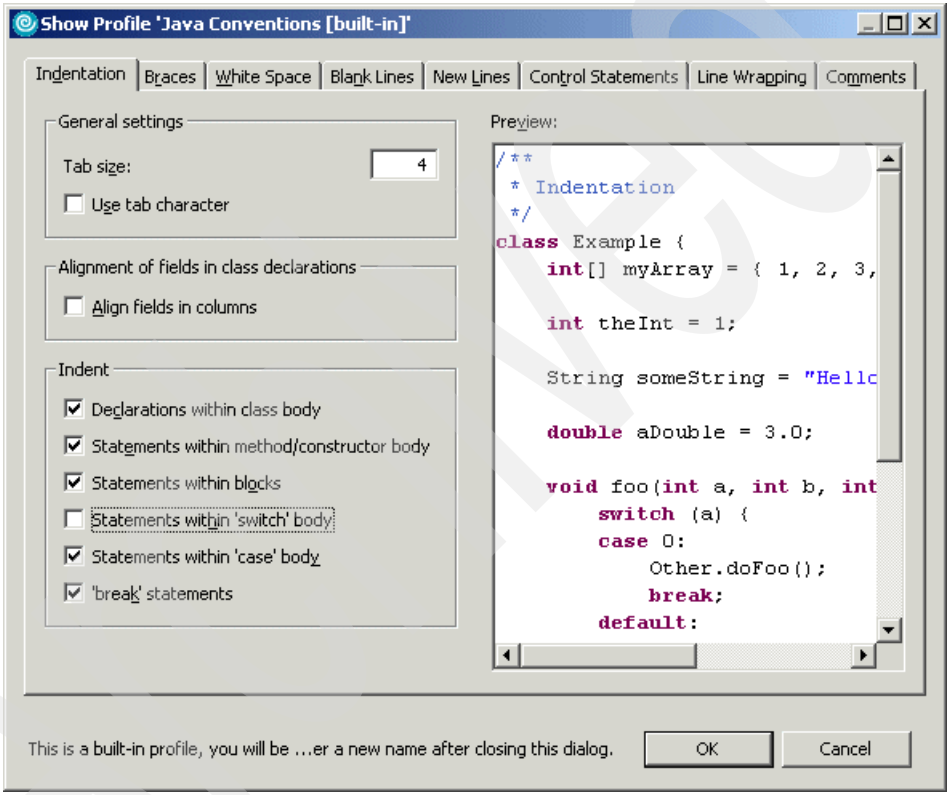


Figure 3-24 Code formatting - Indentations tab

- Braces

  The braces tab formats the Java style of where braces will be placed for a number of Java language concepts. A preview is provided as you check the check boxes to ensure that it fits in with the guidelines established in your team. The options are displayed in Figure 3-25 on page 109.

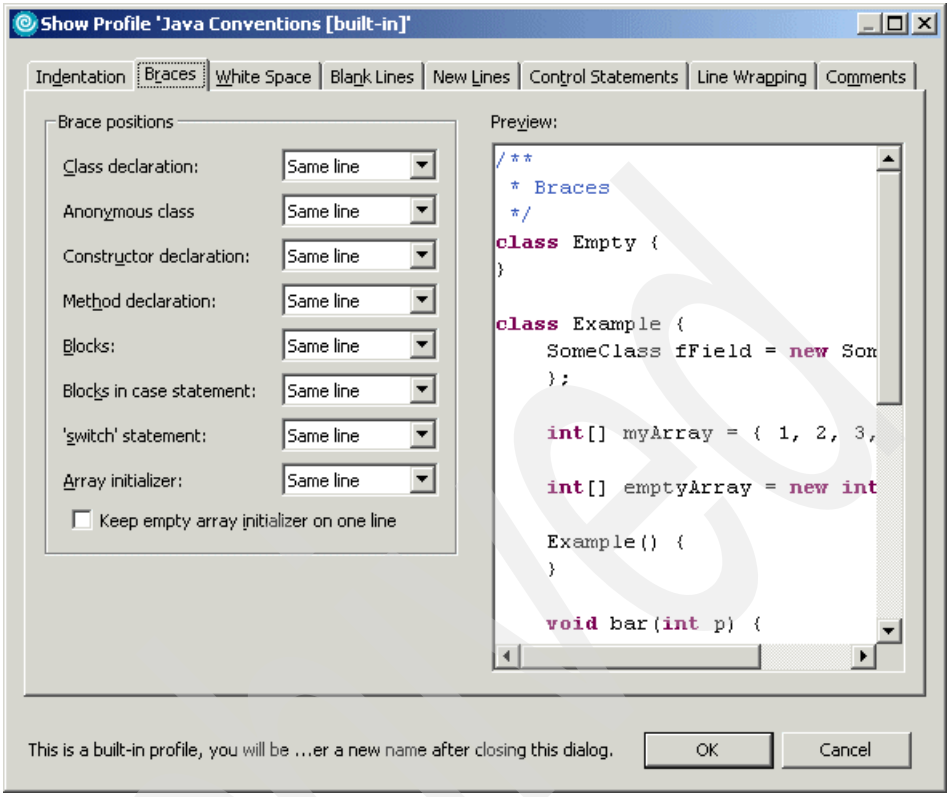


Figure 3-25 Code formatting - Braces tab

- White Space

  The White Space tab is used to format where the spaces are placed in the code based on a number of Java constructs. The screen shot in Figure 3-26 on page 110 provides some of the options that are available.

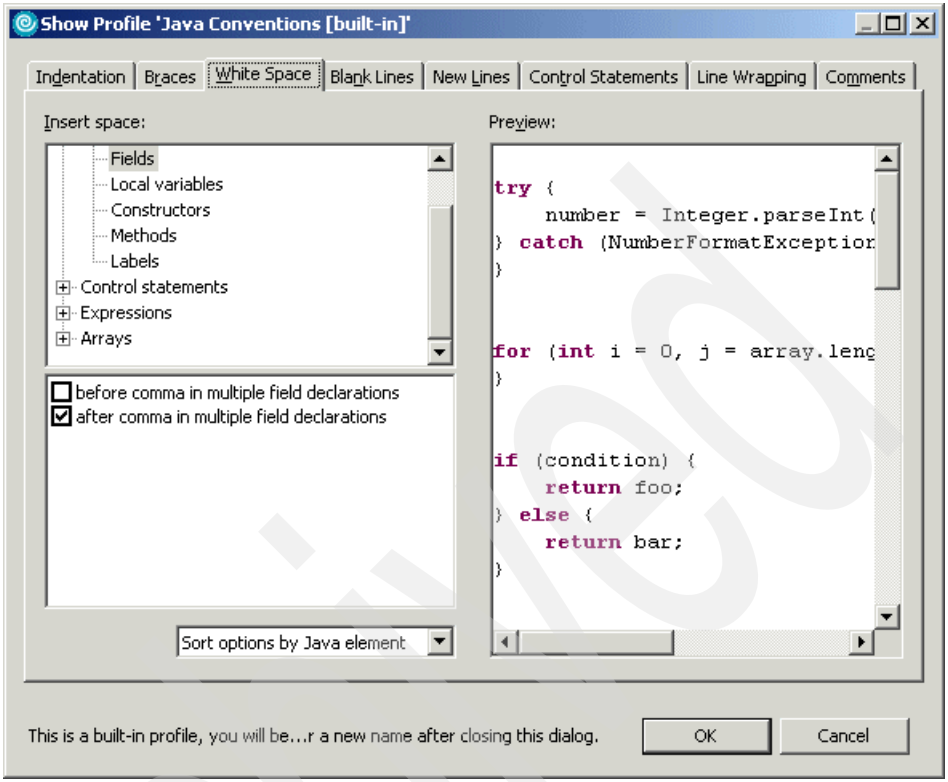


Figure 3-26 Code formatting - White Space tab

- Blank Lines

  The Blank Lines tab is used to specify where you wish to place blank lines in the code for readability or style guidelines. The screen shown in Figure 3-27 on page 111 provides a view of what is configurable.

Archive

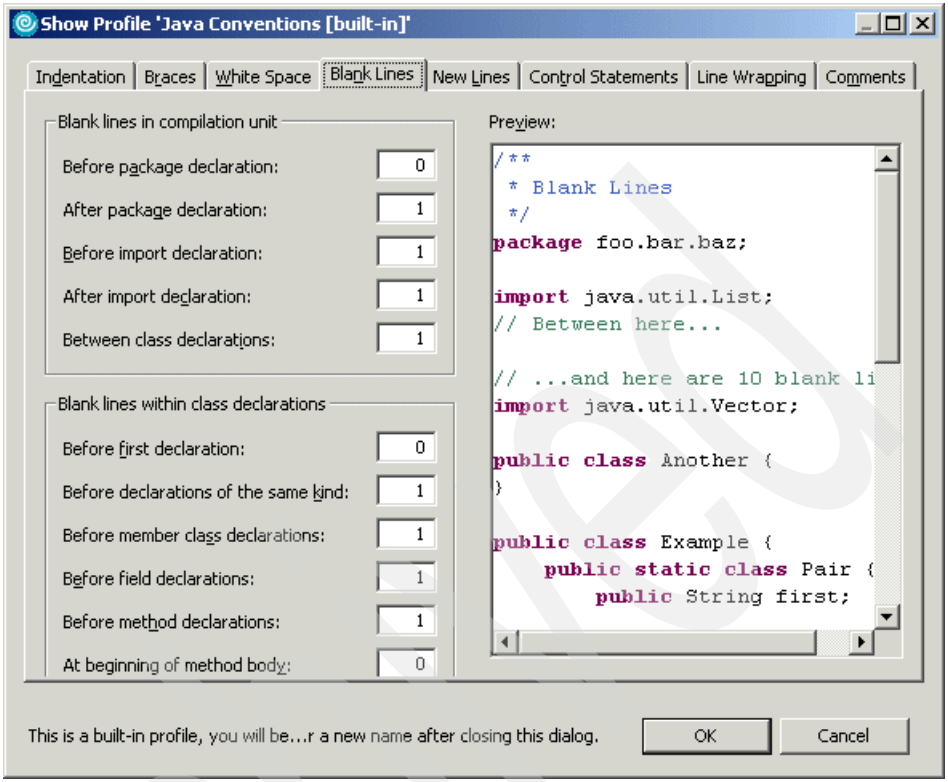


Figure 3-27 Code formatting - Blank Lines

- New Lines

  The New Lines tab specifies the option of where you wish to insert a new line, as shown in Figure 3-28 on page 112.

Archive

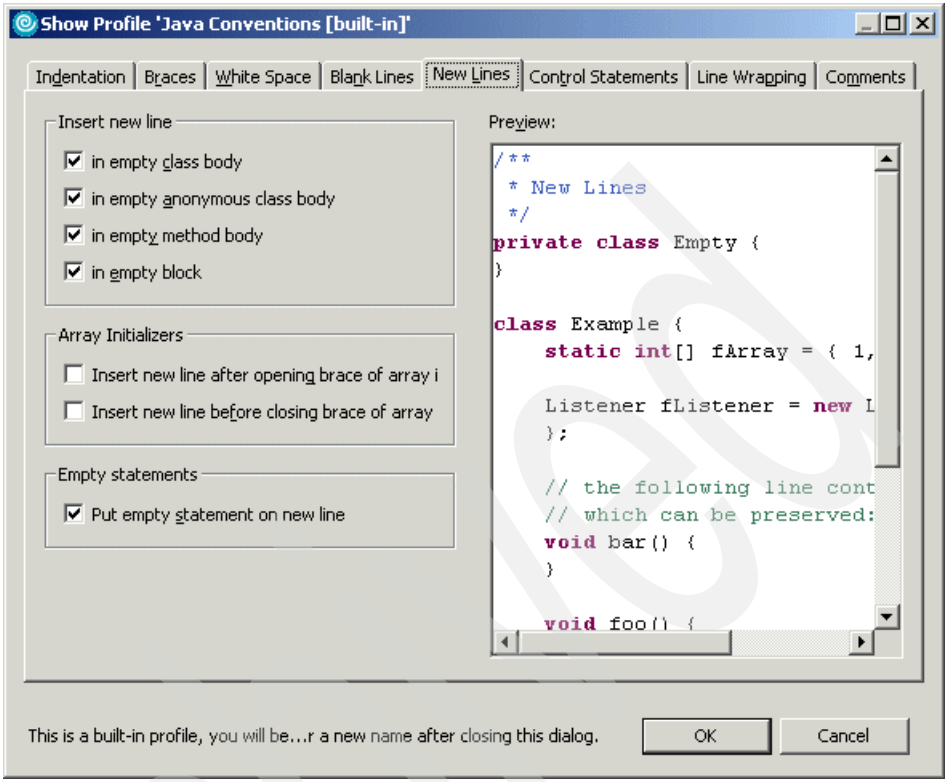


Figure 3-28 Code formatting - New Lines tab

- Control Statements

  The Control Statements tab is used to control the insertion of lines in control statements, as well as the appearance of *if else* statements of the Java code, as shown in Figure 3-29 on page 113.

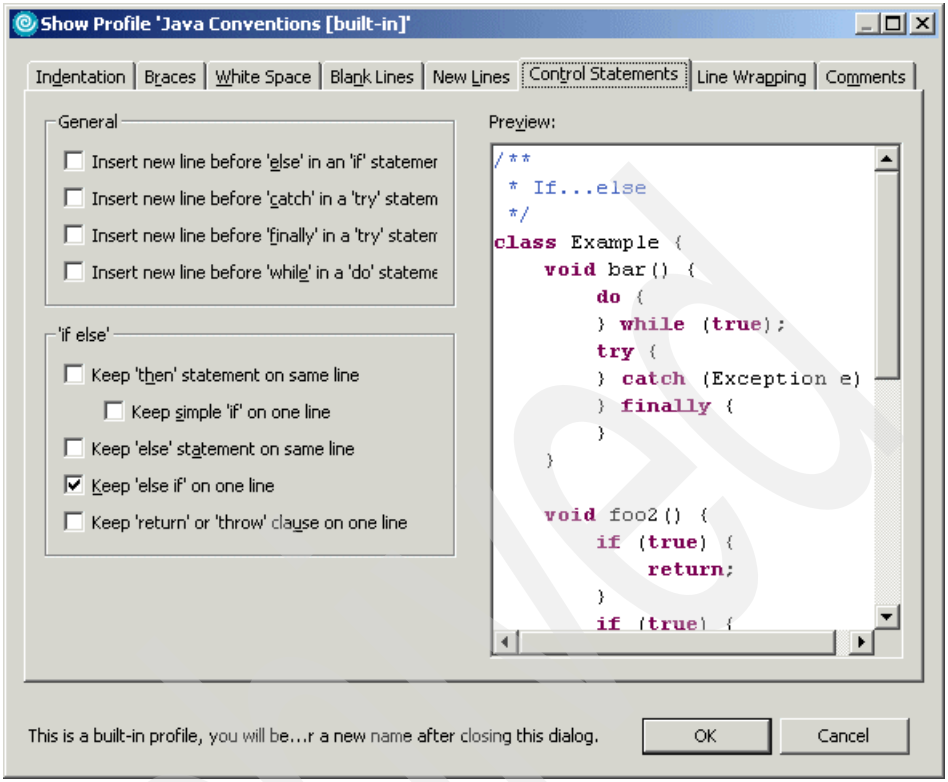


Figure 3-29 Code formatting - Control Statements

- Line Wrapping

  The Line Wrapping tab facility provides the style rule on what should be performed with the wrapping of code, as shown in Figure 3-30 on page 114.

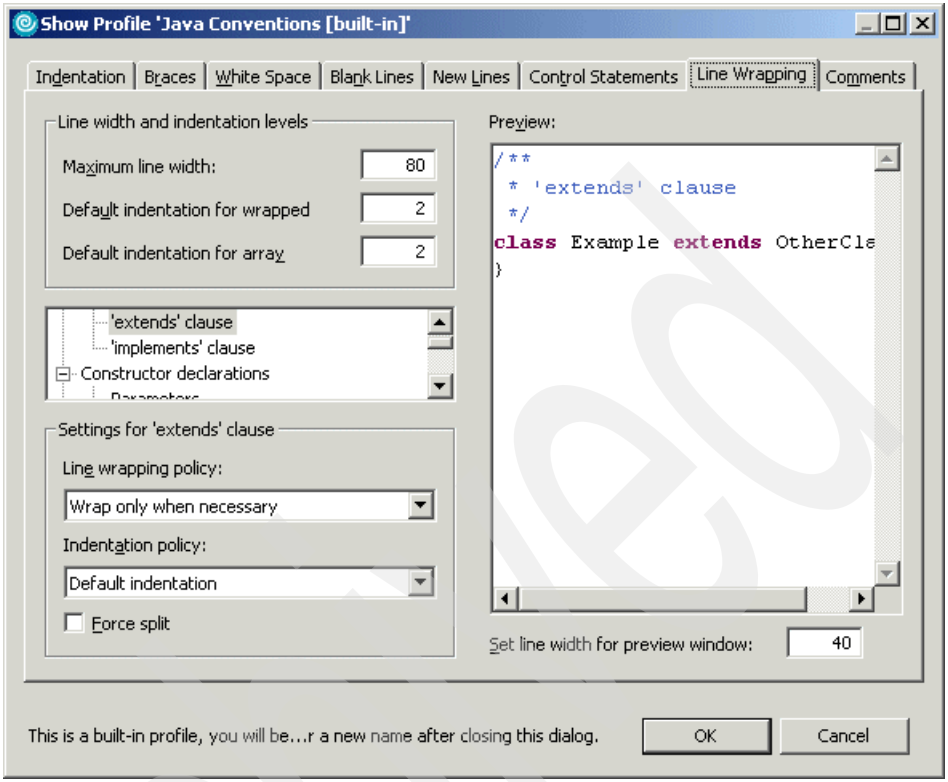


Figure 3-30 Code formatting - Line Wrapping

- Comments

  The Comments tab is used to determine the rules of the look of the comments that are in the Java code. The settings that are possible are shown in Figure 3-31 on page 115.

Archived

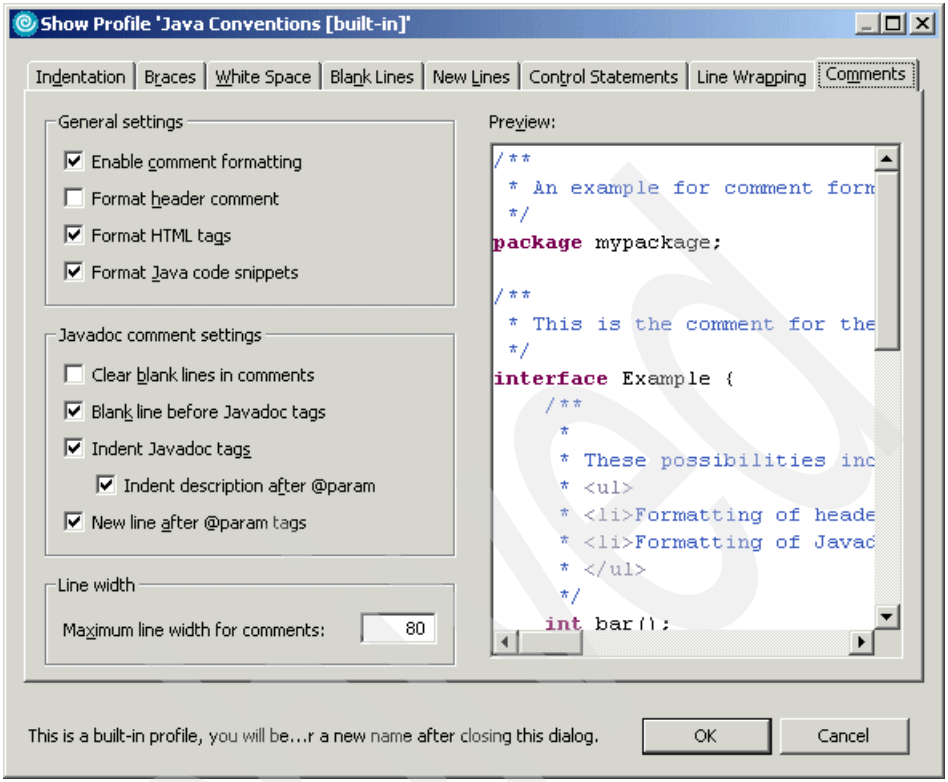


Figure 3-31 Code formatting - Comments

3.3.4 Compiler options

Problems detected by the compiler are classified as either warnings or errors. The existence of a warning does not affect the execution of the program. The code executes as if it had been written correctly. Compile-time errors (as specified by the Java Language Specification) are always reported as errors by the Java compiler.

For some other types of problems you can, however, specify if you want the Java compiler to report them as warnings, errors, or to ignore them. To change the default settings you need to perform the following:

1. From the Workbench, select **Window** → **Preferences**.
2. Select **Java** → **Compiler** (see Figure 3-32 on page 117).
3. With each of the tabs select set the appropriate behavior required to ensure that you obtain the required information from the compiler.

The tab options include:

- Style

  Defines the severity level for the compiler for a number of scenarios.

- Advanced

  Defines the severity level the compiler will identify for advanced error checking.

- Unused code

  Configuration settings to assist in common problems that occur in Java development, such as local variables defined and not used and how to display these problems.

- Javadoc

  Configuration settings on how to deal with Javadoc problems that may arise and what to display as errors.

- Compliance and Classfiles

  Enforces a particular JDK compliance level. This is important to do if your application has to fulfil a minimum JDK level.

- Build Path

  Specifies the error condition in the event of a build path error and how to display this error.

Archived

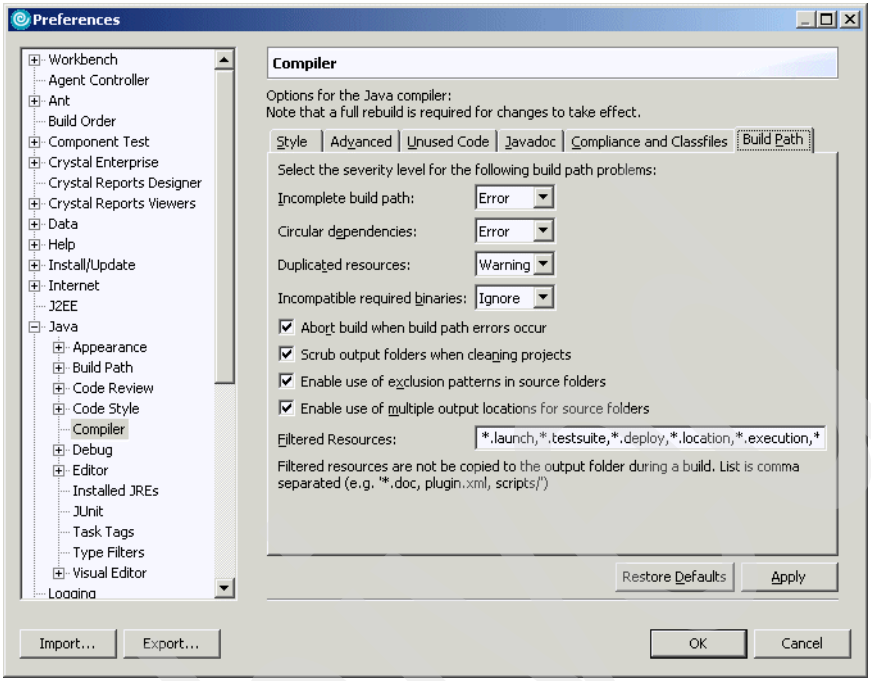


Figure 3-32 Java compiler preferences dialog

Note: The Java compiler can create .class files even in the presence of compilation errors. In the case of serious errors (for example, references to inconsistent binaries, most likely related to an invalid build path), the Java builder does not produce any .class file.

3.3.5 Java editor settings

The Java editor has a number of settings that assist in the productivity of the user in IBM Rational Application Developer V6. Most of these options relate to the look and feel of the Java editor in the Workbench. Preferences can be accessed via the following procedure:

1. From the Workbench, select **Windows** → **Preferences**.
2. Select **Java** → **Editor** and you will be presented with the following tab options to configure:
   - Appearance: The look and feel of the colors and editor features such as line numbers.
   - Syntax: The color options for syntax highlighting for the Java language.

- Typing: Options to assist the user in completion of common tasks, such as closing of brackets.
- Hovers: Options to enable hover assists to view, such as variable selection assistance.
- Navigation: Identifies the carat positioning navigation and keyboard shortcut for hyper text opening.
- Folding: Options to enable and disable the collapsible sections in the Java editor. This enables a user of the Workbench to collapse a method, comments, or class into a concise single line view.

Details on each of the options within these tabs can be obtained from the Rational Application Developer online help.

Code Assist

The Code Assist feature in IBM Rational Application Developer V6 is used in assisting a developer in writing their code rapidly and reducing the errors in what they are writing. It is a feature that is triggered by pressing the keys Ctrl and the space bar simultaneously, and assists the developer in completion of a variable or method name when coding. These features are configured as follows:

1. From the Workbench, select **Windows** → **Preferences**.
2. Select **Java** → **Editor** → **Code Assist**.

   This window provides the configuration settings for the code assist functionality in the editor. A description of each setting can be found in the Rational Application Developer online help.

Mark occurrences

The mark occurrences preferences can be configured as follows:

1. From the Workbench, select **Windows** → **Preferences**.
2. Select **Java** → **Editor** → **Mark Occurrences**.

   When enabled, this highlights all occurrences of the types of entities described in the screen, as shown in Figure 3-33 on page 119. In the Java editor, by selecting an attribute, for example, the editor will display in the far right-hand context bar of the editor all occurrences in the resource of that attribute. This can be navigated to by selecting the highlight bar in that context bar.

Redbooks

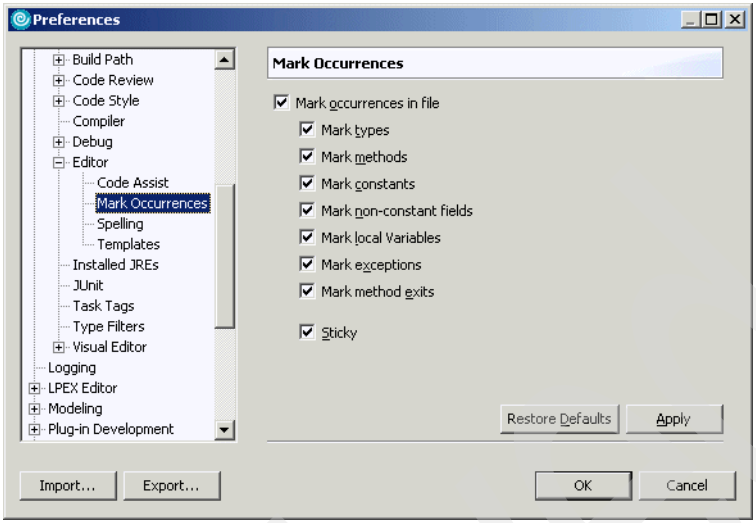


Figure 3-33 Mark Occurrences preferences for the Java editor

3.3.6 Installed JREs

IBM Rational Application Developer V6 allows you to specify which Java Runtime Environment (JRE) should be used by the Java builder. By default the standard Java VM that comes with the product is used; however, to ensure that you application is targeted for the correct platform, the same JRE or at least the same version of the JRE should be used to compile the code. If the application is targeted for a WebSphere Application Server V6, then the JRE should be set to use the JRE associated with this environment in IBM Rational Application Developer V6.

The installed JRE can be changed using the following procedure if you want to point it to the WebSphere Application Server V6 JRE:

1. From the Workbench, select **Windows** → **Preferences**.
2. Select **Java** → **Installed JREs**.

> **Note:** It should be noted that in a new workspace, only the eclipse JRE will appear in the list. The WebSphere JREs will not appear in the list until a test server has been created (and started) for the first time. Once that action takes place, then the full list of available JREs will appear.

Important: A full rebuild is required on setting of the JRE. To initiate, click **Project** → **Clean** followed by **Project** → **Build All**.

By default, the JRE used to run the Workbench will be used to build and run Java programs. It appears with a check mark in the list of installed JREs. If the target JRE that the application will run under is not in the list of JREs, then this can be installed on the machine and added onto the list. You can add, edit, or remove a JRE.

Let us assume that the application you are writing requires the latest JRE 1.4.2_06 located in the directory C:\Program Files\Java\j2re1.4.2_06\. The procedure to add a new JRE using this screen is as follows:

1. Click the **Add** button.
2. Fill out the dialog, as shown in Figure 3-34.
   - JRE type: A drop-down box indicating whether a Standard VM or Standard 1.1.x VM. In most circumstances this will be set to Standard VM.
   - JRE Name: A chosen name for the JRE to identify it.
   - JRE home Directory: The location of the root directory of the install for the JRE.
   - Javadoc URL: The URL for the Javadoc. If located on the disk, this would have the local reference.
   - Default VM arguments: Arguments that are required to be passed to the JRE.
   - Use default system libraries: When checked, will use the libraries associated with the JRE. When unchecked, additional Jar files can be added.

Archived

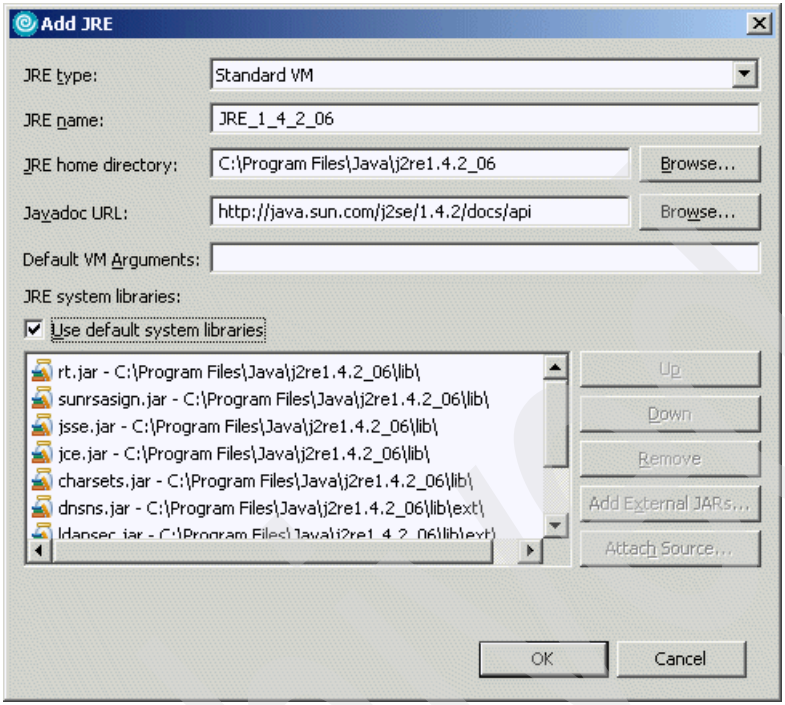


Figure 3-34 Add JRE dialog

3. Click **OK** to add.
4. Select the check box of the JRE to set as the default JRE, press **OK**, and rebuild all Java code in the workspace.

3.3.7 Templates

IBM Rational Application Developer V6 also provides Templates that are often reoccurring source code patterns. The JDT of Application Developer offers support for creating, managing, and using templates.

The templates can be used by typing a part of the statement you want to add; and then by pressing Ctrl + Space bar in the Java editor, a list of templates matching the key will appear in the presented list. For more information on code assist refer to “Code Assist (content)” on page 320. Note that the list is filtered as you type, so typing the few first characters of a template name will reveal it.

The symbol in front of each template, shown Figure 3-35, in the code assist list is colored yellow, so you can distinguish between a template and a Java statement entry.

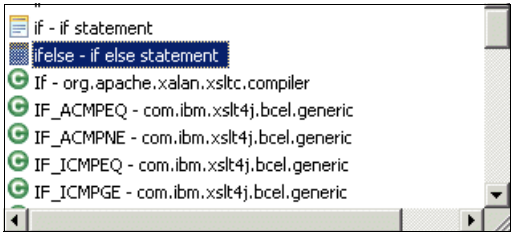


Figure 3-35 Using templates for code assist

The Templates preference page allows you to create new and edit existing templates. A template is a convenience for the programmer to quickly insert often-reoccurring source code patterns. A new template can be created as follows:

1. From the Workbench, select **Windows** → **Preferences**.
2. Select **Java** → **Editor** → **Templates** and the window shown in Figure 3-36 on page 123 is displayed.

Archived

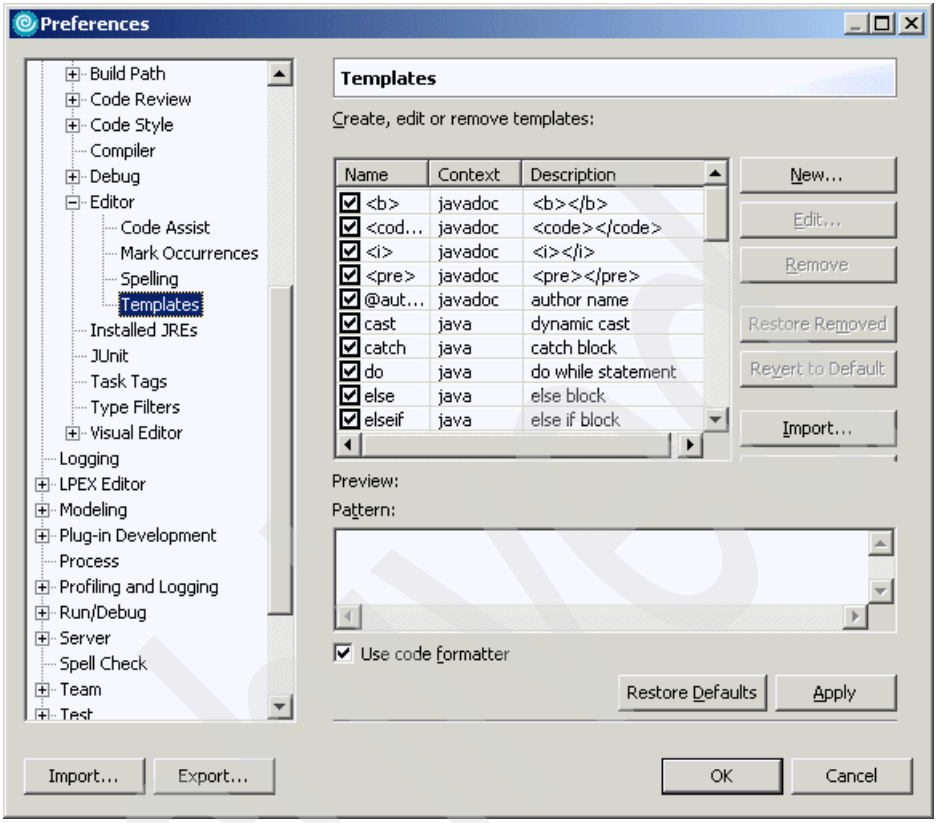


Figure 3-36 Templates preferences

3. Click **New** to add a new template and fill out the information to create a new template for a multi-line if-then-else statement, as shown in Figure 3-37 on page 124, and click **OK** when complete.

> **Note:** Ensure that the context drop-down box is set to either Java or Javadoc, depending on the type of template that this is utilizing.

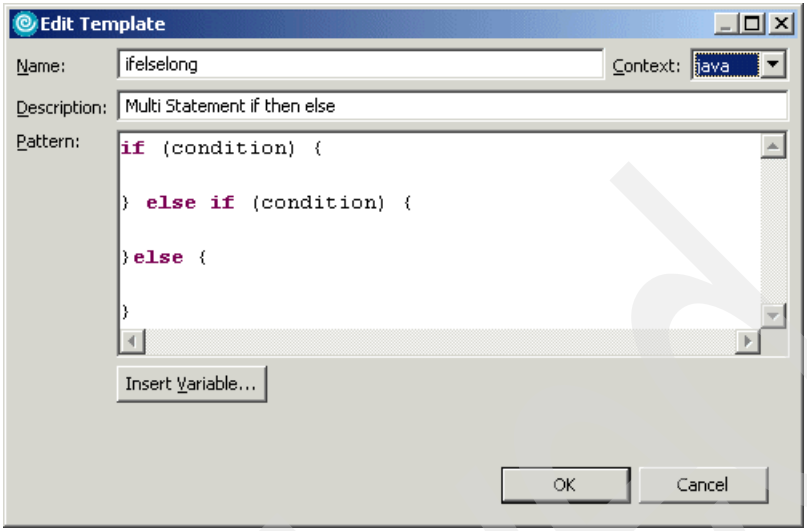


Figure 3-37 Creating a new template

4. Click **OK** to exit the preferences page.

The template will now be available for use in the Java editor.

There are also some predefined variables available that can be added in the template. These variables can be inserted by clicking the **Insert Variable** button in Figure 3-37. This will bring up a list and a brief description of the variable.

Templates that have been defined can be exported and later imported into IBM Rational Application Developer V6 to ensure that a common environment can be set up among a team of developers.

3.3.8 Code review

IBM Rational Application Developer V6 has provided a code review facility to assist in the development of quality code developed in the Workbench. The code review provides the facility to review the code written to ensure that it adheres to common and good coding practices and outputs messages based on these rules.

IBM Rational Application Developer V6 comes with a set of predefined code reviews for a number of scenarios:

- Quick code review

  Consists of three categories of rules: The J2EE Best Practices, J2SE Best Practices and Performance. The purpose of this set is to provide a way to

discover the most problematic code, and does not necessarily reflect fast execution. Rules are applied that check for code with the most serious problems that need to be addressed first.

- Complete code review

  This review contains the full set of rules and rule categories covering a range of severity levels, problems, warnings, and recommendations, and takes the longest to execute.

- J2EE Best Practices code review

  This review contains one review category of J2EE Best Practices, and applies rules to the detection of antipatterns, which are difficult using conventional techniques. These rules only run across servlet code located in Dynamic Web Projects and focus on antipatterns that can cause performance degradation or system outages.

- Globalization code review

  This review consists of one category called Globalization. The purpose of this review is to detect code that is noncompliant with globalization standards and that causes problems with translation and formatting of characters based on assumptions on the bit size of a character.

- J2SE Best Practices code review

  This review consists of one review category called J2SE Best Practices, and applies rules to detect code that can potentially cause problems or be noncompliant with J2SE standards. The rules span a number of categories such as the AWT and Swing interface, as well as rules related to Null.

Each of these type of reviews can be customized to turn off rules within them and have filters added to ignore resources that should not have these rules run against them. A list of customization operations that can be performed for a review are:

- Adding a new user rule
- Changing the severity of the problem
- Filter resources to exclude from a code review
- Filter resources to exclude from a code review for a particular rule

Setting the code review options

The code review can be set within the Workbench using the following procedure:

1. From the Workbench, select **Windows** → **Preferences**.
2. Select **Java** → **Code Review** to display the window in Figure 3-38 on page 126.

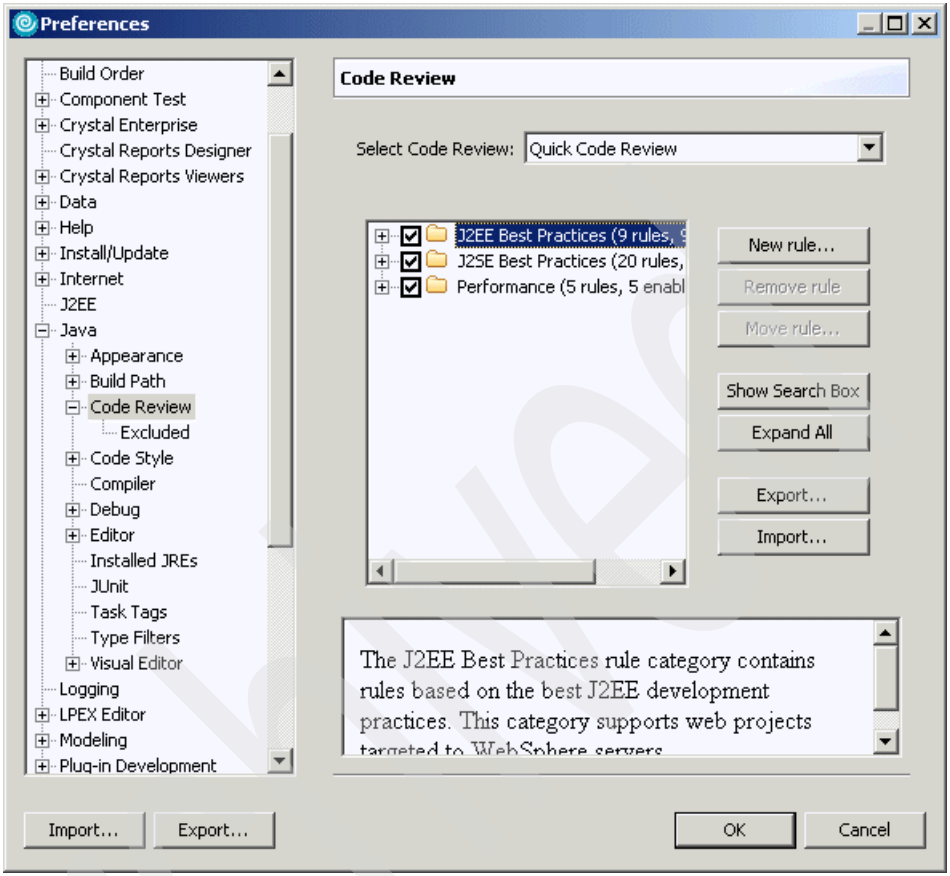


Figure 3-38 Code review preferences

3. Select the code review required based on the application being written from the drop-down box. For example, set the code review to J2EE Best Practices code review for review of dynamic projects.

Add a rule and filter

A rule can be added to the code review we have selected, and a filter to ignore particular resources. The procedure to show this is as follows:

1. From the Workbench, select **Windows** → **Preferences**.
2. Select **Java** → **Code Review**.
3. Click the **New Rule...** displaying the window shown in Figure 3-39 on page 127.

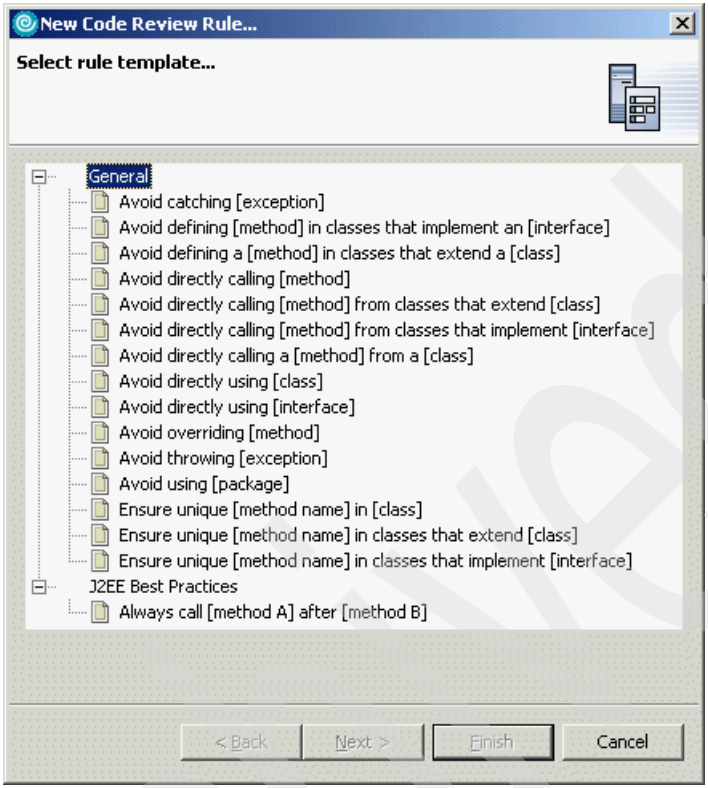


Figure 3-39 Adding a new rule to the J2EE Best Practices code review

4. Select the rule **Avoid throwing [exception]** and click **Next** to display the screen shown in Figure 3-40 on page 128.
   a. Select the category that this rule should fall under. In this case, the category **J2EE Best Practices:Correctness** was selected.
   b. Select the severity of the rule failing from the options Problem, Warning, or Recommendation. In this case, **Problem** was selected.
   c. Specify the class that needs to be avoided being thrown, **javax.servlet.ServletException**.
   d. Click **Finish**.

Archived

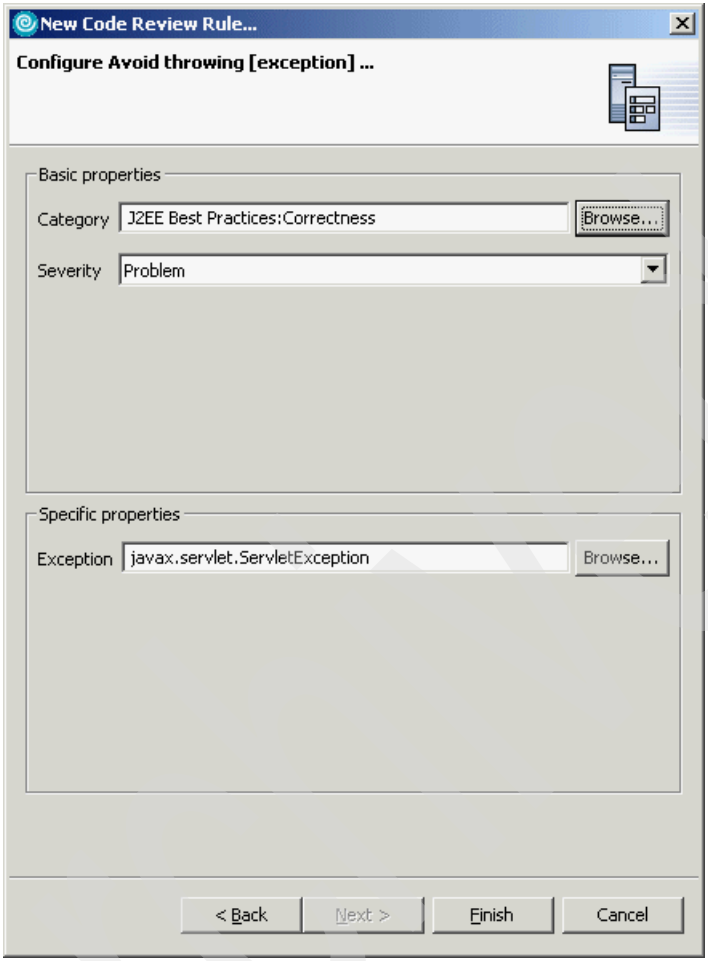


Figure 3-40 Define the parameters for the rule

Tip: Ensure that the specified category for the rule is available for the code review that the rule is being added to. The default category for user-defined rules is *User Defined*, and in all views except the Complete Code Review this will not appear and will not be applied when a code review is performed.

5. Click **OK** to close the windows preferences.

When the rule has been added, we can specify resource filters to ignore applying this rule to certain resources that may not be complete or valid for application of this rule.

For example, if we had a servlet located in the workspace at a location /JSPandServletExample/JavaSource/com/ibm/samples/controller/LogonServlet.java/LoginServlet.java, which we want to not have the code rule run, then the following would be performed:

1. From the Workbench, select **Windows** → **Preferences**.
2. Select **Java** → **Code Review**.
3. Select and expand **J2EE Best Practices** → **Correctness**, and highlight the rule added previously, **Avoid throwing javax.servlet.ServletException**.
4. Select the **Resource Filters** tab at the bottom of the window.
5. Click **Add...** and fill in the information shown in Figure 3-41, and specified as the example above, and click **OK**.

Figure 3-41 Filtering a class for a particular rule

This would prevent this rule from running for the particular resource that has been defined.

Exclusion of resources

Resources can be excluded from being code reviewed completely for a particular code review. Using the example defined in “Add a rule and filter” on page 126, the procedure to do this is as follows:

1. From the Workbench, select **Windows** → **Preferences**.
2. Select **Java** → **Code Review** → **Excluded**.
3. Select the **Resources** tab.
4. Click **Add...** to display the window shown in Figure 3-42 on page 130, and type in the resource name or use the **Browse...** button.

Archived

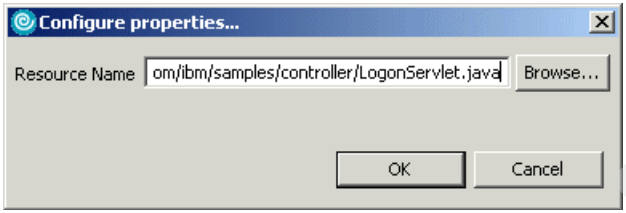


Figure 3-42 Defining an excluded resource from the code review

5. Click **OK** to display the window shown in Figure 3-43.

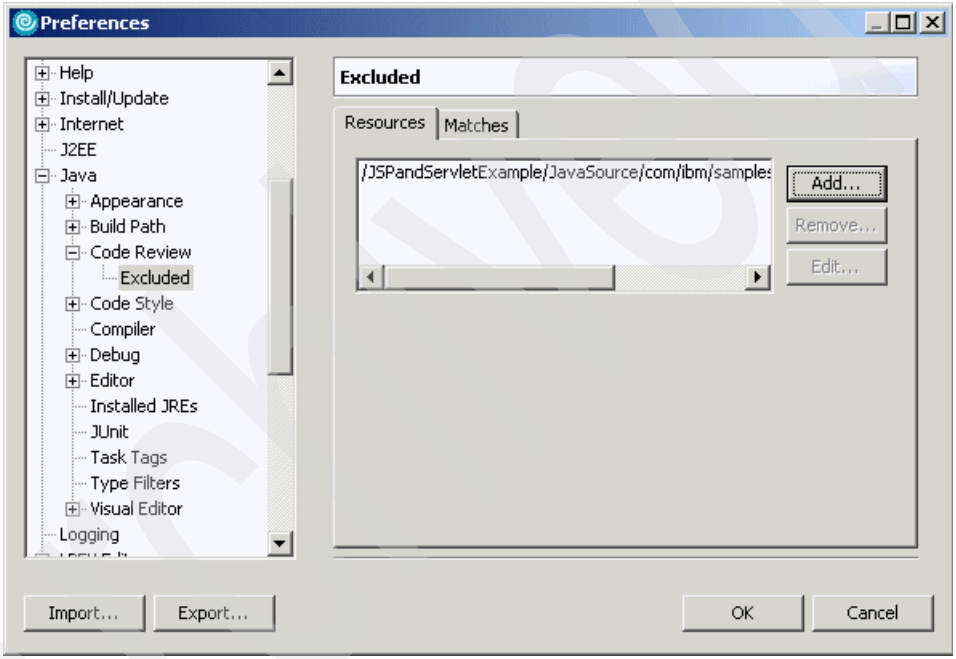


Figure 3-43 Excluded resource has been set for all rules in code review

Excluding matches

Particular matches identified after performing a code review can be excluded in the Code Review view, using the context menu for the particular code review.

1. From the Workbench, select **Windows** → **Preferences**.
2. Select **Java** → **Code Review**
3. Select **Excluded** on the Matches tab. This dialog view only allows the removal of excluded matches.

4

Perspectives, views, and editors

Rational Application Developer supports a role-based development model. It does so by providing several different perspectives on the same project. Each perspective is suited for a particular role and provides the developer with the necessary tools to work on the tasks associated with that role.

This chapter is organized as follows to highlight the features of the perspectives, views, and editors included in IBM Rational Application Developer V6:

- Integrated development environment (IDE)
- Available perspectives
- Rational Product Updater

Archived

© Copyright IBM Corp. 2005. All rights reserved.

4.1 Integrated development environment (IDE)

An integrated development environment is a set of software development tools such as source editors, compilers, and debugger, that are accessible from a single user interface.

In Rational Application Developer, the IDE is called the *Workbench*. Rational Application Developer's Workbench provides customizable perspectives that support role-based development. It provides a common way for all members of a project team to create, manage, and navigate resources easily. It consists of a number of interrelated views and editors.

Views provide different ways of looking at the resource you are working on, whereas editors allow you to create and modify the resource. Perspectives are a combination of views and editors that show various aspects of the project resource, and are organized by developer role or task. For example, a Java developer would work most often in the Java perspective, while a Web designer would work in the Web perspective.

Several perspectives are provided in Rational Application Developer, and team members also can customize them, according to their current role of preference. You can open more than one perspective at a time, and switch perspectives while you are working with Rational Application Developer. If you find that a particular perspective does not contain the views or editors you require, you can add them to the perspective and position them to suit your preference.

Before describing the perspectives, we will take a look at Rational Application Developer's help feature.

4.1.1 Rational Application Developer online help

Rational Application Developer's online help system provides access to the documentation, and lets you browse and print help content. It also has a full-text search engine and context-sensitive help.

Rational Application Developer provides the help content in a separate window, which you can open by selecting **Help** → **Help Contents** from the menu bar (see Figure 4-1 on page 133).

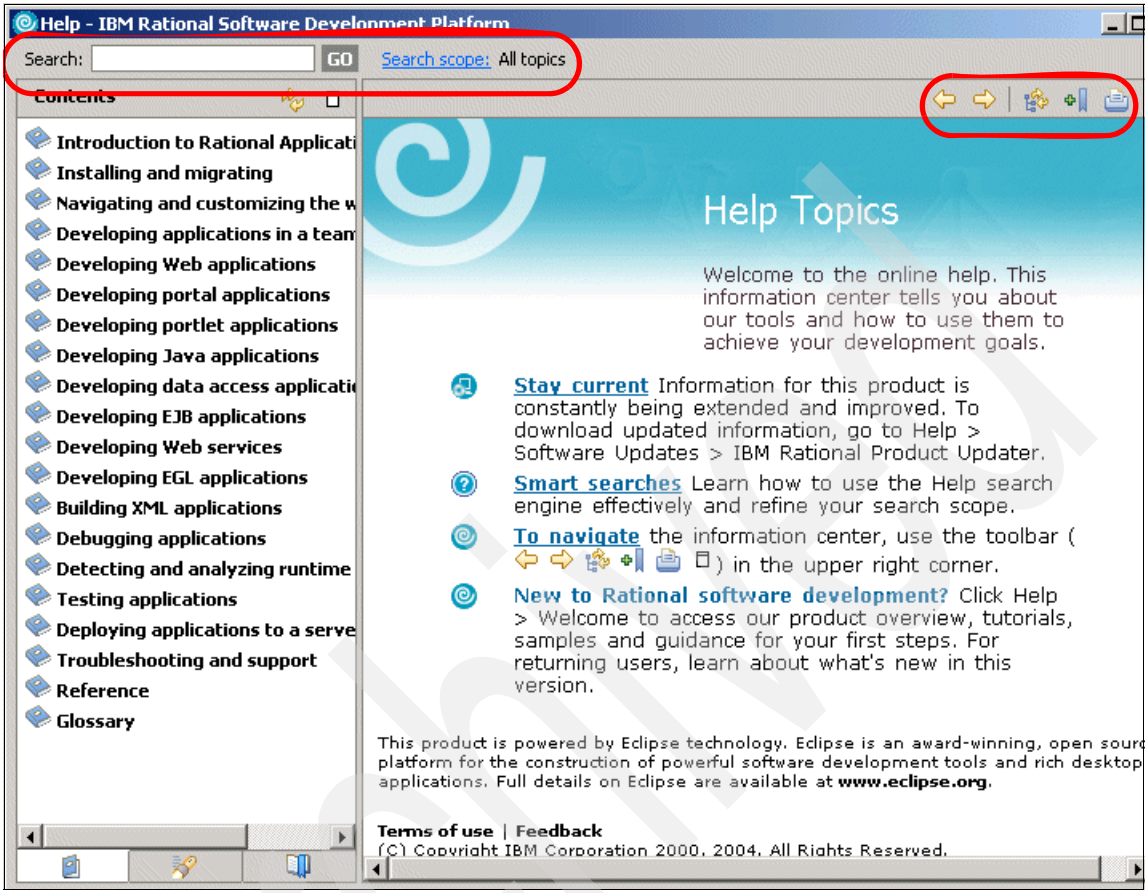


Figure 4-1 Help window

In the help window you see the available books in the left pane and the content in the right pane. When you select a book in the left pane, the appropriate table of contents opens up and you can select a topic within the book. When a page is selected, the page content is displayed in the right pane.

You can navigate through the help documents by using the Go Back and Go Forward buttons in the toolbar for the right pane. There are other buttons in this toolbar:

- Show in Table of Contents: Synchronizes the navigation frame with the current topic, which is helpful if you have followed several links to related topics in several files, and want to see where the current topic fits into the navigation path.

- Bookmark Document: Adds a bookmark to the Bookmarks view in the left pane.
- Print Page.
- Maximize: Maximizes the left pane to fill the whole help window.

Figure 4-2 shows the help window with the table of contents for the *Navigating and customizing the Workbench* book.

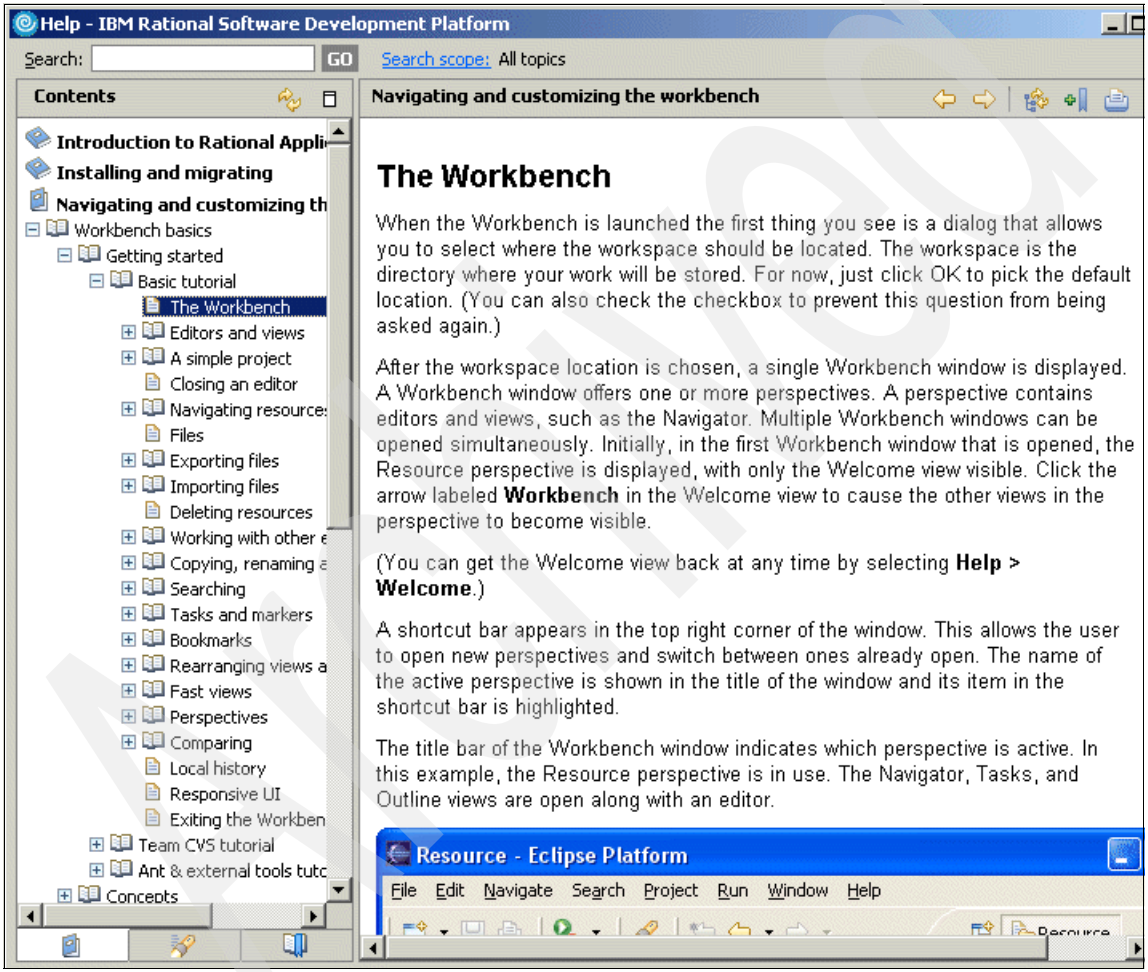


Figure 4-2 Workbench help

Rational Application Developer's help system contains a lot of useful information about the tool and technologies. It provides information about the different

concepts used by the Workbench, the different tasks you can do within the Workbench, and some useful samples and tutorials.

The Search field allows you to do a search over all books by default. The **Search Scope** link opens a dialog box where you can select a scope for your search. If you have previously defined a search list, this list of books and topics can be selected from the dialog, allowing you to choose from particular selections of books that you have found useful in the past. This is shown in Figure 4-3 on page 135.

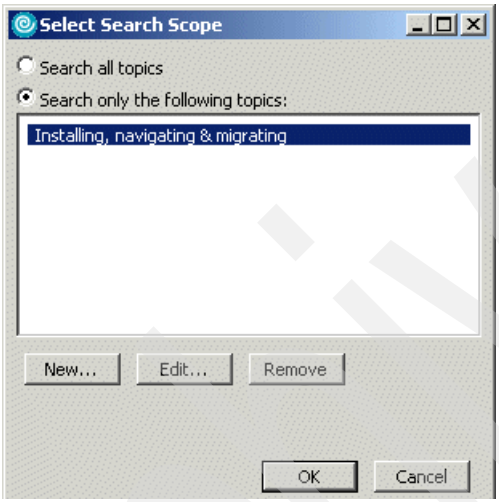


Figure 4-3 Select Search Scope dialog for help

To create a search list, click the **New...** button, or click **Edit...** if you wish to modify an existing search list. In either case, a dialog similar to the one shown in Figure 4-4 on page 136 allows you to make your selection of books and topics.

Archived

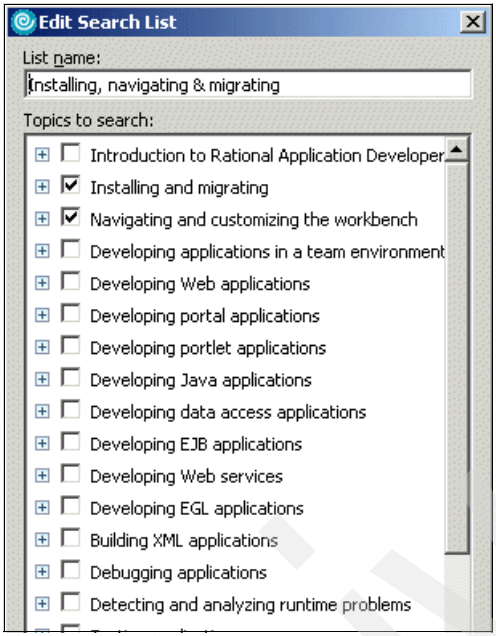


Figure 4-4 New Search List dialog for help

Note: The first time you search the online help, the help system initiates an index-generation process. This process builds the indexes for the search engine to use, which will probably take several minutes. Unlike WebSphere Studio Application Developer, the index is not rebuilt when the product is opened with a different workspace. It is also possible to prebuild the index: See the help entry under **Navigating and customizing the workbench** → **Extending the workbench (advanced)** → **Extending the platform** → **Reference** → **Other reference information** → **Pre-built documentation index**.

Enter your search expression in the Search field and click **Go** to start your search.

4.1.2 Perspectives

Perspectives provide a convenient grouping of views and editors that match the way you use Rational Application Developer. Depending on the role you are in and the task you have to do, you open a different perspective. A perspective defines an initial set and layout of views and editors for performing a particular set of development activities (for example, EJB development or profiling). You

can change the layout and the preferences and save a perspective that you can have customized, so that you can open it again later. We will see how to do this later in this chapter.

4.1.3 Views

Views provide different presentations of resources or ways of navigating through the information in your workspace. For example, the Navigator view displays projects and other resources that you are working as a folder hierarchy, like a file system view. Rational Application Developer provides synchronization between different views and editors. Some views display information obtained from other software products, such as database systems or software configuration management (SCM) systems.

A view might appear by itself, or stacked with other views in a tabbed notebook arrangement. A perspective determines the initial set of views that you are likely to need. For example, the Java perspective includes the Package Explorer and the Hierarchy views to help you work with Java packages and hierarchies.

4.1.4 Editors

When you open a file, Rational Application Developer automatically opens the editor that is associated with that file type. For example, the Page Designer editor is opened for .html, .htm, and .jsp files, while the Java editor is opened for .java and .jpage files.

Editors that have been associated with specific file types will open in the editor area of the Workbench. By default, editors are stacked in a notebook arrangement inside the editor area. If there is no associated editor for a resource, Rational Application Developer will open the file in the default editor, which is a text editor.

4.1.5 Perspective layout

Many of Rational Application Developer's perspectives use a similar layout. Figure 4-5 shows a layout of a perspective that is quite common.

Active!

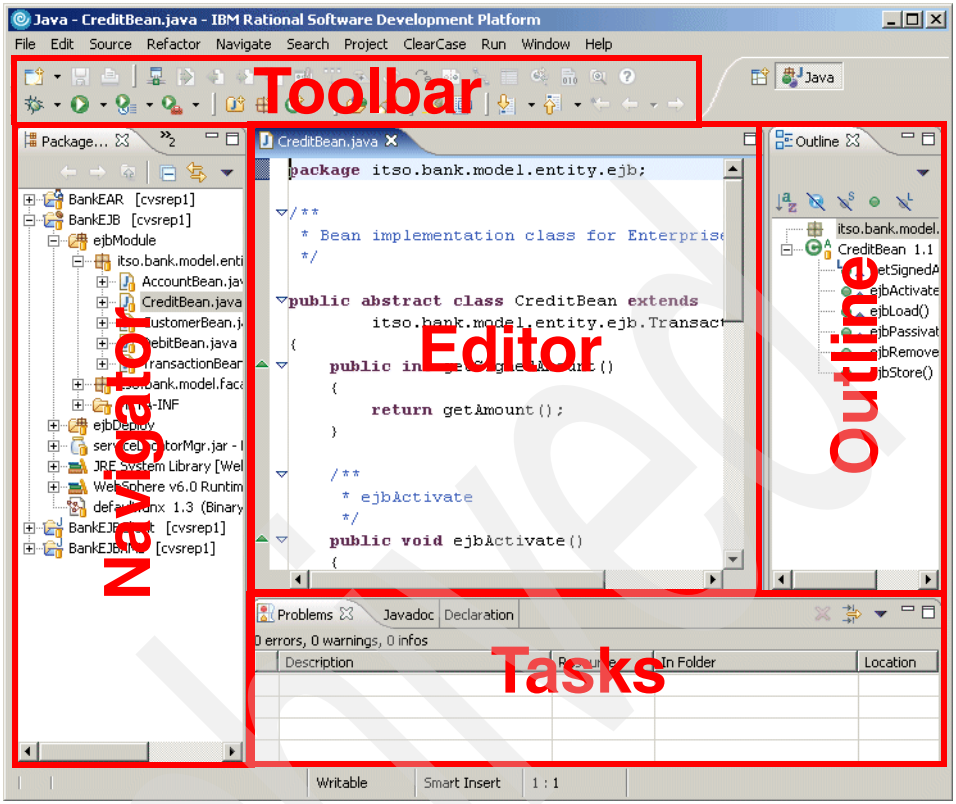


Figure 4-5 Perspective layout

On the left side are views for navigating through the workspace. In the middle of the Workbench is larger pane, usually the source editor or the design pane, which allows you to change the code and design of files in your project. The right pane usually contains outline or properties views. In some perspectives the editor pane is larger and the outline view is at the bottom left corner of the perspective.

The content of the views is synchronized. For example, if you change a value in the Properties view, the update will automatically be reflected in the Editor view.

4.1.6 Switching perspectives

There are two ways to open another perspective:

- Click the Open a perspective icon ( ) in the top right corner of the Workbench working area and select the appropriate perspective from the list.
- Select **Window** → **Open Perspective**.

In either case, there is also an **Other...** option, which displays the Select Perspective dialog (see Figure 4-6). Select the required perspective and click **OK**.

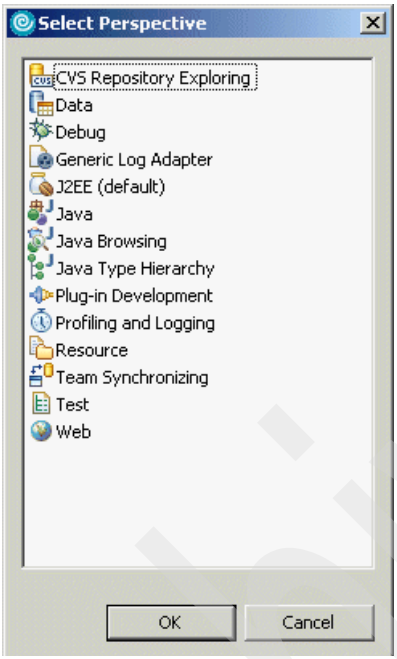


Figure 4-6 Select Perspective dialog

A button appears in the top left corner of the Workbench (an area known as the shortcut bar) for each open perspective, as shown in Figure 4-7. To switch to a particular perspective, click the appropriate button.

Figure 4-7 Buttons to switch between perspectives

Archived

Tips:

- The name of the perspective is shown in the window title area along with the name of the file open in the editor, which is currently at the front.
- To close a perspective, right-click the perspective's button on the shortcut bar (top right) and select **Close**.
- To display only the icons for the perspectives, right-click somewhere in the shortcut bar and deselect **Show Text**.
- Each perspective requires memory, so it is a good practice to close perspectives when not being used to improve performance.

4.1.7 Specifying the default perspective

The J2EE perspective is Rational Application Developer's default perspective, but this can be changed using the Preferences dialog:

1. From the Workbench, select **Window** → **Preferences**.
2. Expand **Workbench** and select **Perspectives**.
3. Select the perspective that you want to define as the default, and click **Make Default**.
4. Click **OK**.

4.1.8 Organizing and customizing perspectives

Rational Application Developer allows you to open, customize, reset, save, and close perspectives. These actions can be found in the Window menu.

To customize a perspective, select **Window** → **Customize Perspective...**. The Customize Perspective dialog opens (Figure 4-8 on page 141).

Archive

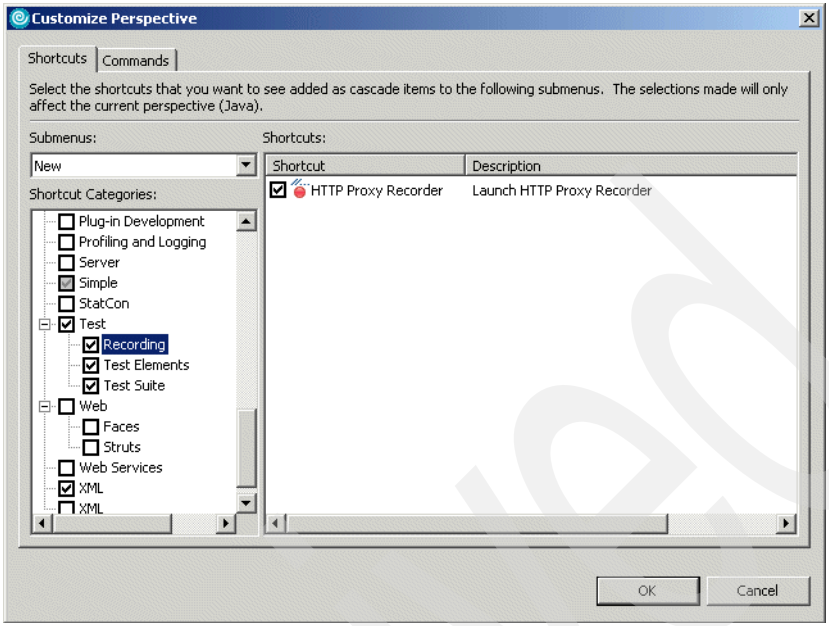


Figure 4-8 Customize Perspective dialog

In the dialog you can use the check boxes to select which elements you want to see in the New, Open Perspective, and Show View menus of the selected perspective. Items you do not select are still accessible by clicking the **Other...** menu option. The Commands tab of the dialog allows you to select command groups that will be added to the menu bar or tool bar for Rational Application Developer in this perspective.

- Customize

  You can also customize a perspective by adding, closing, moving, and resizing views. To add a view to the perspective, select **Window** → **Show View** and select the view you would like to add to the currently open perspective.

- Move

  You can move a view to another pane by using drag and drop. To move a view, select its title bar, drag the view, and drop it on top of another view. The views are stacked and you can use the tabs at the top of the view to switch between them.

- Drag

  While you drag the view, the mouse cursor changes into a drop cursor. The drop cursor indicates what will happen when you release the view you are

dragging. In each case, the area that will be filled with the dragged view is highlighted with a rectangular outline.

The view will dock below the view under the cursor.

The view will dock to the left of the view under the cursor.

The view will dock to the right of the view under the cursor.

The view will dock above the view under the cursor.

The view will appear as a tab in the same pane as the view under the cursor.

The view will dock in the status bar (at the bottom of the Rational Application Developer window) and become a *Fast View* (see below).

The view will become a separate child window of the main Rational Application Developer window.

- Fast View

  A Fast View appears as a button in the status bar of Rational Application Developer. Clicking the button will toggle whether or not the view is displayed on top of the other views in the perspective.

- Maximize and minimize a view

  To maximize a view to fill the whole working area of the Workbench, you can double-click the title bar of the view, press Ctrl+M, or click the Maximize icon ( )in the view's toolbar. To restore the view, double-click the title bar or press Ctrl+M again. The Minimize button in the toolbar of a view minimizes the tab group so that only the tabs are visible; click the **Restore** button or one of the view tabs to restore the tab group.

- Save

  Once you have configured the perspective to your preferences, you can save it as your own perspective by selecting **Window** → **Save Perspective As...**. In practice it is often unnecessary to save a perspective with a new name since Rational Application Developer remembers the settings for all your perspectives. It might be useful in situations where a development is being shared between several developers.

- Restore

  To restore the currently opened perspective to its original layout, select **Window** → **Reset Perspective**.

Archived

4.2 Available perspectives

In this section we describe the perspectives that are available in Rational Application Developer. The perspectives are covered in the order in which they appear in the Select Perspective dialog (alphabetical order).

Rational Application Developer allows a developer to disable or enable capabilities to simplify the interface or make it more capable for specific types of development work. This is described in 3.2.3, “Capabilities” on page 86. For this section, all capabilities have been enabled in the product—otherwise certain perspectives, associated with specific capabilities, would not be available.

IBM Rational Application Developer V6.0 includes the following perspectives:

- CVS Repository Exploring perspective
- Data perspective
- Debug perspective
- Generic Log Adapter perspective
- J2EE perspective
- Java perspective
- Java Browsing perspective
- Java Type Hierarchy perspective
- Plug-in Development perspective
- Profiling and Logging perspective
- Resource perspective
- Team Synchronizing perspective
- Test perspective
- Web perspective

4.2.1 CVS Repository Exploring perspective

The CVS Repository Exploring perspective (see Figure 4-9 on page 144) lets you connect to Concurrent Versions System (CVS) repositories. It allows you to add and synchronize projects with the workspace and to inspect the revision history of resources.

- CVS Repositories view

  Shows the CVS repository locations that you have added to your Workbench. Expanding a location reveals the main trunk (HEAD), project versions, and branches in that repository. You can further expand the project versions and branches to reveal the folders and files contained within them.

  The context menu for this view also allows you to specify new repository locations. Use the CVS Repositories view to check out resources from the repository to the Workbench, configure the branches and versions shown by the view, view resource history, and compare resource versions.

Archived

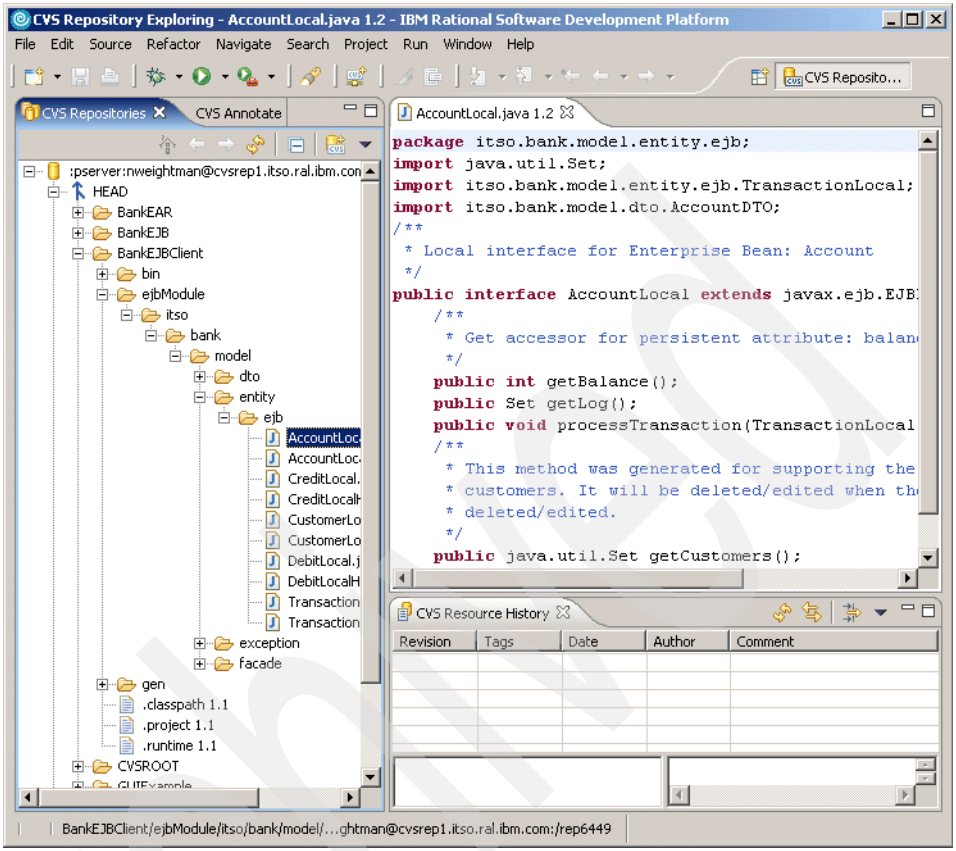


Figure 4-9 CVS Repository Exploring perspective

- Editor

  Files that exist in the repositories can be viewed by double-clicking them in a branch or version. This opens the version of the file specified in the editor pane.

- CVS Resource History view

  Displays more detailed history of each file. This view provides a list of all the revisions of a resource in the repository. From this view you can compare two revisions or open an editor on a revision.

- CVS Annotate view

  Select a resource in the CVS Repositories view, right-click a resource, and select **Show Annotation**. The CVS Annotate view will come to the front and will display a summary of all the changes made to the resource since it came under the control of your CVS server.

More details about using the CVS Repository Exploring perspective, and other aspects of CVS functionality in Rational Application Developer, can be found in Chapter 26, “CVS integration” on page 1299.

4.2.2 Data perspective

The Data perspective lets you access relational database tools, where you can create and manipulate the data definitions for your project. This perspective also lets you browse or import database schemas in the DB Servers view, create and work with database schemas in the Data Definition view, and change database schemas in the table editor. You can also export data definitions to another database installed either locally or remotely. The Data perspective is shown in Figure 4-10.

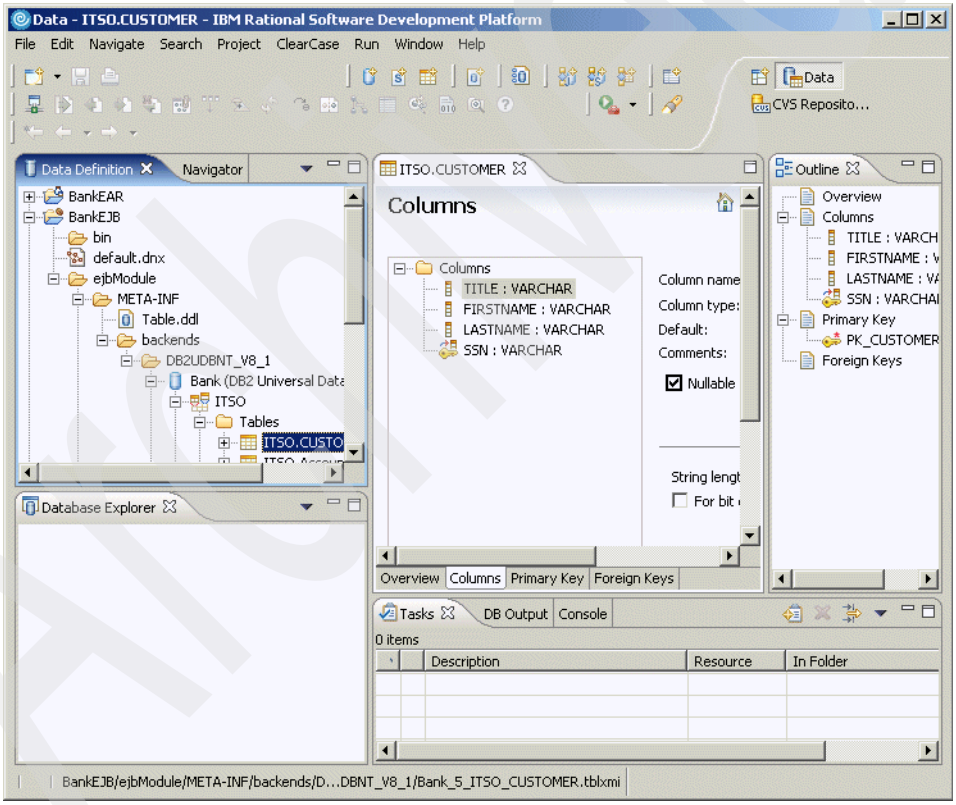


Figure 4-10 Data perspective

The views are:

- Data Definition view

This view of lets you work directly with data definitions, defining relational data objects. It can hold local copies of existing data definitions imported from the DB Servers view, designs created by running DDL scripts, or new designs that you have created directly in the Workbench.

- Database Explorer view

  Using this view, you can work with database connections to view the design of existing databases, and import the designs to another folder in the Data Definition view, where you can extend or modify them.

- Tasks view

  The Tasks view displays system-generated errors, warnings, or information associated with a resource, typically produced by builders. Tasks can also be added manually and optionally associated with a resource in the Workbench.

- Navigator view

  The Navigator view provides a hierarchical view of all the resources in the Workbench. By using this view you can open files for editing or select resources for operations such as exporting. The Navigator view is essentially a file system view, showing the contents of the workspace and the directory structures used by any projects that have been created outside the workspace.

- Console view

  The Console view shows the output of a process and allows you to provide keyboard input to a process. The console shows three different kinds of text, each in a different color: Standard output, standard error, and standard input.

- DB Output view

  The DB Output view shows the messages, parameters, and results that are related to the database objects that you work with, such as SQL statements, stored procedures, and user-defined functions.

More details about using the Data perspective can be found in Chapter 8, “Develop Java database applications” on page 333.

4.2.3 Debug perspective

The Debug perspective, shown in Figure 4-11 on page 147, contains five panes:

- Top left: Shows Debug and Servers views
- Top right: Shows Breakpoints and Variables views
- Middle left: Shows the editor or the Web browser when debugging Web applications
- Middle right: Shows the Outline view

- Bottom: Shows the Console and the Tasks view

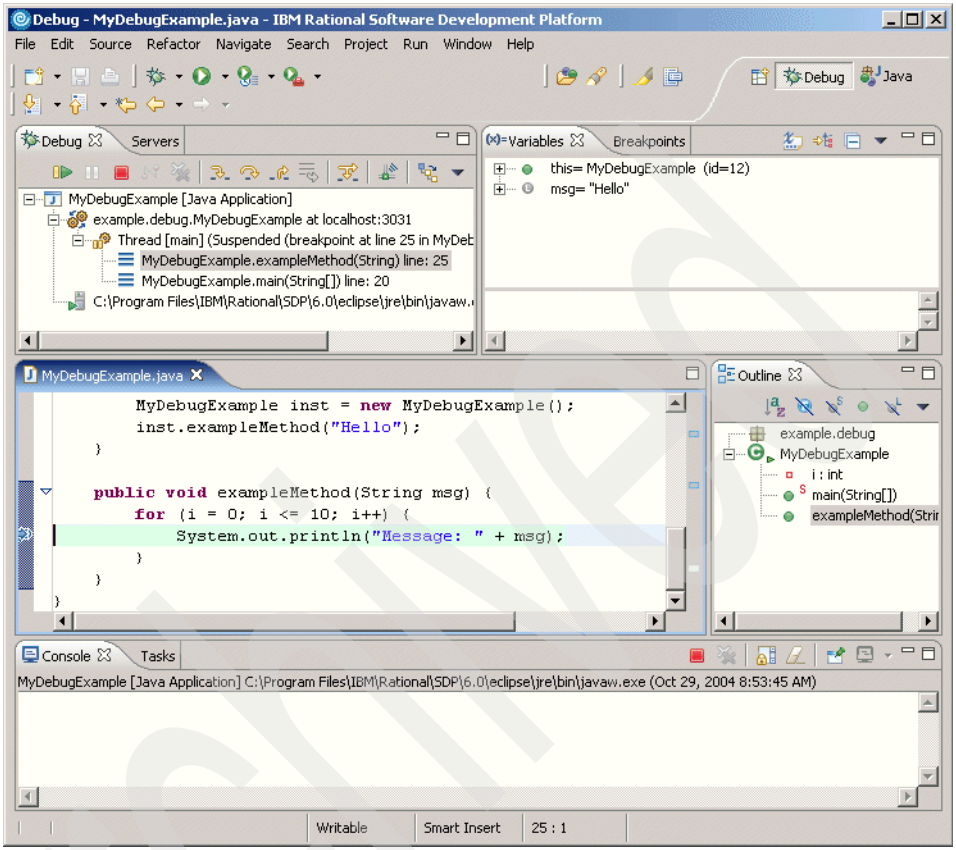


Figure 4-11 Debug perspective

The views are:

- Debug view

  The Debug view displays the stack frame for the suspended threads for each target you are debugging. Each thread in your program appears as a node in the tree. If the thread is suspended, its stack frames are shown as child elements.

  If the resource containing a selected thread is not open and/or active, the file opens in the editor and becomes active, focusing on the source with which the thread is associated.

  The Debug view enables you to perform various start, step, and terminate debug actions, as well as enable or disable step-by-step debugging.

- Variables view

Archived

The Variables view displays information about the variables in the currently selected stack frame.

- Breakpoints view

  The Breakpoints view lists all the breakpoints you have set in the Workbench projects. You can double-click a breakpoint to display its location in the editor. In this view, you can also enable or disable breakpoints, delete them, change their properties, or add new ones. This view also lists Java exception breakpoints, which suspend execution at the point where the exception is thrown.

- Servers view

  The Servers view lists all the defined servers and their status. Right-clicking a server displays the server context menu, which allows the server to be started, stopped, etc.

- Outline view

  The Outline view shows the elements (imports, class, fields, and methods) that exist in the source file in the front editor. Clicking an item in the outline will position you in the editor view at the line where that structure element is defined.

The Console and Tasks views have already been discussed in earlier sections of this chapter.

More information about the Debug perspective can be found in Chapter 21, “Debug local and remote applications” on page 1121.

4.2.4 Generic Log Adapter perspective

This perspective is used when working with Hyades, which is an integrated test, trace, and monitoring environment included with Rational Application Developer. See Chapter 20, “JUnit and component testing” on page 1081, for more information on using this feature of the Workbench. Hyades is an Eclipse project dealing with the area of Automated Software Quality; for more information, see http://www.eclipse.org/hyades/.

This perspective includes:

- Problems view

  This view shows all errors, warnings, and information messages relating to resources in the workspace. The items listed here can be used to navigate to the line of code containing the error, warning, or information point.

- Sensor Results view

A sensor monitors a log or trace resource for changes. This view shows what sensor events have taken place.

- Extractor Results view

  Extractors take content from a log or trace resource and convert it into a standardized internal representation, known as Common Base Event (CBE) format.

- Formatter Results view

  A formatter takes CBEs and presents them in a customizable format.

The Navigator view has already been discussed in earlier sections of this chapter.

4.2.5 J2EE perspective

The default perspective of Rational Application Developer is the J2EE perspective, shown in Figure 4-12 on page 150. Rational Application Developer lets you change the default perspective. See 4.1.7, “Specifying the default perspective” on page 140, for instructions on how to change the default perspective.

The J2EE perspective contains the following views that you would typically use when you develop J2EE resources:

- Project Explorer view

  This shows information about your J2EE and other projects in the workspace, including your project resources, J2EE deployment descriptors, and the files in the project directories, like the Navigator view.

Archived

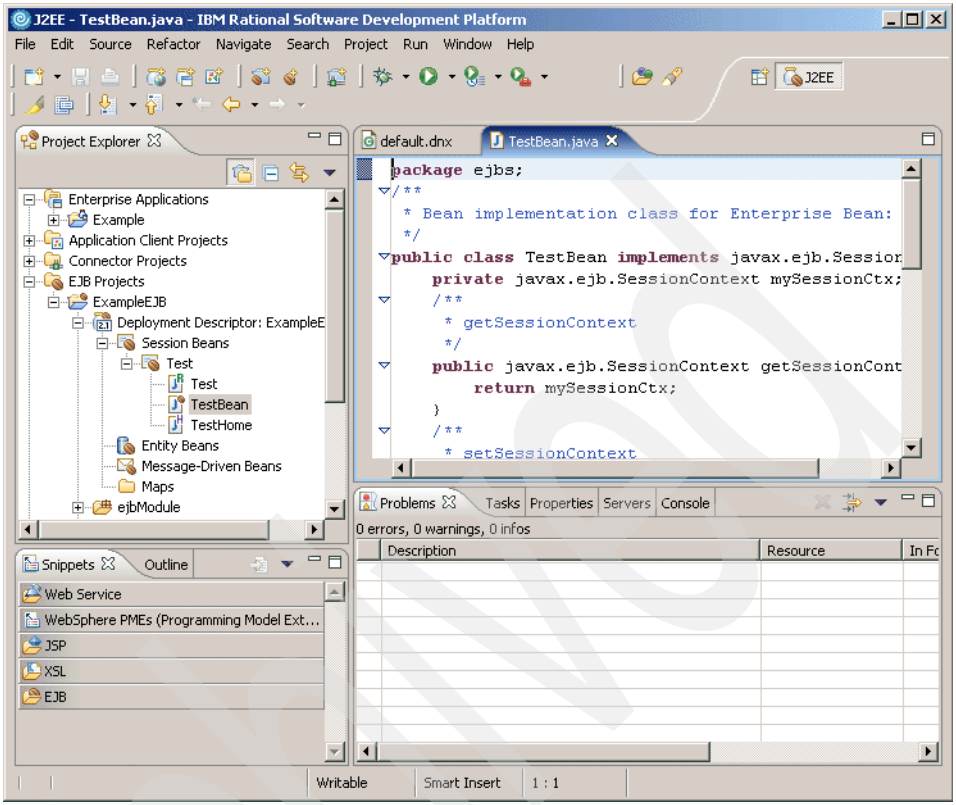


Figure 4-12 J2EE perspective

- Snippets view

  The Snippets view lets you catalog and organize reusable programming objects, such as HTML tagging, JavaScript, and JSP code, along with files and custom JSP tags. The view can be extended based on additional objects that you define and include. The available snippets are arranged in *drawers*, such as JSP or EJB, and the drawers can be customized by right-clicking a drawer and selecting **Customize...**.

- Properties view

  Provides a tabular view of the properties and associated values of objects in files you have open in an editor. For example, you can specify converters in the Properties view of the Mapping editor.

The Outline, Servers, Problems, Tasks, and Console views have already been discussed in earlier sections of this chapter.

More details about using the J2EE perspective can be found in Chapter 15, “Develop Web applications using EJBs” on page 827.

4.2.6 Java perspective

The Java perspective (Figure 4-13 on page 152) supports developers who create, edit, and build Java code.

The Java perspective consists of an editor area and displays, by default, the following views:

- Package Explorer view

  This shows the Java element hierarchy of all the Java projects in your Workbench. It provides you with a Java-specific view of the resources shown in the Navigator view (which is not shown by default in the Java perspective). The element hierarchy is derived from the project's build classpath. For each project, its source folders and referenced libraries are shown in the tree view. From here you can open and browse the contents of both internal and external JAR files.

  Note: When in the Package Explorer view, the JavaSource folder is off the project root folder. In most other views, the JavaSource folder is under Java Resources.

Archived

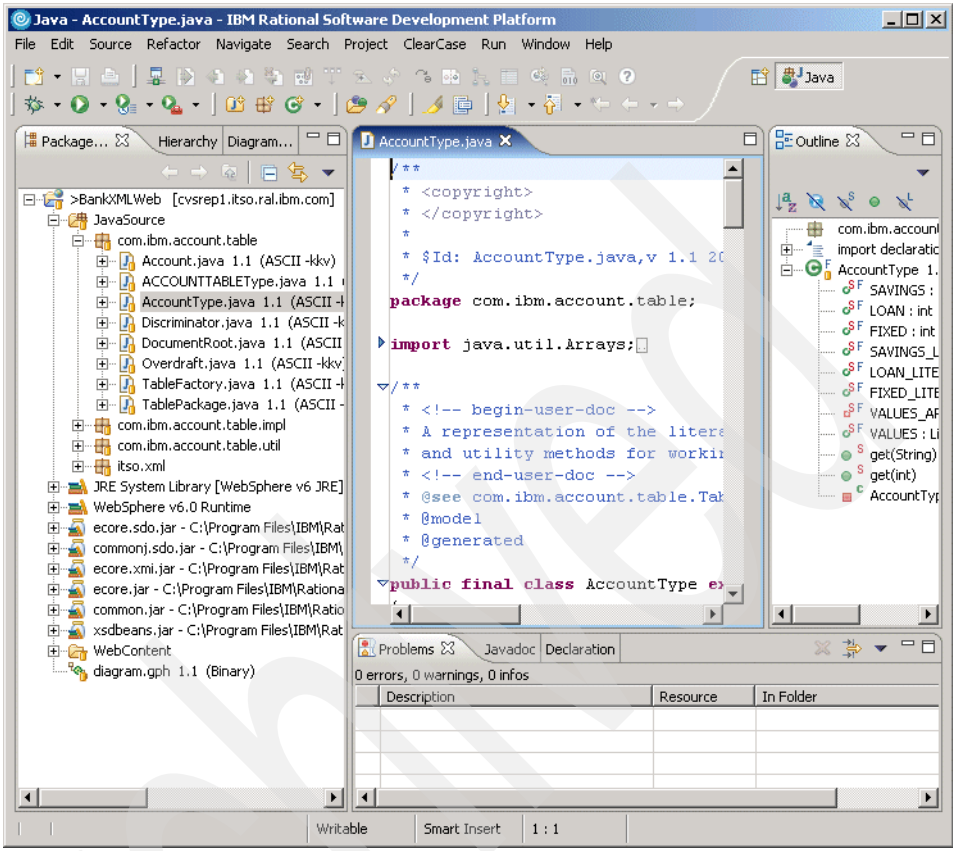


Figure 4-13 Java perspective

- Hierarchy view

  Can be opened for a selected type to show its superclasses and subclasses. It offers three different ways to look at a class hierarchy:

  - The Type Hierarchy displays the type hierarchy of the selected type, that is, its position in the hierarchy, along with all its superclass and subclasses.
  - The Supertype Hierarchy displays the supertype hierarchy of the selected type and any interfaces the type implements.
  - The Subtype Hierarchy displays the subtype hierarchy of the selected type or, for interfaces, displays classes that implement the type.

  More information about the Hierarchy view is provided in 4.2.8, “Java Type Hierarchy perspective” on page 155.

- Diagram Navigator view

This view shows a hierarchical view of the workspace and allows a developer to find UML visualization diagrams.

- Javadoc view

  This view shows the Javadoc comments associated with the element selected in the editor or outline view.

- Declarations view

  Shows the source of the element selected in the editor or in outline view.

The Outline and Problems views have already been discussed in earlier sections of this chapter.

See Chapter 7, “Develop Java applications” on page 221, for more information about how to work with the Java perspective and the following two perspectives.

4.2.7 Java Browsing perspective

The Java Browsing perspective also addresses Java developers, but it provides different views (Figure 4-14 on page 154).

Archived

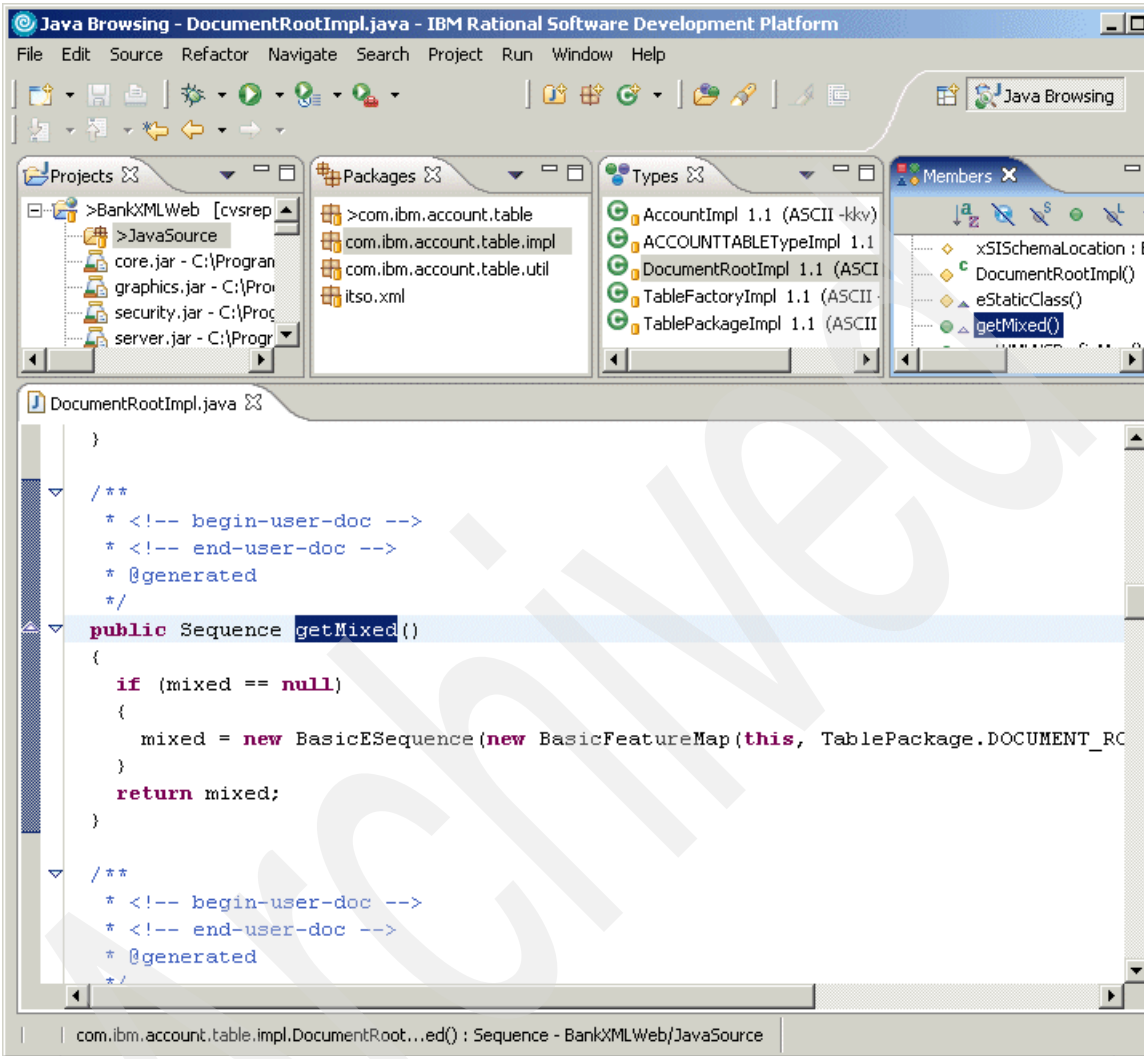


Figure 4-14 Java Browsing perspective

The Java Browsing perspective has a large editor area and several views to select the programming element you want to edit:

- Projects view: Lists all Java projects
- Packages view: Shows the Java packages within the selected project
- Types view: Shows the types defined within the selected package
- Members view: Shows the members of the selected type

Archive

The Show Selected Element Only button ( ) toggles between showing all the content of the selected type and showing only the code (and comments) for the element selected in the Members view.

4.2.8 Java Type Hierarchy perspective

This perspective also addresses Java developers and allows them to explore a type hierarchy. It can be opened on types, compilation units, packages, projects, or source folders, and consists of the Hierarchy view and an editor (Figure 4-15).

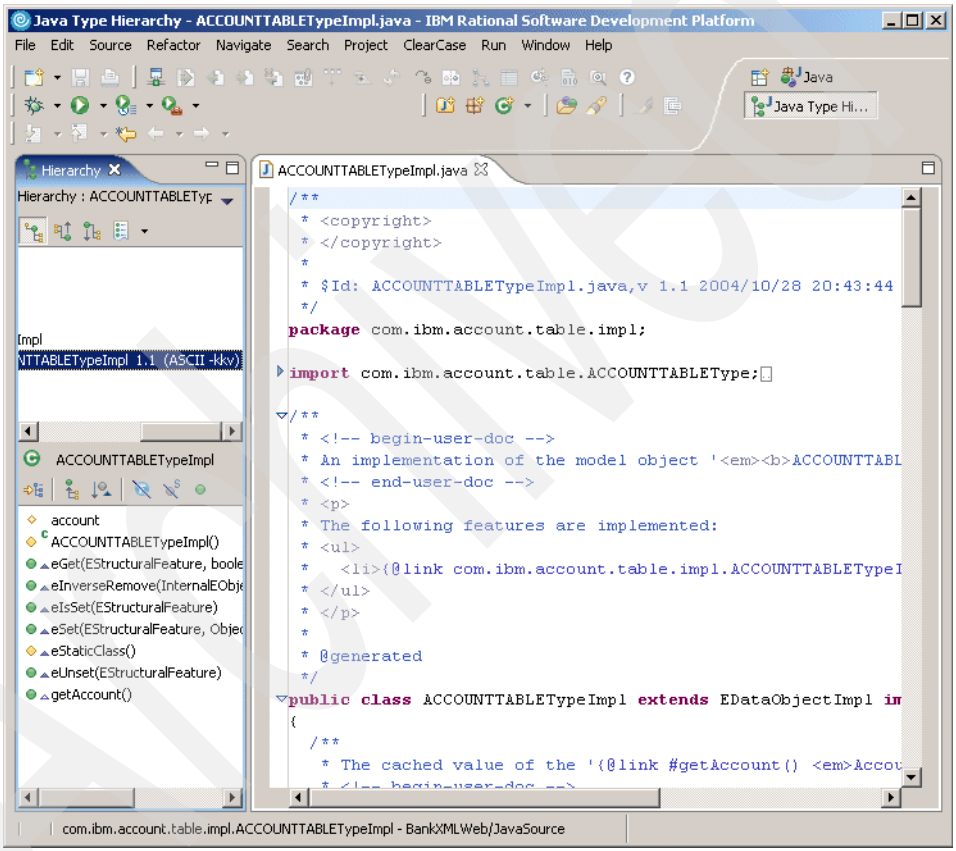


Figure 4-15 Java Type Hierarchy perspective

The Hierarchy view does not display a hierarchy until you select a type, so the initial appearance is as shown in Figure 4-16 on page 156.

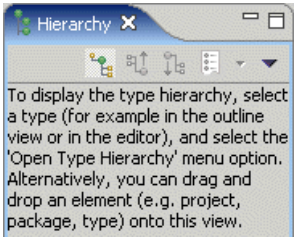


Figure 4-16 Hierarchy view - initial appearance

To open a type in the Hierarchy view, open the context menu from a type in any view and select **Open Type Hierarchy**. The type hierarchy is displayed in the Hierarchy view. Figure 4-17 shows the Hierarchy view of the Swing class JList.

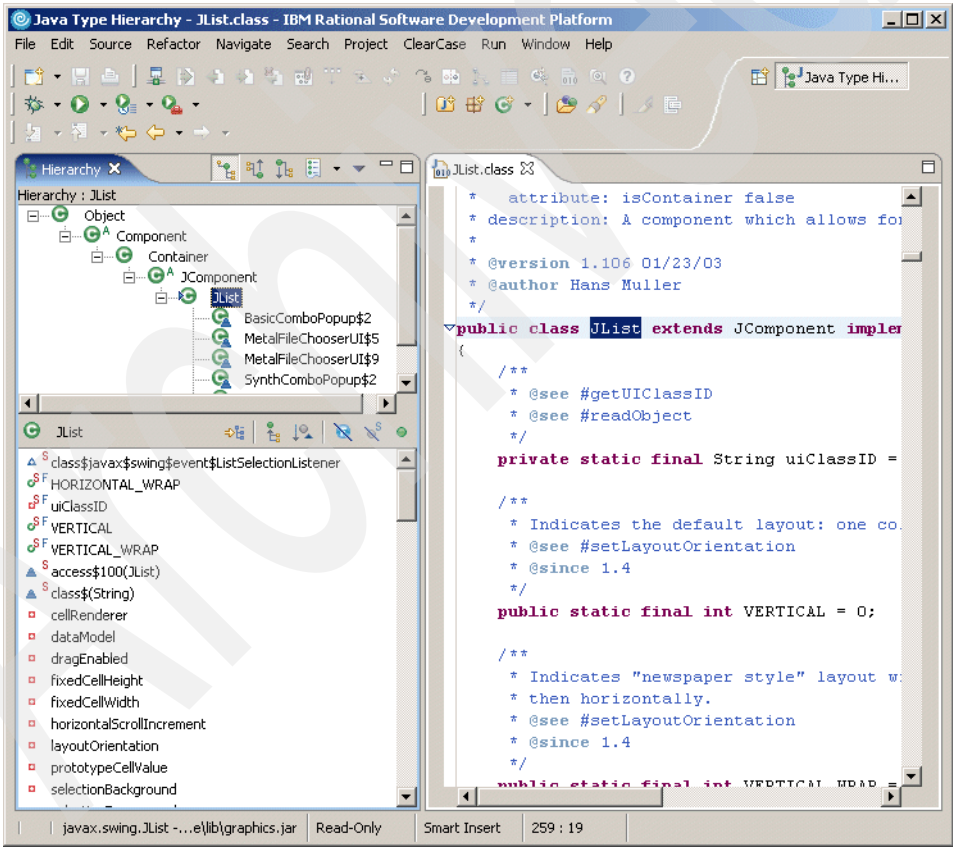


Figure 4-17 Hierarchy view

Icons are provided at the top of the Hierarchy view to display the type hierarchy ( ), the supertype hierarchy (bottom-up) ( ), or the subtype hierarchy (top-down) ( ). The supertype hierarchy also shows interfaces that are implemented for each class in the hierarchy.

Since the Hierarchy view, which has been described earlier in this chapter, is also displayed in the Java perspective, and can be added to any other perspective as required, the only advantage of the Java Type Hierarchy perspective is the large editor area.

4.2.9 Plug-in Development perspective

You can develop your own Rational Application Developer tools by using the Plug-in Development perspective, which is inherited by Rational Application Developer from the Eclipse framework. This perspective includes:

- Plug-ins view: Shows the combined list of workspace and external plug-ins
- Error Log view: Shows the error log for the software development platform, allowing a plug-in developer to diagnose problems with plug-in code

The other views in this perspective have already been described earlier in this chapter.

In this book we do not describe how to develop plug-ins for Rational Application Developer or Eclipse. Figure 4-18 on page 158 shows the Plug-in Development perspective.

Archived

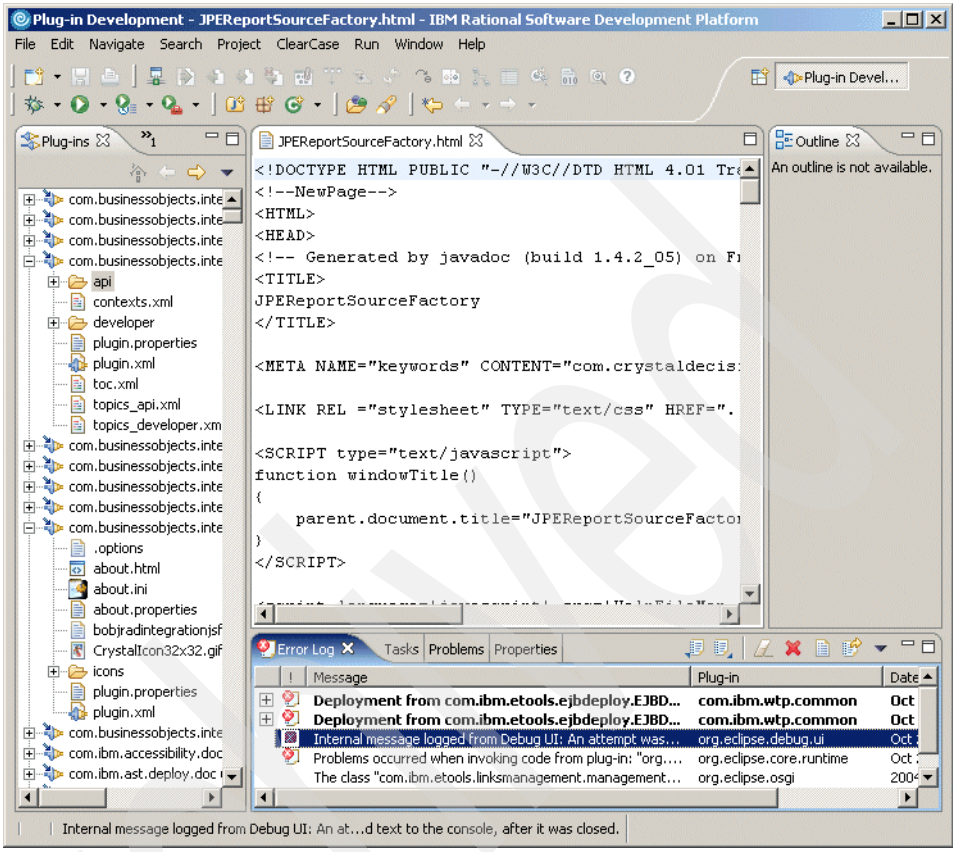


Figure 4-18 Plug-in Development perspective

To learn more about plug-in development, refer to *Eclipse Development using the Graphical Editing Framework and the Eclipse Modeling Framework*, SG24-6302, and *The Java Developer's Guide to Eclipse* by Shavor et al.

4.2.10 Profiling and Logging perspective

The Profiling and Logging perspective provides a number of views for working with logs and for profiling applications.

- Log Navigator view

  This view is used for working with various types of log files, including the service logs for WebSphere Application Server. Logs and symptom databases can be loaded using this view and opened in an editor or viewer, as shown in Figure 4-19 on page 159.

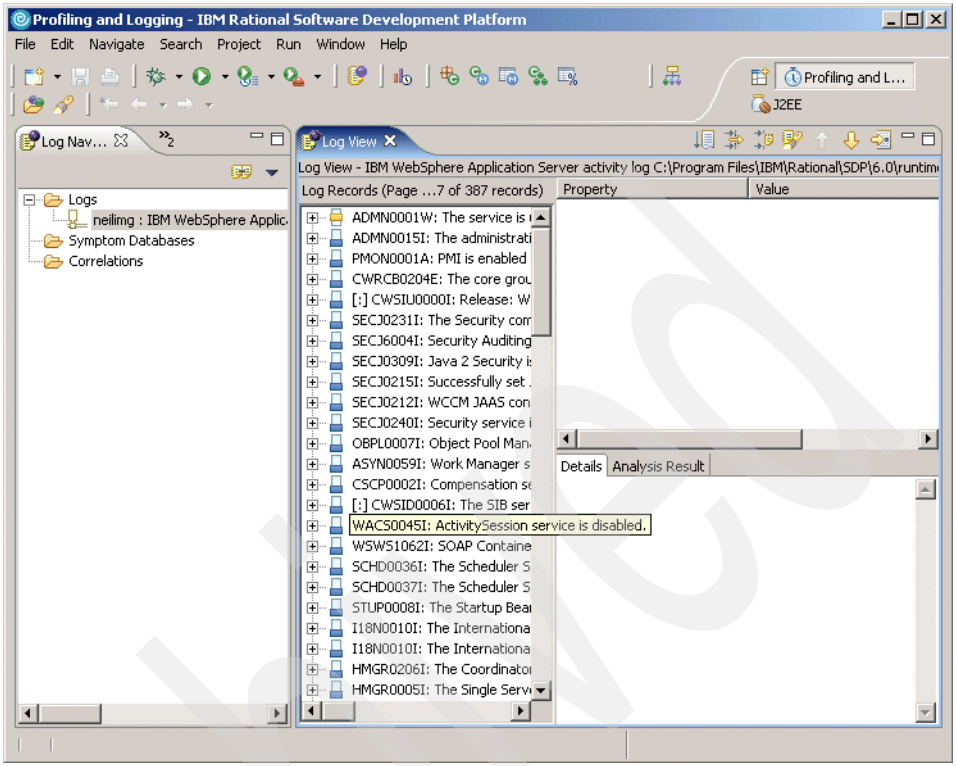


Figure 4-19 Working with logs in the Profiling and Logging perspective

- Profiling Monitor view (not shown in Figure 4-19)

  This view shows the process that can be controlled by the profiling features of Rational Application Developer. Performance and statistical data can be collected from processes using this feature and displayed in various specialized views.

The Navigator view has been discussed earlier in this chapter.

More details about these views and the techniques required to use them can be found in Chapter 24, “Profile applications” on page 1237.

4.2.11 Resource perspective

The Resource perspective is a very simple perspective (see Figure 4-20 on page 160).

Archived

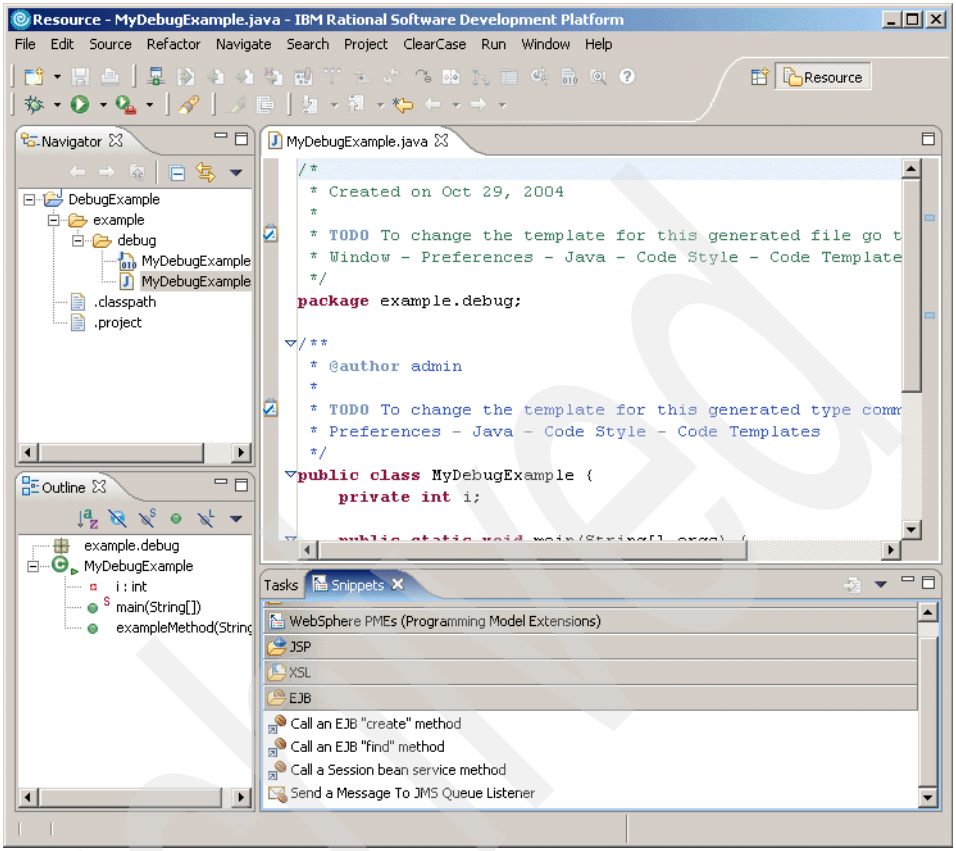


Figure 4-20 Resource perspective

All the views in this perspective have already been described.

4.2.12 Team Synchronizing perspective

The Team Synchronizing perspective provides the features necessary to synchronize the resources in the workspace with resources held on an SCM repository system, regardless of the specific system being used in your team.

- Synchronize view

  Clicking the Synchronize button on the toolbar of this view leads you through the process of specifying the SCM repository you want to synchronize with. The view will display the list of synchronization items that result from the analysis of the differences between the local and repository versions of your projects. Double-clicking an item will open the comparison view to help you in completing the synchronization. This is shown in Figure 4-21.

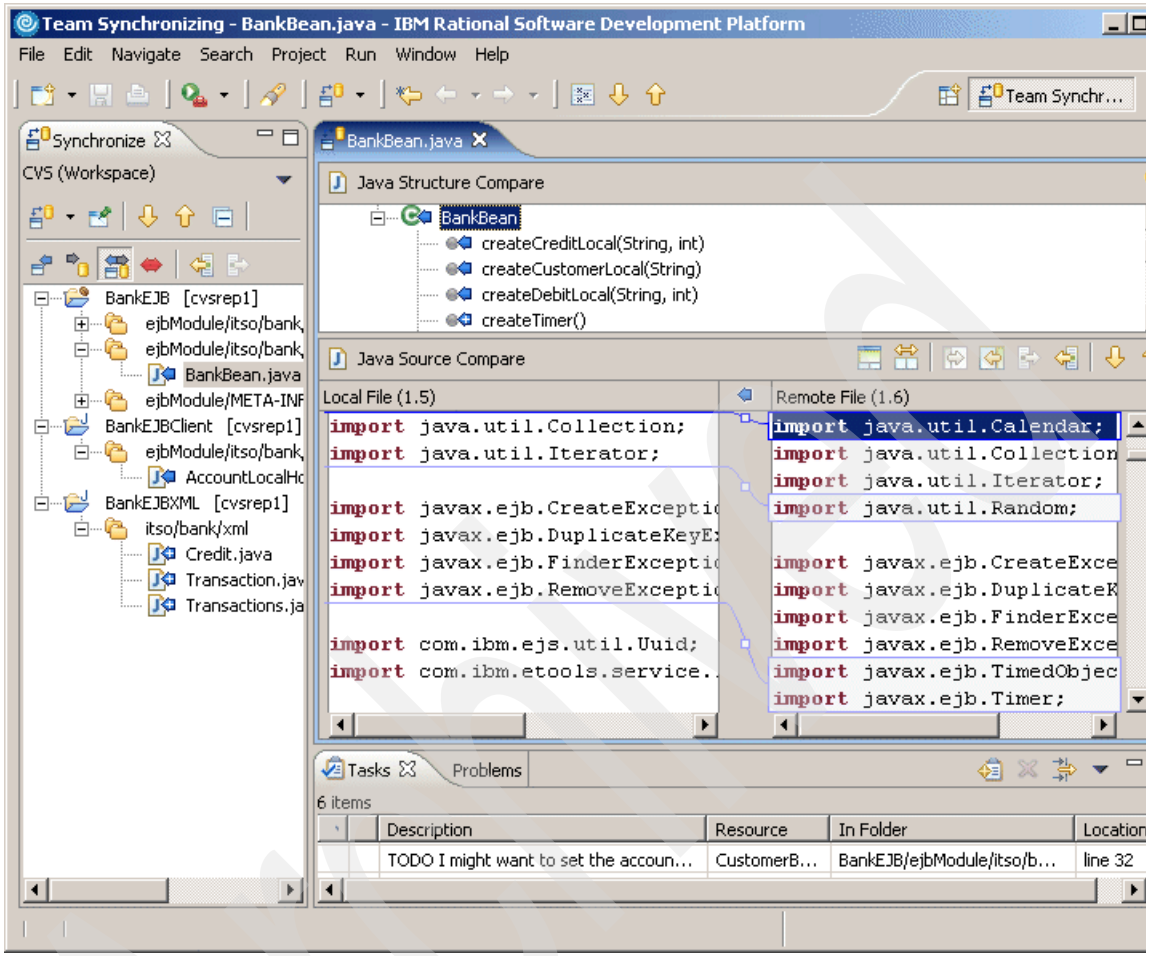


Figure 4-21 Synchronizing resources using the Team Synchronizing perspective

4.2.13 Test perspective

The Test perspective (see Figure 4-22 on page 162) provides a framework for defining and executing test cases and test suites. The main view provided is the Test Navigator view. The Test Navigator is the main navigation view for browsing and editing test suites and viewing test results. It shows an execution-oriented view of test projects.

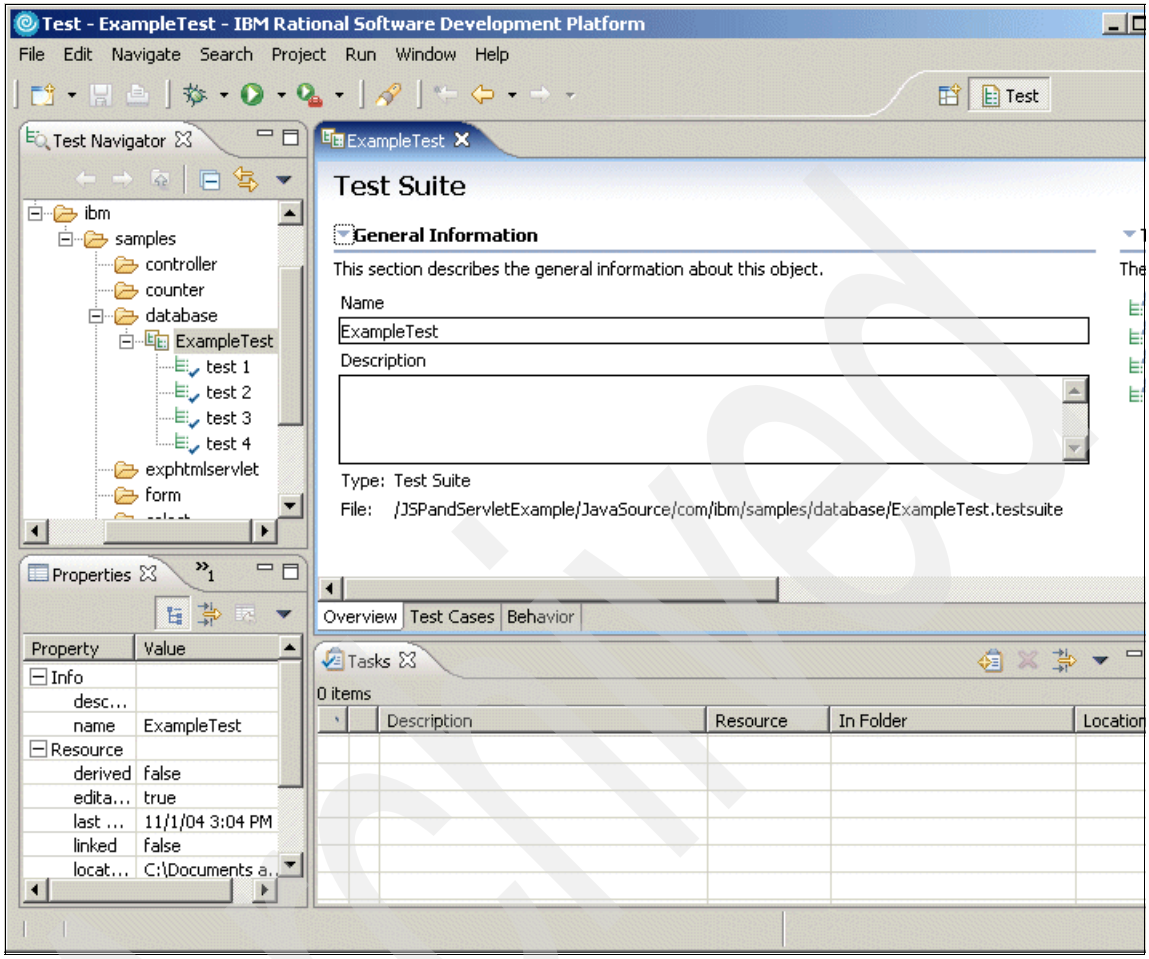


Figure 4-22 Test perspective

More information about Component Testing is located in Chapter 20, “JUnit and component testing” on page 1081.

The Tasks, Properties, and Outline views have already been covered in this chapter.

4.2.14 Web perspective

Web developers can use the Web perspective to build and edit Web resources, such as servlets, JSPs, HTML pages, Style sheets, and images, as well as the deployment descriptor file web.xml (Figure 4-23 on page 163).

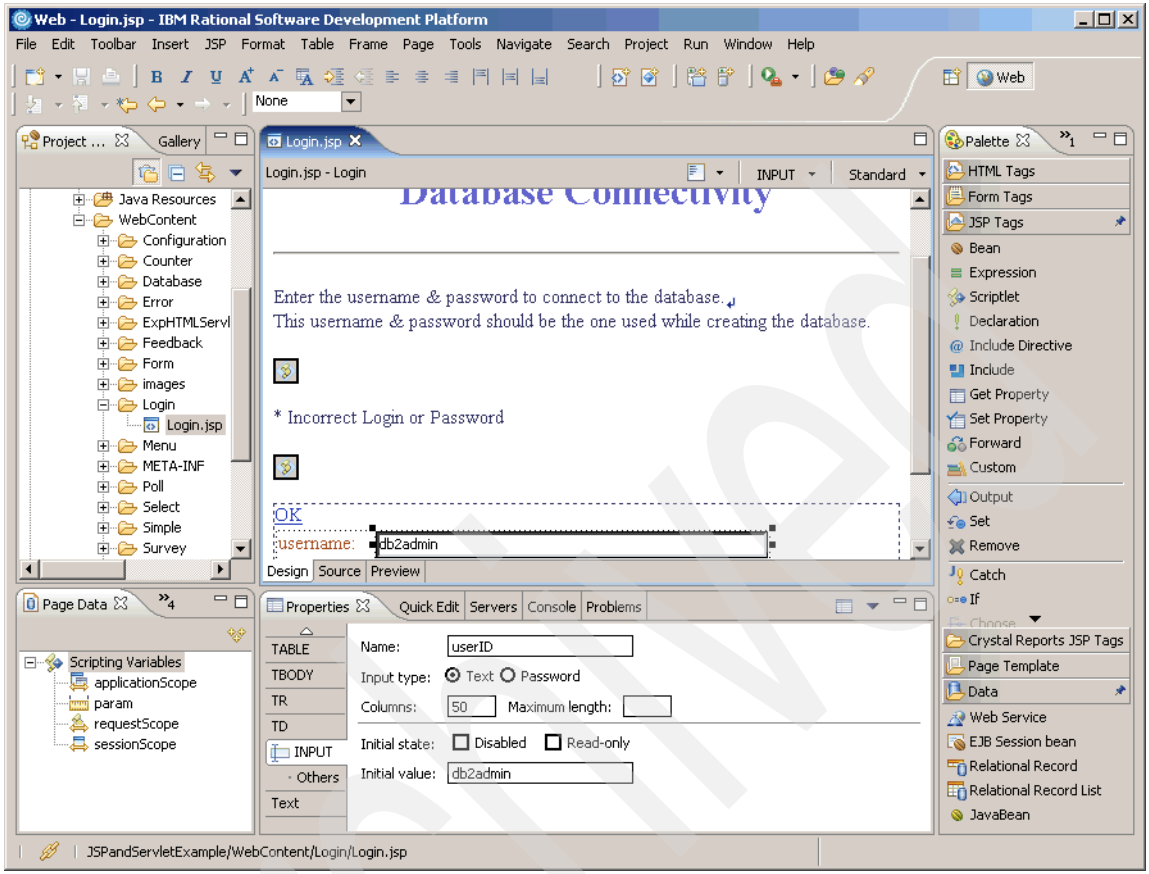


Figure 4-23 Web perspective

The Web perspective contains the following views:

- Gallery view

  Contains a variety of catalogs of reusable files that can be applied to Web pages. The file types available include images, wallpaper, Web art, sound files, and style sheet files.

- Page Data view

  Allows you manage data from a variety of sources, such as session EJBs, JavaBeans, and Web services, which can be configured and dropped onto a JSP.

- Client Data view

Archive

This view relates to developing Web applications using JavaServer Faces (JSF). User interface components on JSF pages can be connected to server-side data. This view provides a mechanism for configuring the link.

- Styles view

  Provides guided editing for cascading style sheets and individual style definitions for HTML elements.

- Thumbnails view

  Shows thumbnails of the images in the selected project, folder, or file. This view is especially valuable when used with the Gallery view to add images from the artwork libraries supplied by Rational Developer to your page designs. When used with the Gallery view, thumbnail also displays the contents of a selected folder. You can drag and drop from this view into the Project Explorer view or the Design or Source page of Page Designer.

- Quick Edit

  Allows you to edit small bits of code, including adding and editing actions assigned to tags. You can drag and drop items from the Snippets view into the Quick Edit view.

- Palette view

  Contains expandable drawers of drag and drop objects. Allows you to drag objects, such as tables or form buttons, onto the Design or Source page of the Page Designer.

- Links view (as a fast view button in the status area)

  Shows the resources used by or linked to from the selected file and files that use or link to the selected resource. This view provides you with a way to navigate through the various referenced parts of a Web application and can be used as an overview of the structure of the application being developed. The button toggles the display of this view.

  Figure 4-24 on page 165 shows the Web perspective with the Links view toggled open.

Archived

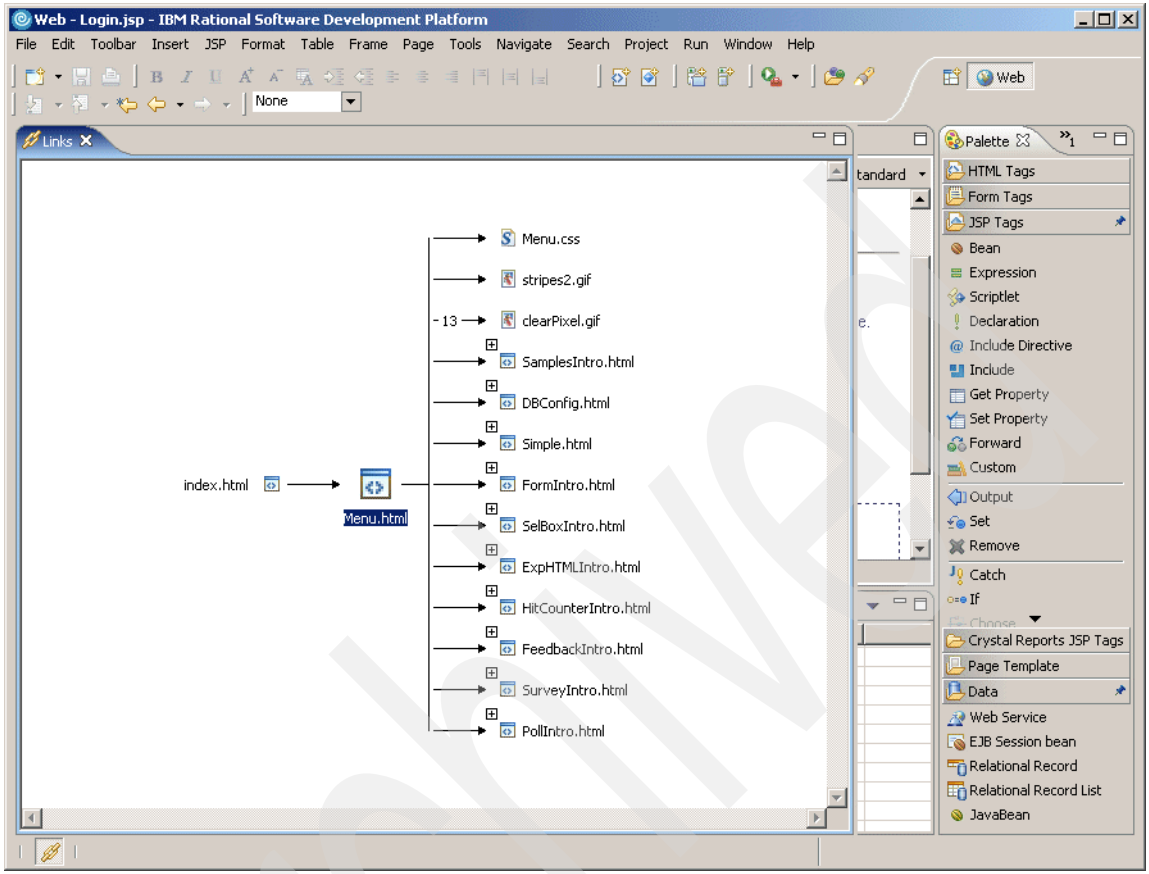


Figure 4-24 Web perspective - Links view

- Page Designer editor

  Page Designer allows you to work with HTML files, JSP files, and embedded JavaScript. Within the Page Designer, you can move among three pages that provide different ways for you to work with the file that you are editing. You can switch pages by clicking the tabs at the bottom of the editor pane. These pages work in conjunction with the Outline and Properties views, tool bar buttons, menu bar options, and context menus.

  – Design

    The Design page of Page Designer is the WYSIWYG mode for editing HTML and JSP files. As you edit in the Design page, your work reflects the layout and style of the Web pages you build without the added complexity of source tagging syntax, navigation, and debugging. Although many tasks can also be performed in the same way in the Source page, the

Design page provides full access to Page Designer menu options, context menu actions, view-specific GUI options (such as those in the Styles view), and drag and drop behavior.

– Source

  The Source page enables you to view and work with a file's source code directly. The Outline and Properties views both have features that supplement the Source page.

– Preview

  Shows how the current page is likely to look when viewed in a Web browser. JSPs shown in this view will contain only static HTML output.

The Project Explorer, Outline, Properties, Servers, Console, Problems, and Snippets views have already been discussed in this chapter.

4.2.15 Progress view

The Progress view is not part of any perspective by default, but is a very useful part of using Rational Application Developer. When Rational Application Developer is carrying out a task that takes a substantial amount of time, two options are available: The user looks at the dialog until the operation completes, or the user clicks the **Run in Background** button and the task continues in the background. If the second option is selected, Rational Application Developer runs more slowly, but the developer can carry out other tasks while waiting. Examples of tasks that might be worth running in the background would be publishing and running an enterprise application, checking a large project into CVS or rebuilding a complex set of projects.

As an example, we will see what happens when a project is published to a newly created server, which is then started. Right-clicking **index.html** and selecting **Run** → **Run on Server** begins the process. Once a server has been created or selected in the wizard, a dialog appears, as shown in Figure 4-25 on page 167.

The user can wait or click the Run in Background button. Clicking the button causes the process to minimize to the status area, shown as an icon in the bottom-right corner (Figure 4-26 on page 167). This section of the status bar now shows the name of the process being run in the background, a progress bar, and the “Shows background tasks in Progress view” button, which looks like a conveyor belt.

Clicking the **Shows background tasks in Progress view** button opens the Progress view, as shown in Figure 4-27 on page 168. This view lists all the active background processes and allows them to be monitored or controlled.

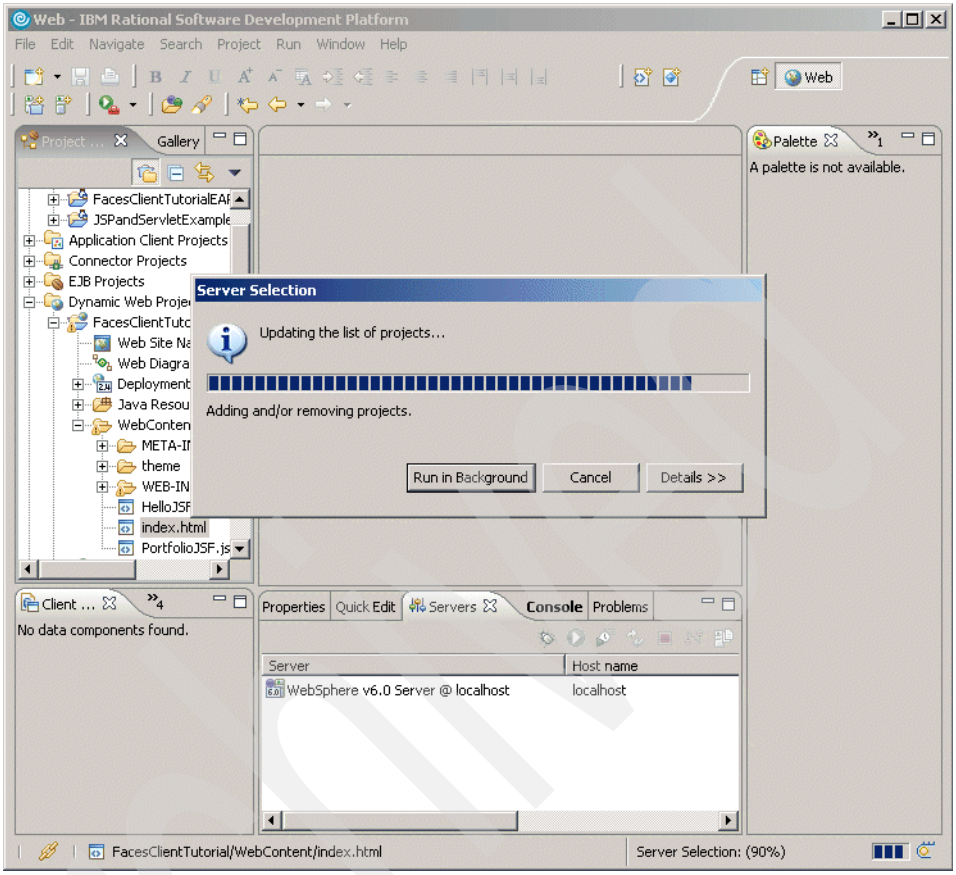


Figure 4-25 Blocking process

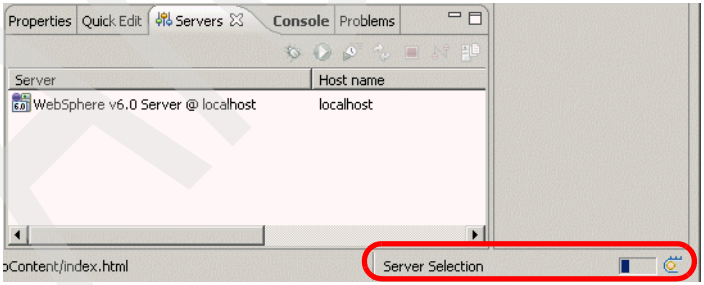


Figure 4-26 Process information in status bar

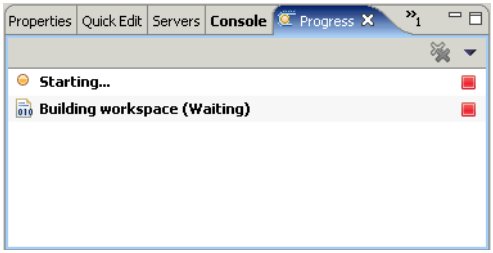


Figure 4-27 Progress view

4.3 Rational Product Updater

To obtain and install updates for Rational Application Developer, run the Rational Software Development Platform *Product Updater* tool, which is installed along with Rational Application Developer. This application will search for updates to Rational Software Development Platform products that are installed on the host computer and download them from IBM's update site. The interface is shown in Figure 4-28.

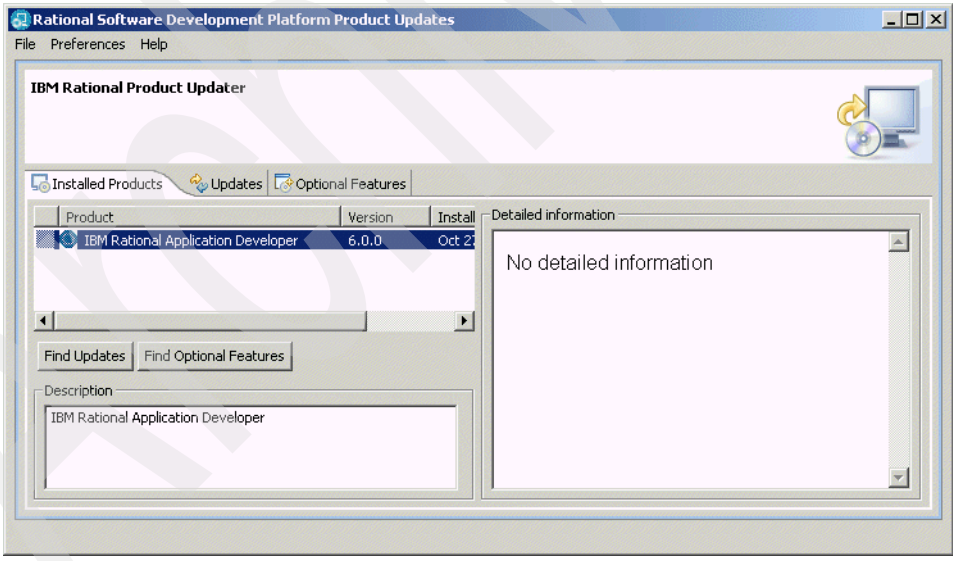


Figure 4-28 Rational Software Development Platform Product Updates

5

Projects

This chapter reviews J2EE packaging rules, explains how Rational Application Developer represents these rules, introduces the main project types, and looks at some of the options for configuring projects.

The chapter is organized into the following sections:

- J2EE architecture
- Projects and folders
- Rational Application Developer projects
- Creating a new project
- Project properties
- Rational Application Developer samples

Archived

© Copyright IBM Corp. 2005. All rights reserved.

5.1 J2EE architecture

The Java 2 Platform Enterprise Edition (J2EE) is a highly available, reliable, scalable, and secure platform for developing client-server applications in Java. In this book we are mainly concerned with J2EE V1.4.

The J2EE specification, along with many other resources relating to J2EE, is available at http://java.sun.com/j2ee/. The specification includes a description of the component model for developing J2EE applications and the runtime architecture. For more information on the programming technologies and how they are designed to interact, refer to Chapter 2, “Programming technologies” on page 31.

5.1.1 EAR files

Enterprise archive (EAR) files represent a J2EE application that can be deployed to WebSphere Application Server. The EAR file format is based on the Java archive (JAR) file format. An .ear file contains a deployment descriptor (application.xml), which describes the contents of the application and contains instructions for the entire application, such as security settings to be used in the runtime environment.

An EAR file contains the following modules (zero or more of each type—at least one module):

- Web modules - With a file extension of .war
- EJB modules - Packaged in a .jar file
- Application client modules - Packaged in a .jar file
- Resource adapter modules - Packaged in a .rar file

An EAR file can also contain utility JAR files required by other modules and other files as necessary.

5.1.2 WAR files

Web archive (WAR) files contain all the components of a Web application. These components will often include:

- HyperText Markup Language (HTML) files
- Cascading Style Sheets (CSS) files
- JavaServer Pages (JSP) files
- Compiled Java Servlet classes
- Other compiled Java classes
- Image files
- Portlets (portal applications)

WAR files also include a deployment descriptor (web.xml), which describes how to deploy the various components packaged within the Web module.

5.1.3 EJB JAR files

An EJB JAR file is used to package all the classes and interfaces that make up the EJBs in the module. The EJB JAR file includes a deployment descriptor (ejb-jar.xml), which describes how the EJBs should be deployed.

5.1.4 J2EE Application Client JAR files

A J2EE Application Client is packaged in a JAR file. The JAR file includes the classes required for the user interface for the application, which will run on a client machine and access resources provided by a J2EE server, such as data sources, security services, and EJBs. The file also includes a deployment descriptor (application-client.xml).

5.1.5 RAR files

A resource adapter archive (RAR) file is used to package J2EE resource adapters, which are designed to provide access to back-end resources using services provided by the application server to manage the lookup of the resource and to manage connections. Resource adapters are often provided by vendors of enterprise information systems to facilitate access to resources from J2EE applications.

The resource adapter may be installed as a stand-alone module within the server, to allow it to be shared among several applications, or as part of an application, in which case it will be available only to the containing application. The RAR file includes the Java classes, providing the Java-side of the resource adapter, and platform-specific code, providing the back-end connection functionality. The deployment descriptor is called ra.xml.

J2EE resource adapters are known by several names:

- Java 2 Connectors (J2C)
- J2EE Connector Architecture (JCA)
- J2EE Resource Adapter (RA)

5.2 Projects and folders

Rational Application Developer organizes all resources into projects, which contain the required files and folders. In the Workbench you can create different

Archived

kinds of projects with different structures. These will be described in 5.3, “Rational Application Developer projects” on page 173.

Unless otherwise specified, projects are created in Rational Application Developer’s workspace directory, which is normally chosen as the tool is started.

When projects are deleted, Rational Application Developer offers the choice of whether to `Also delete contents under <directory>` or `Do not delete contents` (this is shown in Figure 5-1).Choosing the second option, which is the default, removes the project from Rational Application Developer’s list of projects, but leaves the file structure of the project intact—the project can later be imported using the “Existing Project into Workspace” option for imports. A project that has been deleted from the workspace takes up no memory and is not examined during builds, so deleting projects in this way can improve the performance of Rational Application Developer.

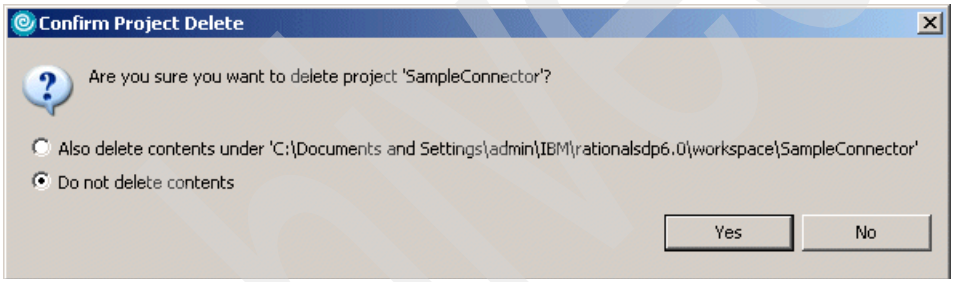


Figure 5-1 Confirm Project Delete dialog

Rational Application Developer complies with the J2EE specifications for developing components. The packaging rules, described above in 5.1, “J2EE architecture” on page 170, are only applied by Rational Application Developer when a J2EE application or module is exported. While these applications and modules are within Rational Application Developer, they are stored as projects in the same way as anything else. The relationship between the enterprise application projects and the modules they contain is managed by Rational Application Developer and applied on export to produce a properly packaged .ear file. This arrangement, and the correspondence between applications, modules and projects is shown in Figure 5-2 on page 173.

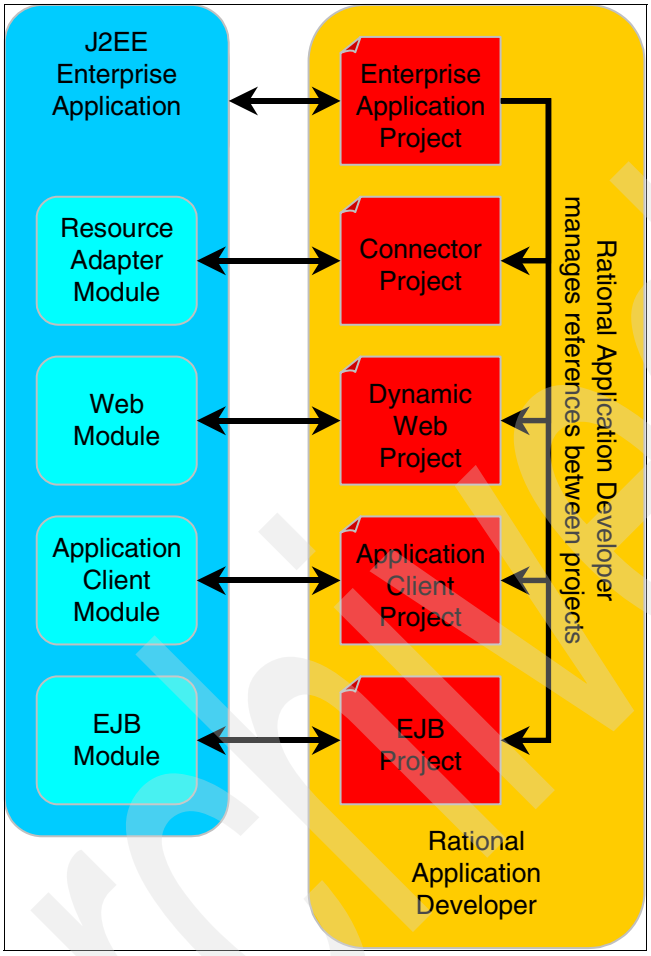


Figure 5-2 Relationship between J2EE modules and projects

5.3 Rational Application Developer projects

Table 5-1 shows the types of project for which Rational Application Developer provides creation wizards.

Table 5-1 Project types in Rational Application Developer

Enterprise Application Project	J2EE Application Client Project
Dynamic Web Project	Static Web Project
EJB Project	Connector Project

Archive

Java Project	Simple Project
Server Project	Component Test Project
EMF Project	Checkout Projects from CVS
Feature Patch	Feature Project
Fragment Project	Plug-in Project
Update Site Project	

EMF, Feature, Fragment, Plug-in, and Update Site Projects and Feature Patches are concerned with extending the functionality of the Workbench, which is beyond the scope of this book. This section does cover the other project types.

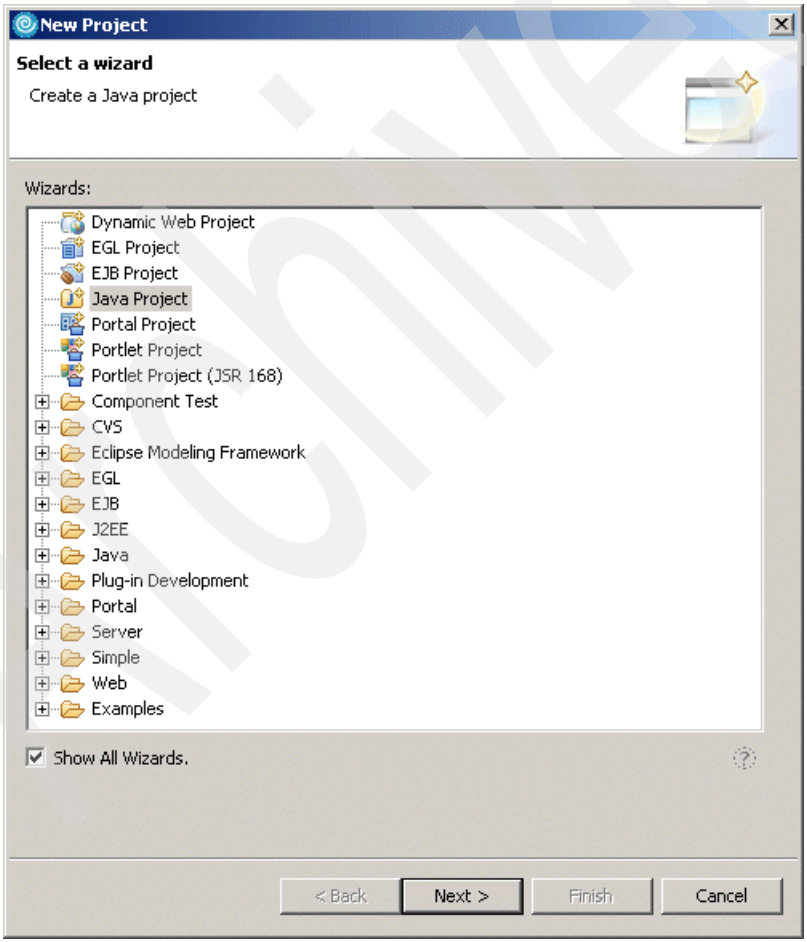


Figure 5-3 New Project dialog

Archived

To create a project, do the following:

1. Select **File** → **New** → **Project...**.
2. This displays the New Project dialog, which lists all the project types for which wizards exist. This dialog is shown in Figure 5-3 on page 174. Check **Show All Wizard**. Select the desired project type and click **Next >** to start the wizard.

5.3.1 Enterprise Application project

Enterprise Application projects contain references to the resources needed for enterprise applications and can contain a combination of Web modules, EJB modules, application client modules, resource adapter modules, utility Java projects, and JAR files.

The wizard for creating an enterprise application project also includes an option for creating one of each type of contained module (dynamic Web, EJB, application client, and connector). The J2EE level and target server can also be specified.

For more information on developing J2EE enterprise applications, see Chapter 23, “Deploy enterprise applications” on page 1189.

5.3.2 J2EE Application Client project

J2EE Application Client projects contain the resources needed for J2EE application client modules. An application client module is used to contain a full-function client Java application (non Web-based) that connects to and uses the J2EE resources defined in your server, such as the Java Naming and Directory Interfaces (JNDI) service provided by the server and, through JNDI, EJBs, and data sources.

The wizard allows the J2EE version, target server, and containing EAR file to be specified.

5.3.3 Dynamic Web Project

A Dynamic Web Project contains resources needed for Web applications, such as JSPs, Java Servlets, HTML, and other files. The Dynamic Web Project wizard provides the capability to configure the version of the Java Servlet specification, target server, EAR file name, and context root. The wizard also allows various other features to be added to the dynamic Web Project:

- A CSS file
- Struts support

Archived

- A Web diagram
- JSP tag libraries
- Crystal Reports

The dynamic Web Project can also be built using a template—various samples are provided or user-defined templates can be created.

For more information on developing Web applications, see Chapter 11, “Develop Web applications using JSPs and servlets” on page 499.

5.3.4 Static Web Project

A Static Web Project contains only static Web content, such as HTML pages, images, sounds, and movie files. These resources are designed to be deployed to a Web server and do not require the services of a J2EE application server. The wizard allows the CSS file for the project to be selected and can build the static Web Project from a template.

For more information on developing Web applications, see Chapter 11, “Develop Web applications using JSPs and servlets” on page 499.

5.3.5 EJB project

EJB projects contain the resources for EJB applications. The EJB project contains classes and interfaces that make up the EJBs, the deployment descriptor for the EJB module, IBM extensions and bindings files, and files describing the mapping between entity beans in the project and relational database resources.

The wizard allows the EJB version, target server, and containing EAR file to be specified. In addition, support can be added for annotated Java classes (an extension to EJB V2.1, which looks forward to the EJB V3.0 specification), and the wizard can create a default session bean, called DefaultSession. An EJB Client JAR can also be created, which includes all the resources needed by client code to access the EJB module (the interfaces and stubs).

For more information on developing EJBs, see Chapter 15, “Develop Web applications using EJBs” on page 827.

5.3.6 Connector project

A Connector project contains the resources required for a J2EE resource adapter. The wizard allows the JCA version, target server, and containing EAR file to be specified. Unlike most of the projects discussed in this chapter, there is no explicit support for developing J2EE resource adapters.

5.3.7 Java project

A Java project contains Java packages and Java code as .java files and .class files. Java projects have an associated Java builder that incrementally compiles Java source files as they are changed. Java projects can be exported as JAR files or into a directory structure.

Java projects are used for stand-alone applications or to build utility JAR files for an enterprise application.

For more information on developing Java applications, see Chapter 7, “Develop Java applications” on page 221.

5.3.8 Simple project

A Simple project in Application Developer does not have any default folders or required structure.

5.3.9 Server project

A Server project stores information about test servers and their configurations. If all testing will be performed on the default test server, no server project is required, but if you want to test your EJB or Web components on another type of server, the configuration information is stored in a server project.

For more information on working with server configurations, see Chapter 19, “Servers and server configuration” on page 1043.

5.3.10 Component test project

A component test project serves as a grouping mechanism for the test artifacts that are generated when you create a component test. Test projects also serve to define the scope of the test project by limiting the number of files that will be analyzed when you create tests or stubs, especially when you have many projects or large projects in your workspace. This can be performed as you go through the wizard.

For more information on testing, see Chapter 20, “JUnit and component testing” on page 1081.

5.3.11 Checkout projects from CVS

This wizard allows you to copy projects from a Concurrent Versions System (CVS) repository to your local workspace. The wizard takes you through the

Archived

process of connecting to the CVS repository server and allows you to specify a project to check out by name or to browse for it from a list provided by the server.

For more information on using CVS, see Chapter 26, “CVS integration” on page 1299.

5.4 Creating a new project

You normally start developing a new application by creating one or more projects. The Workbench provides wizards to create each specific type of project.

As an example of running through the process of creating a new project, we will look at the New Enterprise Application Wizard.

1. To launch this wizard, select **File** → **New** → **Project**.
2. Select **J2EE** → **Enterprise Application Project**.

   The New Enterprise Application Project wizard is shown in Figure 5-4.

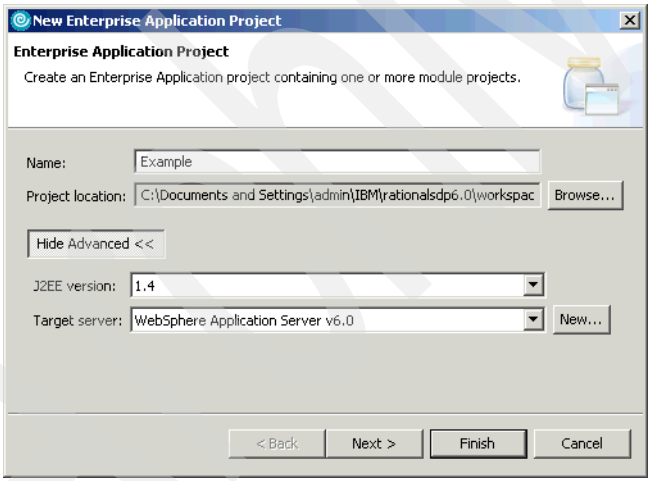


Figure 5-4 New Enterprise Application Project wizard page 1

In this case, the things that can be specified are:

- Name: The project name—Example in this case.
- Location: By default, projects are stored in a subdirectory that is created for the project in the workspace directory, but another location may be specified.
- J2EE version: Here, 1.2, 1.3, or 1.4 can be selected.

– Target server: The options in the drop-down list will depend on the test servers that have been installed.

3. The second page of the wizard is shown in Figure 5-5. If J2EE module projects are already in the workspace, they are listed in this dialog and can be added to the project as it is created. If you want to create some basic modules as part of creating the enterprise application project, click **New Module...**. The resulting dialog is shown in Figure 5-6 on page 180.

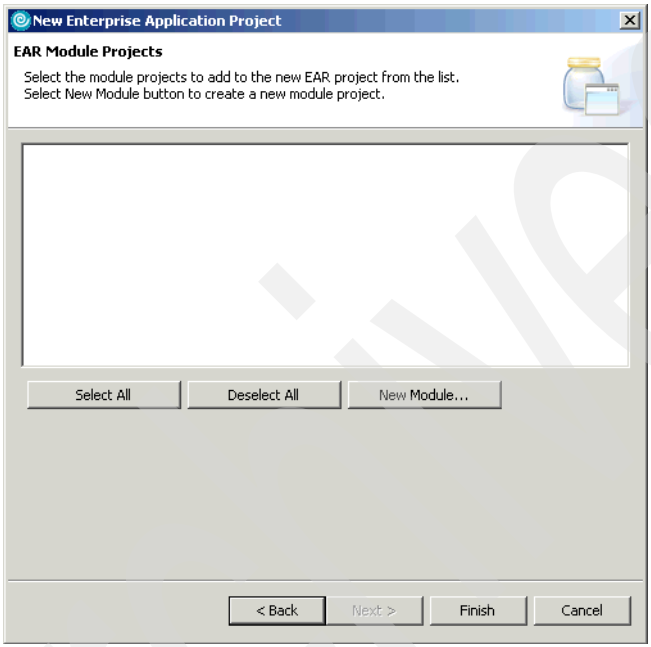


Figure 5-5 New Enterprise Application Project wizard page 2

Archived

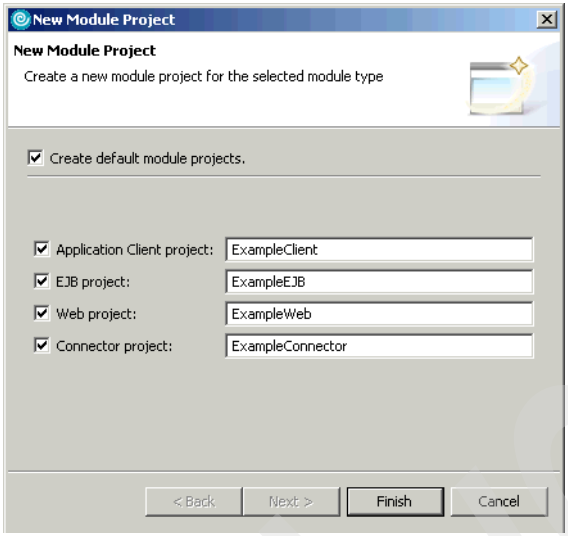


Figure 5-6 New Module Project dialog

4. Select the check boxes for the projects you wish to create, change the names if desired, and click **Finish**. Click **Finish** again on the New Enterprise Application Project wizard.

If the project you have created is associated with a particular perspective (as in this case, with the J2EE perspective), Rational Application Developer will offer to switch over to the perspective concerned.

5.5 Project properties

To make changes to the properties of a project, right-click the project and select **Properties** from the context menu. Figure 5-7 on page 181 shows the properties dialog for an EJB project.

Archived

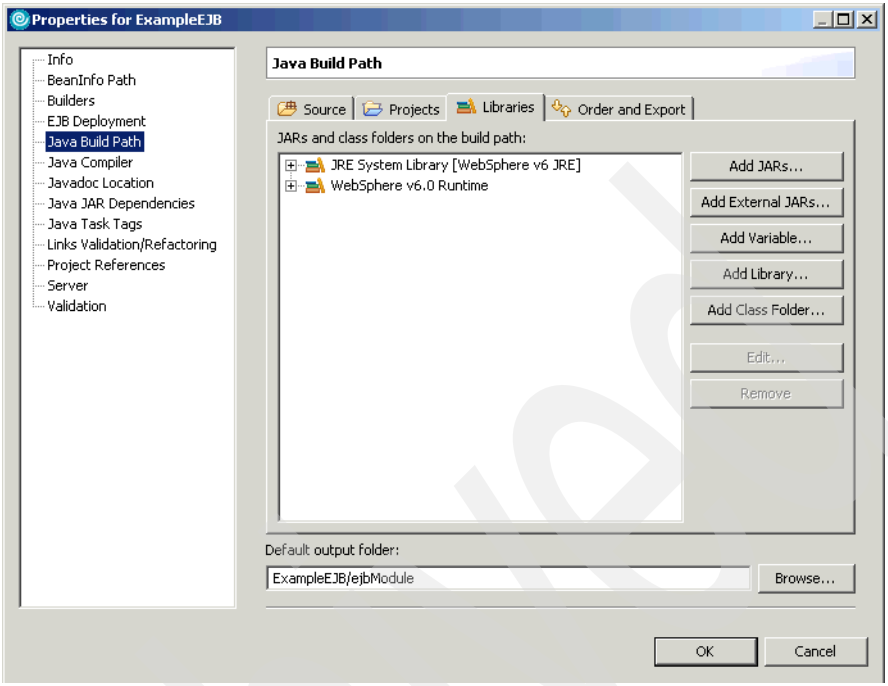


Figure 5-7 Project properties

In the Properties dialog you may want to change:

- Java Build Path: Project dependencies and JAR files.
- Server: Which test server to start when testing the project.
- Validation: Which validation tools should run after making changes.
- Java JAR Dependencies: Configure project dependencies and classpath entries.

Different types of projects have different options available in the Properties dialog.

5.6 Rational Application Developer samples

Rational Application Developer provides a wide range of sample applications that can help you to explore the features provided by the software development platform and the technologies supported by it.

The simplest way to access the samples is through the Welcome page for Rational Application Developer.

1. This page is shown when the product is first started. (If you have closed it, it can be reopened by selecting **Help** → **Welcome)**. The Welcome page is shown in Figure 5-8.

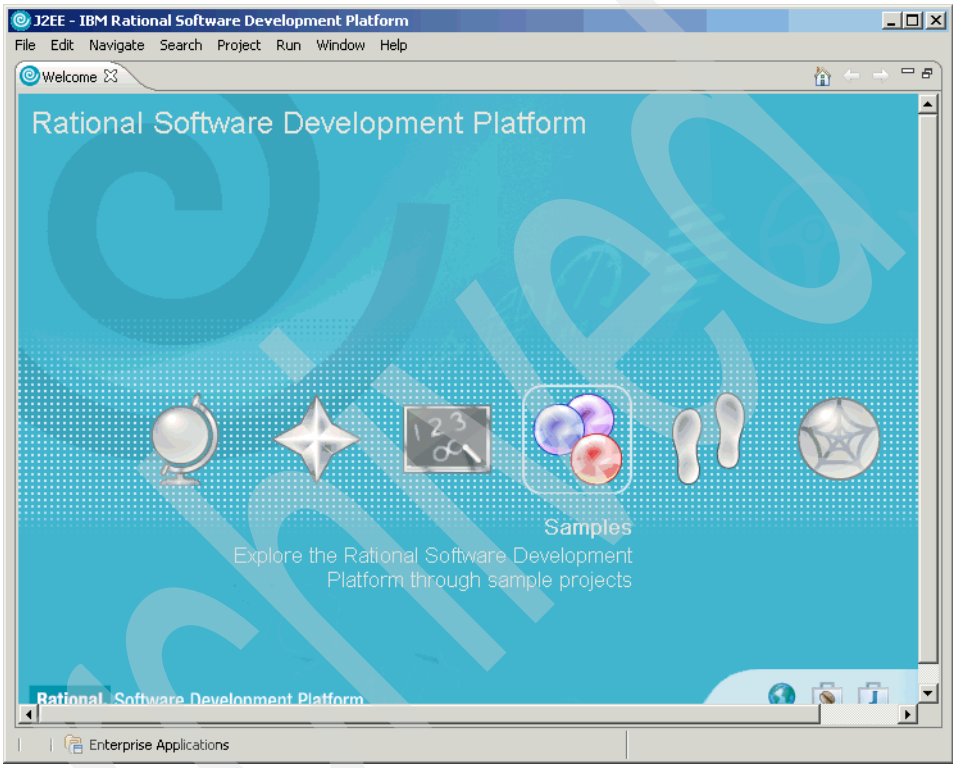


Figure 5-8 Welcome page

2. Click the **Samples** button to display the Samples page.
3. This page is shown in Figure 5-9 on page 183. It includes a few sample applications that can be imported into Rational Application Developer, but a far larger number can be accessed by clicking the **Launch the Samples Gallery** button.
4. This opens a separate Samples Gallery window. The Samples Gallery can also be opened by selecting **Help** → **Samples Gallery**.

Archived

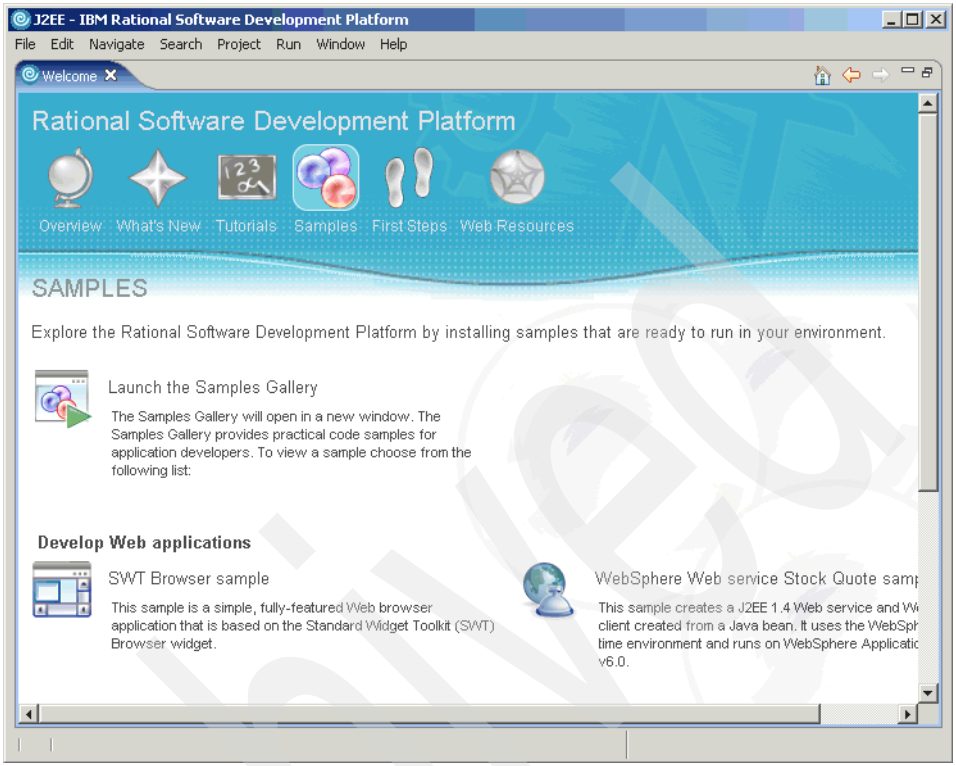


Figure 5-9 Samples page

5.6.1 The samples gallery

The samples gallery lists all the samples available in Rational Application Developer. At the time of writing, this included 42 samples. The samples are arranged in three main categories:

- Showcase samples: The most extensive, robust samples provided, consisting of end-to-end applications that follow best practices for application development
- Application samples: Applications created using more than one tool or API, showing how different tools within Rational Application Developer interact with each other
- Technology samples: More granular, code-based samples that focus on a single tool or API

The samples can be selected from a hierarchical list in the left-hand pane, as shown in Figure 5-10 on page 184.

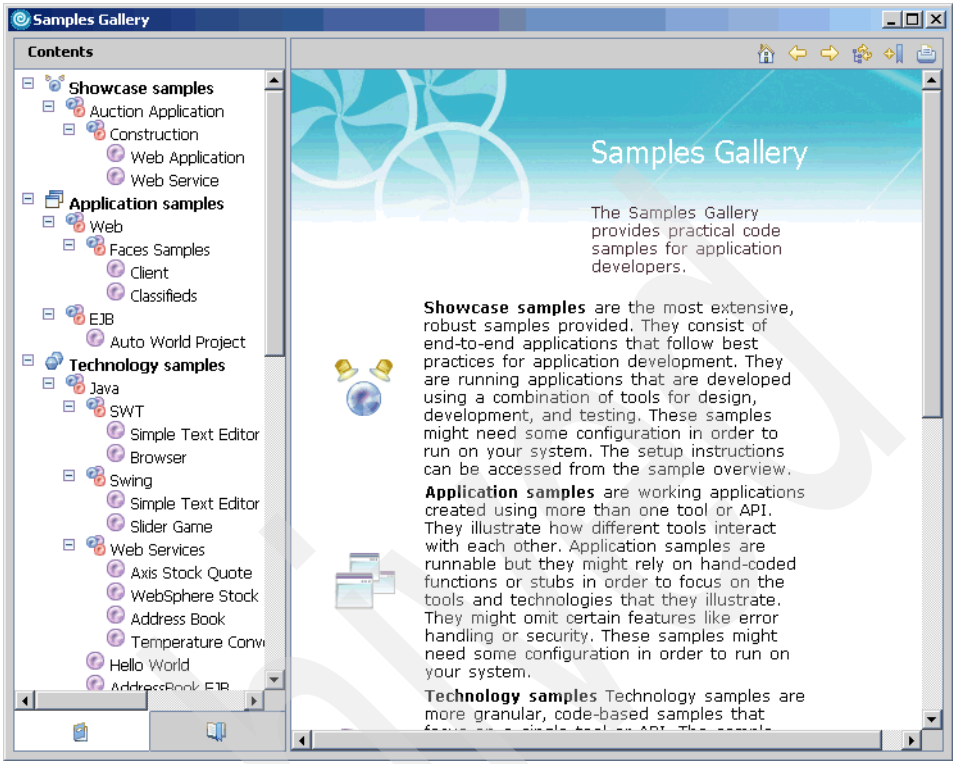


Figure 5-10 Samples Gallery

As an example, the Web Application Showcase sample is an auction application. The page for this is shown in Figure 5-11 on page 185. The Setup instructions link displays a page of instructions on how to configure and run the sample application. The Import the sample link will import the sample application into Rational Application Developer as a project (or, in this case, set of nine projects) so that it can be run on a server and tested.

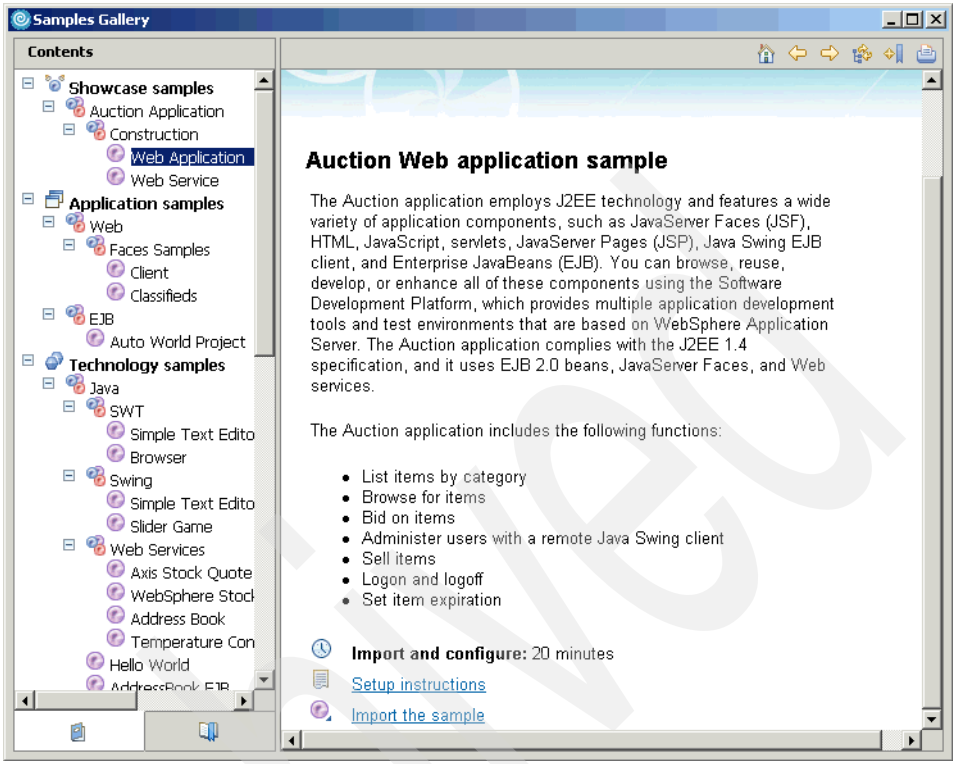


Figure 5-11 Auction Web application sample

For a detailed example of how to import a sample, see 21.2, “Prepare for the sample” on page 1131.

Archive

Archived

Part 2

Develop applications

Archived

© Copyright IBM Corp. 2005. All rights reserved.

Archived

6

RUP and UML

In this chapter we illustrate how a software development project can be assisted by using the Rational Unified Process (RUP) tooling provided within Rational Application Developer. In addition, we demonstrate how the visual UML tooling included in Application Developer can be used within the context of RUP with an emphasis on the implementation phase.

The chapter is organized into the following sections:

- Overview
- Rational Unified Process (RUP)
- Visualize applications with UML
- More information on UML

Archived

© Copyright IBM Corp. 2005. All rights reserved.

6.1 Overview

The implementation phase takes the design and implements code, which can be transformed into one or more executable applications.

We are assuming that at this point at least early versions of the use case model and architectural design are available to us. By now we should have a good idea of how the use cases will be realized, and we can base our implementation work on the design model. Note that since RUP is an iterative process, these artifacts are further updated during each iteration. It is important that the development team and the tools are well integrated.

Rational Application Developer is primarily aimed at the Developer working at the implementation phase, but is also well suited for roles and tasks via the integrated capabilities provided by the Rational Software Development Platform.

The UML visualization capabilities provided with Rational Application Developer fall under the category of code visualization and visual editing. This will be the focus in the second part of this chapter. If you are more interested in the analysis and design discipline, then we recommend looking into the Rational Software Architect and Rational Software Modeler products.

6.2 Rational Unified Process (RUP)

The Rational Unified Process (or simply RUP) is a software engineering process. It provides a disciplined approach to assigning tasks and responsibilities within a development organization. Its goal is to ensure the production of high-quality software that meets the needs of its end users within a predictable schedule and budget.

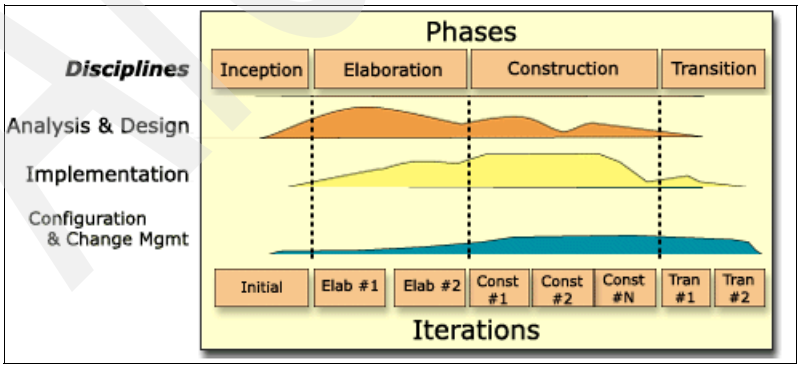


Figure 6-1 RUP overview

Figure 6-1 on page 190 illustrates the overall RUP architecture, which has two dimensions:

- Lifecycle phases

  The horizontal axis represents time and shows the life cycle aspects of the process as it unfolds. This dimension illustrates the dynamic aspect of the process as it is enacted and is expressed in terms of phases, iterations, and milestones.

- Disciplines

  The vertical axis represents disciplines that logically group activities by nature. This dimension portrays the static aspect of the process—how it is described in terms of process components, disciplines, activities, workflows, artifacts, and roles.

The graph in Figure 6-1 on page 190 shows how the emphasis varies over time. For example, in early iterations you spend more time on requirements; in later iterations more time is dedicated towards implementation.

You can access and configure process information directly in Rational Application Developer, using the following features:

- Process Advisor
- Process Browser
- Setting process preferences

6.2.1 Process Advisor

The Process Advisor provides contextual process guidance on model elements within diagrams.

When you select elements in a UML diagram, the system conducts a dynamic search of RUP content based on the context of your selection and the settings on the Process page of the Preferences window. The results of the search are displayed as topic links in the Process Advisor. Clicking a link opens the topic content in the Process Browser view.

To launch the Process Advisor directly, select **Help** → **Process Advisor**.

You can also specify the default set of roles you are performing and the topics and the type of content that you are interested in (see 6.2.3, “Setting process preferences” on page 193).

A search can be initiated by selecting **Search** → **Search** from the Workbench. Then click the **Process Search** tab.

Archive

From there you can search process information, but you also have access to many kind of searches over your workspace or Java resources (see Figure 6-2). If you are connected to the network then you may also search the IBM developerWorks Web site for related information.

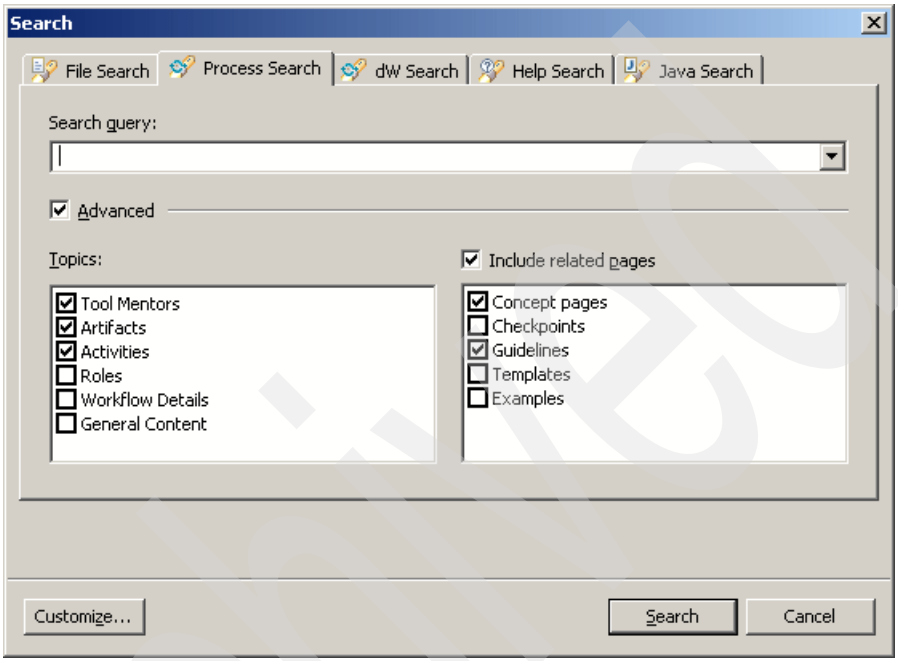


Figure 6-2 Process Search

6.2.2 Process Browser

The Process Browser displays the full set of RUP content from the installed RUP Configuration and provides navigation to topics with the use of three tabs:

- Process view
- Search Results
- Index

To launch the Process Browser, select **Help** → **Process Browser**.

A *Process view* is the hierarchical set of process elements represented in and displayed by the RUP content tree in the view pane, and associated with a particular role or major category. Each process view is represented by a tabbed pane, as shown in Figure 6-3 on page 193.

You can manage the views of these process elements by using the toolbar buttons at the top of the window.

You can perform the following tasks in the Process Browser:

- Browse topics in the process view.
- Browse the index.
- Search for words or phrases.
- Customize RUP process content by creating tailored process views.

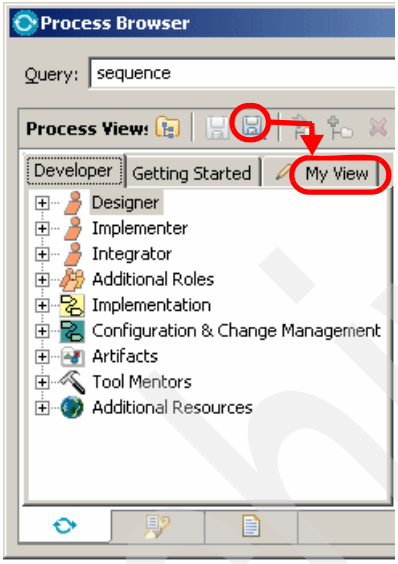


Figure 6-3 Process Browser views

The Process Browser comes with one pre-defined RUP configuration for the Developer role. You can customize this configuration by creating your own view. Click either the **Developer** or the **Getting Started** view, and then click **Save Tree As** to create your own copy. That view can then be modified by adding nodes or selecting the desired information for display.

The Welcome view includes a tutorial that demonstrates the customization steps.

6.2.3 Setting process preferences

As installed by default, the RUP Configuration provides basic process guidance to help you get started. This configuration is a subset of the full process information available in the standard configuration of the RUP methodology.

If you want to, it is easy to switch to another RUP configuration by selecting **Window** → **Preferences** → **Process**, as seen in Figure 6-4 on page 194.

Archived

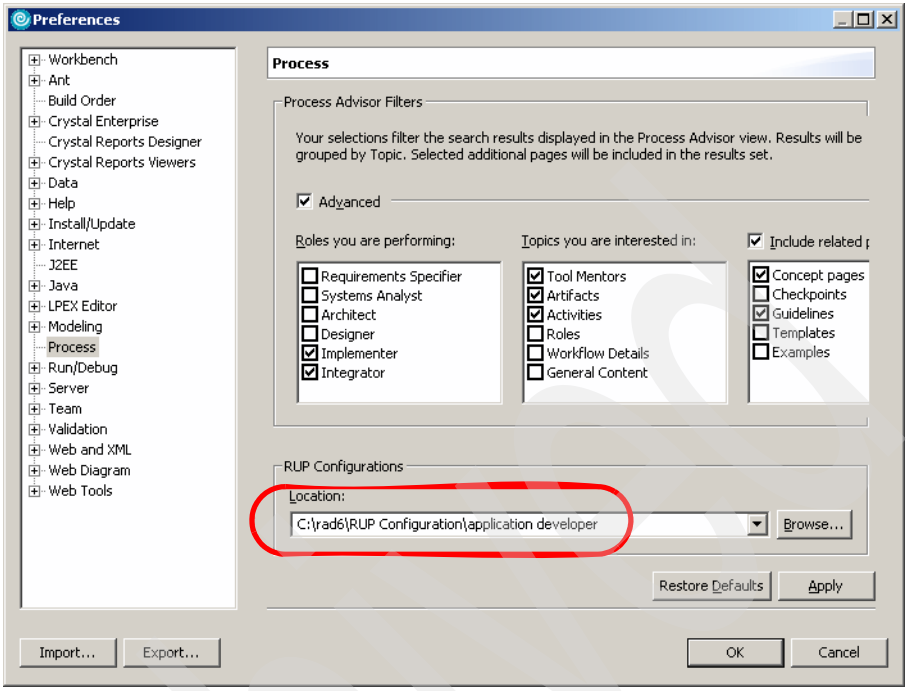


Figure 6-4 Process preferences window

6.3 Visualize applications with UML

There are several types of visualization available with Rational Application Developer. As in the prior releases, IBM Rational Application Developer V6 still offers J2EE and Java visualization capabilities. This section touches on some of the new features associated with each of these types of visualization. Also, new in this release is support for data visualization. Data visualization enables developers to edit and view their database schemas in a variety of diagram notations.

There are some visualization semantics and actions that are unique to the type of artifacts that are being visualized such as J2EE, Java, or data. However, regardless of the type of project, the visualization capabilities allow you to examine existing projects or build a new project from the start.

To get started using the visualization diagrams you can create the particular diagram manually by selecting **File** → **New** → **Other** → **Modeling** → <modeling_type>. Where the <modeling_type> can be one of the following:

- Class Diagram

- Sequence Diagram
- Static Method Sequence Diagram
- Topic Diagram
- E-R Modeling
  - IDEFIX Diagram
  - IE Diagram

In some cases the diagram will be generated by default by running through a particular wizard.

UML visualization features provided by Rational Application Developer offer the advantage of code visualization and visual editing.

Visual editing offers the ability to build code without explicitly typing the code in a text editor, but rather adding attributes and methods through the diagram editor in a visual manner.

The code visualization capabilities allow developers to view an existing code base as either class or sequence diagrams. This type of visualization helps developers understand and communicate application code by showing relationships between components, and also provides insight into items included with deployment descriptors as is the case with J2EE visualization. The diagrams used to visualize code can also be copied into design documents and used in design and code reviews.

6.3.1 Unified Modeling Language (UML)

Rational Application Developer uses Unified Modeling Language (UML) V2.0 for the diagrams.

UML is a notation for object-oriented analysis and design. It is standardized by the Object Management Group (OMG).

UML 2.0 contains 13 types of diagrams, up from 9 in UML 1.x. Each type of diagram shows different aspects of the system being modeled. The diagrams are divided into two major groups:

- Static diagrams focus on the structure of the system.
- Dynamic diagrams are used to describe the behavior of the system.

In Rational Application Developer, you can create UML visualization diagrams to examine and design the artifacts of your J2SE, J2EE, and database applications.

UML diagrams are a powerful way to visualize the structure and behavior of your application. A UML diagram can depict some or all of the components in an application. You can use a UML diagram to create your own context to examine,

Archived

understand, collaborate, and design, using a subset of the components in a project.

You can create UML diagrams to show a high-level view of Java, Enterprise JavaBeans, and (with this release) database applications. You can use these diagrams, which are a visual abstraction of the system's architecture, to identify a system's key components and relationships. These diagrams can help you maintain existing application components or create new ones.

Rational Application Developer provides the following diagrams for UML visualization:

- Browse Diagram: A temporary, non-editable diagram that provides a quick way to explore existing relationships between elements in an application. Great for quick exploring and lookup jobs.
- Topic Diagram: A non-editable diagram that provides a quick way to show existing relationships between elements in an application. You can think of the topic diagram as an enhanced browse diagram. You can customize the underlying query, open multiple topic diagrams at the same time, and save them away for further use.
- Static Method Sequence Diagram: A non-editable diagram that provides a quick way to illustrate the logic inside a method. Works much like the topic diagram, but is looking at a single method only.
- Class Diagram: Shows a collection of static model elements such as classes and types, their contents, and their relationships.
- Sequence Diagram: Shows the interactions between objects with focus on the ordering of the interactions. The interactions are represented by messages sent from one object to another. The messages can be conditional and can be iterated.

Data visualization is new in Rational Application Developer. There are several types of diagrams that can be associated with the data being visualized in the relational database. If you are familiar with UML visualization already, you are probably already familiar with the class diagram. Rational Application Developer also provides two other diagram types that differ only in the notation that is used to diagram the database schema:

- IE notation diagram: Information Engineering (IE) is a widely practiced and well-established data modeling notation. IE and has been around for many decades, before the object-oriented modelling started.
- IDEF1X notation diagram: IDEF1X also supports data modelling, capturing a logical view of the enterprise data. IDEF1X is based on an Entity Relationship model, and it is intended for logical database design.

The basic elements of IDEF1X are an *entity* (referring to a collection), *attribute* (associated with each member of the set), and the *classification structure* (for modelling logical data types).

For more details about data visualization, refer to Chapter 8, “Develop Java database applications” on page 333.

In addition to the above UML visualization, Rational Application Developer also includes a special diagram type for designing the application Web site. A *Web Diagram* shows the application flow of a Faces-based or Struts-based Web application. This diagram is not UML based, and is discussed separately in Chapter 12, “Develop Web applications using Struts” on page 615, and Chapter 13, “Develop Web applications using JSF and SDO” on page 673.

6.3.2 Browse diagram

You can use browse diagrams to explore artifacts in your workspace, without having to create, persist, and maintain diagrams. They are not meant for design and editing, but you can save them as regular UML diagram files (.dnx) and then edit those diagrams normally afterwards.

At any one time, there is no more than one copy of the Visualizer Browse Diagram open. When you explore the next element, this is displayed in the same Visualizer Browse view. However, you can scroll to the previous and next diagrams using the Visualizer tool bar.

As mentioned, there is no editing capability, and you cannot change the appearance or layout of a browse diagram. However, you can filter the relationships that the diagram shows, and you can control the depth to which an element is queried. Zooming in and out, and the outline view are also available.

1. Start Rational Application Developer.

   If desired, you can create a new workspace.

2. Install the Classifieds sample from the Samples Gallery.

   Click **Help** → **Welcome** → **Samples** → **Faces Classifieds sample.** Import the sample by following the instructions. We will use this source code for the visualization examples in this chapter.

3. To verify that the sample was imported correctly, you may want to check that you can start the application and see that it is working as expected.

Note: Occasionally we needed to manually start WebSphere Application Server V6.0 prior to run on server for the application to start successfully. We also noticed some scripting errors during the first application startup.

4. Let us examine the application using the available visualizing support. In Project Explorer, select and expand **Dynamic Web Projects** → **classifieds** → **Java Resources** → **JavaSource** → **beans**.
5. Right-click **Login.java** and select **Visualize** → **Explore in Browse diagram** from the context menu. A diagram similar to Figure 6-5 should appear. The browse diagram contains only the single element that we selected.
6. Let us find out which other elements refer to this Java class. Click the fourth arrow **References (Association)** so that it becomes selected, and click **Apply** (see annotations in Figure 6-5).

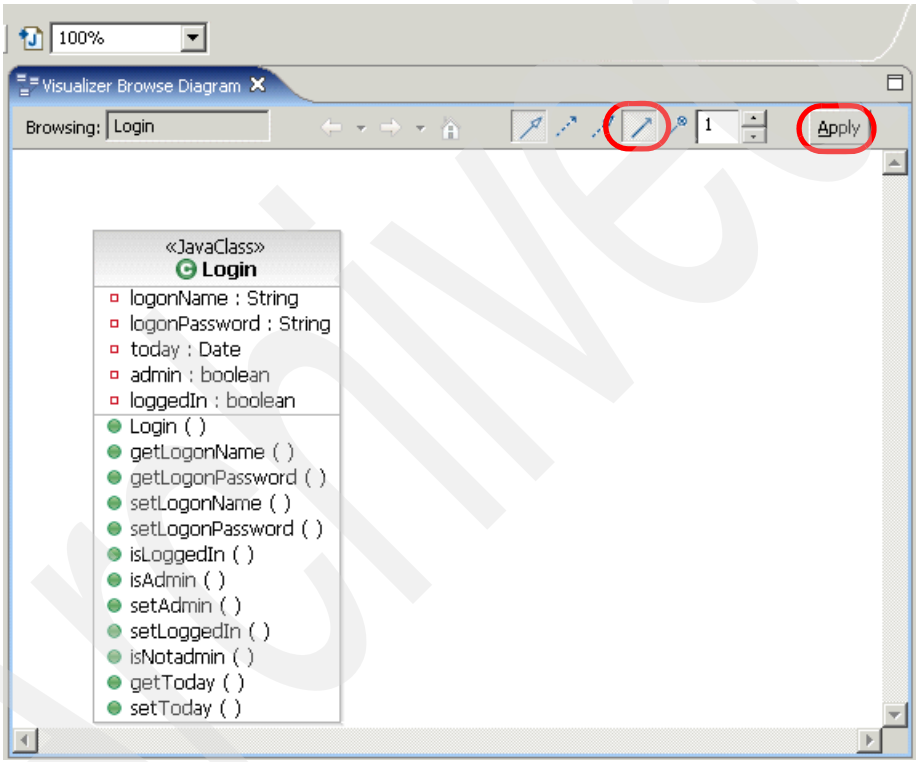


Figure 6-5 Visualizer Browse Diagram - Login.java

7. The diagram is updated (see Figure 6-6 on page 199), and we can see that there are five other classes that refer to the Login class. You may need to zoom in for a better view. You can also bring up the Outline view by selecting **Window** → **Show View** → **Outline**.
8. We can now continue browsing to other elements in this diagram simply by right-clicking the element and selecting the same menu option. Browse diagrams retain a history of what you browse. You can then use the menu bar

Archived

Back and Forward buttons to change context without creating a new browse diagram.

9. Another useful option on the menu bar is the auto-navigate to Java source button ( ) (found in the upper left part of the Workbench). When you turn that on, selecting an element in the diagram will automatically bring up the relevant source file.
10. Yet another nice feature is the navigation help. Suppose we wanted to know what the Results class is and where it is defined. Right-click **Results** and select **Navigate** → **Show in** → **Package Explorer** from the context menu**.** We can see that it is defined under the pagecode folder.
11. To explore an element in the diagram, simply double-click the element.
12. So far this is all temporary browsing and exploration activity. However, if you need to, you can also save the browse diagram as a file. The diagram is saved as a regular class diagram, and the class diagram editor is automatically launched (see Figure 6-6).

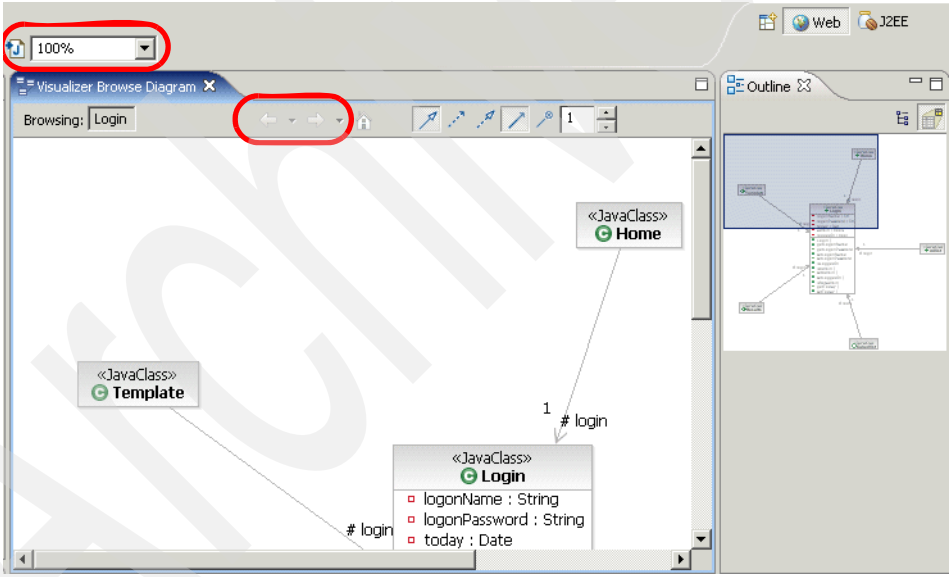


Figure 6-6 Visualizer Browse diagram and the Outline view

6.3.3 Topic Diagram

Topic Diagrams show the results of a query. They are similar to Browse Diagram since they are not editable either. There is an important distinction though. A Topic Diagram is created based on a context and a query. Only the context and

query are persisted in the diagram, so when the underlying elements change then this is automatically reflected in the diagram.

Because topic diagrams show the current state of your workspace when you create or open them, you can use them to generate diagrams for documentation or Javadoc reports.

A Topic Diagram is created in the context of a code project and is stored in the project as a diagram file with a .tpx file name extension. The appearance and layout of a topic diagram cannot be customized.

Let us look at an example how this works.

1. You can continue further exploration directly from the same browse diagram we had open before. Right-click **Results** again and select **Visualize in Topic Diagram** from the context menu.

   Or, if you already closed the browse diagram, you can alternatively expand the **JavaSource** → **pagecode** folder in the Explorer, right-click **Results.java**, and then select **Visualize** → **Add to New Diagram File** → **Topic Diagram** from the context menu.

2. When the Topic Wizard Location dialog appears, we entered the following (as seen in Figure 6-7 on page 201) and then clicked **Next**:

   – Parent folder: `classifieds/Design`
   – Filename: `Results topicsdiagram`

   Note: It is important to understand the ramification of selecting the parent folder when creating diagrams.

   In our example, we chose to create the diagrams in a separate folder off the root of the project. We observed the following behavior when working with Web Projects.

   - When creating the folder off the root of the project, the folder and diagrams will be preserved when using a Project Interchange file.
   - When creating the folder off the root of the project, the folder and diagrams will not be preserved when exporting to an EAR or WAR even if Export source files is checked.
   - When creating the folder in JavaSource, we found that each time diagram is saved, the builder will copy the diagram to the WEB-INF\classes folder. Since the WAR export utility will include all resources located in the WebContent tree, any diagrams located in the JavaSource tree will be included in the WAR file, regardless of the setting of the Export source files check box.

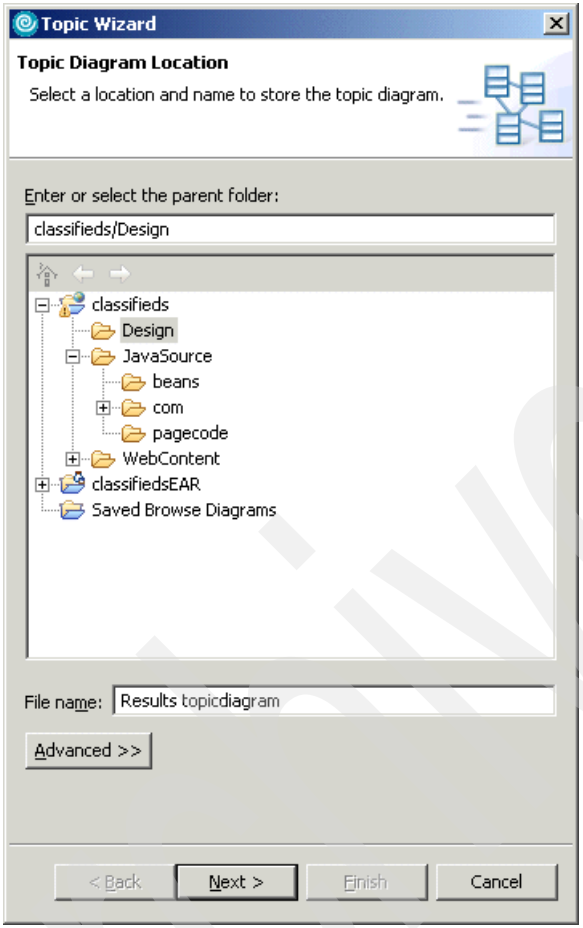


Figure 6-7 Creating a topic diagram

3. When the Topics dialog appears with a list of predefined queries, select **Utilized Java Elements** and then click **Next**.

4. The Related Visualizer Elements dialog appears (as seen in Figure 6-8 on page 202). First of all, this dialog basically shows the detail for our selection on the previous screen, and allows for modification. You can specify direction of search, the visual layout type (default/radial), and how many levels down the search should reach. Note that this particular query that we selected maps to the *Uses dependency* relationship. Accept the default and click **Finish**.

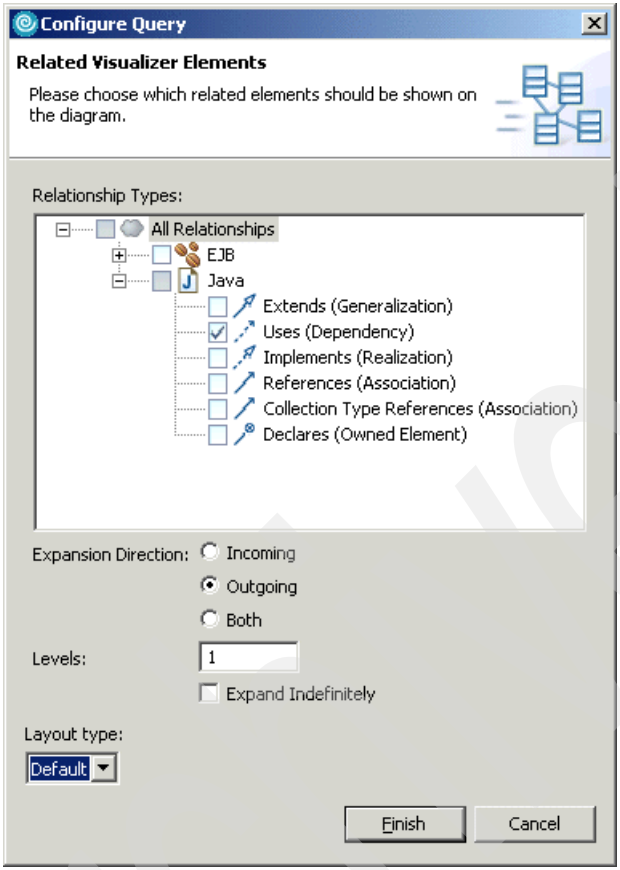


Figure 6-8 Creating a topic diagram: The related elements selection

5. The topic diagram is now created and displayed. You can see that Results uses Login and SearchBean beans to populate the page. Again, the editing functions on this diagram are disabled. But, you can easily go back to the query from the diagram context menu and change it with the Customize Query option.
   - You can also add more elements to it—or even an entire folder—by right-clicking and selecting **Visualize** → **Add to Current Diagram** from the context menu.
   - Topic diagrams are saved as .tpx files by default. You can turn them into regular edit diagrams by saving them, and then they become a user modifiable .dnx diagram file. For instance, you could then annotate the diagram with notes and text.

Note: All the topic diagrams can be accessed via the Diagram Navigator view. You can use this view to find and edit all the topic diagrams in your workspace.

6.3.4 Static Method Sequence Diagram

This diagram type generates a non-editable sequence diagram from a method of your choosing. The diagram models the flow of logic within the method in a visual manner, enabling you both to document and validate your logic. The static method sequence diagram can be used to explore the logic of a complex operation, function, or procedure.

Sequence diagrams are the most popular UML artifact for dynamic modeling, which focuses on identifying the behavior within the system.

Let us experiment with the Static Method Sequence Diagram with a simple example.

1. Again in the same browse diagram for the Login class, select the class and then the first method in the list, **Login()**.
2. Right-click the selection and select **Navigate** → **Open as Interaction** from the context menu (see Figure 6-9).

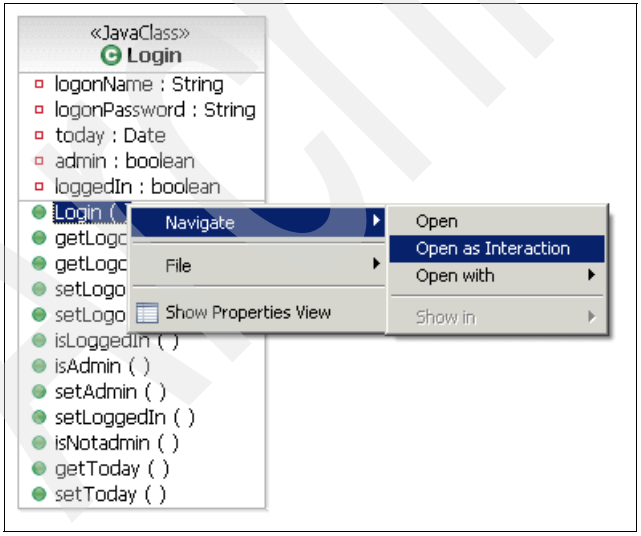


Figure 6-9 Creating static method sequence diagram

If you do not have the browse diagram open, an alternative way is to expand the class in the Explorer, right-click the method, and select **Visualize** → **Add**

to New Diagram File → **Static Method Sequence Diagram** from the context menu.

3. The diagram is created and displayed. The diagram file is also automatically stored under the same folder (..\classifieds\JavaSource\beans in our case), as sequencediagramN.tpx, where N is a running number. See Figure 6-10.

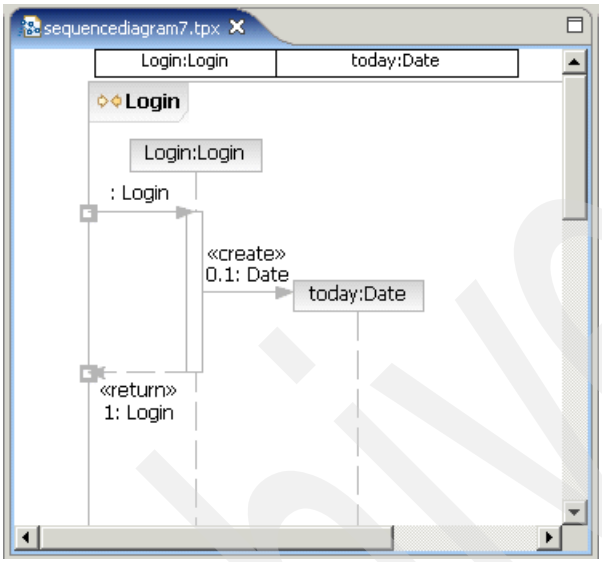


Figure 6-10 Static method sequence diagram for Login() method

4. This is a simple logic where we create a new date object and that is it. Try visualizing some other methods too, for instance the doBrowseFetchAction() method in the Categories class. This diagram contains some of the new UML2 features like the option combinations fragments (see Figure 6-11 on page 205).

Archived

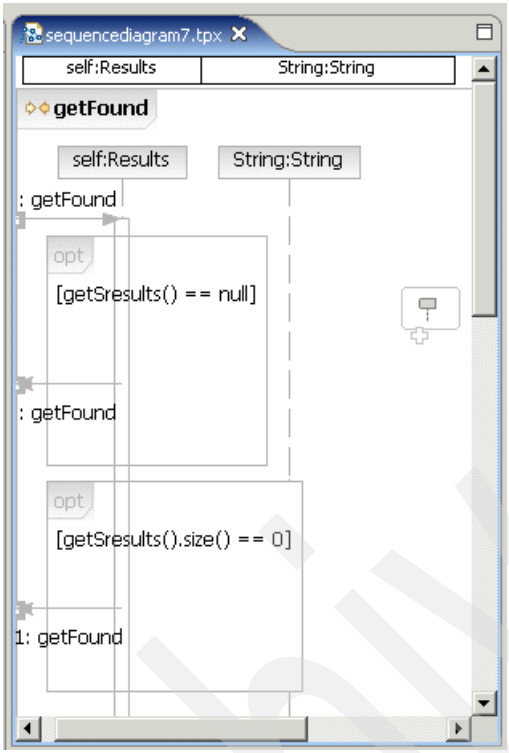


Figure 6-11 Static method sequence diagram with option fragments

6.3.5 Class Diagram

Class Diagrams are the mainstay of object-oriented analysis and design. UML 2 class diagrams show the classes of the system, their interrelationships (including inheritance, aggregation, and association), and the operations and attributes of the classes. Class diagrams are used for a wide variety of purposes, including both conceptual and domain modeling and detailed design modeling.

You may create class diagrams in many different ways in Rational Application Developer. If you are working in the Java perspective, class diagrams are available under the **New** → **Other** → **Modeling** → **Class Diagram** option in the context menu. You may also create a class diagram from the Diagram Navigator view. This is where all your diagrams are conveniently accessible in one place.

Attention: The examples found in this section are intended to highlight the UML tooling. This is not intended to be a complete working example application.

Archived

Create Java project

The followings steps demonstrate some of the functionality that comes with class diagrams. We will create a new Java project and design some basic classes for a sample banking application.

1. From the Workbench select **File** → **New** → **Project** → **Java Project**. Click **Next**.
2. In the New Java Project dialog, enter `ITSO Banking` as the project name and then click **Finish**.
3. If the Confirm Perspective Switch dialog appears, click **Yes**.

   We will design the classes in the Java perspective. However, the J2EE perspective can be used as well.

Create class diagram

There are many ways to create the classes that we need. In our example, we will create the classes from a class diagram.

To create a class diagram, do the following:

1. Right-click **ITSO Banking**, and select **File** → **New** → **Other**.
2. Expand **Modeling**, select **Class Diagram**, and then click **Next**.
3. When the Create Class Diagram dialog appears, we entered the following and then clicked **Finish**:
   - Parent folder: `ITSO Banking/Design`
   - Filename: `Banking Classes`

   Note that for information about the placement of diagrams within the project structure, refer to the note box in 6.3.3, “Topic Diagram” on page 199.

The class diagram is created and opened for editing with associated palette on the right side, as seen in Figure 6-12 on page 207.

If you have existing classes, you can create a new class diagram by visualizing them. In that case, select the classes and select **Visualize** → **Add to New Diagram File** → **Class Diagram** from the context menu. This context menu option is available from Explorer or from another diagram—basically, whenever you can select the class you can also visualize it.

Archived

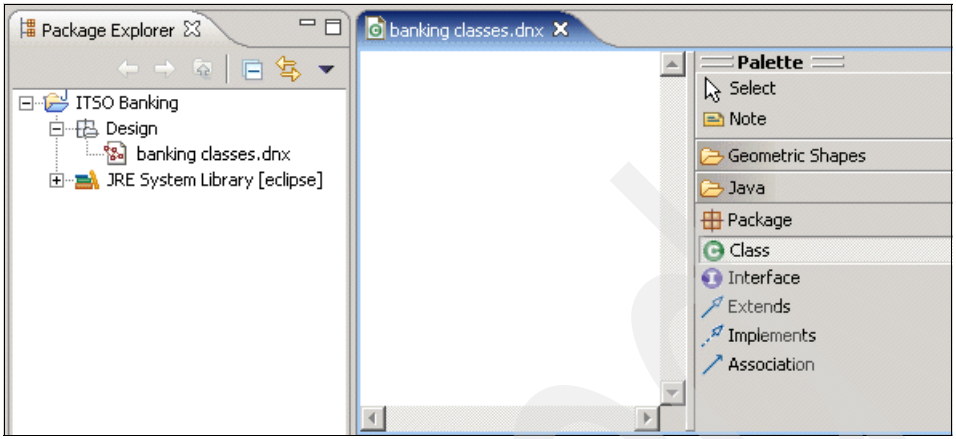


Figure 6-12 Creating a new class diagram

Add classes to the class diagram

To add the Customer, Account, ATM, and Bank classes to the class diagram, do the following:

1. In the Palette, click **Class** and move your cursor to the drawing area.

   The cursor changes to a rectangle shape, with a **+** on the top left corner. This indicates we can insert our new class here.

2. Click where you want the class to be placed on the diagram.

3. When the New Java Class dialog appears, enter `Customer` in the Name field, and `itso.uml.sample` in the Package field (see Figure 6-13 on page 208), then click **Finish**.

Archive

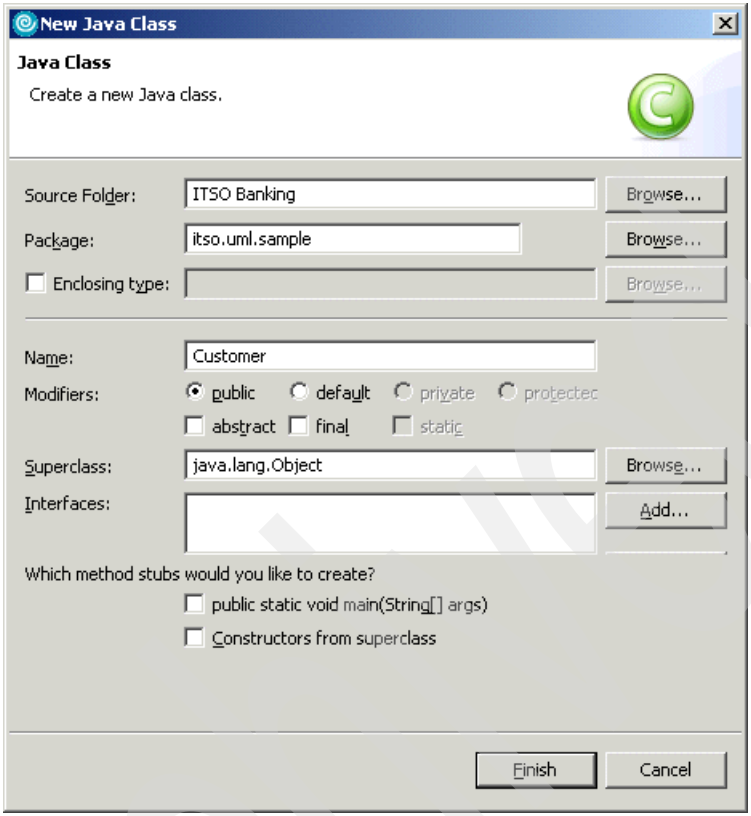


Figure 6-13 Adding a Java class

4. The Customer class is created, both in the diagram and under the project. Expand the **itso.uml.sample** folder to verify this.
5. Repeat the previous instructions to create the Account, ATM, and Bank classes.

Add fields and methods to a class

This section describes how to add fields and methods to a class in the class diagram. In UML visualization, an action bar appears by default when you place the cursor over a shape in a diagram, as seen in Figure 6-14 on page 209. To see a short description for an action, move the cursor over the action (hover help).

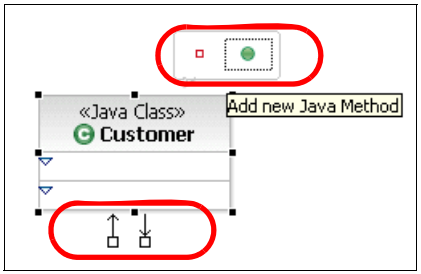


Figure 6-14 Shape action bar

1. To add a field name to the Customer class, do the following:
   a. In the diagram, place the cursor over Customer and click the little red square Add new Java Field icon from the action bar.
   b. When the Create Java Field dialog appears, enter `name` in the Name field, and select **java.lang.String** in the Type drop-down list, as seen in Figure 6-15 on page 210, and then click **Finish**.

Archived

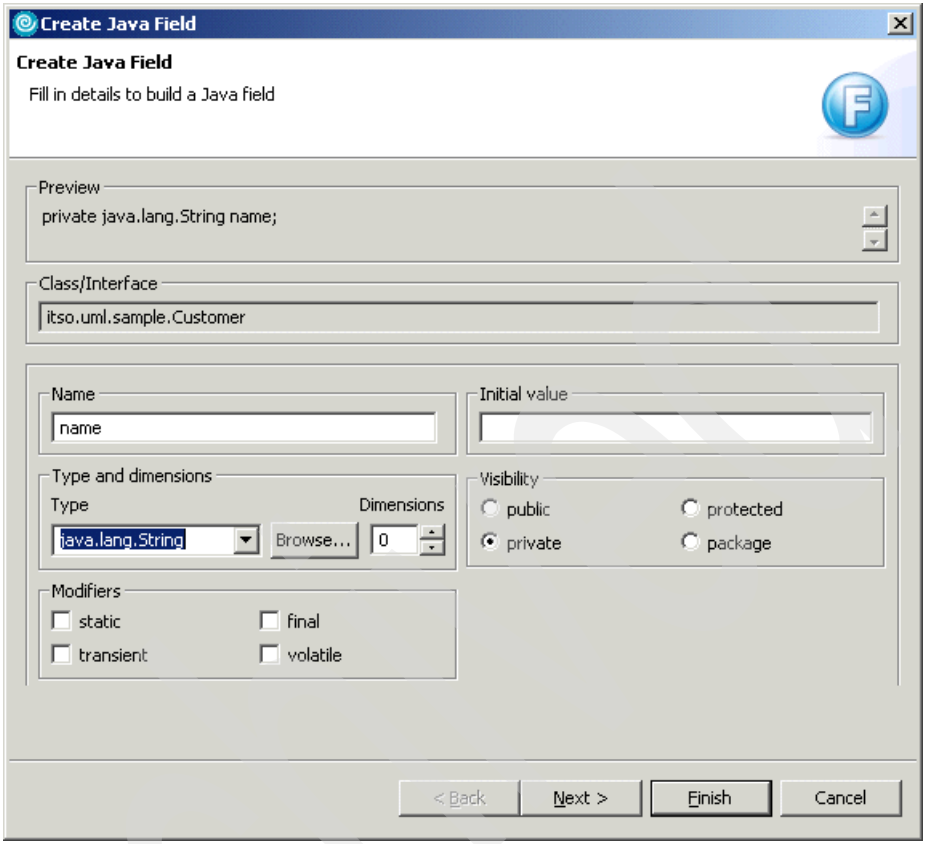


Figure 6-15 Adding a new field to Customer class

2. To add the getter and setter methods for the name field, do the following:
   a. In Customer class, select the **name** field that we added in the previous step.
   b. Right-click and select **Refactor** → **Encapsulate Field** from the context menu.
   c. The Save all Modified Resources dialog will appear. Click **OK** to save the changes to the diagram.
   d. In the Encapsulate Field dialog, click **OK**. The getname() and setName(String) methods are now added to the class.
3. To create the withdrawCash method for the ATM class, do the following:
   a. Right-click the **ATM** class and select **Add Java** → **Method**.

b. When the Create Java Method dialog appears, do the following:

   i. Enter `withdrawCash` as the name for the method.

      This method requires two parameters, the account number and amount. We can enter them in this same dialog.

   ii. In the Parameters section, click **Add**.

   iii. When the Create Parameter dialog appears, enter `accountNumber` in the name field, `Account` in the Type field, and click **OK**.

   iv. Repeat the previous steps to add a parameter named `amount` of type `java.math.BigDecimal`.

c. Click **Finish**.

4. To create the getBalance method for the Bank class, do the following:

   a. Right-click the Bank class and select **Add Java** → **Method**.

   b. When the Create Java Method dialog appears, do the following:

      i. Enter `getBalance` as the name for the method.

         This method requires two parameters, the account number and amount. We can enter them in this same dialog.

      ii. In the Parameters section, click **Add**.

      iii. When the Create Parameter dialog appears, enter `accountNumber` in the name field and `Account` in the Type field, then click **OK**.

      iv. Click **Add**.

   c. Click **Finish**.

5. If you do not see the <<use>> associations, right-click the drawing area and select **Filters** → **Show/Hide Relationships** from the context menu. Make sure Uses (Dependency) is checked and then click **OK**. Visiting this selection dialog is sometimes required to update the diagram with current relationships.

6. Save the diagram by selecting **File** → **Save**.

You should now have a very basic class diagram that looks like the diagram in Figure 6-16 on page 212.

Archived

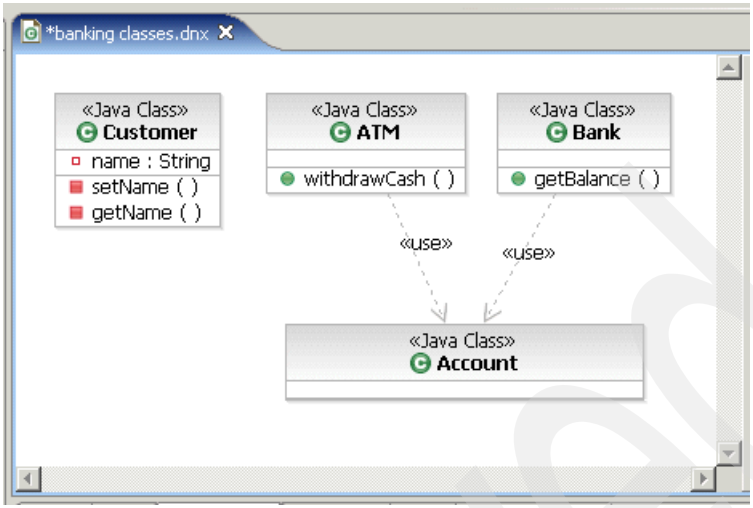


Figure 6-16 Completed class diagram

Note the following capabilities exist when working with class diagrams:

- Class is selected when you have the eight dot rectangle surrounding it. You can also select things inside the class, but then the context menu is different. If you want to reset any selection (deselect everything), click empty space in the diagram.
- Auto-navigation to Java source can be turned on and off with the button on the toolbar.
- Deleting a class from the diagram does not delete it from your project or model. This means you can quickly hide unnecessary classes from your diagrams, without affecting your code or semantics.

> **Note:** When you delete Java elements from a project (for example, the Explorer view, the Java elements are deleted from both the diagram and the project.

- All other changes, including deleting content *inside* a class, are connected to sources and affect the project directly. The actual file will be immediately updated on the disk then. A separate save step to effect the changes that were made in the diagram is not needed.
- Undo and redo are available on the context menu for recovering from unwanted changes or actions.

Archived

- The toolbar and context menu provide many different ways to arrange, align, and customize the elements in the diagram. You can also manually move the elements around, and change their visual layout (fonts, colors, line styles).
- Compartments can be hidden or displayed by clicking the little triangle at the top left corner of that compartment, or by using the compartment switch on the toolbar (all compartments vs name only).
- To navigate and view more of the diagram, the usual double-click the diagram title will maximize it. Resizing and zooming works well too. For larger diagrams, you can also bring up the Outline view. Usually it is already visible, but select **Window** → **Show View** → **Outline** if you do not see it.

Class diagram preferences

You can customize some of the preferences for the Java assisted modeling function. Click **Window** → **Preferences** → **Modeling**. From there, you can customize the appearance and set the defaults for fields and methods that are generated from the diagrams. You can also enable the query for related elements to include binary types also. By default they are filtered out.

A new feature in Rational Application Developer is the option to add an EJB to a new or existing class diagram from the Create an Enterprise Bean wizard. This option is enabled by default but can be turned off from the preferences menu under **Modeling** → **EJB**.

The new J2EE and EJB visualization features in class diagrams are discussed in Chapter 15, "Develop Web applications using EJBs" on page 827.

6.3.6 Sequence Diagram

Sequence diagrams are the most popular UML artifact for dynamic modeling, which focuses on identifying the behavior within your system.

With the new capabilities to represent fragments, interaction occurrences, and loops, sequence diagrams can be used in two forms:

- Instance form: Describes a specific scenario in detail, documenting one possible interaction, without conditions, branches, or loops. This form is used to represent one use case scenario. Different scenarios of the same use case are represented in different sequence diagrams. Modeling tools that support UML 1.x semantics only allow this form of representation.
- Generic form: Describes all possible alternatives in a scenario, taking advantage of new UML 2.0 capabilities like conditions, branches, and loops. This form can be used to represent several scenarios of the same use case in a unique sequence diagram, where it makes sense.

Archive

Rational Application Developer supports both the instance and generic form, and the UML2 generic form is available when drawing a sequence diagram by hand. As we have seen, the generic form is also used when a static method sequence diagram is generated.

Let us continue our example and design a sequence diagram that would handle the use case of a customer withdrawing cash from an ATM. We will use some of the new UML2 elements in this example, including conditional fragments and reference to another diagram.

Create a sequence diagram

To create a sequence diagram, do the following:

1. Select **File** → **New** → **Other**.
2. Expand **Modeling**, and select **Sequence Diagram**.
3. When the Create Sequence Diagram dialog appears, select the previously created **ITSO Banking/Design** folder, enter `Withdrawal Cash` in the file name field, and click **Finish**.

   The sequence diagram and the associated drawing palette will appear. The title of the new diagram is Interaction1.
4. To change the title, click **Interaction1** in the top left corner and enter `Withdraw Cash` as the new title, as seen in Figure 6-17.

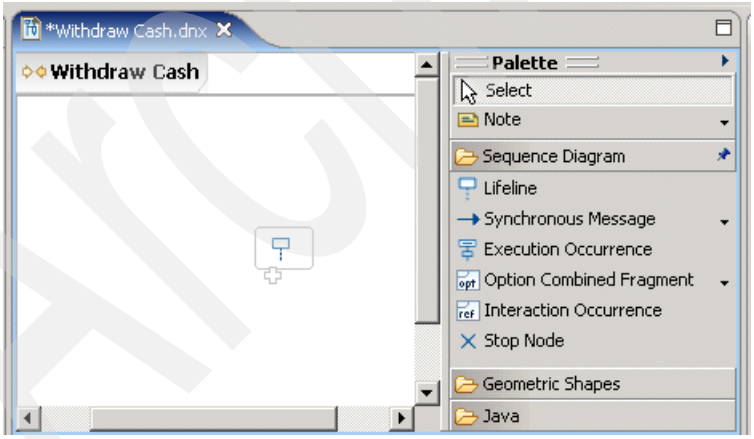


Figure 6-17 Drawing a sequence diagram

Archived

Add objects

To add Customer, ATM, and Bank objects to the sequence diagram, do the following:

1. From Package Explorer, select the **Customer** class and drag the class to the sequence diagram. The property:Customer lifeline will appear.
2. Repeat the previous step to add the ATM and Bank classes.

> **Note:** You can drag and drop C++, data, and other artifacts onto the sequence diagram (or class diagram for that matter) as well. However, code generation is only supported for Java/EJB artifacts.
>
> A new lifeline can be added from the Palette also, but this does not generate the class. You could this for unspecified lifeline types.

3. Select the **property:Customer** lifeline and press **F2**.
4. Enter `cust` and press **Enter**.
5. Repeat the previous steps to rename property2:ATM to teller:ATM and property3:Bank to theirBank:Bank.
6. Save the diagram.

   The diagram should now look like the one in Figure 6-18.

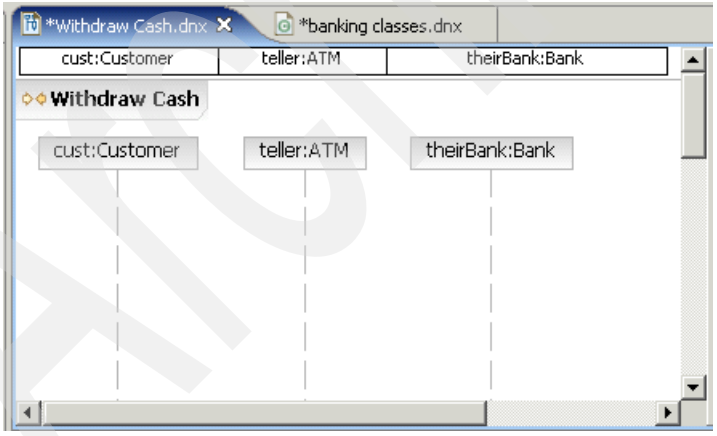


Figure 6-18 Sequence diagram with three lifelines

Archived

Create synchronous messages

Now that the necessary objects have been created, we can design the interactions between the objects.

1. From the Palette, expand **Sequence Diagram**, and select **Synchronous Message**.
2. Once the association icon is selected click the **Customer** lifeline and drag with the left-mouse button pressed down to the Teller lifeline.
3. From the listbox, select **withdrawCash** and click in the empty area in the diagram. The resulting diagram should look like Figure 6-19 on page 216.

> **Note:** If the lines are overlapping, you can re-position any line by selecting it and moving it up or down the lifeline.

4. Repeat the previous steps to create the Synchronous Message `getBalance` from `Teller` to `Bank`. The message should originate from within the withdrawCash activation box of the Teller object.

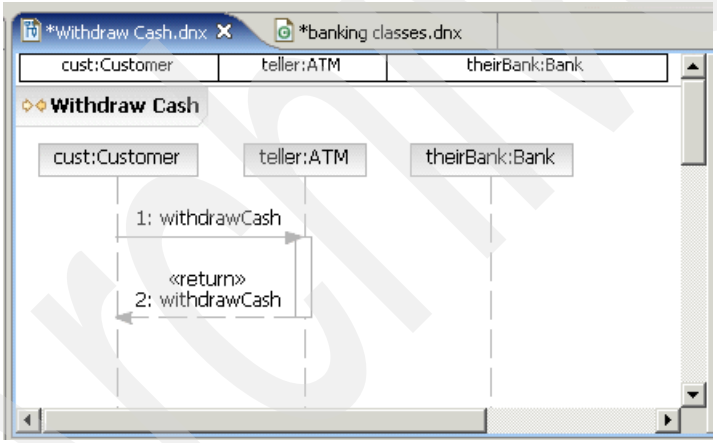


Figure 6-19 Adding a message to sequence diagram

Create a reference to an external diagram

We would like to indicate that the balance lookup is actually described in another sequence diagram. We can draw that using the Interaction Occurrence selection in the palette.

1. From the Palette view, select **Interaction Occurrence**.
2. Click the getBalance activation box on the Bank lifeline and select **Create New Interaction** from the context menu.

3. Enter `Balance Lookup(accountNumber) : BigDecimal` inside the new frame. This indicates that the Balance® Lookup diagram contains the flow of that sequence and will return a value of the type BigDecimal.

Create a conditional fragment

Now we would like specify that if sufficient funds were available, we will proceed with the transaction. This can be illustrated with a conditional fragment.

To add a conditional fragment to the sequence diagram, do the following:

1. From the palette, select **Option Combined Fragment**.
2. Drag a rectangle covering the Teller and Bank lifelines. The rectangle must be drawn below the previously created messages.
3. When the Add Covered Lifelines dialog appears with the Bank and Teller lifelines checked, click **OK**.

   The diagram should now appear similar to Figure 6-20.
4. To specify the condition for the new frame, click **[]** and enter `balance > amount` as the test for further execution inside the frame.

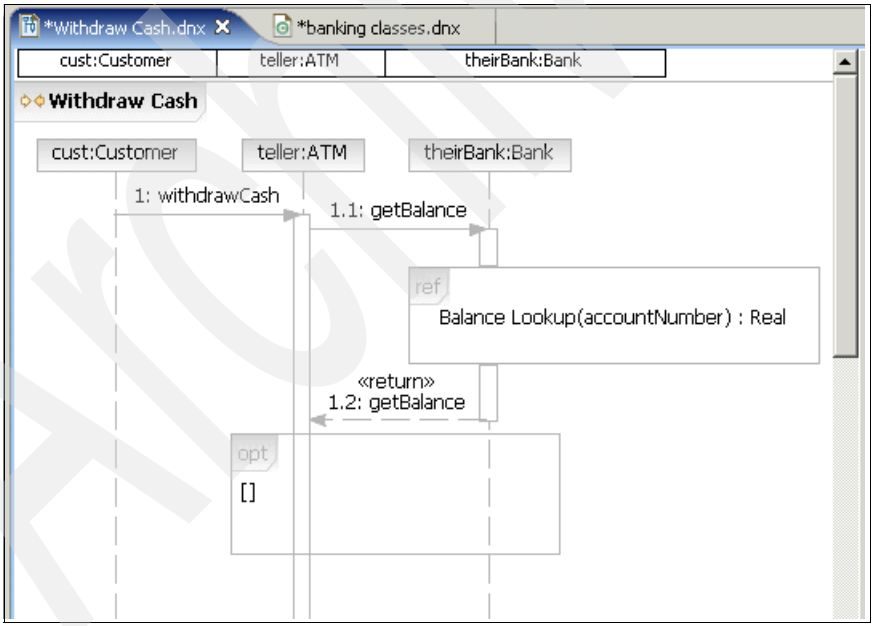


Figure 6-20 Sequence diagram with conditional fragment

Archived

Add interactions inside option frame

To add the debit message from Teller to Bank, do the following:

1. Create a Synchronous Message from Teller to Bank inside the opt frame created in the previous section.
2. In the Create New Operation listbox, press Enter.
3. When the Enter Operation name dialog appear, enter `debit` in the Operation name field and click **OK**.

   Note: At the time of writing, entering parameters in the Enter Operation name dialog was not supported and resulted in invalid Java code being generated for the new method.

4. Using the method described in “Create a reference to an external diagram” on page 216, add a ref frame to indicate that the debit sequence is designed in the Debit Account(accountNumber, amount) diagram.

The completed sequence diagram should be similar to Figure 6-21.

Archived

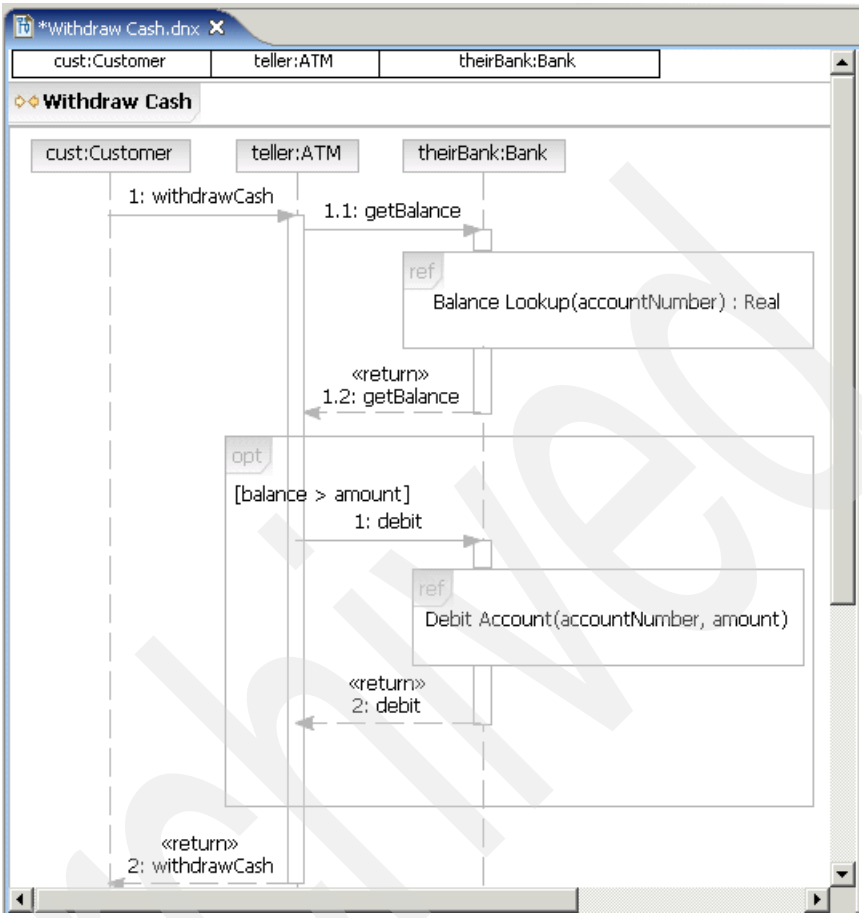


Figure 6-21 Completed sequence diagram: Withdraw Cash

6.3.7 J2EE visualization

A new feature in IBM Rational Application Developer V6.0 is the option to add an EJB to a new or existing Class diagram from the Create an Enterprise Bean wizard. This option is enabled by default; however, this option can be turned off from the preferences menu under **Modeling** → **EJB**.

Archived

6.4 More information on UML

For more information about UML, we recommend the following resources. These three Web sites provide information on modeling techniques, best practices, and UML standards:

- IBM developerWorks Rational

  http://www.ibm.com/developerworks/rational/

  Provides guidance and information that can help you implement and deepen your knowledge of Rational tools and best practices. This network includes access to white papers, artifacts, source code, discussions, training, and other documentation.

  In particular we would like to highlight the following series of high quality Rational Edge articles focusing on UML topics:

  http://www.ibm.com/developerworks/rational/library/content/RationalEdge/archives/uml.html

- The IBM Rational Software UML Resource Center

  http://www.ibm.com/software/rational/uml/index.html

  This is a library of UML information and resources that IBM continues to build upon and update. In addition to current news and updates about the UML, you can find UML documentation, white papers, and learning resources.

- Object Management Group

  http://www.omg.org
  http://www.uml.org

  These OMG Web sites provide the following resources:

  - Formal specifications on UML that have been adopted by the OMG and are available in either published or downloadable form
  - Technical submissions on UML that have not yet been adopted

Archived

7

Develop Java applications

This chapter provides an introduction to the Java development capabilities and tooling for IBM Rational Application Developer V6.0.

The chapter is organized into the following four major sections:

- Java perspective overview
- Develop the Java Bank application
- Additional features used for Java applications
- Java editor and Rapid Application Development

Note: The sample code described in this chapter can be completed by following along in the procedures documented. Alternatively, you can import the sample Java code provided in the c:\6449code\java\BankJava.zip Project Interchange file. For details refer to Appendix B, “Additional material” on page 1395.

© Copyright IBM Corp. 2005. All rights reserved.

7.1 Java perspective overview

The Java perspective was briefly introduced in Chapter 4, “Perspectives, views, and editors” on page 131. Throughout this chapter we look at the commonly used views within the Java perspective. The highlighted areas in Figure 7-1 indicate the views used within the Java perspective.

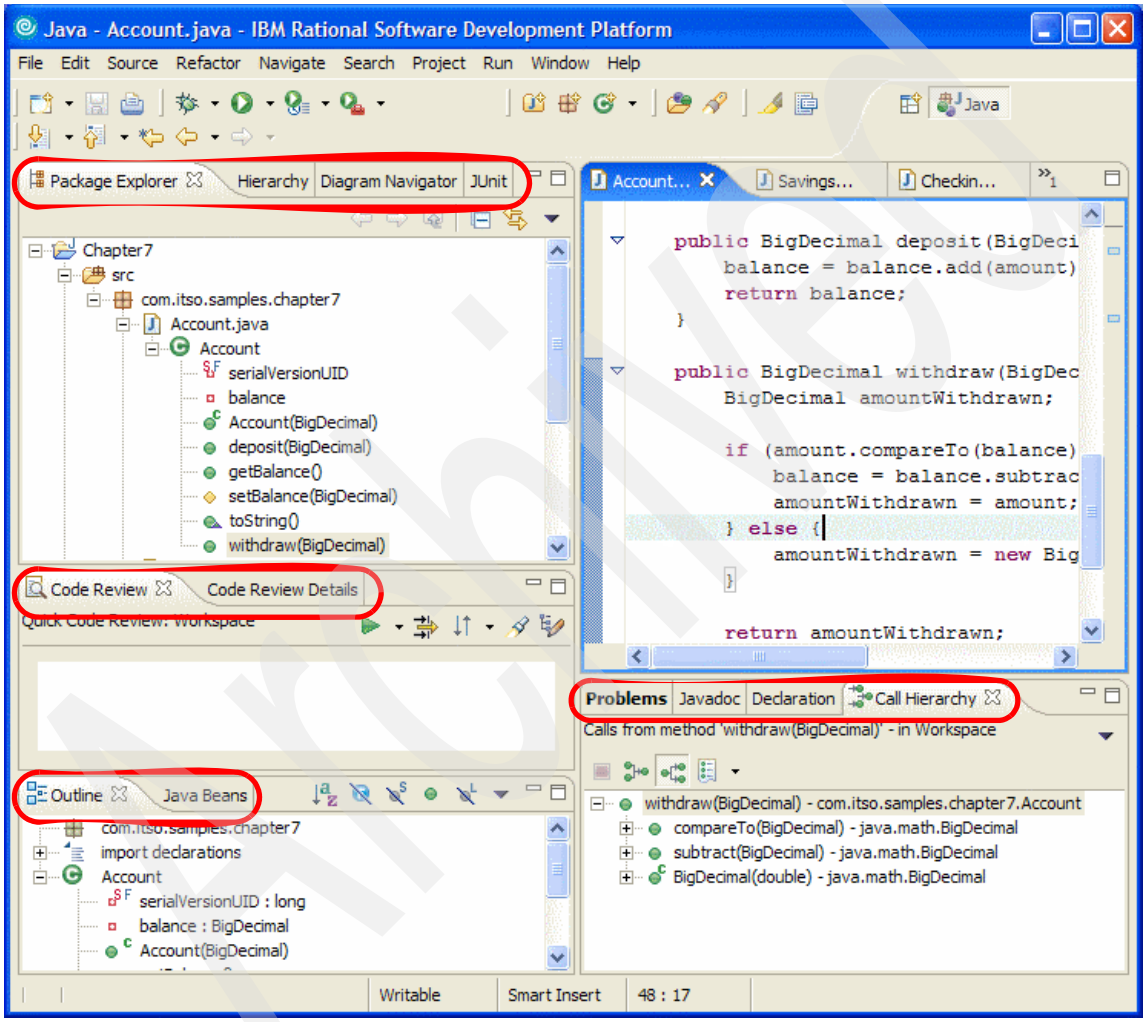


Figure 7-1 Views in the Java perspective

7.1.1 Package Explorer view

The Package Explorer view displays all packages, classes, interfaces, member variables, and member methods contained in the project workspace, as shown in Figure 7-2.

To view the Package Explorer, select **Window** → **Show View** → **Other**. In the Show View dialog box, expand **Java** and select **Package Explorer**.

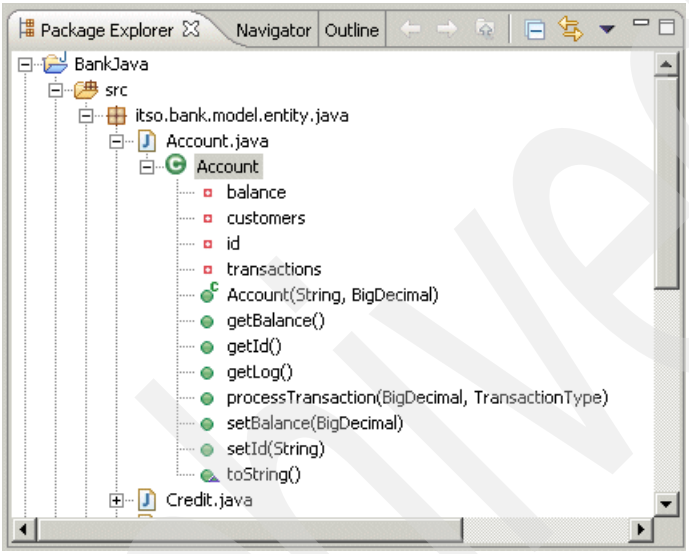


Figure 7-2 Package Explorer view

7.1.2 Call Hierarchy view

The Call Hierarchy view displays all the members calling the selected method (Caller Hierarchy) or all the members called by the selected method (Callee Hierarchy), and presents it in a tree hierarchy.

To view the call hierarchy of a method, select the method in the Package Explorer, right-click, and choose **Open Call Hierarchy**. The Call Hierarchy for that method is displayed as shown in Figure 7-3 on page 224. To toggle between the Caller Hierarchy and Callee Hierarchy use the and buttons, respectively.

Archive

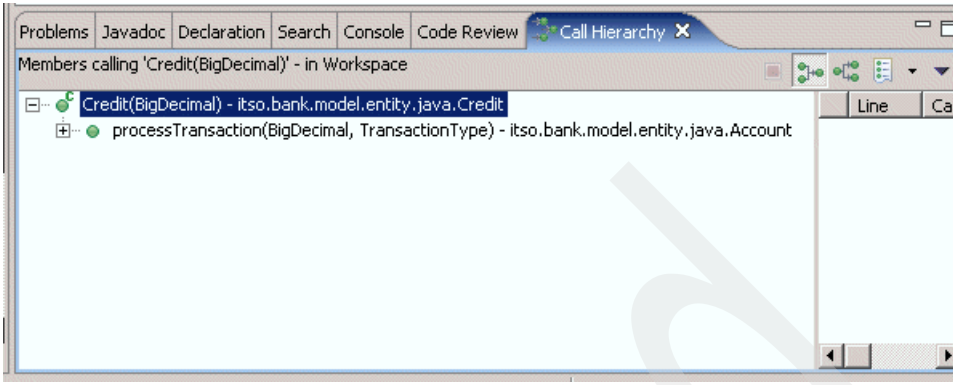


Figure 7-3 Call Hierarchy view

7.1.3 Type Hierarch view

The Type Hierarchy view, also known as Hierarchy, displays the class type hierarchy of the selected class.

To view the type hierarchy of a class type, select the type in the Package Explorer, right-click, and choose **Type Hierarchy**. The type hierarchy for that method is displayed in the Type Hierarchy view, as shown in Figure 7-4. To Toggle between SuperType and SubType Hierarchy use the and buttons, respectively.

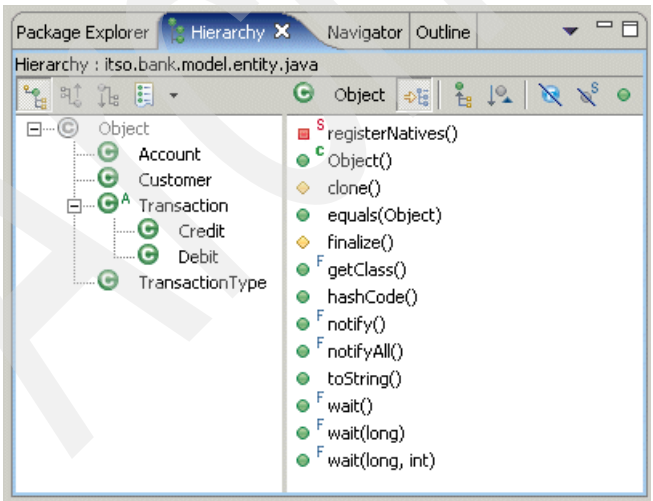


Figure 7-4 Type Hierarchy view

Archived

7.1.4 Problems view

The Problems view displays information regarding errors generated by the Workbench. Errors are typically generated by various builders within the Workbench.

To verify all the builders within a project, select the project in the Package Explorer, right-click, and select **Properties** from the context menu. In the properties window choose **Builders**. You will now be able to see all the builders associated with the project. You can enable/disable a builder from this view. Figure 7-5 shows an example of the Problems view with a problem detected by the Java builder.

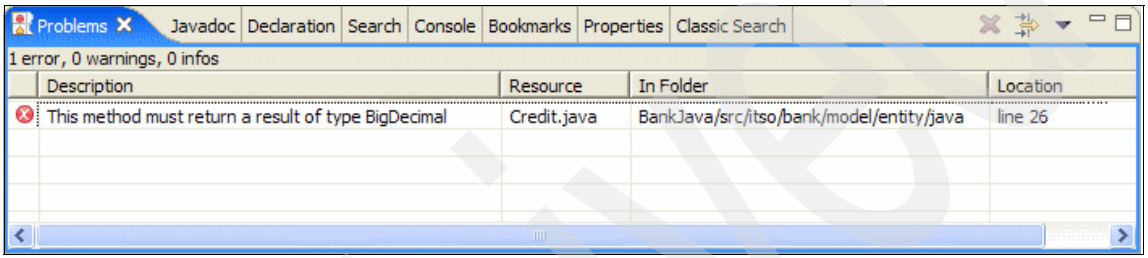


Figure 7-5 Problems view

Also, the Problems view can be filtered to show only specific types of problems. For example, you may want to see only errors or tasks related to a specific resource. For more information on this issue, refer to 7.3.1, “Locating compile errors in your code” on page 287.

7.1.5 Declaration view

The Declaration view displays the declaration and definition of the selected type.

When a class is selected in the Package Explorer, the class declaration along with its member variables and method definitions are displayed in the Declaration view. Figure 7-6 on page 226 displays the Declaration view when a method is selected in the Package Explorer. To open the type and to edit the source in the editor use the button.

Archive

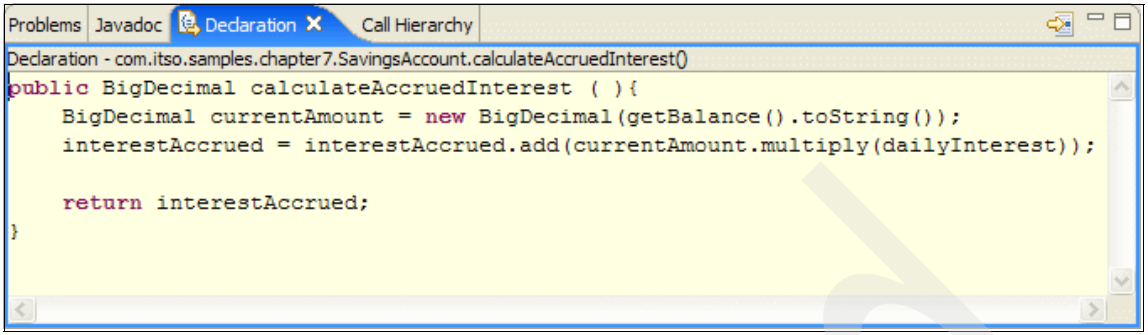


Figure 7-6 Declaration view

7.1.6 Code review

IBM Rational Application Developer V6.0 introduces the Code Review tool that facilitates verification of code quality and automates code reviews of source code in the workspace.

Rules can be created and configured through the Preference Window by selecting **Window** → **Preference**. From the Preference window expand **Java** and select **Code Review**.

There are five categories of code review, each with varying sets of rules that will be used to verify quality of code:

- Quick Code Review
- Complete Code Review
- Globalization Code Review
- J2SE Best Practices Code Review
- J2EE Best Practices Code Review

For more information on the various categories of code review and creating new rules based on existing rule templates, refer to Chapter 3, “Workbench setup and preferences” on page 75. Once rules are configured, code in the workspace, a selected project, or selected java files can be run against the configured rules.

Code Review view

To display the Code Review view, select **Window** → **Show View** → **Other**. In the Show View dialog box, expand **Java** and select **Code Review**.

Code Review example

To demonstrate the code review feature of Rational Application Developer, we will use the base set of best practice rules that come with Rational Application

Developer for a full code review and the CodeReviewSample.java, which has a method as shown in Example 7-1. CodeReviewSample.java can be found in **BankJava** → **src** → **itso.bank.model.example.java**, provided you imported the sample BankJava.zip Project Interchange file (see Appendix B, “Additional material” on page 1395).

This method violates one of the rules set up that mandates that as a best practice zero length arrays should be returned instead of null to minimize NullPointerExceptions.

Example 7-1 CodeReviewSample.java

```
public Object[] verifyCodeReviewTool(boolean dummy){
   Object[] dummyValues = { new Object(), new Object() };
   if ( dummy ){
      return dummyValues;
   } else {
      return null;
   }
}
```

To run the Code Review tool, select the Java element (Project, Package, or Java File) in the Package Explorer, right-click, and select **Code Review** → **Review CodeReviewSample.java**. Alternatively, use the Review Button ( ) to select the workspace, project, package, or Java file to review.

The Code Review tool runs and displays all the code artifacts that do not adhere to the rules configured previously. At the top of the window, a brief report indicating the number of rules against which the code was compared, number of files that were included in the current run of the code review tool, number of problems found, number of warnings found, and number of recommendations provided is displayed.

For the code shown in Example 7-1, the code review tool returns the result shown in Figure 7-7 on page 228. An icon is displayed in the editor beside the offending line (Figure 7-8 on page 228) to help developers easily spot the problem.

Archived

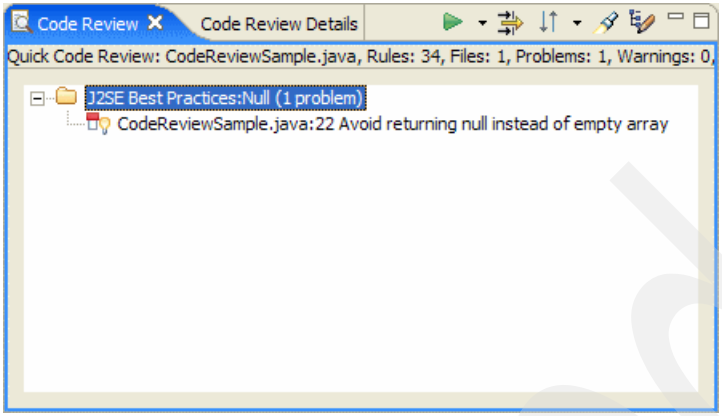


Figure 7-7 Code Review view with the result of reviewing the code in CodeReviewViewSample.java

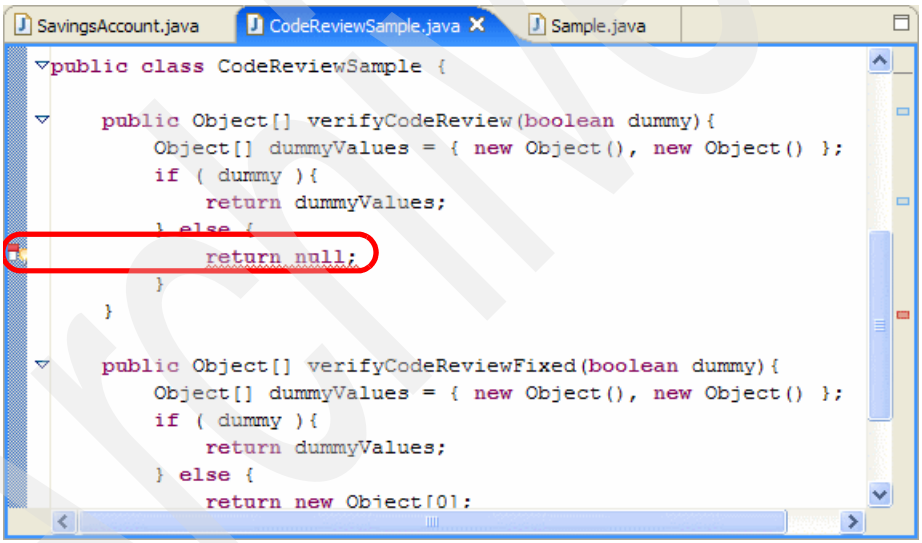


Figure 7-8 Editor displays the problem icon beside the offending line

Archived

Note: Rules and rule configurations can be exported and imported through the Preferences window. Importing and exporting of settings is described in Chapter 3, “Workbench setup and preferences” on page 75.

This allows project teams to decide upon a standard set of rules that all code in the project needs to conform to to pass the quality code of the project. Developers can run the code review tool with the agreed upon rule set for the project to ensure that the code being developed adheres to the project standards. This tool is typically invaluable during the development and code review process and in helping maintain software quality.

Code Review Details view

The Code Review Details view displays detailed information about the diagnosed problem by providing a description of the problem, examples, and solutions to fix the problem detected by the Code Review tool.

To open the Code Review Details view, double-click a problem displayed in the Code Review view shown in Figure 7-7 on page 228. Developers can take advantage of the Code Review Details view by using the description, examples, and solutions to fix any problem detected during code review. Figure 7-9 shows the code review details window describing the problem found as a result of running the Code Review tool against the code shown in Example 7-1.

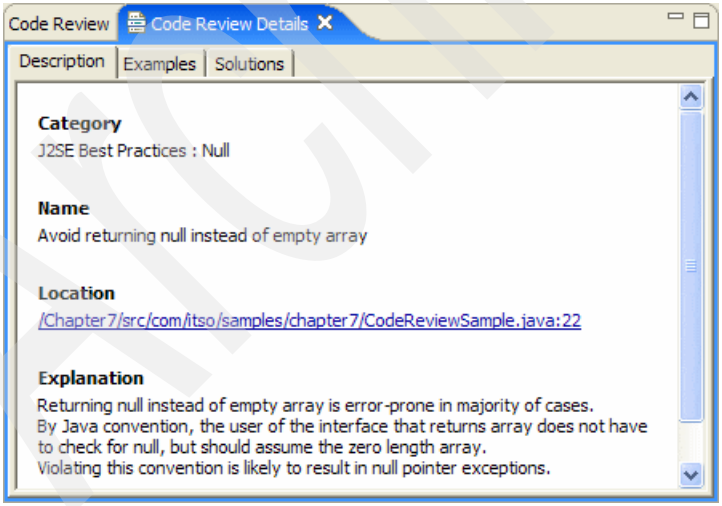


Figure 7-9 Code Review Details

Example 7-2 lists the updated code fixed as a result of the information provided in the Code Review Details view. The line that was modified is shown in bold in the Example 7-2.

Example 7-2 Listing of code fixed

```
public Object[] verifyCodeReviewFixed(boolean dummy){
   Object[] dummyValues = { new Object(), new Object() };
   if ( dummy ){
      return dummyValues;
   } else {
      return new Object[0];
   }
}
```

7.1.7 Outline view

The Outline view is context sensitive, and in the Java perspective context displays all the Java elements including package, imports, class, fields, and methods corresponding to the file currently active in the Java editor. The Outline view in the Java perspective is shown in Figure 7-10. This view allows the developer to filter Java elements that are displayed in this view by providing the buttons highlighted in Figure 7-10.

The Outline view can be displayed by selecting **Window** → **Show View** → **Outline**.

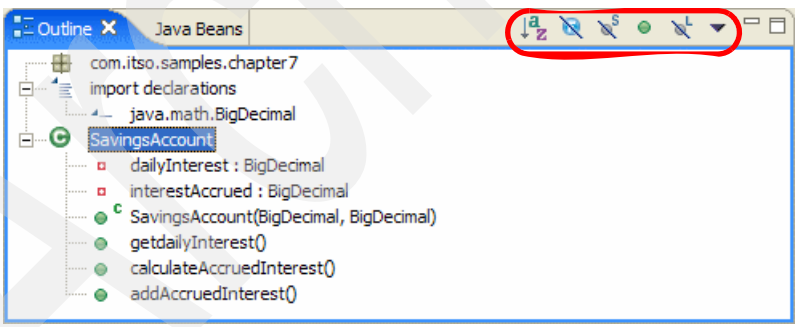


Figure 7-10 Outline view

7.1.8 Diagram Navigator view

The Diagram Navigator view is new to IBM Rational Application Developer V6.0. The Diagram Navigator supports creation class diagrams, sequence diagrams, and topic diagrams as visualization models. These visualization models can be used to design and develop J2SE solutions. Modifications to the visualization

model are propagated to the code immediately, and changes made to the code is propagated to the visualization model.

We will explore class and sequence diagrams later in this chapter.

- Javadoc: This view displays the java documentation of the selected java element selected in the package explorer. We will discuss this view in detail in 7.3.11, “Javadoc” on page 303.
- Java Beans: This view is used with Visual Editor for Java for designing and developing applications in which Java renders the Graphical User Interface for the application. This is discussed in detail in Chapter 9, “Develop GUI applications” on page 415.
- JUnit: This view assists developers running JUnit test cases and visualizes the test results of executed test cases. This view is discussed in detail in Chapter 20, “JUnit and component testing” on page 1081.

Note: For a detailed description of UML and Rational Application Developer’s support for UML refer to Chapter 6, “RUP and UML” on page 189.

7.2 Develop the Java Bank application

In this section we demonstrate how Rational Application Developer can be used to develop the various Java elements including projects, packages, classes, interfaces, member variables, and methods.

- Java Bank application overview.
- Create a Java Project.
- Create a class diagram.
- Create Java packages.
- Create a Java interface.
- Create Java classes.
- Create the Java attributes and accessor methods.
- Add method declarations to an interface.
- Add Java methods and constructors.
- Define relationships (extends, implements, association).
- Implement the methods for each class.
- Run the Java Bank application.

7.2.1 Java Bank application overview

We will use the ITSO RedBank banking example to work our way through the remaining sections of this chapter. The banking example is deliberately over

simplified. Exception handling is ignored to keep the example concise and relevant to our discussion.

Classes and interfaces for the example

The classes and interfaces for the Java Bank application example are as follows:

- Bank: A bank interface allows common operations a bank would perform, and typically includes customer, account, and transaction related operations.
- ITSOBank: A concrete instance of bank that will implement operations in the Bank interface.
- Customer: A customer is a client of the bank who holds one or more bank accounts with a bank, ITSOBank in our example. A customer can hold one or more accounts with a bank.
- Account: An account is the representation of a bank account and allows manipulations of the account information. An account can be associated to one or more customers.
- Transaction: Transaction is a single operation that will be performed on an account. In the Java implementation of this class, this is represented as an abstract base class. The two kinds of valid transactions in our case are credit and debit transactions. An account composes a list of transactions performed on this account for logging and querying purposes.
- Credit: Credit is a sub-type of transaction and inherits from transaction. This transaction results in an account being credited with the amount indicated.
- Debit: Debit is a sub-type of transaction and inherits from transaction. This transaction results in an account being debited by the amount indicated.
- TransactionType: A simple typesafe implementation for the indication of the transaction type.
- Exceptions: Exception classes to indicate common account and customer related exceptions.

Packaging structure

The bank sample uses the package structure listed in Table 7-1 to organize the classes discussed above.

Table 7-1 Bank sample packages

Package name	Description
itso.bank.model	Contains the business model objects for the ITSO Bank
itso.bank.facade	Contains the interface for the ITSOBANK

Package name	Description
itso.bank.exception	Contains the exception classes
itso.bank.client	Contains the BankClient class used to verify the bank sample
itso.bank.test	Contains a JUnit test case and an example for using the Code Review tool

Class diagram

The complete class diagram for the bank example is shown Figure 7-11 on page 234, which indicates the classes and their relationships discussed above. We will create this diagram using Rational Application Developer's Visual UML tool throughout the course of this chapter, and at the end of this chapter we will have created the diagram displayed in Figure 7-11 on page 234.

Archived

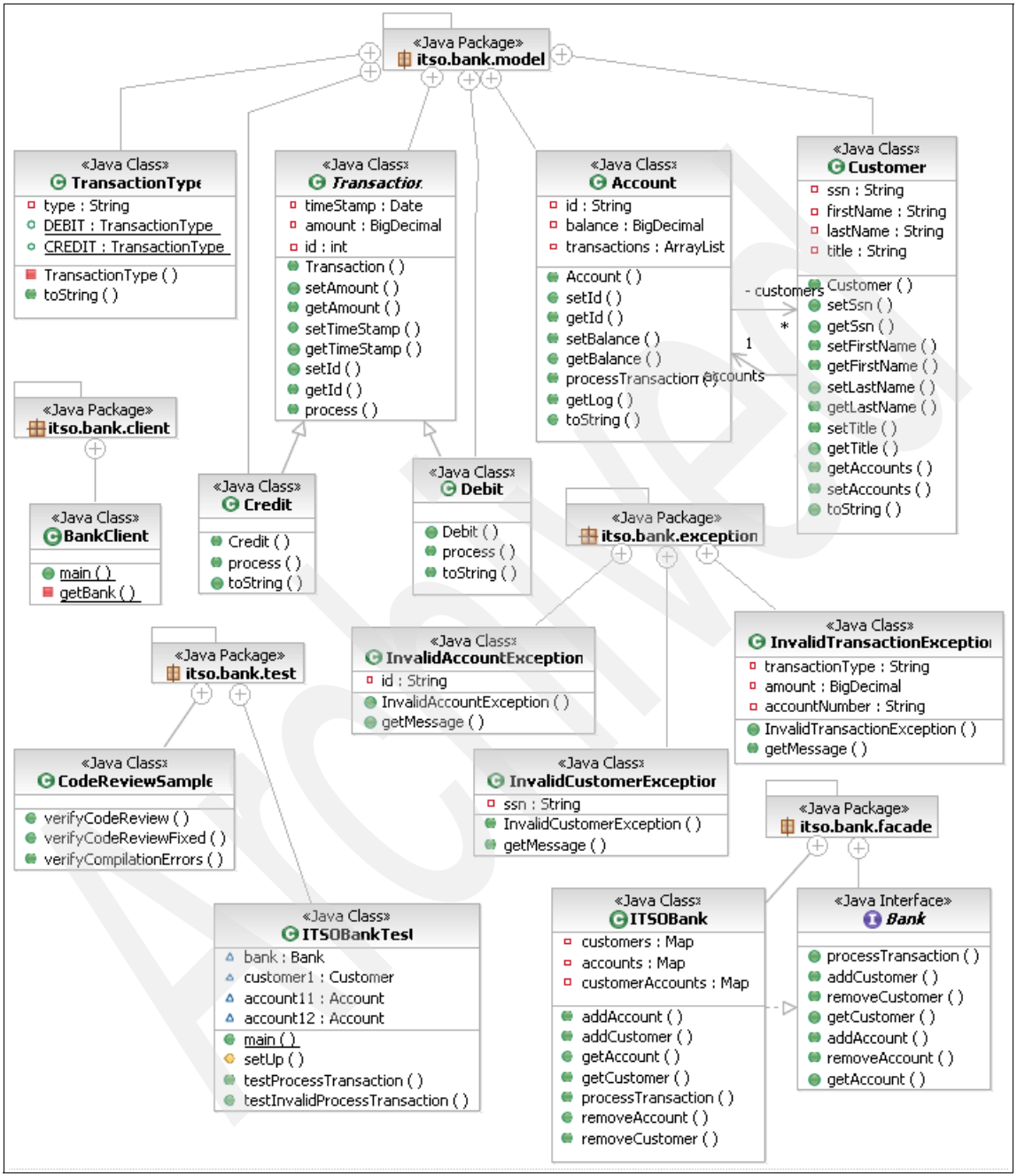


Figure 7-11 Class diagram - Banking sample

7.2.2 Create a Java Project

A Java project contains the resources needed for Java applications, including Java files and class files. Java projects are used to create Java packages. When you create a new Java project, the environment is set up for Java development.

To create a new Java project, do the following:

1. Start Rational Application Developer by clicking **Start** → **Programs** → **IBM Rational** → **IBM Rational Application Developer V6.0** → **Rational Application Developer**.
2. From the Workbench, select **File** → **New** → **Project**.

> **Tip:** Alternatively, create a Java Project by using the Alt+Shift+N hot key sequence and select **Project**.

3. When the Select a wizard dialog appears, select **Java Project** (or **Java** → **Java Project**), as seen in Figure 7-12, and then click **Next**.

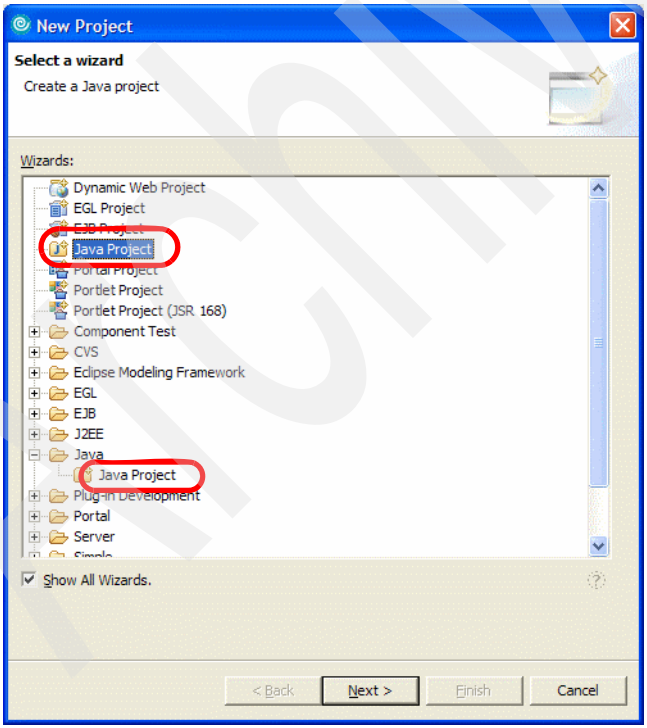


Figure 7-12 New Project dialog

Archived

4. When the Create a Java project dialog appears, we entered the following, as seen in Figure 7-13 on page 237, and then clicked **Next**:
   - Project name: `BankJava`
   - Location: Select **Create project in workspace** (default).

     Note: By default, the project files will be stored in a directory created under the Rational Application Developer workspace directory. You can change the project directory by selecting the radio button marked as "Create project at external location" and specifying another directory.

   - Project layout: Select **Create separate source and output folders**.

     Note: By default, **Use project folder as root for source and class files** is selected. If you select **Create separate source and output folders**, you can optionally change the name for of the directories to something other than src and bin by clicking the ***Configure Defaults*** button. Figure 7-14 on page 238 displays the Preferences window for new projects.

Archive

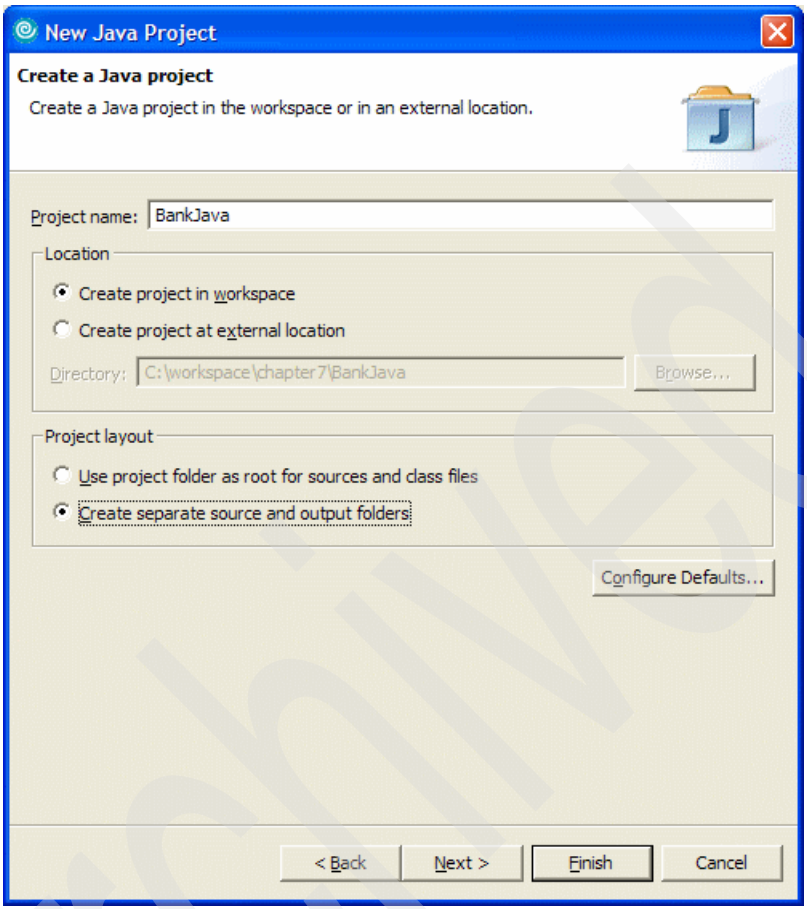


Figure 7-13 New Java project: Project name and directory

Archive

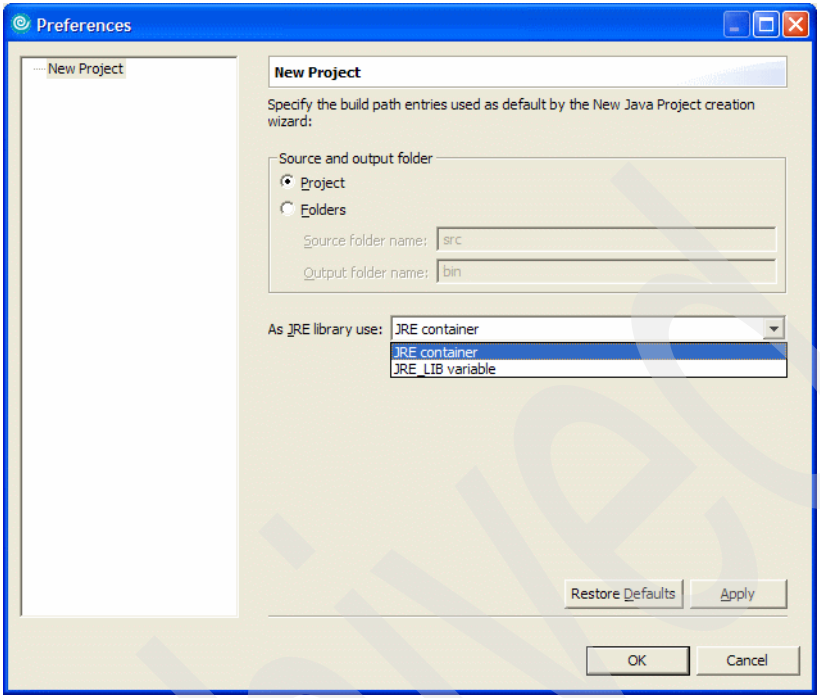


Figure 7-14 Preferences window for setting defaults for new projects

You can change the default folder names and the JRE that will be used for new Java Projects. In our case we will accept the default settings of src/bin for folder names and JRE container for the JRE to be used. Click **Cancel** to return to the Java Project Wizard.

5. When the Java Settings dialog appears (see Figure 7-15 on page 239), you may want to change the settings for the following tabs.

Note: We accepted the default settings for each of the tabs.

– Source tab

On the Source tab you decide whether it is appropriate to store source code directly in the project folder, or if you want to use separate source folders. For our sample project, we already created a simple model where Java source is stored under src and the class files are stored under bin. The src folder is included in the build path. Additionally, if you want to include more directories with Java source files into the build path, you can add these folders by using the Add Folder... button.

You can also change the output folder; we will leave it as bin in this case.

Note: In the Package Explorer view you cannot see the generated .class files. If you open the Navigator view (**Window** → **Show View** → **Navigator**), you can see the source under src and class files under bin, for our example.

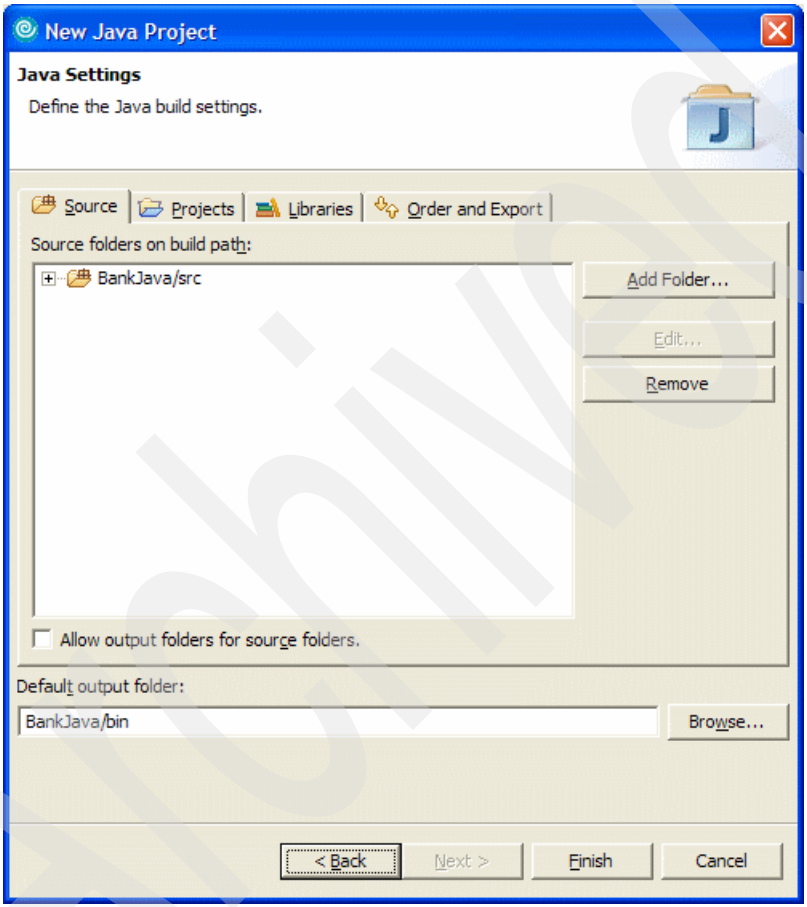


Figure 7-15 New Java project: Source settings

– Projects tab

On the Projects tab you can specify any other projects in your workspace that should be in the Java build path for the new project (Figure 7-16 on page 240). You might have some common code in a project in the workspace that already exists that you want to reuse it in the new project.

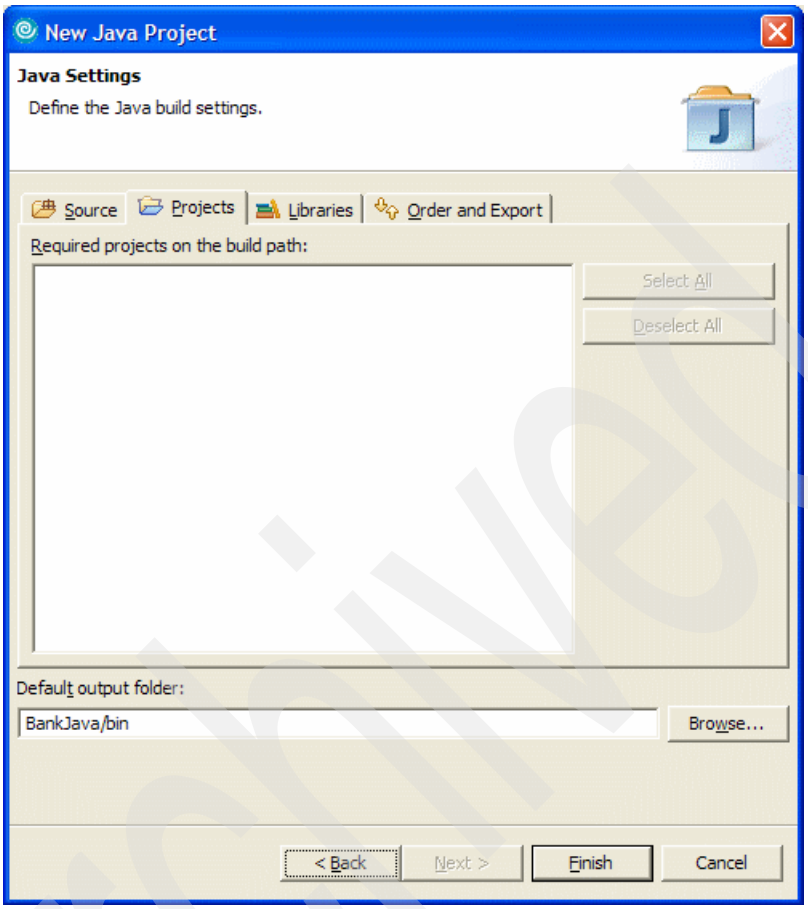


Figure 7-16 New Java project: Projects in build path

– Libraries tab

On the Libraries tab (shown in Figure 7-17 on page 242) you can add other code libraries that have to be included in the build path of your project. By default, the library list contains an entry representing the Java runtime library.

You can also add the following types of resources to your build path:

- Jar files: These are jar files that are available within the current workspace, To add jar files available within the current workspace use the Add Jars... button.
- External jar files: These are jar files that are in the file system external to the workspace. To add jar files available externally to the workspace use the **Add External Jars...** button.

- Variables: Variables are *symbolic pointers* to JAR files with the benefit of avoiding local file system paths specified explicitly in build and runtime classpaths. To add variables to the build path use the **Add Variable...** button.

  For more information on defining new variables, refer to Chapter 4, “Perspectives, views, and editors” on page 131. It is a good idea to define and use project-specific variables when projects are shared in a team environment.

- Libraries: Libraries are a logical grouping of a group of jar files and are treated as a single artifact. To add libraries to the build path use the **Add Library...** button.

  Rational Application Developer introduces the concept of User Libraries, allowing users to define their own libraries consisting of multiple jar files through the Preference window. It is a good idea to define and use user libraries consisting of common jar files used throughout the project when developing in a team environment.

  For more information on defining user libraries, refer to Chapter 4, “Perspectives, views, and editors” on page 131.

- Folders: Folders can be directly added to the build path to include all resources within the folder, including class and property files. To add folders to the build path use the **Add Folder...** button.

Archived

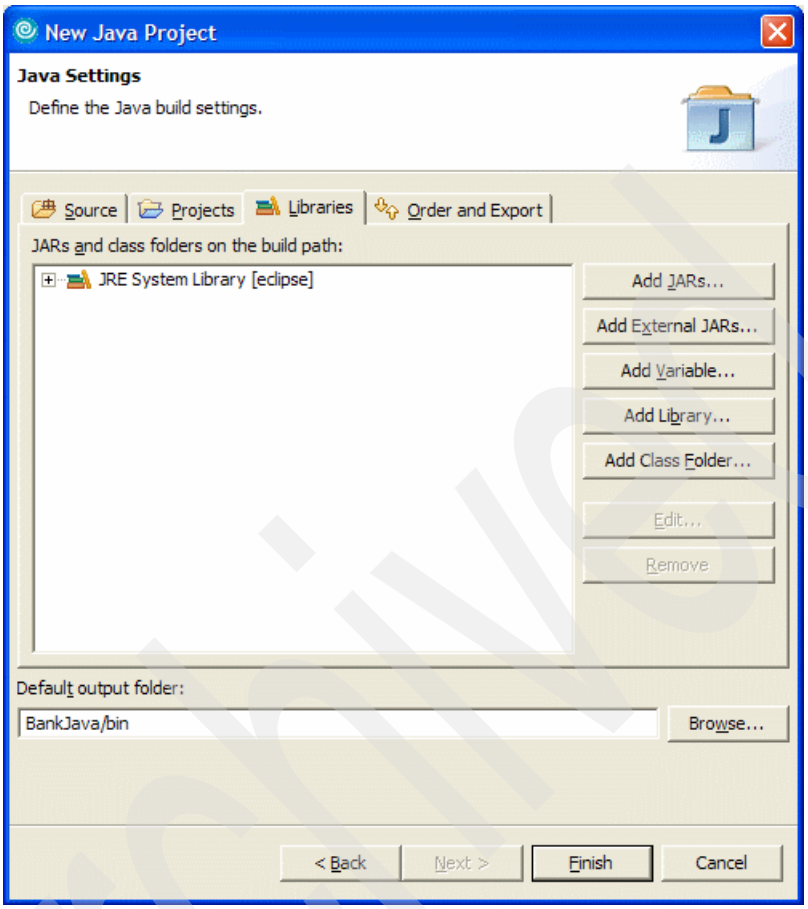


Figure 7-17 New Java Project: Library Settings

– Order and Export tab

On the last tab, Order and Export, you can specify the order in which you want items in the build path to be searched. Using the Up and Down buttons allows you to arrange the order of the classpath entries in the list (Figure 7-18 on page 243).

The checked list entries are marked as exported. Exported entries are visible to client projects that will use the new Java project being created. Use the Select All and Deselect All buttons to change the checked state of all entries. The source folder itself is exported, and not deselectable.

Archive

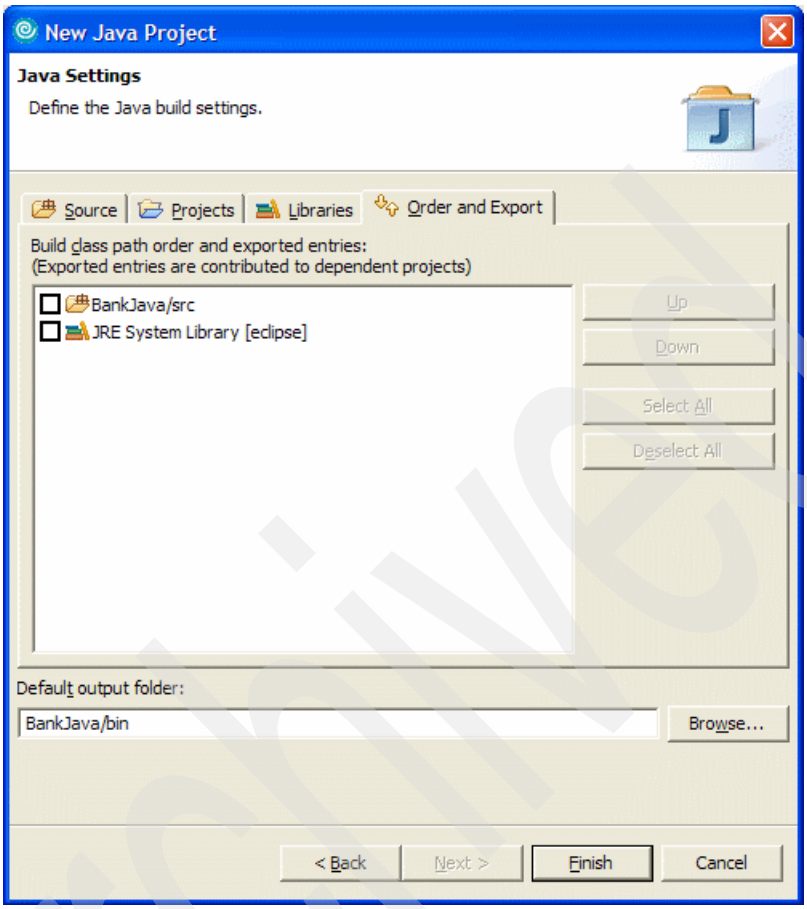


Figure 7-18 New Java project: Order and export settings

6. Click **Finish** to create the new project.

7.2.3 Create a class diagram

Before we can create Java elements (interfaces, classes, etc.) using the Diagram Navigator, we need to create a Class Diagram. In this example we will create a seperate folder named *diagrams* to hold the class and sequence diagrams.

Note: It is important to understand where you create the folder in which the diagrams will be placed.

1. Create the diagrams folder under the BankJava project.

Archived

a. Select the **BankJava** project from the Project Explorer, right-click, and select **New** → **Other**.

b. Expand **Simple**, select **Folder**, and then click **Next**.

c. Enter `diagrams` for the folder name field. Make sure that the parent folder name is BankJava. Click **Finish**.

2. From the Workbench, select **Window** → **Show View** → **Other**.
3. In the Show View dialog box, expand **Modeling** and select **Diagram Navigator**. Click **OK**.
4. Create an empty class diagram from the Open Diagram Navigator view.

   a. Select and expand **My Diagrams** → **BankJava** → **diagrams**.

   b. Right-click the newly created folder **diagrams** and select **New Diagram File** → **Class Diagram**.

   c. When the New Class Diagram wizard opens, enter the following and then click **Finish** (see Figure 7-19 on page 245):

      - Parent folder: `BankJava/diagrams`
      - File name: `BankJava-ClassDiagram`

Note: It is important to understand the ramification of the selection of the parent folder when creating diagrams.

In our example, we chose to create the diagrams in a separate folder off the root of the project. We observed the following behavior when working with Web Projects.

- When creating the folder off the root of the project, the folder and diagrams will be preserved when using a Project Interchange file.
- When creating the folder off the root of the project, the folder and diagrams will not be preserved when exporting to an EAR or WAR even if Export source files is checked.
- When creating the folder in JavaSource, we found that each time the diagram is saved, the builder will copy the diagram to the WEB-INF\classes folder. Since the WAR export utility will include all resources located in the WebContent tree, any diagrams located in the JavaSource tree will be included in the WAR file, regardless of the setting of the Export source files check box.

Refer to “Filtering the content of the EAR” on page 1222 for information on techniques for filtering files (include and exclude) when exporting the EAR.

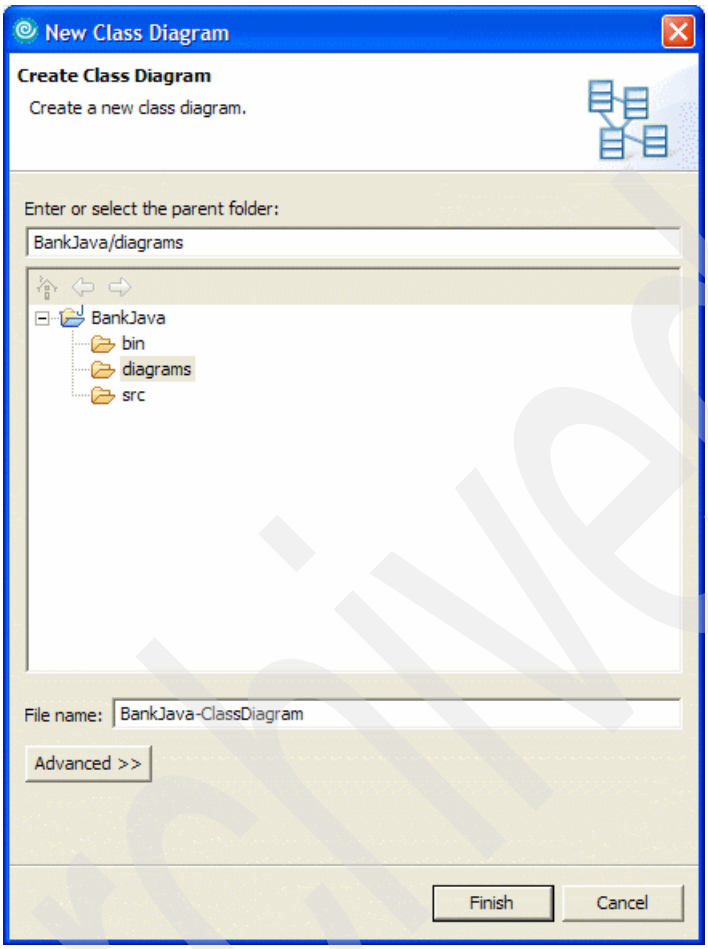


Figure 7-19 New Class Diagram

The file BankJava-ClassDiagram.dnx should now appear under the diagrams folder. If the visualization model for this file is not already open in the editor, double-click the file to open the visualization model.

Notice that the Java drawer is open in the Palette, as shown in Figure 7-20 on page 246. Rational Application Developer automatically opens the Java drawer by default for a Java project.

Archived

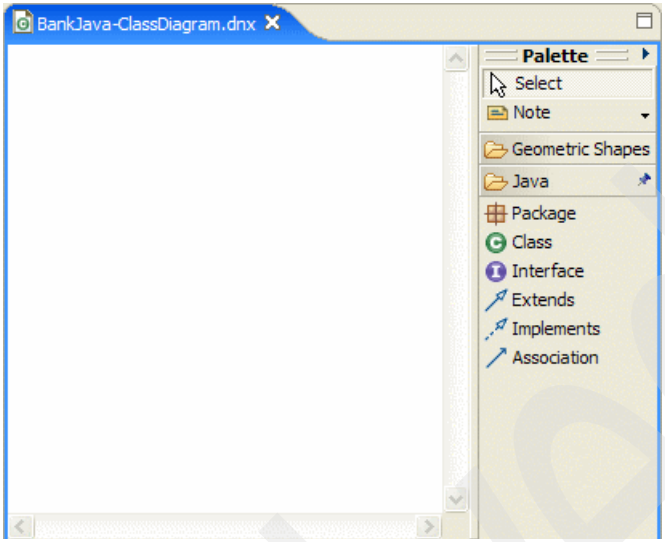


Figure 7-20 Blank Class Diagram with Java drawer opened in the Palette

7.2.4 Create Java packages

Once the Java project has been created, Java packages can be added to the project using either the Diagram Navigator or the Java Package wizard. Both of these methods are similar in the fact that the Diagram Navigator wizard pops up the Java Package wizard; from this point on the mechanism for creating packages is the same.

You will need to create all four packages listed in Table 7-2 by either using the Diagram Navigator or the Java Package wizard.

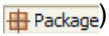
Table 7-2 Bank sample packages

Package name	Description
itso.bank.model	Contains the business model objects for the ITSO Bank
itso.bank.facade	Contains the interface
itso.bank.exception	Contains the exception classes
itso.bank.test	Contains a JUnit test case and an example for using the Code Review tool
itso.bank.client	Contains instance of bank with hardcoded customer data in main to run the bank application

Archive

Create a Java package using the Diagram Navigator

To create a Java package using the Diagram Navigator, do the following:

1. Click the Package icon () in the Java drawer.
2. Once the package icon is selected, click anywhere in the visualization model.
3. When the Java Package Wizard appears, enter the name of the package and then click **Finish**. For example, we entered the following values, as seen in Figure 7-21, to create the itso.bank.model package.
   - Source Folder: `BankJava/src`
   - Name: `itso.bank.model`

   The package is now created under src.

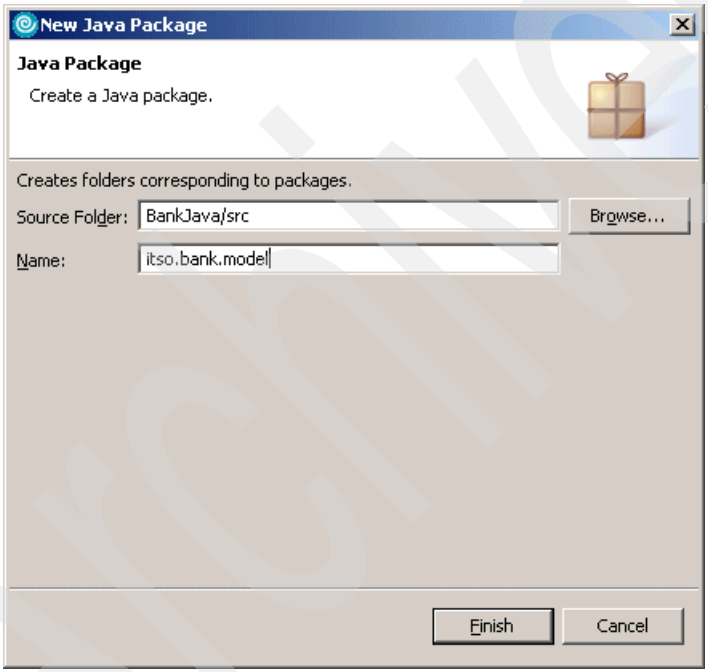


Figure 7-21 Create a Java package using the Diagram Navigator

4. Use Ctrl+S to save the Class Diagram.

Create a Java Package using the Java Package wizard

To create a Java package using the Java Package wizard, do the following:

1. In the Package Explorer, select the folder (for example, BankJava/src) where you want to create the package.
2. Right-click and select **New** → **Package**.

3. When the Java Package wizard appears, enter the Java package name and click **Finish**. For example, we entered the following:
   - Source Folder: `BankJava/src`
   - Name: `itso.bank.facade`
4. To add this package to the visualization model created previously, right-click the **itso.bank.model.facade.java** in the Package Explorer and then select **Visualize** → **Add to Current Diagram** from the context menu.
5. Press Ctrl+S to save the Class Diagram.

When you have added all four packages listed in Table 7-2 on page 246, the Class Diagram should look like Figure 7-22.

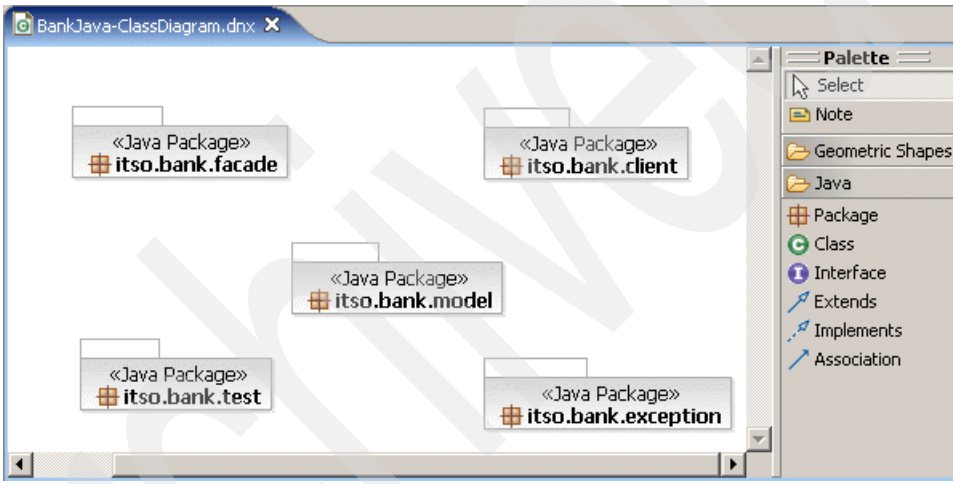


Figure 7-22 Visualization model after adding packages

7.2.5 Create a Java interface

This section describes how to create an interface in the visualization model for the Bank interface.

To create the Java interface, do the following:

1. Click the Interface icon (Interface) in the Java drawer to select it.
2. Once the Interface icon is selected, click anywhere in the visualization model.
3. When the Java Interface Wizard appears, enter the interface details and click **Finish**. For example, we entered the following (as seen in Figure 7-23 on page 249) for the Bank interface:
   - Source folder: `BankJava/src`

Archived

– Package: `itso.bank.facade`
– Name: `Bank`

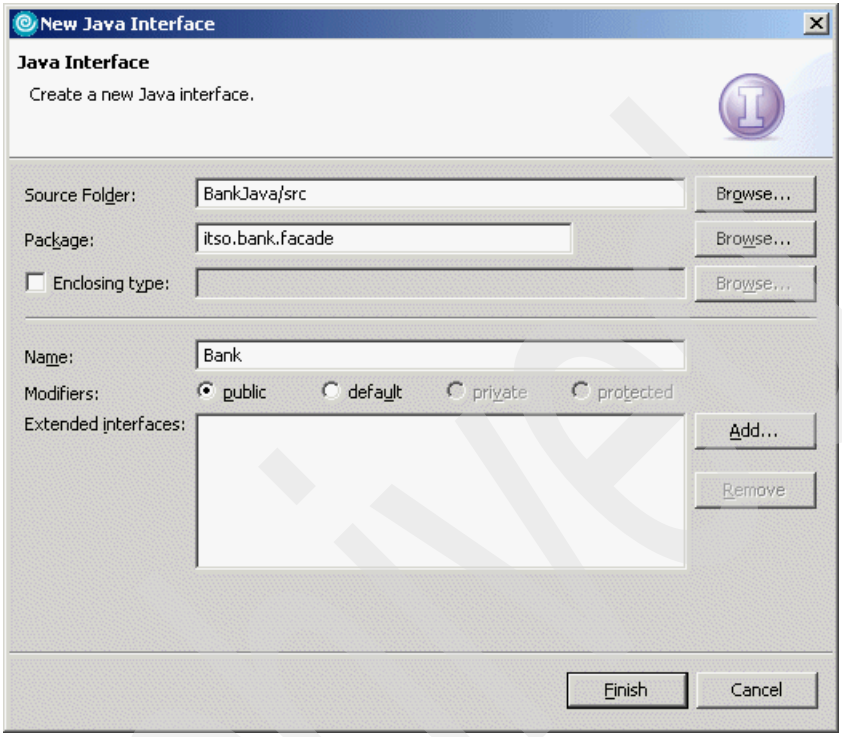


Figure 7-23 New Java Interface Wizard - Create the bank interface

4. Press Ctrl+S to save the Class Diagram.

7.2.6 Create Java classes

Classes can be created using either the Diagram Navigator or the Java Class wizard. For the Java Bank application, we will need to create the classes listed in Table 7-3.

Table 7-3 Bank Java classes

Class name	Package	Superclass	Modifiers	Interfaces
Transaction	itso.bank.model	Object	Public, abstract	Serializable
Credit	itso.bank.model	Transaction	Public	
Debit	itso.bank.model	Transaction	Public	

Class name	Package	Superclass	Modifiers	Interfaces
TransactionType	itso.bank.model	Object	Public	
Account	itso.bank.model	Object	Public	Serializable
Customer	itso.bank.model	Object	Public	Serializable
InvalidAccountException	itso.bank.exception	Exception	Public	
InvalidCustomerException	itso.bank.exception	Exception	Public	
InvalidTransactionException	itso.bank.exception	Exception	Public	
ITSOBank	itso.bank.facade	Object	Public	Bank
BankNames	itso.bank.facade	Object	Public	
BankClient **Note**: Check public static void main(String[] args).	itso.bank.client	Object	Public	

Important: Prior to creating the ITSOBank class, you must create the bank interface as described in 7.2.5, “Create a Java interface” on page 248. It is not possible in the Diagram Navigator or Java Class wizard to create a class that refers to a non-existing interface.

Create a Java class using the Diagram Navigator

To create the Java classes using the Diagram Navigator, do the following:

1. Click the Class icon ( ) in the Java drawer to select it.
2. Once the Class icon is selected, click anywhere in the visualization model.
3. When the Java Class Wizard appears, enter the details for the class to be created. For example, to create the Transaction class we entered the following:
   - Source folder: `BankJava/src`
   - Package: `itso.bank.model`
   - Name: `Transaction`
   - Modifiers:
     - Select **Public**.
     - Check **Abstract**.
4. Click **Add** next to Interfaces. Enter `serializable` in the Choose interfaces field. Select **Serializable** from the list, as seen in Figure 7-24, and click **OK**.

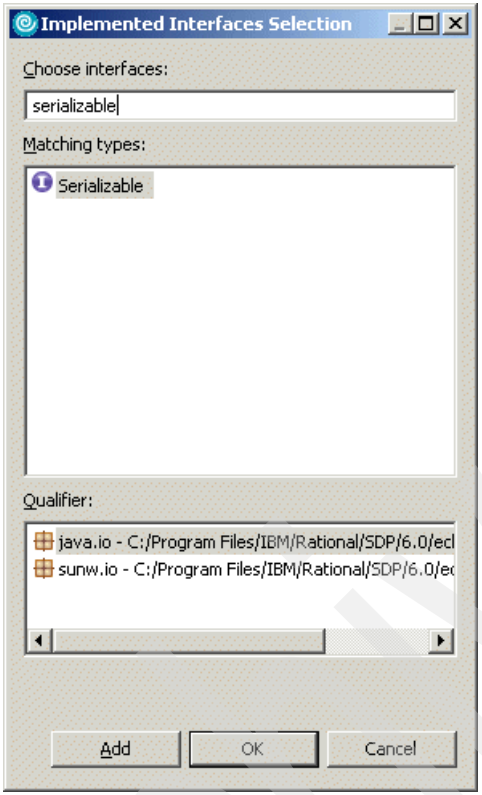


Figure 7-24 Seriazable interface dialog

After entering the serializable interface, the Create Java Class dialog should look like Figure 7-25 on page 252.

Archived

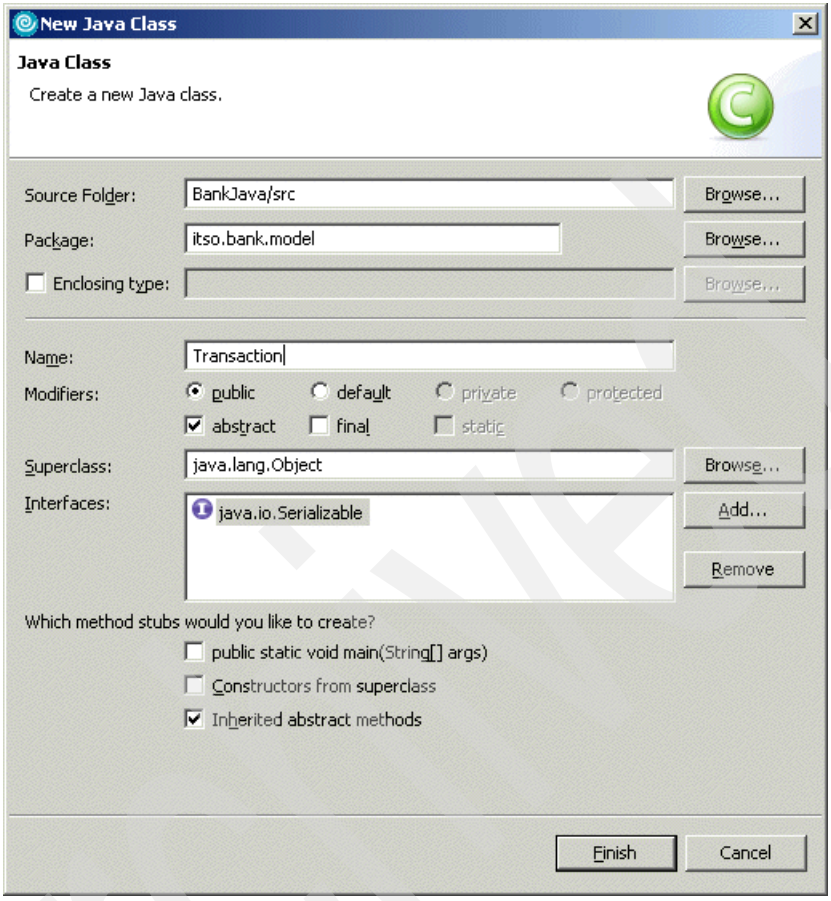


Figure 7-25 Create a class with a serializable interface

5. Click **Finish**.

   The package is now created under src.

6. Press Ctrl+S to save the Class Diagram.

   The newly created class and the updated visualization model are shown in Figure 7-26 on page 253.

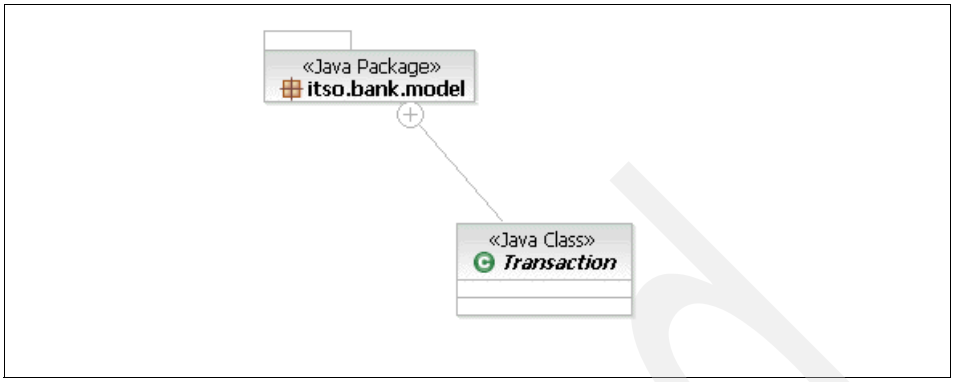


Figure 7-26 Visualization model after adding the Transaction class

Create a Java class using the Java Class wizard

To create a Java class using the Java Class wizard, do the following:

1. Right-click the package (for example, itso.bank.model), and select **New** → **Class**.
2. When the New Java Class wizard appears, enter the class details. For example, enter the following to create the Transaction class:
   - Source folder: `BankJava/src`
   - Package: `itso.bank.model`
   - Name: `Transaction`
   - Modifiers:
     - Select **Public**.
     - Check **Abstract**.
3. Click **Add** next to Interfaces. Enter `serializable` in the Choose interfaces field. Select **Serializable** from the list and click **OK**.
4. Add the class to the class diagram.
   a. Double-click **BankJava-ClassDiagram.dnx**, found in the diagrams folder.
   b. Right-click the class (for example, Transaction) in the Project Explorer and select **Visualize** → **Add to Current Diagram** from the context menu.

After creating all the classes listed in Table 7-2 on page 246 you may right-click in the class diagram and select **Arrange All**. You then manually move the classes as desired for better placement in the diagram.

Archived

7.2.7 Create the Java attributes and accessor methods

This section describes how to create the Java attributes and accessor (setter and getter) methods using the Java Field wizard.

Table 7-4 lists the Java attributes and accessor methods that need to be created for the Java Bank application classes.

Table 7-4 Java attributes and accessor methods

Class name	Attribute name	Initial value	Type	Visibility, modifier	Accessor methods
Transaction	timeStamp	Null	Date	Private	Yes
	amount	Null	BigDecimal	Private	Yes
	id	0	int	Private	Yes
TransactionType	type	Null	String	Private	No
	DEBIT	New TransactionType (“DEBIT”)	TransactionType	Public, static final	No
	CREDIT	New TransactionType (“CREDIT”)	TransactionType	Public, static final	No
Account	id	Null	String	Private	Yes
	balance	Null	BigDecimal	Private	Yes
Customer	ssn	Null	String	Private	Yes
	firstName	Null	String	Private	Yes
	lastName	Null	String	Private	Yes
	title	Null	String	Private	Yes
InvalidAccountException	id		String	Private	No
InvalidCustomerException	ssn		String	Private	No

Archive

Class name	Attribute name	Initial value	Type	Visibility, modifier	Accessor methods
InvalidTransactionException	transaction Type		String	Private	No
	amount		BigDecimal	Private	No
	accountNumber		String	Private	No
BankNames	name	Null	String	Private	No
	ITSOBANK	New BankNames (“ITSOBANK”)	BankNames	Public, static final	No

Create Java attributes using the Java Field wizard

To add attributes to a Java class (for example, Transaction class), do the following:

1. Move the cursor anywhere over the class (for example, Transaction).
2. When the action bar appears, as shown in Figure 7-27, click the Add new Java field icon (▫) to add attributes to the class.

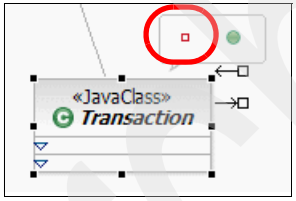


Figure 7-27 The action bar is used to add attributes and methods to a Java class

3. When the Create Java Field wizard appears, enter the attribute details. For example, we entered the following, as seen in Figure 7-28 on page 256, to create the timeStamp attribute:
   - Name: `timeStamp`
   - Type: `Date`
   - Initial value: `null`

Archive

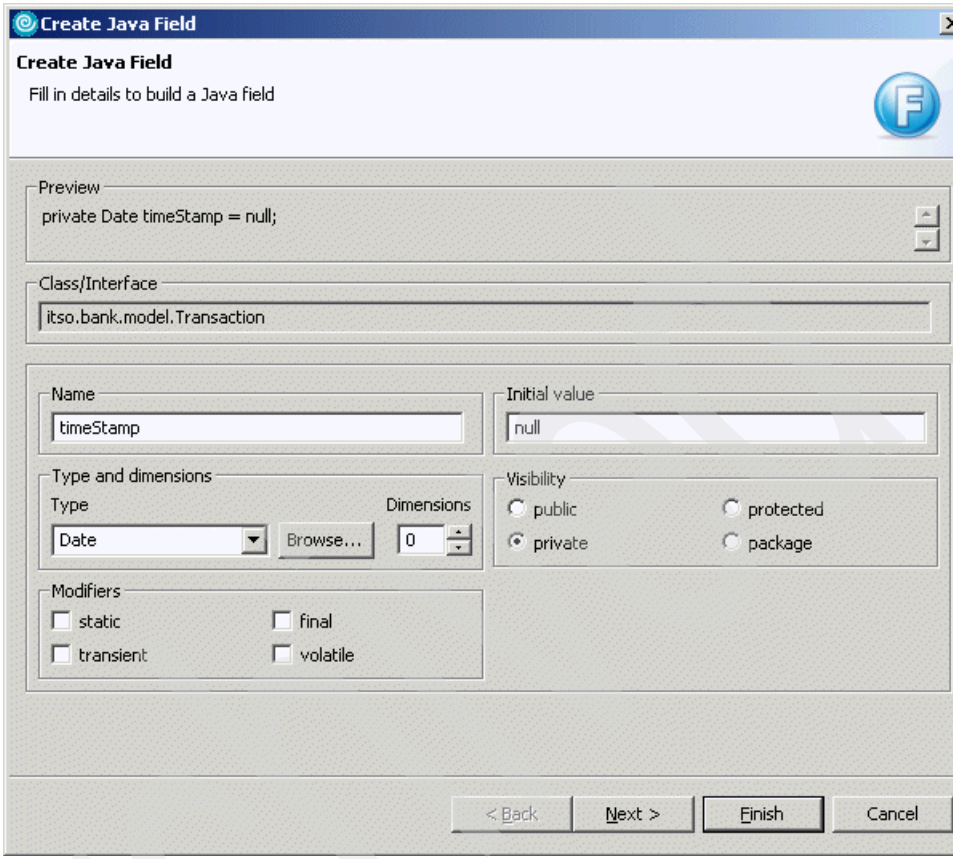


Figure 7-28 Java Field wizard - Add attribute

4. Click **Finish**.

Note: When clicking Finish, there are two issues that can cause the attribute to not be displayed in the class diagram:

- Type cannot be resolved.

  If the type is not resolved (for example, Date), press Ctrl+Shift+O to organize the imports. If one or more of the types are ambiguous, the Organize Import dialog will then appear. Select the full import name (for example, java.util.Date) and then click **OK**. In some cases, the type may not be defined yet, and the error is normal.

- Class diagram attribute compartment is collapsed (hides attributes).

  If the attribute compartment arrow is collapsed, the attributes will not be displayed. Simply click the arrow to expand the attributes compartment.

For this example, the Transaction class is updated and timeStamp now appears in the Attribute Compartment of Transaction in the class diagram.

5. Resolve the type by adding the import.

   If the type is not resolved (for example, Date), press Ctrl+Shift+O to open the Organize Imports dialog. Select the full import name (for example, java.util.Date) and then click **OK**.

6. Press Ctrl+S to save.

7. Repeat the process to add all the attributes listed in Table 7-4 on page 254.

Create accessor methods using the Java Field wizard

Create the accessor (getter and setter) methods for each of the attributes listed in Table 7-4 on page 254, using the Java Field wizard.

1. Select the attribute (for example, timeStamp) in the Attribute Compartment of the class (for example, Transaction).

2. Right-click and select **Refactor** → **Encapsulate Field**.

3. If prompted to Save all Modified Resources, click **OK**.

4. When the Encapsulate Field dialog appears, we entered the following, as seen in Figure 7-29 on page 258, and then clicked **OK**:

   – Getter name: `getTimeStamp` (supplied by default)

   – Setter name: `setTimeStamp` (supplied by default)

   – Insert new methods after: Select **As First Method**.

     This will insert the getter and setter methods as the initial methods in the Java source file.

     Tip: When adding subsequent accessors, you may consider inserting after the get accessor (set, then get) so that the get and set accessors are grouped in pairs within the Java source.

   – Access modifier: Select **Public**.

     This will create the setter and getter methods as public methods.

   – Field access in declaring class: Select **Keep field reference**.

     This will force any access to the timeStamp attribute within the Transaction class to use the field directly (not setter and getter).

   – Check **Generate Javadoc comments**.

   The public accessor methods are created in both the visualization model and Transaction class.

Archived

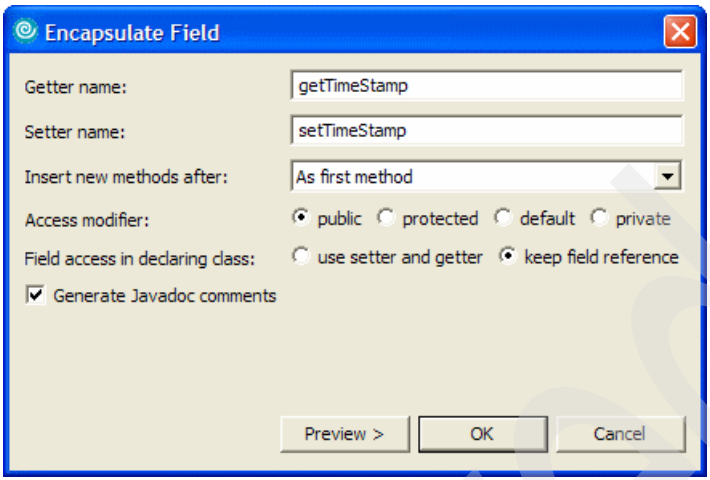


Figure 7-29 Encapsulate Field

5. Press Ctrl+S to save.
6. Repeat this process to create the accessor methods for each of the attributes listed in Table 7-4 on page 254.

7.2.8 Add method declarations to an interface

This section describes how to add method declarations to an interface using the Create Java Method wizard. For our working example, we will need to add all the method declarations listed in Table 7-5 to the bank interface.

Table 7-5 Method declarations for the bank interface

Method name	Type	Parameters (type name)	Exception name
processTransaction	void	- Customer customer - Account account - BigDecimal amount - TransactionType type	- InvalidCustomerException - InvalidAccountException - InvalidTransactionException
addCustomer	void	- Customer customer	- InvalidCustomerException
removeCustomer	void	- Customer customer	
getCustomer	Customer	- String ssn	- InvalidCustomerException
addAccount	void	- Customer customer - Account account	- InvalidCustomerException

Archived

Method name	Type	Parameters (type name)	Exception name
removeAccount	void	- Customer customer - Account account	
getAccount	Account	- Customer customer - String id	

To add a method declaration to an interface, do the following:

1. Move the cursor over the interface (for example, Bank).
2. When the action bar appears, click the method icon ( ) to add methods to the interface.
3. When the Java Method wizard appears, enter the method details. For example, we entered the following to create the processTransaction method declaration for the `Bank` interface:
   - Name: `processTransaction`
   - Type: `void`
4. For each exception the method should throw (if needed), do the following:
   a. Click **Add** next to the Throws table.
   b. Enter the search text for the exception name, select the exception from the list, and then click **OK**. For example, we added the following from the itso.bank.exception package:
      - `InvalidCustomerException`
      - `InvalidAccountException`
      - `InvalidTransactionException`
5. For each parameter in the method declaration, do the following:
   a. Click **Add** next to the Parameters table.
   b. Enter the Name and Type, and then click **OK**. For example, we added the following parameters to the table:
      - `Customer customer`
      - `Account account`
      - `BigDecimal amount`
      - `TransactionType type`
6. Resolve the type by adding the import.

   If the type is not resolved (for example, Date), press Ctrl+Shift+O to open the Organize Imports dialog. Select the full import name (for example, java.util.Date), and then click **OK**.

Archived

7. When complete, the Create Java Method wizard should look like Figure 7-30 on page 261. Click **Finish**.
8. Press Ctrl+S to save.

Note: When adding method declarations with exceptions or parameters to an interface using the Method wizard in IBM Rational Application Developer V6.0, we found that the parameters or exceptions where not added to the declaration in the Java source code. This works correctly for classes. To work around this issue in V6.0, we added the parameters and exceptions to the code manually.

When using Interim Fix 0002 or later this problem is fixed.

Archived

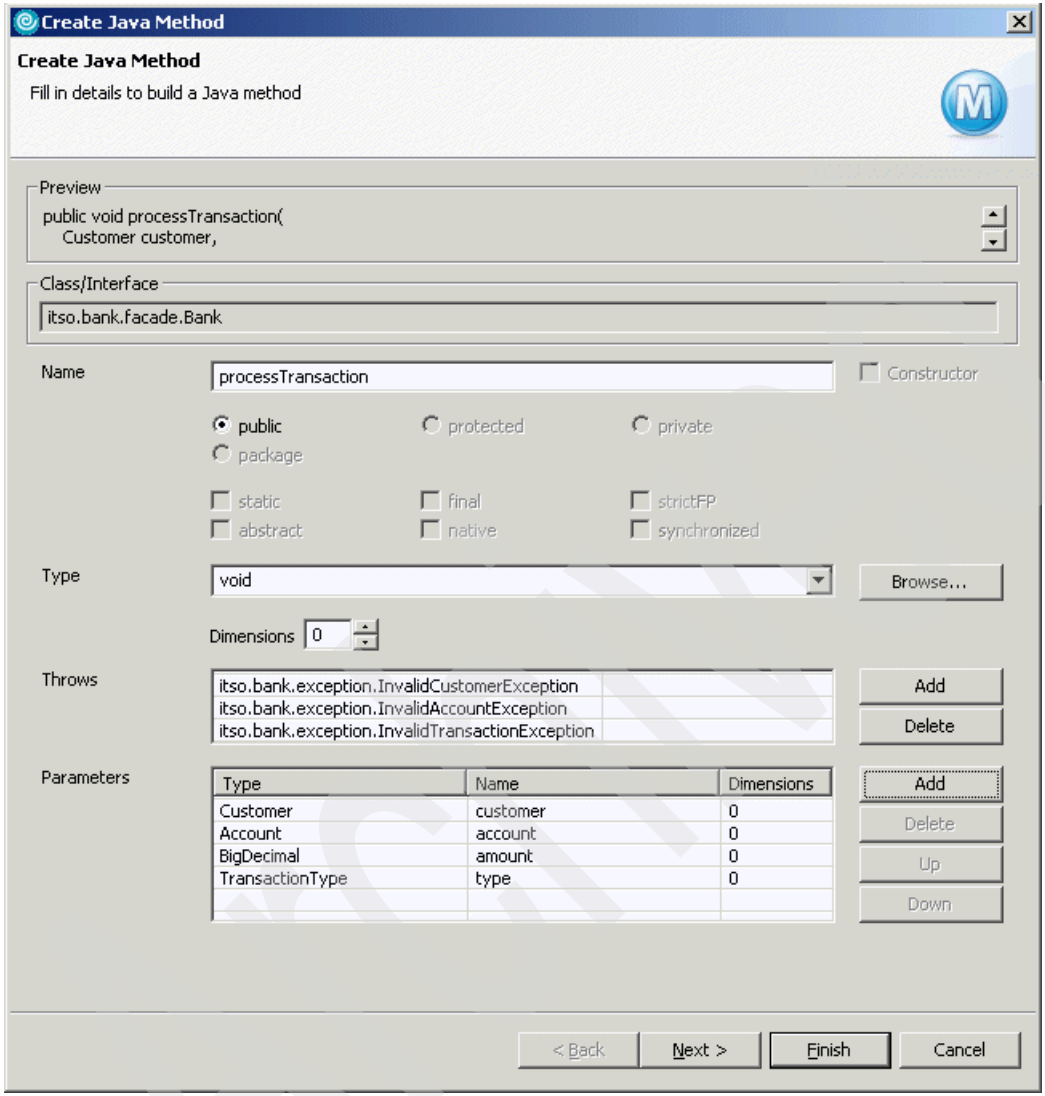


Figure 7-30 Create Java Method - Interface method declarations

9. Repeat the process to add each of the method declarations listed in Table 7-5 on page 258.

7.2.9 Add Java methods and constructors

This section describes how to create new Java methods and constructors using the Java Method wizard. Table 7-6 lists the methods and constructors that need to be added for the bank application sample.

Tip: It is a good practice to insert the constructors after the attributes and before the methods. Since we are adding constructors after adding attributes and accessors, the constructor will need to be moved manually.

Table 7-6 Methods and constructors to add for Bank sample

Method name	Indentifier, modifier	Type	Parameters (type name)	Exception
Transaction class				
Transaction	public, constructor		- BigDecimal amount	
process	public, abstract	BigDecimal	- BigDecimal accountBalance	
Credit class				
Credit	public, constructor		- BigDecimal amount	
process	public	BigDecimal	- BigDecimal accountBalance	
toString	public	String		
Debit class				
Debit	public, constructor		- BigDecimal amount	
process	public	BigDecimal	- BigDecimal accountBalance	
toString	public	String		
Account class				
Account	public, constructor		- String ID - BigDecimal balance	
processTransaction	public	void	- BigDecimal amount - TransactionType transactionType	- InvalidTransactionException

Archived

	Method name	Indentifier, modifier	Type	Parameters (type name)	Exception
	getLog	public	List		
	toString	public	String		
Customer class					
	Customer	public, constructor		- String ssn - String firstName - String lastName	
	toString	public	String		
InvalidCustomerException class					
	InvalidCustomerException	public, constructor		- String ssn	
	getMessage	public	String		
InvalidAccountException class					
	InvalidAccountException	public, constructor		- String ssn	
	getMessage	public	String		
InvalidTransactionException class					
	InvalidTransactionException	public, constructor		- String accountNumber - String transactionType - BigDecimal amount	
	getMessage	public	String		
TransactionType class					
	TransactionType	private, constructor		- String type	
	toString	public	String		
ITSOBank class					
	processTransaction	public	void	- Customer customer - Account account - BigDecimal amount - TransactionType transactionType	- InvalidCustomerException - InvalidAccountException - InvalidTransactionException

Archived

	Method name	Indentifier, modifier	Type	Parameters (type name)	Exception
	addAccount	public	void	- Customer customer - Account account	- InvalidCustomerException
	addCustomer	public	void	- Customer customer	- InvalidCustomerException
	getAccount	public	void	- Customer customer - String id	- InvalidCustomerException
	getCustomer	public	Customer	- String id	
	removeAccount	public	void	- Customer customer - Account account	
	removeCustomer	public	void	- Customer customer	
BankNames class					
	BankNames	private, constructor		- String name	
BankClient class					
	main	public, static	void	- String[] args	
	getBank	private, static	Bank	BankNames bank	

To add methods to a Java class, do the following:

1. Move the cursor over the class (for example, Transaction).
2. When the action bar appears, click the method icon ( ) to add methods to the class.
3. When the Java Method wizard appears, enter the method details. For example, we entered the following, as seen in Figure 7-31 on page 266, for the process method:
   - Name: `process`
   - Select **public**.
   - Check **abstract**.
   - Type: `BigDecimal`
4. For each parameter in the method declaration, do the following:
   a. Click **Add** next to the Parameters table.
   b. Enter the Name and Type, and then click **OK**.

Archived

For example, we added the following parameters to the table for the process method:

```
BigDecimal accountBalance
```

The process method takes in the current account balance, processes the transaction, and returns the new account balance after the transaction has been processed successfully.

5. Click **Finish**.
6. Press Ctrl+S to save.

Archived

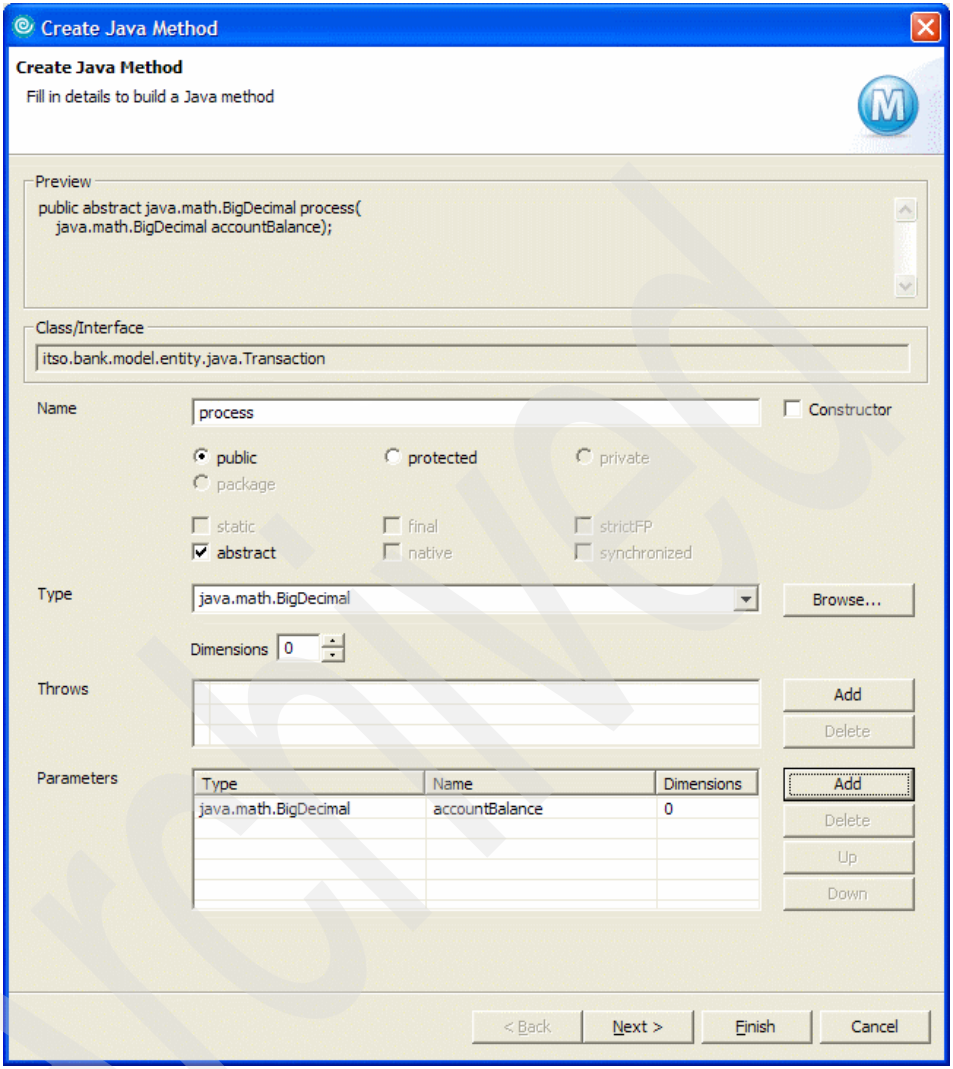


Figure 7-31 Java Method Wizard

7. Repeat the process to add each method and contructor listed in Table 7-6 on page 262.

> **Note:** After adding the methods you will have many errors in the problems view. These will be resolved in subsequent sections.

7.2.10 Define relationships (extends, implements, association)

This section includes the following tasks:

- Extends relationship
- Implements relationship
- Association relationship

Extends relationship

In Rational Application Developer it is possible to model the class inheritance via the Diagram Navigator. To create an *extends* relationship between existing classes in the Diagram Navigator, click the Extends icon(Extends) and drag the mouse with the left mouse button down from any point on the child class to the parent class.

You will notice that the Diagram Navigator already contains the extends relationship, due to the fact that the transaction was specified as the superclass when the credit and Debit classes were created in 7.2.6, “Create Java classes” on page 249. You do not need to do anything else to relate the credit or Debit classes to the Transaction class.

Implements relationship

You will notice that the visualization model already contains the *implements relationship* due to the fact that the bank was specified in the Interfaces field in the ITSOBank New Class Java Wizard. You do not need to do anything else to relate ITSOBank and Bank using the implements relationship for the sample.

To create an implements relationship between existing classes and interfaces in the visualization model, click the **Implements** relationship icon (Implements) and drag the mouse with the left mouse button down from any point on the child class to the parent class.

Association relationship

The Customer class can have one or more accounts, and an Account class can be associated with one or more customers. Additionally, each account is composed of zero or more transactions for the purpose of logging based on the transaction history associated with the account.

We will now demonstrate how to define a two-way association between account and customer:

1. Create an association from account to customer.
   a. Click the Java **Association** icon (Association) in the Java drawer to select it.

b. Once the association icon is selected, click the **Account** class and drag with the left-mouse button pressed down to the Customer class in the visualization model.

c. When the Create Association wizard appears, enter the association details. For example, we entered the following, as seen in Figure 7-32, and then clicked **Finish**:

- Name: `customers`
- Initial value: `null`
- Dimensions: Select **1** (default is 0).

d. Press Ctrl+S to save.

This creates an attribute in the Account class that is an array of type Customer (the target type) to reflect the association.

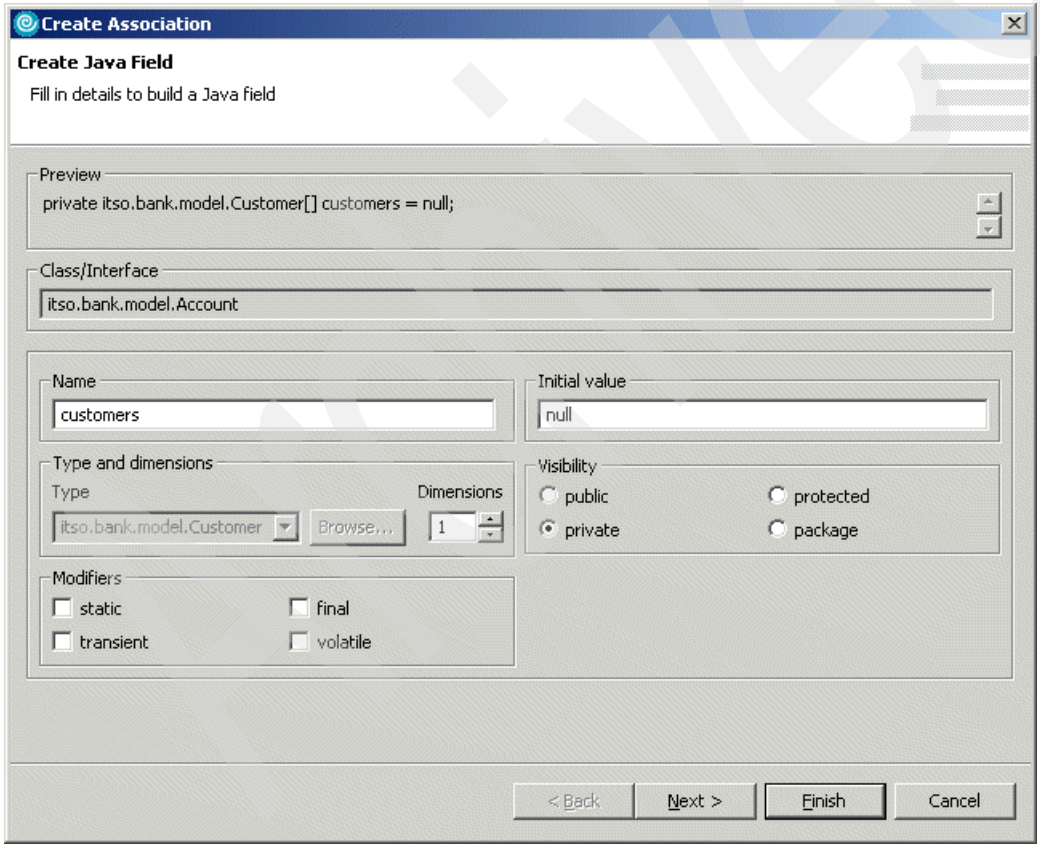


Figure 7-32 Create associations between an account and one or more customers

2. Repeat the steps above to create an association from customer to account.

- Name: `accounts`
- Initial value: `null`
- Dimensions: Select **1** (default is 0).

3. Repeat the steps above to create an association from account to transactions.
   - Name: `transactions`
   - Initial value: `null`
   - Dimensions: Select **1** (default is 0).
4. Repeat the steps above to create an association from ITSOBank to account.
   - Name: `customerAccounts`
   - Initial value: `null`
   - Dimensions: Select **1** (default is 0).

Modify collection type for multi-dimensional association

When a multi-dimensional association is created, the wizard will implement the association using Java arrays. This is not always the desired behavior. In our example, we need the accounts, transactions, and customerAccounts associations to be of dynamic size. Both accounts and transactions should use the java.util.ArrayList class, while the customerAccounts should use java.util.Map.

We will modify the types as follows:

1. From the Project Explorer, double-click the **Customer** class to open the Customer.java in the Java editor. Modify the following and save the file.
   - From:

     ```
     private itso.bank.model.Account[] accounts = null;
     ```

   - To:

     ```
     private java.util.ArrayList accounts = null;
     ```

2. From the Project Explorer, double-click the **Account** class to open the Account.java in the Java editor. Modify the following and save the file.
   - From:

     ```
     private itso.bank.model.Transaction[] transactions = null;
     ```

   - To:

     ```
     private java.util.ArrayList transactions = null;
     ```

3. From the Project Explorer, double-click the **ITSOBank** class to open the ITSOBank.java in the Java editor. Modify the following and save the file.
   - From:

     ```
     private itso.bank.model.Account[] customerAccounts = null;
     ```

– To:

```
private java.util.Map customerAccounts = null;
```

The Updated visualization model after the associations are created between the account, customer, and transactions in our example are shown in Figure 7-33.

Note: We found that in some instances, the association arrow disappears when modifying the type and is displayed as a field (attribute).

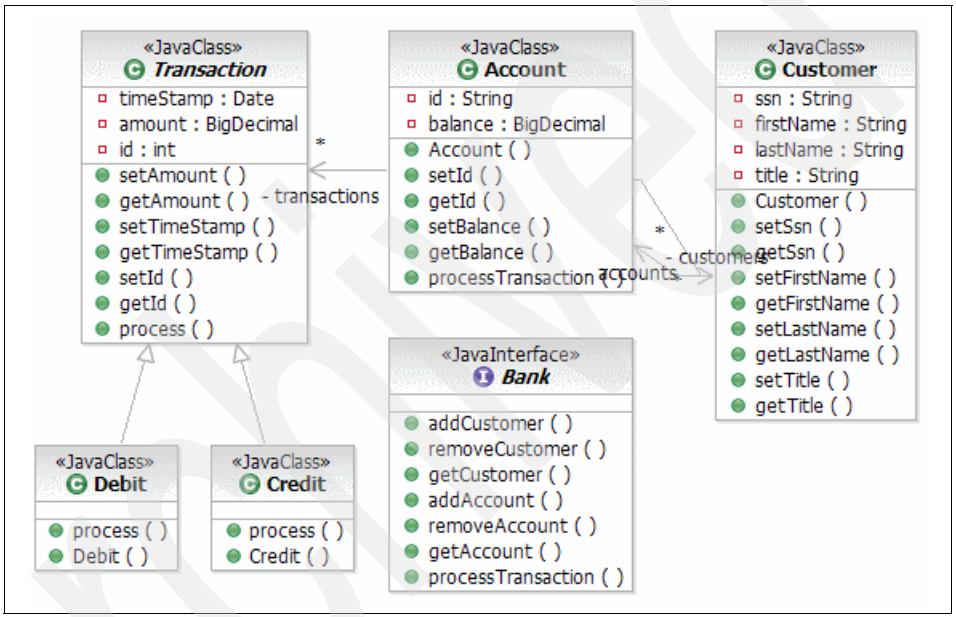


Figure 7-33 Class diagram with associations

7.2.11 Implement the methods for each class

Thus far we have included a step-by-step approach with the objective of demonstrating the tooling and logical process. This section provides the source code for each of the methods to be implemented in each of the classes.

Note: To save time you can import the sample Java code provided in the c:\6449code\java\BankJava.zip Project Interchange file. For details refer to Appendix B, “Additional material” on page 1395.

Implement the Transaction class methods

Example 7-3 lists the Java source for the Transaction class methods.

Example 7-3 Transaction class

```
package itso.bank.model;

import java.io.Serializable;
import java.math.BigDecimal;
import java.util.Date;

public abstract class Transaction implements Serializable {
   private Date timeStamp = null;
    private BigDecimal amount = null;
    private int id = 0;

    public Transaction(BigDecimal amount){
       this.amount = amount;
   }

   public void setAmount(BigDecimal amount) {
      this.amount = amount;
   }

   public BigDecimal getAmount() {
      return amount;
   }

   public void setTimeStamp(java.util.Date timeStamp) {
      this.timeStamp = timeStamp;
   }

   public java.util.Date getTimeStamp() {
      return timeStamp;
   }

   public void setId(int id) {
      this.id = id;
   }

   public int getId() {
      return id;
   }
   public abstract BigDecimal process(
         BigDecimal accountBalance);
}
```

Archived

Implement the Credit class methods

Example 7-4 lists the Java source for the Credit class methods.

Example 7-4 Credit class

```
package itso.bank.model;

import java.math.BigDecimal;

public class Credit extends Transaction {

   public Credit(BigDecimal amount) {
      super(amount);
   }

   public BigDecimal process(BigDecimal accountBalance) {

      BigDecimal newAccountBalance = accountBalance;
      if (getAmount() != null
         && getAmount().compareTo(new BigDecimal(0))>0){
         newAccountBalance = accountBalance.add(getAmount());
         setTimeStamp(new java.util.Date());
      } else {
         String amount = null;
         if (getAmount() != null){
            System.out.println("Credit: Could not process "
                  +"Transaction. Reason: Negative/Zero Credit Amount. Amount: $"
                  +getAmount().setScale(2,BigDecimal.ROUND_HALF_EVEN));
         } else {
            System.out.println("Credit: Could not process "
                  +"Transaction. Reason: Null Credit Amount.");

         }
      }
      return newAccountBalance;
   }

   public String toString(){
      if (getAmount()!=null){
         String message;
         if (getTimeStamp()!=null){
            message = "Credit: --> Amount $"+
            getAmount().setScale(2,BigDecimal.ROUND_HALF_EVEN)
            + " on " + getTimeStamp();
         } else {
            message = "Amount that will be Credited to the Account is $"+
            getAmount().setScale(2,BigDecimal.ROUND_HALF_EVEN);
         }
         return message;
```

Archived

```
      }

      return super.toString();
   }
}
```

Implement the Debit class methods

Example 7-5 lists the Java source for the Debit class methods.

Example 7-5 Debit class

```
package itso.bank.model;

import java.math.BigDecimal;

public class Debit extends Transaction {

   public Debit(BigDecimal amount) {
      super(amount);
   }

   public BigDecimal process(BigDecimal accountBalance) {
      BigDecimal newAccountBalance = accountBalance;
      if (getAmount() != null
            && accountBalance.compareTo(getAmount())>0){
         newAccountBalance = accountBalance.subtract(getAmount());
         setTimeStamp(new java.util.Date());
      } else {
         String amount = null;
         if (getAmount() != null){
            System.out.println("Debit: Could not process "
                  +"Transaction. Reason: Negative/Zero Debit Amount. Amount: $"
                  +getAmount().setScale(2,BigDecimal.ROUND_HALF_EVEN));
         } else {
            System.out.println("Debit: Could not process "
                  +"Transaction. Reason: Null Debit Amount.");
         }
      }
      return newAccountBalance;
   }

   public String toString(){
      if (getAmount()!=null){
         String message;
         if (getTimeStamp()!=null){
            message = "Debit: --> Amount $"+
            getAmount().setScale(2,BigDecimal.ROUND_HALF_EVEN)
            + " on " + getTimeStamp();
```

Archived

```
         } else {
            message = "Amount that the Account will be Debited by is $"+
            getAmount().setScale(2,BigDecimal.ROUND_HALF_EVEN);
         }
         return message;
      }

      return super.toString();
   }
}
```

Implement the Account class methods

Example 7-6 lists the Java source for the Account class methods.

Example 7-6 Account class

```
/*
 * File Name:  Account.java
 */
package itso.bank.model;

import itso.bank.exception.InvalidTransactionException;
import java.io.Serializable;
import java.math.BigDecimal;
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

public class Account implements Serializable {

    private String id = null;
    private BigDecimal balance = null;
    private Customer[] customers = null;
    private ArrayList transactions = null;

   public Account(String id, BigDecimal balance) {
      this.id = id;
      this.balance = balance;
      this.transactions = new ArrayList();
   }

   public void setId(String id) {
      this.id = id;
   }

   public String getId() {
      return id;
   }
```

Archived

```
public void setBalance(BigDecimal balance) {
   this.balance = balance;
}

public BigDecimal getBalance() {
   return balance;
}
public void processTransaction(BigDecimal amount,
      TransactionType transactionType) throws InvalidTransactionException{
   BigDecimal newBalance = null;
   Transaction transaction = null;
   if (transactionType==TransactionType.CREDIT){
      transaction = new Credit(amount);
   } else if (transactionType==TransactionType.DEBIT){
      transaction = new Debit(amount);
   } else {
      System.out.println ( "Invalid Transaction, Please use Debit/Credit." +
            "No other Transactions are currently supported.");
      return;
   }

   newBalance=transaction.process(balance);
   if(newBalance.compareTo(balance)!=0){
      balance = newBalance;
      transactions.add(transaction);
   } else {
      throw new InvalidTransactionException(id, transactionType.toString(), amount);
   }
}

public List getLog(){
   return Collections.unmodifiableList(transactions);
}

public String toString(){
   StringBuffer account = new StringBuffer();
   account.append("Account"+id+": --> Balance: $"
      + balance.setScale(2,BigDecimal.ROUND_HALF_EVEN));
   if (transactions.size()>0){
      account.append(System.getProperty("line.separator"));
      account.append("    Transactions: ");
      account.append(System.getProperty("line.separator"));
      for(int i=0;i<transactions.size();i++){
         account.append("        "+i+". "+(Transaction)transactions.get(i));
      }
   }
   return account.toString();
}
```

Archived

```
}
```

Implement the Customer class methods

Example 7-7 lists the Java source for the Customer class methods.

Example 7-7 Customer class

```
package itso.bank.model;

import java.io.Serializable;
import java.util.ArrayList;

public class Customer implements Serializable {
    private String ssn = null;
    private String firstName = null;
    private String lastName = null;
    private String title = null;
   private ArrayList accounts = null;

   public Customer(String ssn, String firstName,
         String lastName) {
      this.ssn=ssn;
      this.firstName=firstName;
      this.lastName=lastName;
   }
   /**
    * @param ssn The ssn to set.
    */
   public void setSsn(String ssn) {
      this.ssn = ssn;
   }
   /**
    * @return Returns the ssn.
    */
   public String getSsn() {
      return ssn;
   }
   /**
    * @param firstName The firstName to set.
    * @deprecated
    */
   public void setFirstName(String firstName) {
      this.firstName = firstName;
   }
   /**
    * @return Returns the firstName.
    */
   public String getFirstName() {
```

Archived

```
        return firstName;
    }
    /**
     * @param lastName The lastName to set.
     */
    public void setLastName(String lastName) {
        this.lastName = lastName;
    }
    /**
     * @return Returns the lastName.
     */
    public String getLastName() {
        return lastName;
    }
    /**
     * @param title The title to set.
     */
    public void setTitle(String title) {
        this.title = title;
    }
    /**
     * @return Returns the title.
     */
    public String getTitle() {
        return title;
    }
    /**
     * @return Returns the accounts.
     */
    public ArrayList getAccounts() {
        return accounts;
    }
    /**
     * @param accounts The accounts to set.
     */
    public void setAccounts(ArrayList accounts) {
        this.accounts = accounts;
    }

    public String toString(){
        StringBuffer buffer = new StringBuffer("Customer: --> ssn = " + ssn + " : FirstName = "
+ firstName + " : LastName = " + lastName);
        if (accounts!=null){
            for(int i=0;i<accounts.size();i++){
                buffer.append(System.getProperty("line.separator"));
                buffer.append("    "+(Account)accounts.get(i));
            }
        }
        return buffer.toString();
```

Archived

```
    }
}
```

Implement the TransactionType class methods

Example 7-8 lists the Java source for the TransactionType class methods.

Example 7-8 TransactionType class

```
package itso.bank.model;

public class TransactionType {
   // Simple Implementation of a type-safe enumeration,
   // NOTE: this class is used for serializable uses
   private String type = null;

   public static final TransactionType DEBIT = new TransactionType ("DEBIT");
   public static final TransactionType CREDIT = new TransactionType ("CREDIT");

   private TransactionType(String type){
      this.type = type;
   }

   public String toString(){
      return type;
   }
}
```

Implement the InvalidAccountException class methods

Example 7-9 lists the Java source for the InvalidAccountException class methods.

Example 7-9 InvalidAccountException class

```
package itso.bank.exception;


public class InvalidAccountException extends Exception {
    private String id;

   public InvalidAccountException(String id){
      this.id=id;
   }

   public String getMessage(){
      return "Invalid Account. Id: " + id;
   }
```

Archived

```
}
```

Implement the InvalidCustomerException class methods

Example 7-10 lists the Java source for the InvalidCustomerException class methods.

Example 7-10 InvalidCustomerException class

```
package itso.bank.exception;

public class InvalidCustomerException extends Exception {
    private String ssn;

   public InvalidCustomerException(String ssn){
      this.ssn=ssn;
   }

   public String getMessage(){
      return "Invalid Customer. SSN: " + ssn;
   }
}
```

Implement the InvalidTransactionException class methods

Example 7-11 lists the Java source for the InvalidTransactionException class methods.

Example 7-11 InvalidTransactionException class

```
package itso.bank.exception;

import java.math.BigDecimal;

public class InvalidTransactionException extends Exception {
    private String transactionType;
    private BigDecimal amount;
    private String accountNumber;

   public InvalidTransactionException(String accountNumber, String transactionType, BigDecimal
amount){
      this.transactionType=transactionType;
      this.amount=amount;
      this.accountNumber=accountNumber;
   }

   public String getMessage(){
      return "Transaction. AccountNumber: "
         + accountNumber
```

Archived

```
          + " Transaction: "
          + transactionType
          + " Amount: $"
          + amount.setScale(2,BigDecimal.ROUND_HALF_EVEN);
    }
}
```

Implement the Bank interface

Example 7-12 lists the Java source for the Bank interface.

Example 7-12 Bank interface

```
package itso.bank.facade;

import java.math.BigDecimal;
import itso.bank.exception.InvalidAccountException;
import itso.bank.exception.InvalidCustomerException;
import itso.bank.exception.InvalidTransactionException;
import itso.bank.model.Account;
import itso.bank.model.Customer;
import itso.bank.model.TransactionType;

public interface Bank {

    public void processTransaction(Customer customer, Account account,
            BigDecimal amount, TransactionType type)
    throws InvalidCustomerException, InvalidAccountException, InvalidTransactionException;

    public void addCustomer(Customer customer)
    throws InvalidCustomerException;

    public void removeCustomer(Customer customer);

    public Customer getCustomer(String ssn)
    throws InvalidCustomerException;

    public void addAccount(Customer customer, Account account)
    throws InvalidCustomerException;

    public void removeAccount(Customer customer, Account account);

    public Account getAccount(Customer customer, String id);
}
```

Implement the BankNames class methods

Example 7-13 lists the Java source for the BankName class methods.

Archived

Example 7-13 BankNames class

```
package itso.bank.facade;

public class BankNames {

   private String name;
   public static final BankNames ITSOBANK = new BankNames("ITSOBANK");

   private BankNames(String name){
      this.name=name;
   }
}
```

Implement the ITSOBank class methods

Example 7-14 lists the Java source for the ITSOBank class methods.

Example 7-14 ITSOBank class

```
package itso.bank.facade;

import itso.bank.exception.InvalidAccountException;
import itso.bank.exception.InvalidCustomerException;
import itso.bank.exception.InvalidTransactionException;
import itso.bank.model.Account;
import itso.bank.model.Customer;
import itso.bank.model.TransactionType;
import java.math.BigDecimal;
import java.util.ArrayList;
import java.util.Hashtable;
import java.util.Map;

public class ITSOBank implements Bank {
   private Map customers = new Hashtable();
   private Map accounts = new Hashtable();
   private Map customerAccounts = new Hashtable();

   public void addAccount(Customer customer, Account account) throws InvalidCustomerException
{

      Customer validCustomer = (Customer)customers.get(customer.getSsn());
      if (validCustomer!=null){
         ArrayList list = (ArrayList)customerAccounts.get(customer.getSsn());
         list.add(account);
         validCustomer.setAccounts(list);
      } else {
         throw new InvalidCustomerException(customer.getSsn());
      }
```

Archived

```
    }
    /* (non-Javadoc)
     * @see itso.bank.facade.Bank#addCustomer(itso.bank.model.Customer)
     */
    public void addCustomer(Customer customer) throws InvalidCustomerException {
        if (customers.get(customer)==null){
            customers.put(customer.getSsn(),customer);
            customerAccounts.put(customer.getSsn(), new ArrayList());
        } else {
            throw new InvalidCustomerException(customer.getSsn());
        }
    }

    public Account getAccount(Customer customer, String id) {
        // TODO Auto-generated method stub
        return null;
    }

    public Customer getCustomer(String ssn) throws InvalidCustomerException {
        Customer customer=(Customer)customers.get(ssn);
        if (customer==null){
            throw new InvalidCustomerException(ssn);
        }
        return customer;
    }

    public void processTransaction(Customer customer, Account account,
            BigDecimal amount, TransactionType type) throws InvalidCustomerException,
InvalidAccountException, InvalidTransactionException{
        if(customers.get(customer.getSsn())!=null){
            ArrayList accounts = customer.getAccounts();
            if (accounts.contains(account)){
               account.processTransaction(amount,type);
            } else {
               throw new InvalidAccountException(account.getId());
            }
        } else {
            throw new InvalidCustomerException(customer.getSsn());
        }
    }

    public void removeAccount(Customer customer, Account account) {
        // TODO Auto-generated method stub

    }

    public void removeCustomer(Customer customer) {
        // TODO Auto-generated method stub
```

Archived

```
   }
}
```

Implement the BankClient class methods

Example 7-15 lists the Java source for the BankClient class methods.

Example 7-15 BankClient class

```
/*
 * File Name:  BankClient.java
 */
package itso.bank.client;

import itso.bank.exception.InvalidAccountException;
import itso.bank.exception.InvalidCustomerException;
import itso.bank.exception.InvalidTransactionException;
import itso.bank.facade.Bank;
import itso.bank.facade.BankNames;
import itso.bank.facade.ITSOBank;
import itso.bank.model.Account;
import itso.bank.model.Customer;
import itso.bank.model.TransactionType;

import java.math.BigDecimal;

public class BankClient {
   //Customer customer;

   public static void main(String[] args) {
      Bank bank = getBank(BankNames.ITSOBANK);
      try {
         Customer customer1 = new Customer("111-11-1111","Jane","Doe");
         bank.addCustomer(customer1);
         System.out.println ( "Successfully Added customer1. "+customer1);
         Account account11 = new Account("11",new BigDecimal(10000.00D));
         bank.addAccount(customer1,account11);
         Account account12 = new Account("12",new BigDecimal(11234.23));
         bank.addAccount(customer1,account12);
         System.out.println("Successfully Added 2 Accounts to Customer1... ");
         System.out.println(customer1);

         bank.processTransaction(customer1,account11,new
BigDecimal(2399.99D),TransactionType.DEBIT);

         System.out.println("Sucessfully Debited Account11 by $2399.99.. Updated Customer
Accounts...");
         System.out.println(customer1);
```

Archived

```
      }catch(InvalidCustomerException e){
         e.printStackTrace();
      }catch(InvalidAccountException e){
         e.printStackTrace();
      }catch(InvalidTransactionException e){
         e.printStackTrace();
      }
   }

   private static Bank getBank (BankNames bank){
      if(bank==BankNames.ITSOBANK){
         return new ITSOBank ();
      }
      return null;
   }
}
```

Note: The BankJava.zip Project Interchange file includes additional classes for testing in subsequent sections and chapters.

When complete the class diagram should look like Example 7-34.

Archived

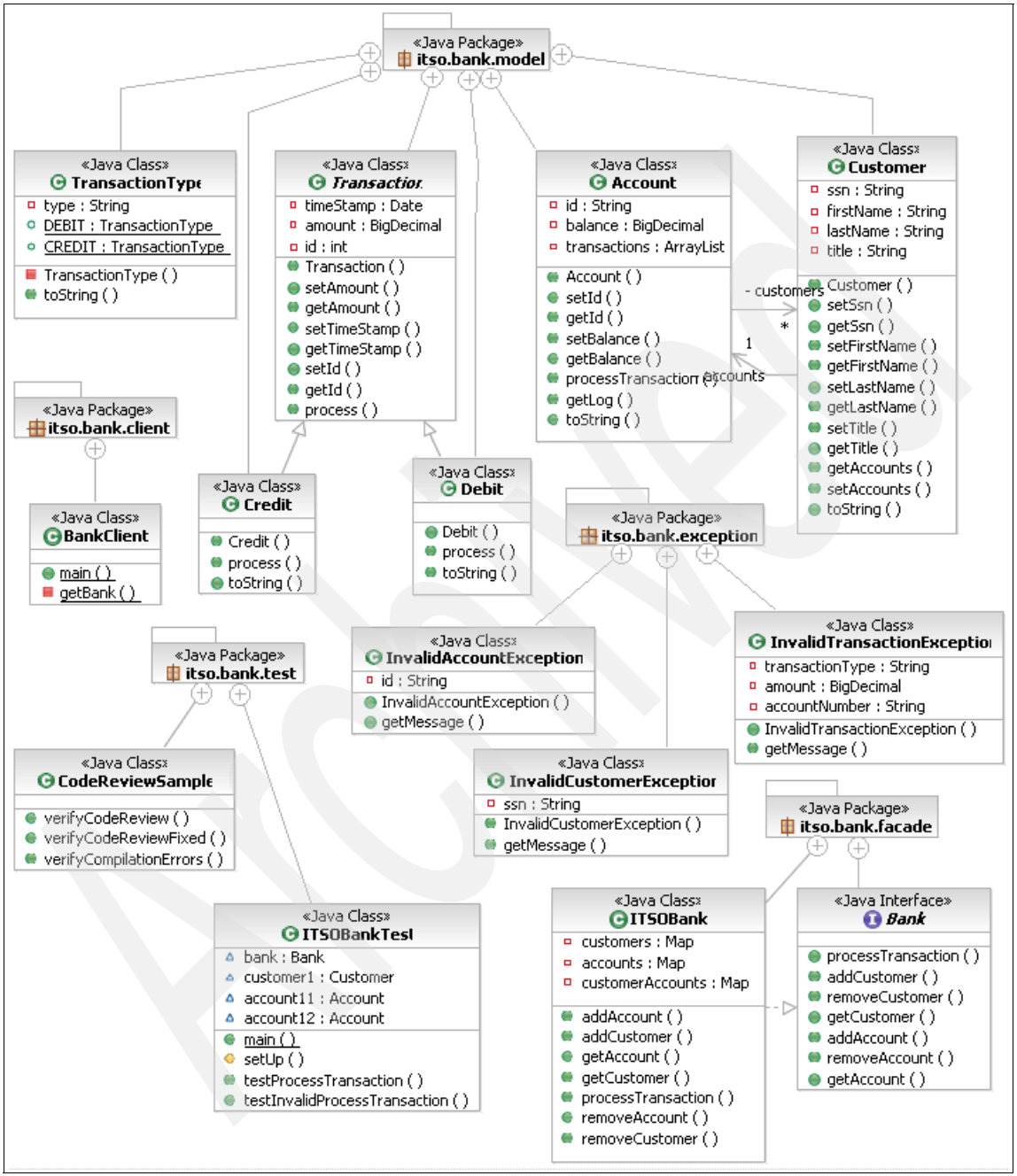


Figure 7-34 Completed class diagram - Banking sample

7.2.12 Run the Java Bank application

Once you have completed the Bank application sample and resolved any outstanding errors, you are ready to test the Bank application.

1. From the Workbench, expand **BankJava** → **src** → **itso.bank.client**.
2. Select **BankClient.java**.
3. Right-click and select **Run** → **Java Application**.

   You should see output in the console like Figure 7-35.

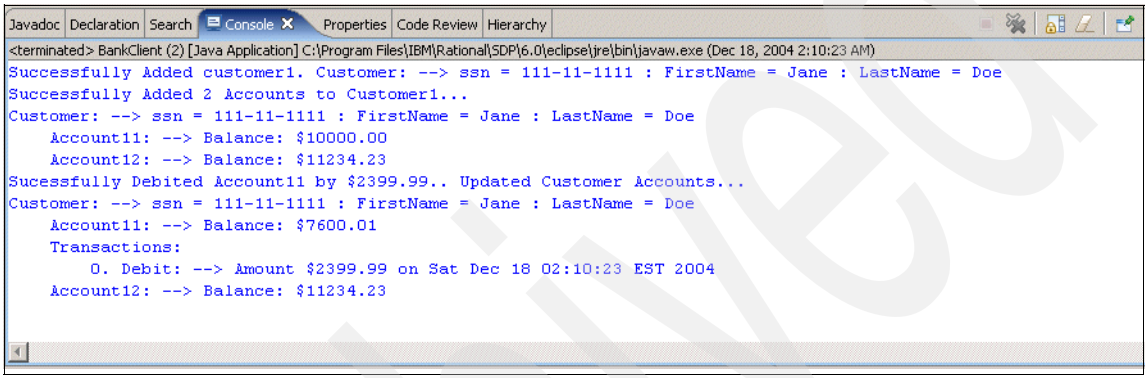


Figure 7-35 Java Bank application console output

> **Note:** For subsequent tests, simply click the Run icon from the Workbench toolbar (remembers previous run configuration).

7.3 Additional features used for Java applications

This section highlights some key features for working with Java projects within Rational Application Developer:

- Locating compile errors in your code
- Running your programs
- Debug your programs
- Java Scrapbook
- Pluggable Java Runtime Environment (JRE)
- Add a JAR file to the classpath
- Export the Java code to a JAR file
- Run the Java application external to Application Developer
- Import a Java JAR file into a project
- Utility Java Projects
- Javadoc

Archived

7.3.1 Locating compile errors in your code

Compile errors in your Java code are shown in the Problems view. Example 7-16 includes a code sample (CodeReviewSample.java) with errors so that we can demonstrate how to use the features of Rational Application Developer to resolve compile errors.

Example 7-16 CodeReviewSample.java

```
package itso.bank.test;

public class CodeReviewSample {

   /**
    * This method is used to demonstrate the capabilities of the
    * Rule based Code Review Tool, this method contains code
    * violating one of the J2SE Best Practices - Null Pointer
    * Category Rules that is set up by default in IBM
    * Rational Applicatin Developer. Based on the input provided
    * by the tool the method above is fixed.
    * @param dummy
    * @return
    */
   public Object[] verifyCodeReview(boolean dummy){
      Object[] dummyValues = { new Object(), new Object() };
      if ( dummy ){
         return dummyValues;
      } else {
         // This line violates the J2SE Best practice
         return null;
      }
   }

   /**
    * This method is used to demonstrate the capabilities of the
    * Rule based Code Review Tool, the verifyCodeReview method
    * violates one of the J2SE Best Practices - Null Pointer
    * Category Rules that is set up by default in IBM
    * Rational Applicatin Developer. Based on the input provided
    * by the tool the method above is fixed.
    * @param dummy
    * @return
    */
   public Object[] verifyCodeReviewFixed(boolean dummy){
      Object[] dummyValues = { new Object(), new Object() };
      if ( dummy ){
         return dummyValues;
      } else {
         return new Object[0];
```

Archived

```
        }
    }

    public String verifyCompilationErrors(boolean dummy){
        //comment the following line to view compilation error
        //return null;
    }

}
```

Find errors in Problems view

As you can see Figure 7-36 on page 289, Rational Application Developer marks the offending Java element in various views including the Package Explorer, Outline, Problems, and the Java editor view with an error symbol ( ) and uses the *Problem highlight line*, making it easier to get to and locate the offending code. Double-clicking the entry in the Problems view's list of problems will navigate to the line in the source editor, where the error was detected. To resolve this problem, simply uncomment the return null.

The line where the error occurs is also indicated by a yellow light bulb. If you move the mouse over the light bulb, the error message is shown.

Archived

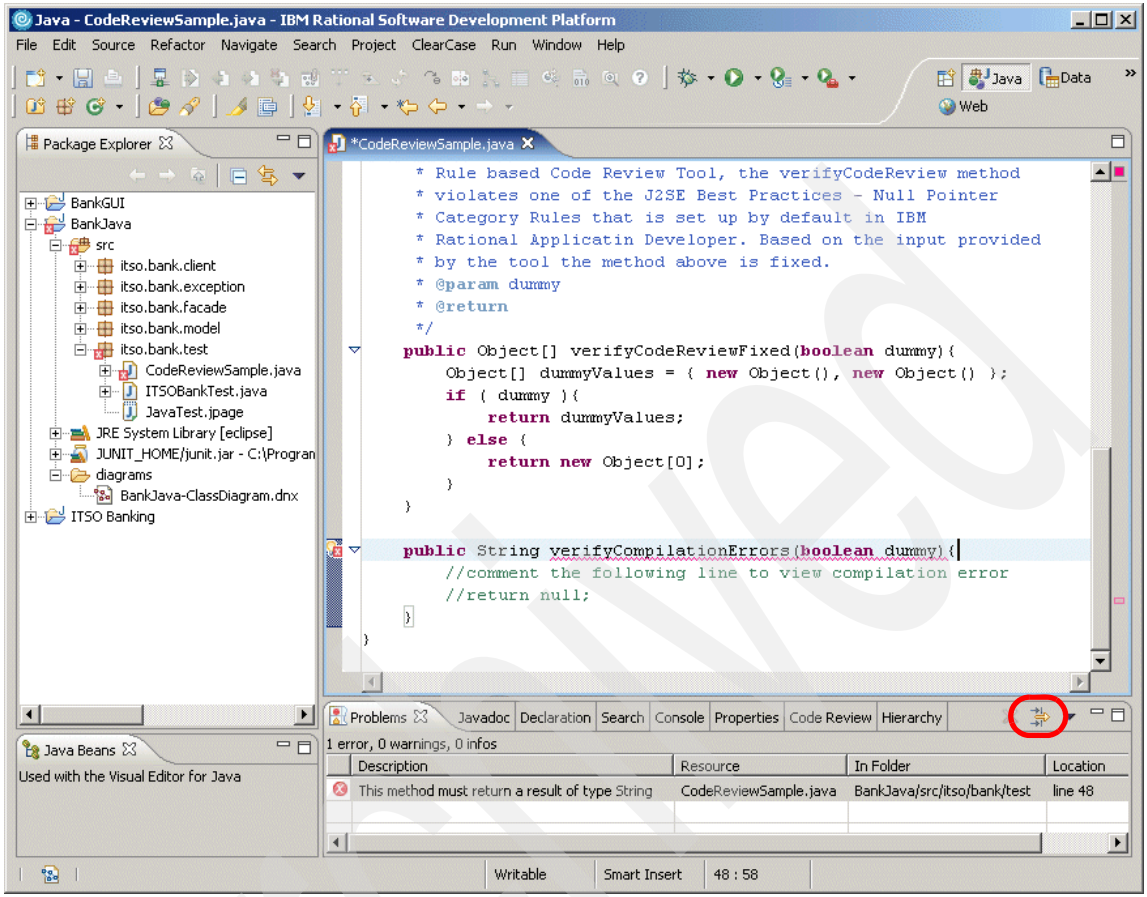


Figure 7-36 Identify errors in Java code

Filter errors displayed in Problems view

To more easily find the errors in the file you are working in, you can filter the Problems view to only show errors related to the current resource. Click the Filter icon in the Problems view (see Figure 7-37 on page 290). Select the entry **On selected resource only** (or **On any resource in same project**; or you can create a *working set*, which is a specific set of resources) in the Create Filter wizard, as shown in Figure 7-37 on page 290.

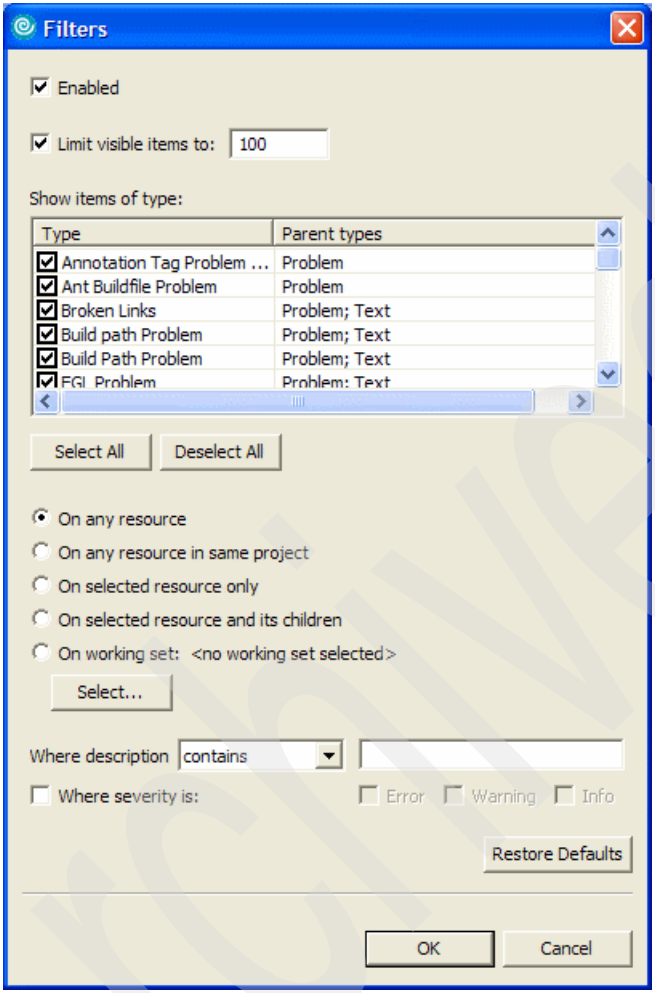


Figure 7-37 Filters - Creating filters to filter the errors shown in the Problems view

7.3.2 Running your programs

After the code has been completed and is free of compile errors, it can be executed using the Workbench Java Application Launcher. To launch the program, select the program—in our case BankClient.java in the Package Explorer—and click the Run icon from the toolbar.

The first time you launch the Run wizard, expand **Java Applications** under Configurations and then click **New**. This will create a new run configuration with appropriate settings for the selected class BankClient, as shown in Figure 7-38 on page 291. You can also specify arguments, select a different JRE, modify

runtime classpath settings, add additional source directories, add environment variables to the JRE, and use either a local or a shared runtime configuration.

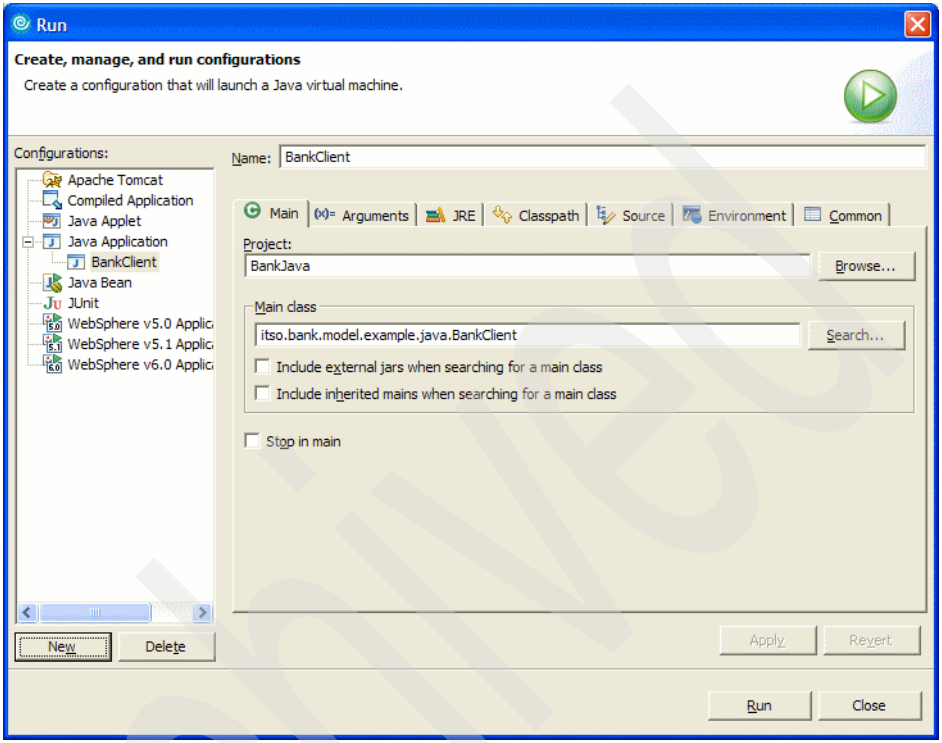


Figure 7-38 Run Configuration wizard

Tip: By defining launch configurations for a program, you can create multiple configurations for the same program with different arguments and settings (such as the JRE, environment variables, etc.).

Archived

Note: The new features introduced with this version of Rational Application Developer in the Run Configuration wizard include:

- Support for running classes that inherit from classes with a main method.
- In the Arguments tab, now the runtime arguments can be specified in terms of using variables. For more information on variables refer to 3.3.1, “Java classpath variables” on page 98.
- You can add additional source directories to the run path, used for debugging. By default, the JRE source and the project source are added. Thus, while debugging, developers will be able to debug the project source and will be able to go into types and methods that are core java classes in the JRE and still view the source.
- Environment variables can be added using the Environment tab; as with arguments, variables can be used.

You can also use the drop-down arrow of the Run icon. Clicking this drop-down arrow the first time allows you either to open the configuration launcher or select the type of application you would like to run, directly.

You can add other launch configurations for any program. Each configuration is displayed when clicking the drop-down arrow of the Run icon. In the sample when BankClient is run it calls a few business methods and displays the output shown in Figure 7-39 in the console.

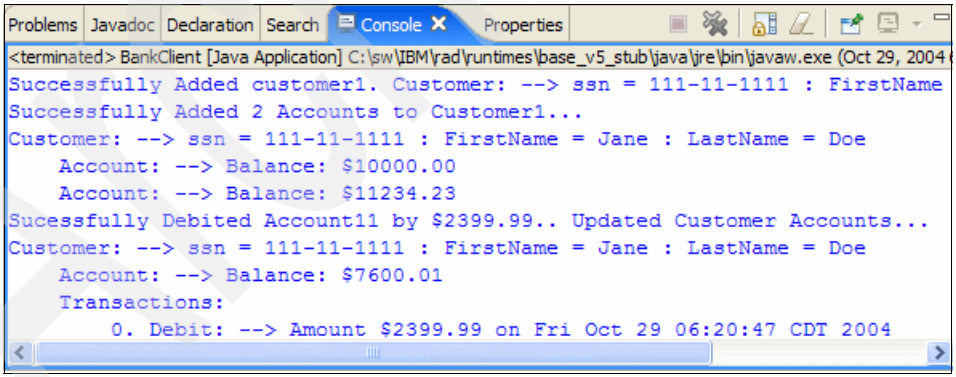


Figure 7-39 Java run console - Partial output from running the BankClient class

7.3.3 Debug your programs

For details on debugging an application, refer to Chapter 21, “Debug local and remote applications” on page 1121.

7.3.4 Java Scrapbook

Snippets of Java code can be entered in a Scrapbook window and evaluated by simply selecting the code and running it. This feature can be used to quickly test code without having to modify any actual Java source files.

These scrapbook pages can be added to any project. The extension of a scrapbook page is jpage, to distinguish them from normal Java source files.

Tip: Content assist (such as code assist) is also available on scrapbook pages.

To create a scrapbook page, create a file with a .jpage extension. Alternatively, select the package in the Package Explorer, right-click, and from the context menu choose **New** → **Other**. In the Select a Wizard dialog, expand **Java** → **Java Run/Debug** → **Scrapbook Page**. Make sure the correct folder is selected and enter a file name (JavaTest) for the new page (Figure 7-40 on page 294).

Archive

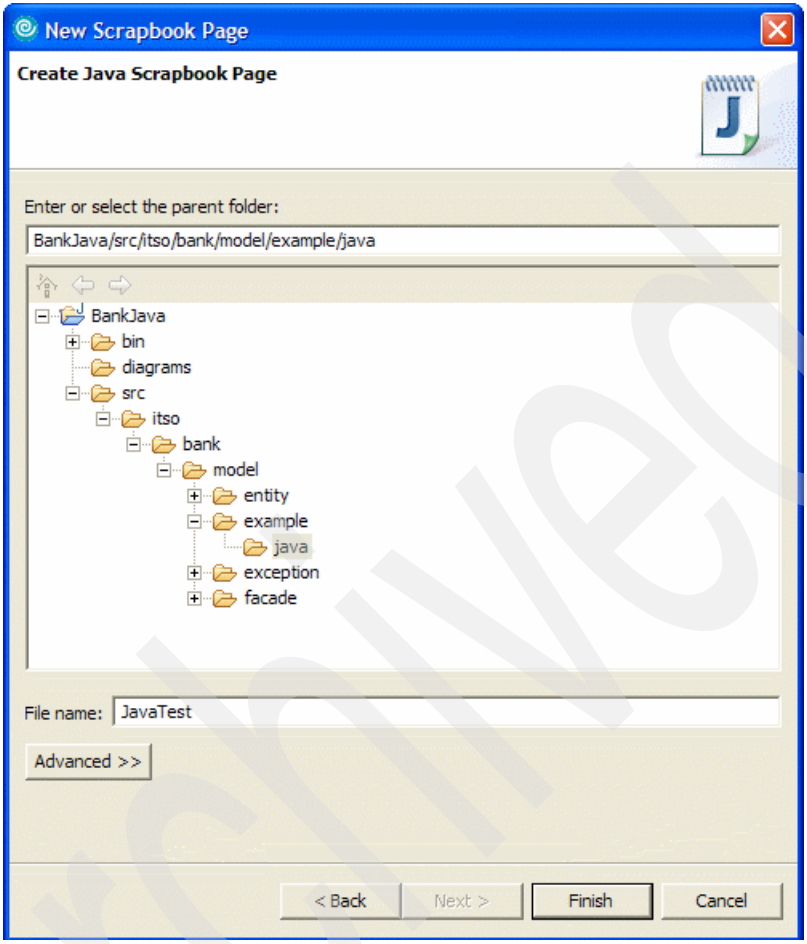


Figure 7-40 Create Java Scrapbook Page dialog

Click **Finish** to create the scrapbook page. After the page has been created and opened in the source editor, you can start entering code snippets in it.

To test a scrapbook page, we use code shown in Example 7-17. The source code is available in the c:\6449code\java\JavaTest.jpage file.

Archived

Tip: All class names in a scrapbook page must be fully qualified, or you have to set import statements:

- Select **Set Imports** from the context menu anywhere in the scrapbook.
- For our example, select the following packages:

```
itso.bank.model
itso.bank.facade
itso.bank.exception
itso.bank.client
itso.bank.test
java.math
```

Example 7-17 JavaTest.jpage sample

```
// Snippet1
   Bank bank = new ITSOBank();
   Customer customer1 = new Customer("111-11-1111","Jane","Doe");
   bank.addCustomer(customer1);
   System.out.println ( "Successfully Added customer1. "+customer1);
   Account account11 = new Account("11",new BigDecimal(10000.00D));
   bank.addAccount(customer1,account11);
   Account account12 = new Account("12",new BigDecimal(11234.23));
   bank.addAccount(customer1,account12);
   System.out.println("Successfully Added 2 Accounts to Customer1... ");
   System.out.println(customer1);

   bank.processTransaction(customer1,account11,new
BigDecimal(2399.99D),TransactionType.DEBIT);

   System.out.println("Sucessfully Debited Account11 by $2399.99.. Updated Customer
Accounts...");
   System.out.println(customer1);

// Snippet 2
   Bank bank = new ITSOBank();
   Customer customer1 = new Customer("111-11-1111","Jane","Doe");
   bank.addCustomer(customer1);
   bank.getCustomer("111-11-1111");
```

After you have added the code, you can run one of the snippets by selecting the code and selecting **Run Snippet** from the context menu, or click the Run the Selected Code icon in the toolbar. In our case, select all of snippet1 and select **Run Snippet**. The result is displayed in the Console view.

Now select **Snippet2** and select **Display** from the context menu, or click the Display icon in the toolbar. The result is shown as in Figure Figure 7-41 on page 296.

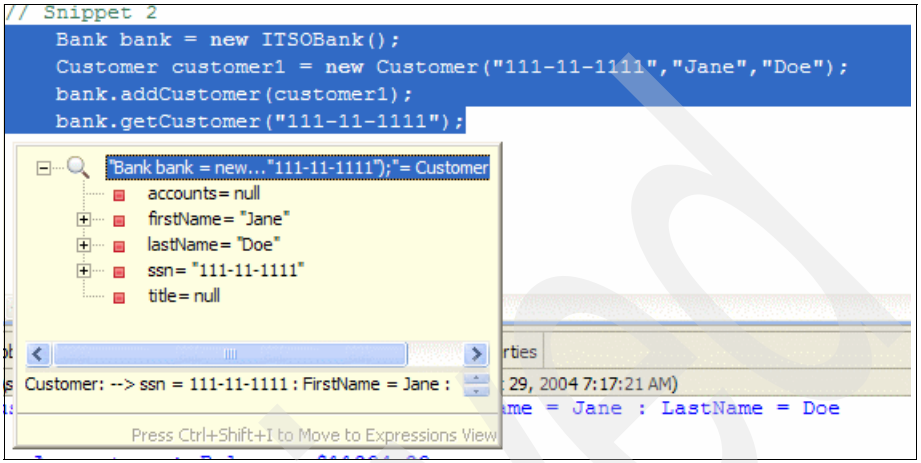


Figure 7-41 Java Scrapbook - Display results

Note: You cannot run code in a scrapbook page until you have at least one statement selected.

You can also select **Display** from the context menu to display the result expression. Alternatively, you can select **Inspect** from the context menu to bring up the Expressions view, which allows you to inspect the result like a variable in the debugger (see Chapter 21, “Debug local and remote applications” on page 1121for more information about debugging and inspecting).

Note: After running snippets of code in the scrapbook, click **the Stop the Evaluation** icon( ) to end the scrapbook evaluation.

7.3.5 Pluggable Java Runtime Environment (JRE)

Rational Application Developer provides support to allow Java projects to run under different versions of the Java Runtime Environments. New JREs can be added to the workspace, and projects can be configured to use any of the JREs available. By default, the Rational Application Developer uses and provides projects with support for IBM Java Runtime Environment V1.4.2. This is the runtime that will be used unless the configuration is changed explicitly to use, as shown in figure using the Run Configuration wizard (see Figure 7-42 on page 297).

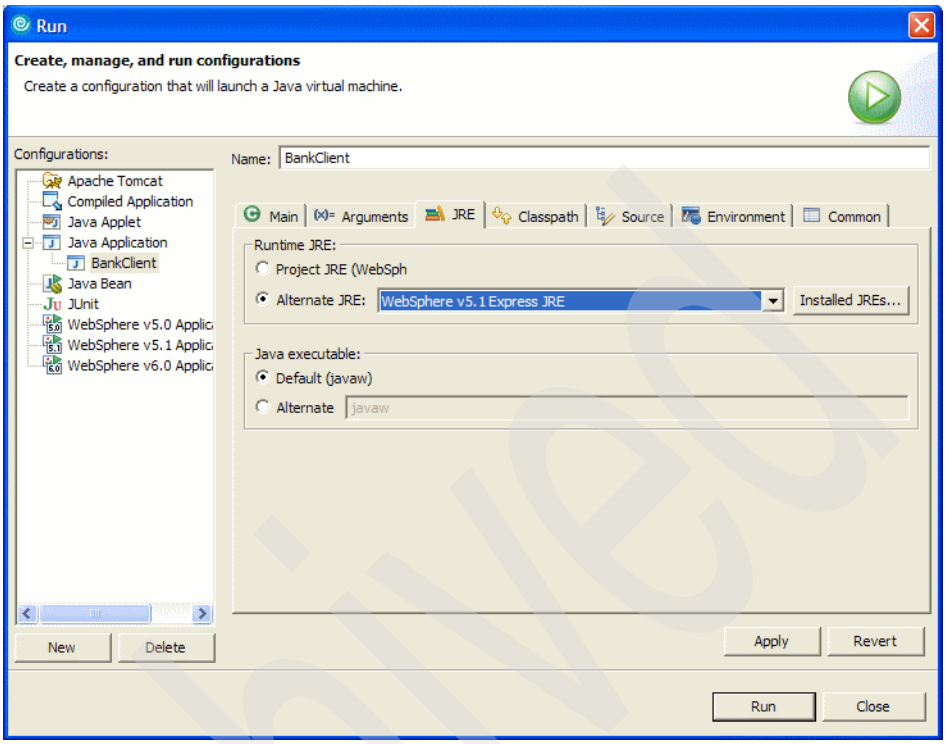


Figure 7-42 Changing the version of the Java Runtime

For more information regarding the JRE environment of Rational Application Developer and how to add a new JRE, see Chapter 3, “Workbench setup and preferences” on page 75.

Note: At the time of this writing, switching the runtime of projects to JREs higher than 1.4.x is not supported.

7.3.6 Add a JAR file to the classpath

Typically classes within Java projects access and depend on types that are developed by third-party companies and are external to the workspace. These jar files are imported into the Java Build path as external jar files. This is done through the *library properties*.

To add an external jar (for example, db2java.zip) to the build path, do the following:

1. Select the project, right-click, and choose **Properties** from the context menu to bring up the Java project properties.
2. Select the **Java Build Path** and then click the **Libraries** tab, as seen in Figure 7-43.

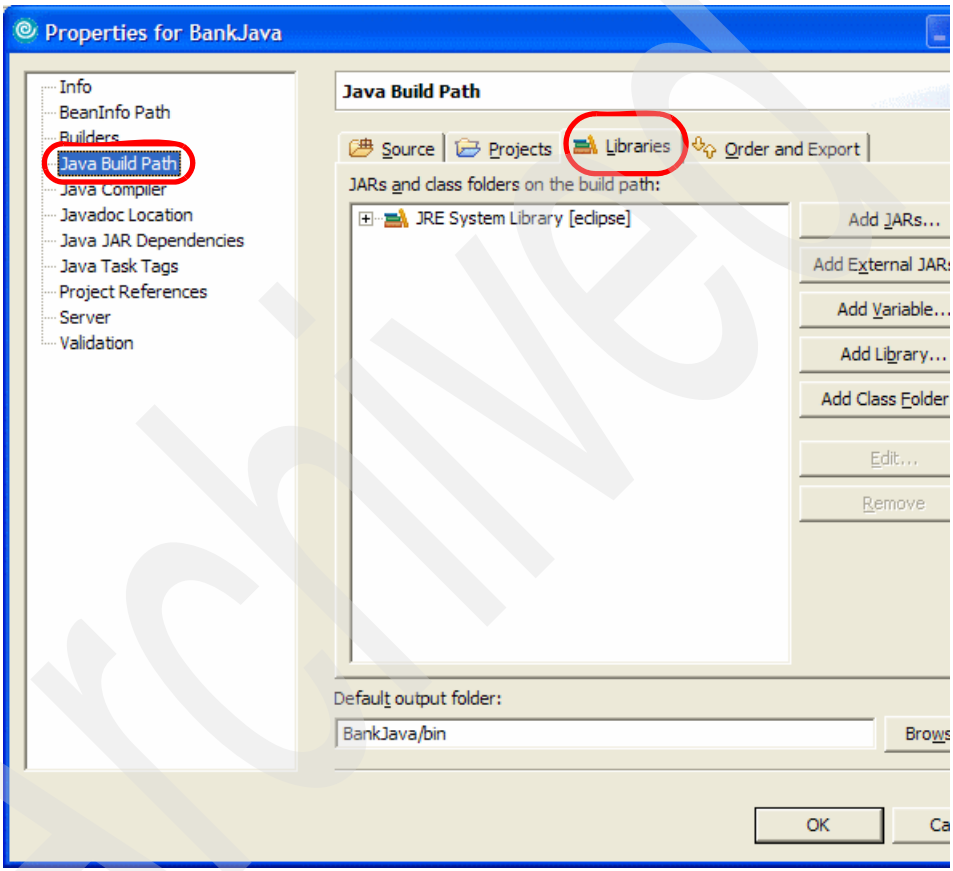


Figure 7-43 Java Build Path settings for the BankJava project

3. There are two ways you can specify access to the required jars.
   - Select **Add External JARs** and locate the db2java.zip in the file system.

   Or:

   - Select **Add Variable** to add a variable that refers to the db2java.zip file.

Archived

Tip: We recommend using the Add Variable option since you are not directly referencing a physical path that could be different for another developer within a team. For this sample it is assumed that a variable DB2_DRIVER_PATH has been defined. If DB2 is installed on your system, Application Developer predefines this variable.

4. To add this variable to the Java build path for the project, select **Add Variable** to display the New Variable Classpath Entry dialog.
5. Select the **DB2_DRIVER_PATH** variable and close the dialog with **OK**.

   If the DB2_DRIVER_PATH variable is not available in the dialog, you have to create a new variable.

   a. Click the **New** button to display the New Variable Entry dialog.

   b. Enter `DB2_DRIVER_PATH` in the name field and enter the path to the db2java.zip in the Path field. Click **OK**.

Now all the types packaged within the jar/zip archive pointed to by the variable DB2_DRIVER_PATH are available in the buildpath of BankJava project.

7.3.7 Export the Java code to a JAR file

This section describes how to export a Java application to a jar file that can be run outside of Rational Application Developer using a JRE via the command line. We will demonstrate how to export and run the BankClient Java application.

To export the Java Bank code to a JAR file, do the following:

1. From the Workbench, select the **BankJava** project.
2. Select **File** → **Export**.
3. When the Export Select dialog appears, select **JAR file** and then click **Next**.
4. When the JAR Package Specification page appears, enter the JAR details. For example, we entered the following, as seen in Figure 7-44 on page 300:

   a. Select the Resources to export: Check “BankJava”.

   b. Check “Export all output folders for checked projects”.

   c. Check “Export java source files and resources”.

Note: In our example, we chose to include the source to demonstrate in a later section how to import JAR files into a project. It may not be necessary or desirable to include Java source in a JAR file for your environment.

Archived

d. JAR file export destination: `C:\BankJava.jar`

e. Click **Next**.

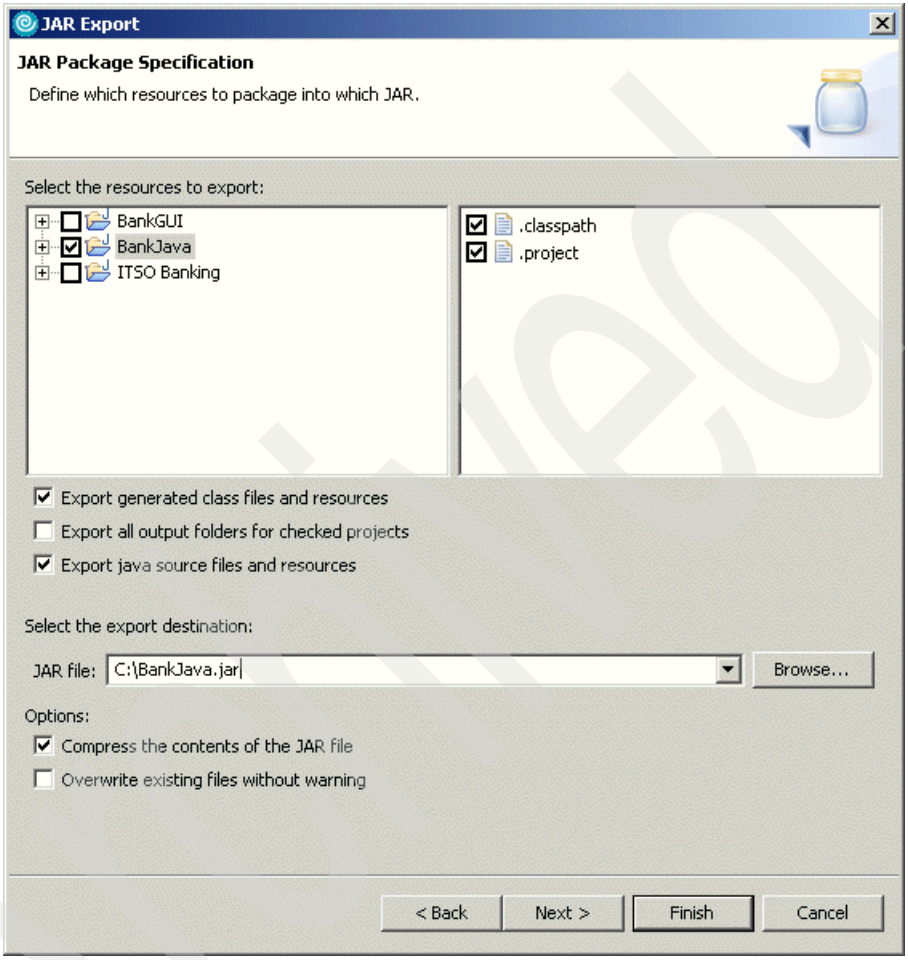


Figure 7-44 Export - JAR Package Specification

5. When the JAR Packaging Options dialog appears, accept the defaults and click **Next**.
6. When the JAR Manifest Specification dialog appears, click the **Browse** button for the Main Class field.
7. Select **BankClient** in the Select Main Class window and click **OK**.
8. Click **Finish** in the Jar Manifest Specification dialog to start the export of the entire Java project as a jar file.

7.3.8 Run the Java application external to Application Developer

Once you have exported the Java code to a jar file as described in the previous step (see 7.3.7, “Export the Java code to a JAR file” on page 299) you can run the Java application external to Rational Application Developer as follows:

1. Open a Windows command window and navigate to the directory to which you have exported the JAR file (for example, C:\).
2. Ensure that the Java Runtime Environment is in your path. If not, add it to the path as follows, in a command window or to the Windows environment path.

   ```
   PATH=%PATH%;C:\Program Files\IBM\Rational\SDP\6.0\eclipse\jre\bin
   ```

3. Enter the following command to run the Java Bank application:

   ```
   java -jar BankJava.jar
   ```

   This results in the main method of BankClient being executed; the results are similar Figure 7-45.

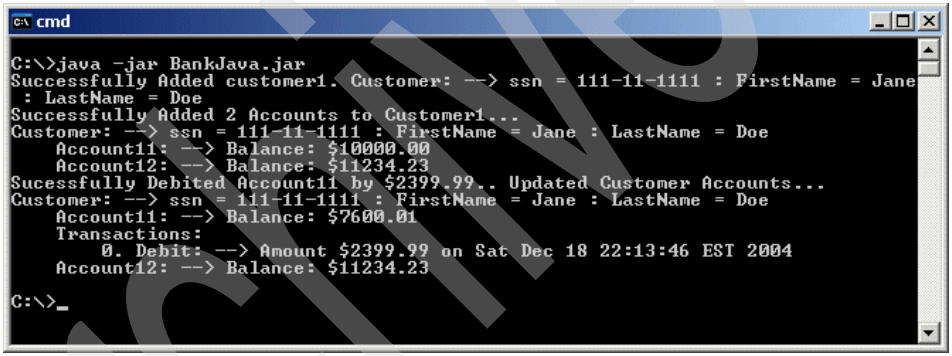


Figure 7-45 Output from running BankJava.jar outside Application Developer

7.3.9 Import a Java JAR file into a project

This section describes how to import a Java JAR file into an existing project. In addition to using the JAR file as a means to import project-related materials, JAR files provide a mechanism for utility projects in enterprise applications. Utility projects are discussed in 7.3.10, “Utility Java Projects” on page 302.

We will use the BankJava.jar file exported in 7.3.7, “Export the Java code to a JAR file” on page 299. Alternatively, you can use the BankJava.jar provided in the c:\6449code\java directory included with the redbook sample code.

1. Create a new project with the default options, for example, BankJava or BankJava1 if you already have a BankJava project.
2. Select the new project, right-click, and select **Import** from the context menu.

Archived

3. Select ZIP file and click **Next**.
4. Click **Browse** and locate the file (for example, c:\6449code\java\BankJava.jar).
5. If you imported the BankJava JAR into a project other than the name BankJava, uncheck .classpath and .project. If you do not do this you will be prompted to overwrite these files.
6. Click **Finish**.

After importing the code you find all the packages discussed in this chapter in the BankJavaJAR project.

We will use the BankJava project as a utility project in Chapter 11, “Develop Web applications using JSPs and servlets” on page 499.

To test the imported project, select and run **BankClient** from the Package Explorer. The program executes a few of the business methods and displays the output in the Console view.

7.3.10 Utility Java Projects

Utility Java projects are Java projects that are designed to contain code that is shared across multiple modules within an enterprise application. J2EE 1.3 and 1.4 provide support for utility JAR files at the enterprise application level. Such JAR files are then made available to Web and EJB modules as JAR file dependencies.

In our example, the banking sample application, we have separated the business logic in a way so that we can use other clients than BankClient.java, which was the client to call business methods in the bank interface. We will use the BankJava project as a utility project in the enterprise sample applications described in Chapter 11, “Develop Web applications using JSPs and servlets” on page 499, and Chapter 12, “Develop Web applications using Struts” on page 615. These chapters also discuss in detail on the enterprise application configuration for using a Java Project as a utility project.

Note: For more information on Java utility JAR files, refer to the following:

- *Developing J2EE utility JARs in WebSphere Studio Application Developer* found at:

  http://www.ibm.com/developerworks/websphere/library/techarticles/0112_deboer/deboer2.html

- Referencing J2EE Utility JARs and Java Projects in WebSphere Studio V5 found at:

  http://www.ibm.com/developerworks/websphere/library/techarticles/0304_manji/manji.html

7.3.11 Javadoc

Javadoc is a tool in the Java JDK used to generate documentation. Documentation is generated with the help of html files in the package folders describing the package and block tags within Doc Comments placed within the Java source describing classes, interfaces, methods, and other Java elements for Java packages, classes, and methods. Rational Application Developer introduces the *Javadoc* view, which is implemented using a SWT browser widget to display HTML. In the Java perspective, the Javadoc view is context sensitive in the that the view displays the Javadoc associated with the Java element where the cursor is currently located within the editor.

To demonstrate the use of Javadoc, we will use the BankJava application created or imported (see 7.3.9, “Import a Java JAR file into a project” on page 301) in previous sections of this chapter.

1. Expand the **BankJava** project.
2. Select the **BankClient.java** from the Package Explorer.
3. You will notice that when the cursor is on the BankClient Type, the Javadoc reflects the documentation associated with BankClient; and when the cursor is over BigDecimal, the Javadoc view changes to the documentation associated with BigDecimal, as shown in Figure 7-46 on page 304.

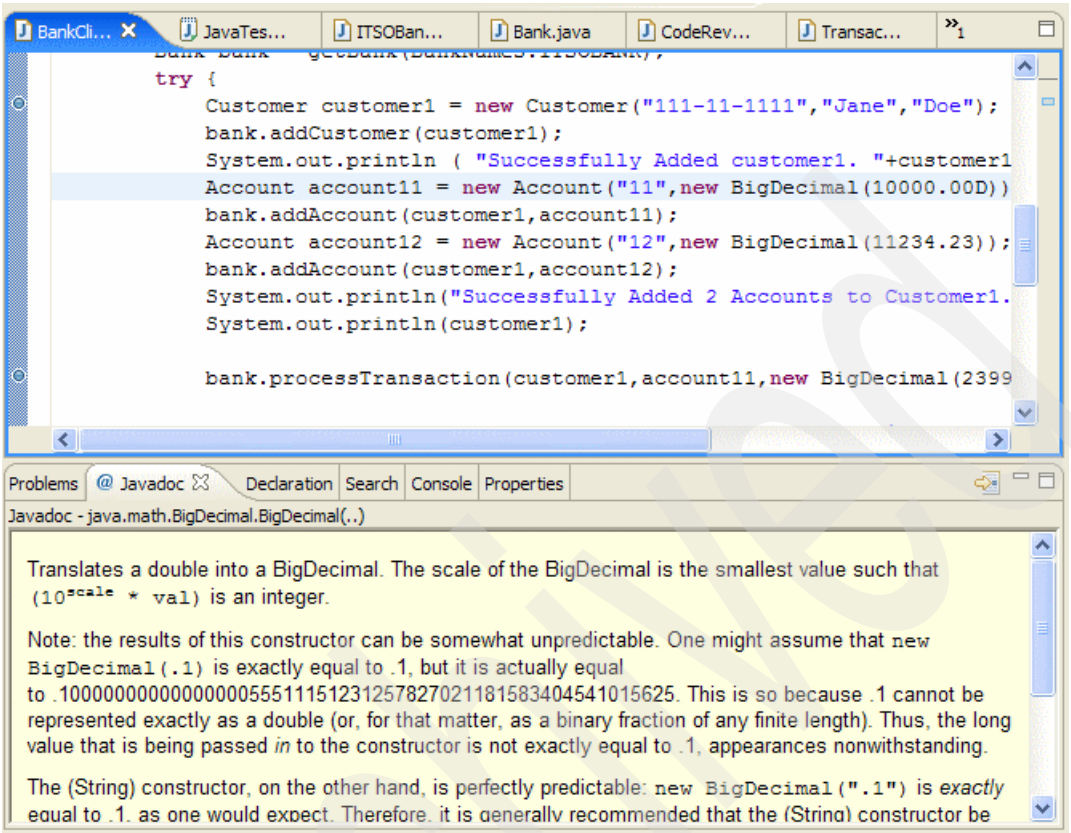


Figure 7-46 Javadoc view - Context sensitive

Methods of generating Javadoc

Rational Application Developer provides the following methods to generate Javadoc:

- Select a project and use **File** → **Export** (or choose **Export** from the context menu), and select **Javadoc** from New wizard.
- Select a project and use **File** → **Export** (or choose Export from the context menu), sand elect **Javadoc with diagram tags**. This wizard is introduced in Rational Application Developer.

  The Javadoc with Diagram Tags wizard enables you to embed a @viz.diagram javadoc tag on the class, interface, or package level. This tag must refer to an existing diagram, and the wizard will then export that diagram into a GIF, JPG, or BMP, and embed that file to the generated javadoc for the class, interface, or package.

Example 7-18 shows an example of the use of the @viz.diagram tag.

Example 7-18 Using the @viz.diagram tag to generate diagrams in javadoc

```
package itso.bank.client;

/**
 * Bank client class.
 *
 * @viz.diagram BankJava-ClassDiagram.dnx
 */
public class BankClient {

   // ...

}
```

Restriction: At the time of writing, the @viz.diagram tag assumed that the diagram being referenced is placed in the same folder as the Java file containing the @viz.diagram tag.

If you plan on using this feature, you will have to place your diagrams along with the Java source code. For Web applications, this has the side effect of the class diagrams being packaged into the WAR file with the compiled Java code.

We found two possible work-arounds:

- You can manually remove these diagrams from the WAR file after exporting.
- Filter the contents by configuring an exclusion filter for the EAR export feature.

  Refer to “Filtering the content of the EAR” on page 1222 for information on techniques for filtering files (include and exclude) when exporting the EAR.

- Select a project. Use **Project** → **Generate Javadoc...** to run the Javadoc Generation Wizard.
- Select a project. Use **Project** → **Generate Javadoc with Diagrams**.

  With this option, you can choose automatic or manual generation. With automatic generation, a topic diagram will be generated for each class by the Javadoc Generation Wizard. With manual generation, the @viz.diagram tag must be present as a block comment, and only topic diagrams pointed to by the @viz.diagram tags will be generated as html into the class specifying the block.

Example for generating Javadoc

This section describes by example how to generate Javadoc for the BankJava project.

1. Open the Java Perspective Package Explorer view.

   Note: We found that when in the Navigator view, we were not able to generate the Javadoc properly (the Generate Javadoc with UML option is greyed out).

2. Select the **BankJava** project.
3. Right-click and select **Export** from the context-sensitive menu.

   Alternatively, select **File** → **Export**.
4. Select **Javadoc** in the Export wizard and click **Next**.
5. When the Javadoc Generation page appears, you will need to specify the path to the javadoc.exe file available under the bin directory of an installed JDK, visibility, output type, and Javadoc destination. For example, we entered the following, as seen in Figure 7-47 on page 307, and then clicked **Next**:
   - Javadoc command:

     `<rad_home>\runtimes\base_v6\java\bin\javadoc.exe`

     Where <rad_home> is the Rational Application Developer installation path (for example, C:\Program Files\IBM\Rational\SDP\6.0).
   - Create Javadoc for members with Visibility: Select **Public** (default).
   - Select **Use Standard doclet** (default; generates HTML output).

     Alternatively, you can specify a custom doclet. To specify a custom doclet, specify the name of the Doclet and the classpath to the doclet implementation.
   - Destination: `c:\workspace\BankJava\doc` (default; generates Javadoc in doc directory of current project)

Archived

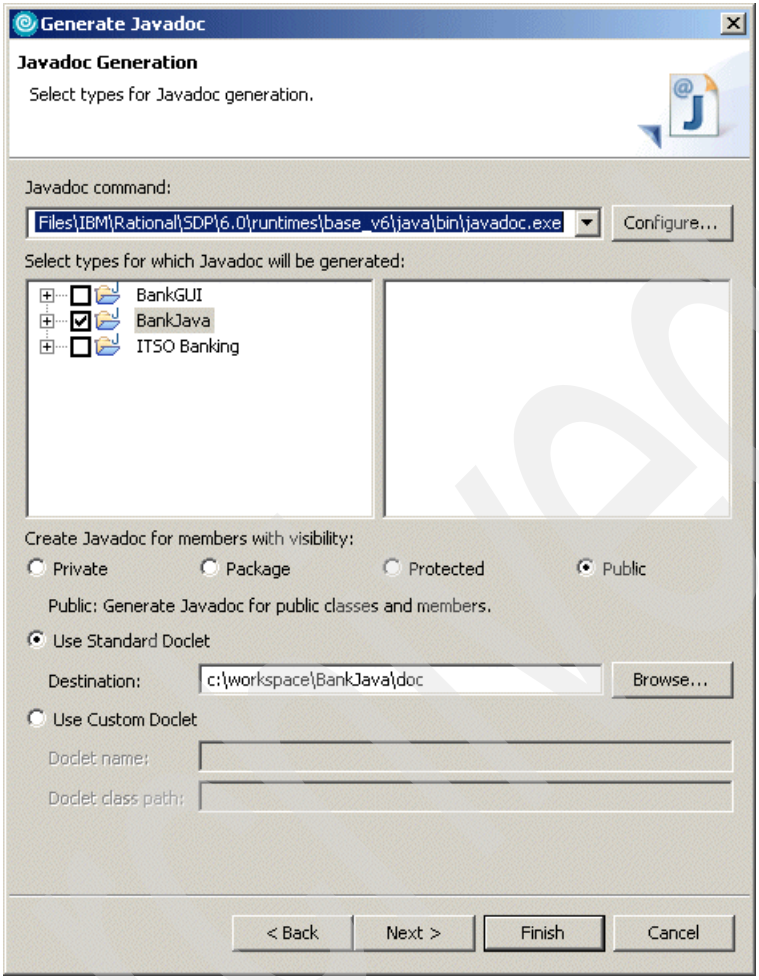


Figure 7-47 Javadoc Generation Wizard

6. When the “Javadoc Generation - Configure Javadoc arguments for standard doclet” window appears, we accept the default settings and click **Next**.
7. When the “Javadoc Generation - Configure Javadoc arguments” window appears, check **Save Settings for this Javadoc export as an Ant script**, as seen in Figure 7-48 on page 308, and then click **Finish**.

   This will generate the c:\workspace\BankJava\javadoc.xml ant script to invoke the Javadoc command.

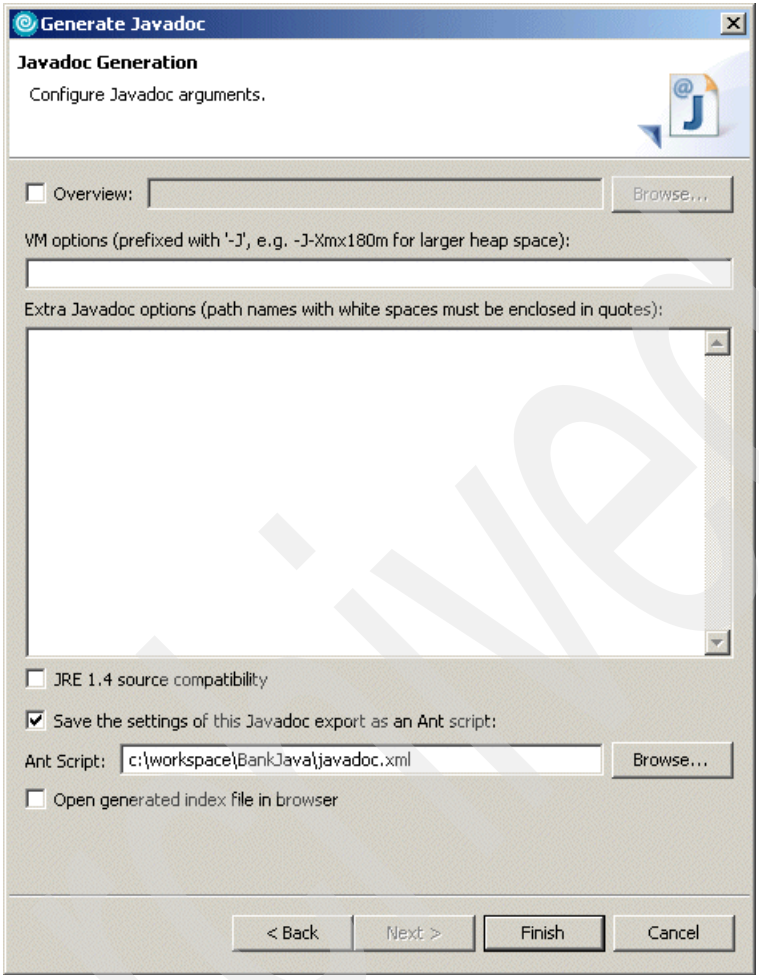


Figure 7-48 Javadoc Generation Wizard - Configure Javadoc arguments

8. When prompted to update the Javadoc location, click **Yes to all**.
9. When prompted, the Ant file will be created; click **OK**.

   You should see console output generated from the Javadoc wizard similar to Figure 7-49 on page 309.

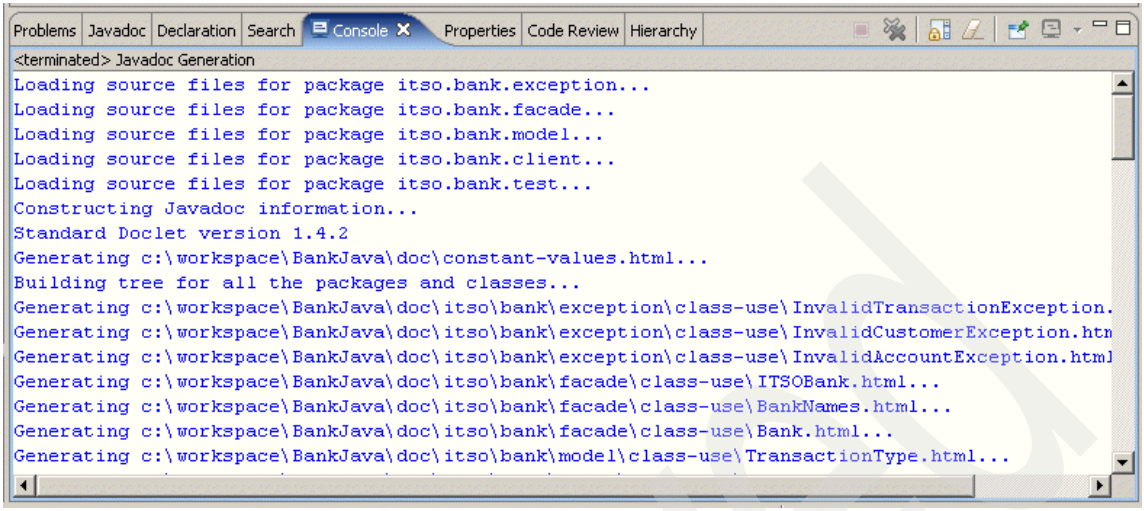


Figure 7-49 Javadoc Wizard - Console output

10. View the generated Javadoc for the BankJava project, as shown in Figure 7-50 on page 310.

    a. Expand **BankJava** → **doc**.

    b. Double-click **index.html** to display the Javadoc contents for the BankJava project.

Archived

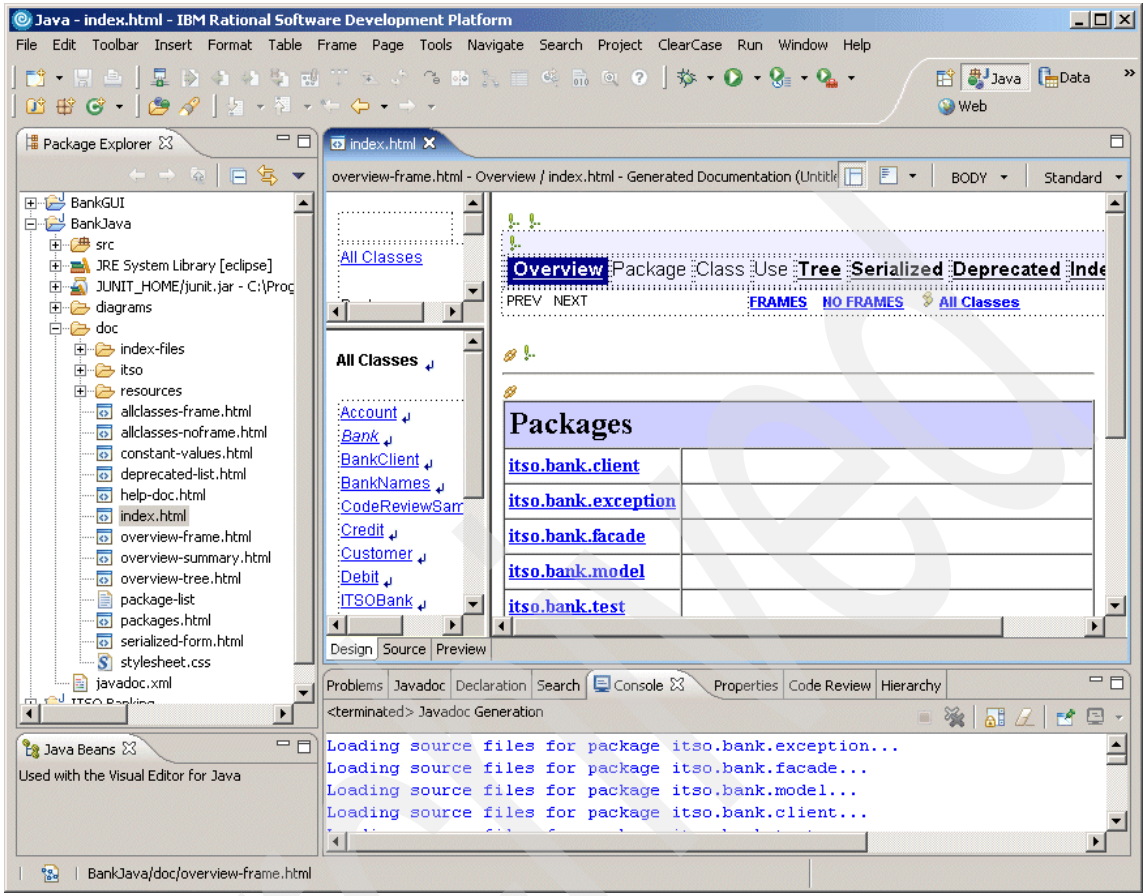


Figure 7-50 Javadoc output generated from the Javadoc Wizard for the BankJava project

Generate Javadoc from an Ant script

In our Javadoc example, we checked “Save Settings for this Javadoc export as an Ant script”. This generated the c:\workspace\BankJava\javadoc.xml ant script, which can be used to invoke the Javadoc command. We will use the javadoc.xml sample Ant script to demonstrate how to generate Javadoc documentation.

1. Select and expand **BankJava**.
2. Select **javadoc.xml** in Package Explorer.
3. From the context menu select **Run** → **Ant Build**.
4. Click **Finish**. The javadoc generation process starts.

Note: For more information on Ant, refer to Chapter 22, “Build applications with Ant” on page 1155.

7.4 Java editor and Rapid Application Development

IBM Rational Application Developer V6.0 contains a number of features that ease and expedite the code development process. These features are designed to make life easier for both experienced and novice Java programmers by simplifying or automating many common tasks.

This section is organized into the following topics:

- Navigate through the code
- Source folding
- Type hierarchy
- Smart Insert
- Mark occurrences
- Word skipping
- Smart compilation
- Java search
- Quick Assist (Quick Fix)
- Code Assist (content)
- Import generation
- Generate getters and setters
- Override/implement methods
- Adding constructors
- Refactoring

Note: This chapter focuses on the key features provided in the Rational Application Developer Java editor. For an exhaustive list of all editor-related features, refer to product help documentation.

7.4.1 Navigate through the code

This section highlights the use of the Outline view, Package Explorer, and bookmarks to navigate through the code.

Use the Outline view to navigate the code

The Outline view displays an outline of a structured file that is currently open in the editor area, and lists structural elements. The contents of the Outline view are editor-specific.

Archived

For example, in a Java source file the structural elements are package name, import declarations, class, fields, and methods. We will use the BankJava project to demonstrate the use of the Outline view to navigate through the code:

1. Select and expand the **BankJava** → **src** → **itso.bank.model** from the Package Explorer.
2. Double-click **Account.java** to open the file in the Java editor.
3. Switch to the Outline view by selecting **Window** → **Show view** → **Outline**.
4. By selecting elements in the Outline view, you can navigate to the corresponding point in your code. This allows you to easily find methods and field definitions without scrolling through the editor window (see Figure 7-51).

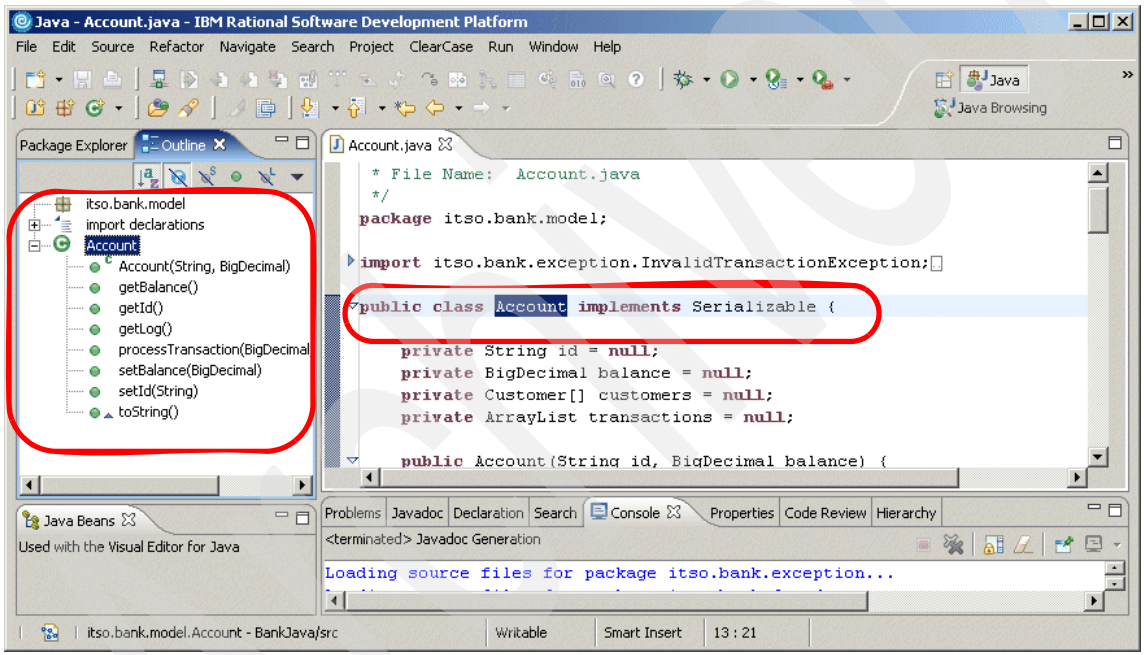


Figure 7-51 Java editor - Outline view for navigation

Note: If you have a source file with many fields and methods, you can use the Show Source of Selected Element Only icon from the toolbar to limit the edit view to the element that is currently selected in the Outline view.

Use the Package Explorer view to navigate the code

The Package Explorer view, which is available by default in the Java perspective, can also be used for navigation. The Package Explorer view provides you with a

Java-specific view of the resources shown in the Navigator. The element hierarchy is derived from the project's build paths.

Use Bookmarks to navigate the code

Bookmarks are another simple way to navigate to resources that you frequently use. The Bookmarks view displays all bookmarks in the Workbench.

Show bookmarks

To show the Bookmarks view select **Window** → **Show View** → **Other** and select **Basic** → **Bookmarks** from the Basic section.

Set bookmark

To set a bookmark in your code, right-click in the gray sidebar to the left of your code in the Java editor and select **Add Bookmark**. The Add Bookmark window is shown in Figure 7-52. Enter the name of the bookmark and click **OK**.

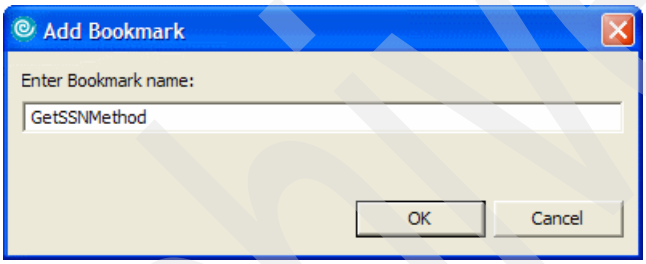


Figure 7-52 Java editor - Add Bookmark

View bookmark

The newly created bookmark is indicated by the symbol in the marker bar, as shown in Figure 7-53, and also appears in the Bookmark view. The Bookmark view, after a couple of bookmarks are created, is shown in Figure 7-54 on page 314. Double-clicking the bookmark entry in the Bookmarks view opens the file and navigates to the line where the bookmark has been created.

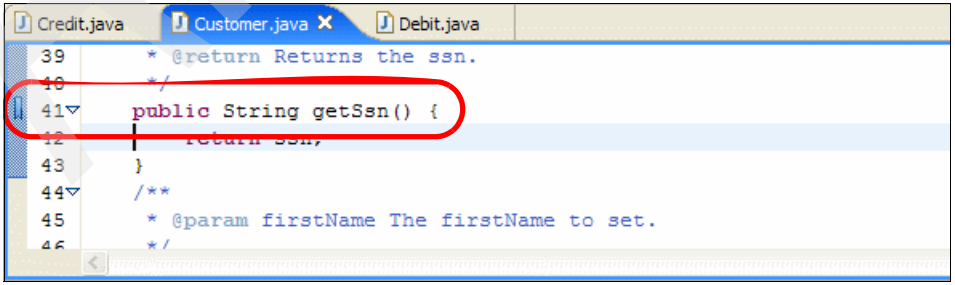


Figure 7-53 Java editor - Bookmark icon in the editor

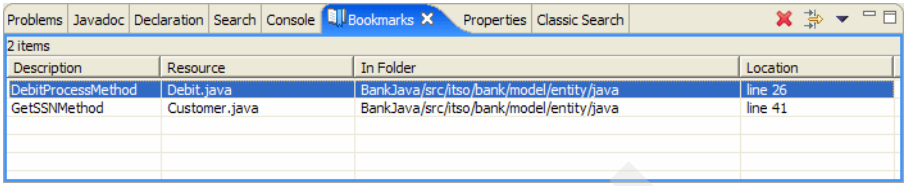


Figure 7-54 Java editor - Bookmark view

Delete bookmarks

You can remove a bookmark by using the bookmark's context menu in the Bookmarks view and select **Delete** or click the delete icon ( ) from the Bookmark view's toolbar, as seen in Figure 7-54.

Note: You can bookmark individual files in the Workbench to open them quickly from the Bookmark's view later. Select the file in the Project Explorer view. Select **Edit** → **Add Bookmark**. When the Bookmark dialog appears, enter the desired bookmark name and click **OK**. The bookmark should now be displayed in the Bookmarks view.

Note: Bookmarks are not specific to Java code. They can be used in any file to provide a quick way of navigating to a specific location.

7.4.2 Source folding

Rational Application Developer introduces source folding of import statements, comments, types, and methods. Source folding is configurable through the Java editor preferences.

To configure this select **Window** → **Preferences**. In the Preference dialog box expand **Java** and select **Editor**. Source folding of an *import* and a *method* are shown in Figure 7-55 on page 315.

Archive

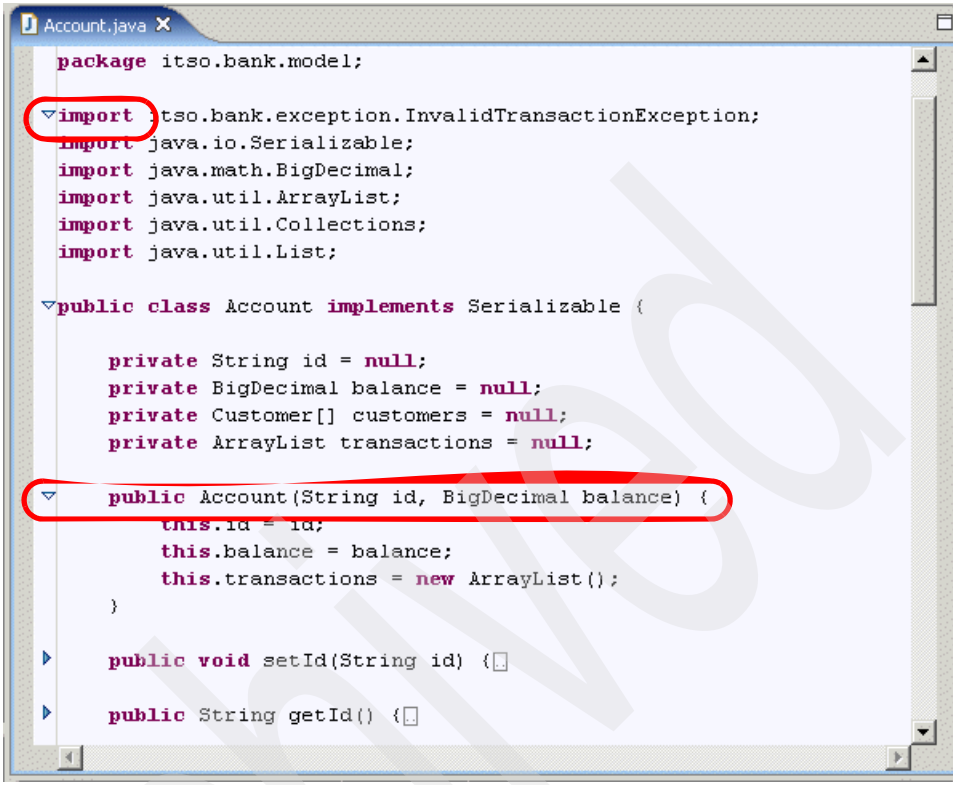


Figure 7-55 Java editor - Source folding

7.4.3 Type hierarchy

Rational Application Developer introduces viewing the quick type hierarchy of a selected type. Select a type with the cursor and press Ctrl+t. This displays the hierarchy, as shown in Figure 7-56 on page 316.

Archived

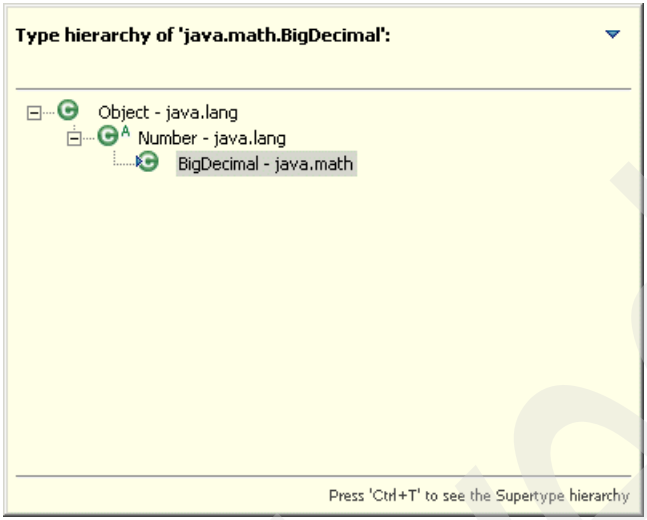


Figure 7-56 Java editor - Type hierarchy

7.4.4 Smart Insert

To toggle the editor between *Smart Insert* and *Insert* modes, press Ctrl+Shift+Insert. When the editor is in Smart Insert mode the editor provides extra features specific to Java. For example, in Smart Mode when you cut and paste code from a source Java file to another target Java file, all the imports needed are automatically added to the target Java file.

To configure the features available when in Smart Insert mode, do the following:

1. Select **Window** → **Preferences**.
2. Expand **Java** and select **Editor**.
3. Select the **Typing** tab.
4. Select/de-select needed features available under “Enable these typing aids when in Smart Insert Mode”.

7.4.5 Mark occurrences

When the cursor is over Java elements, including but not limited to Java types, keywords, variable names, and more importantly method exits (when the cursor is on the method name), and all methods that throw the exception (when the cursor is on the throws clause in a method definition), the editor highlights all occurrences of the type, as shown in Figure 7-57 on page 317.

Archive

Alternately, you can click the Mark Occurrences icon() in the toolbar or click Alt+Shift+O. The Mark Occurrences feature can be configured. To configure Mark Occurrences select **Window** → **Preferences**. Expand **Java**, and select **Mark Occurrences**.

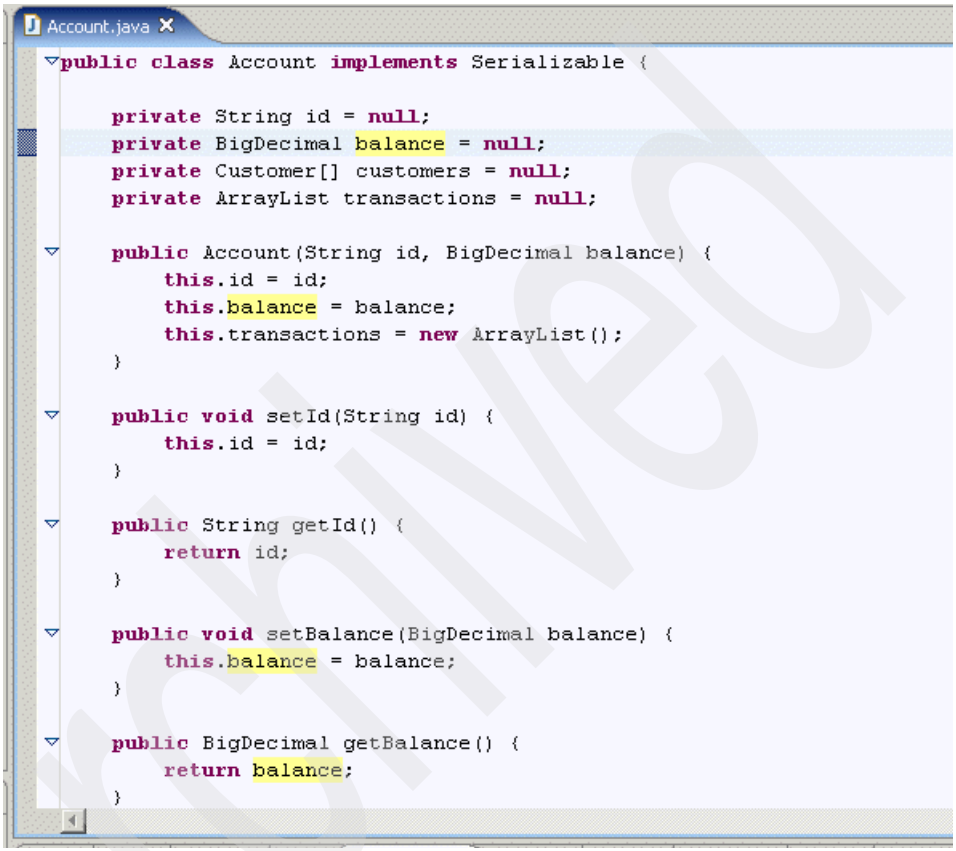


Figure 7-57 Java editor - Mark occurrences

7.4.6 Word skipping

The Java editor now truly supports camel case notation for all word skipping operations. For example, if there was a method named getAllAccounts, the editor would stop at all characters in bold, as in **g**et**A**ll**A**ccounts, when skipping through words, as opposed to **g**etAllAccounts.

Archived

7.4.7 Smart compilation

The Java Builder in the Rational Application Developer Workbench incrementally compiles the Java code in the background as it is changed and displays any compilation errors in the editor window automatically, unless you disable the automatic build feature. Refer to Chapter 4, “Perspectives, views, and editors” on page 131, for information on enabling/disabling automatic builds and running Workbench tasks in the background.

7.4.8 Java search

Rational Application Developer provides support for various searches. To display the search dialog click the search icon( ) in the toolbar or press Ctrl+H. The search dialog box is shown in Figure 7-58.

The Search dialog can be configured to display different searches by clicking the **Customize...** button in the Search dialog box. In Figure 7-58, the search dialog has been customized to not display the Process Search tab. The Process Search tab is not selected and is not displayed. For more information on Process Search refer to Chapter 6, “RUP and UML” on page 189.

Note: If you cannot see the tab you are looking for (for example, the NLS Key tab), make sure the tab is selected in the Search Page Selection window when you click on the **Customize...** button.

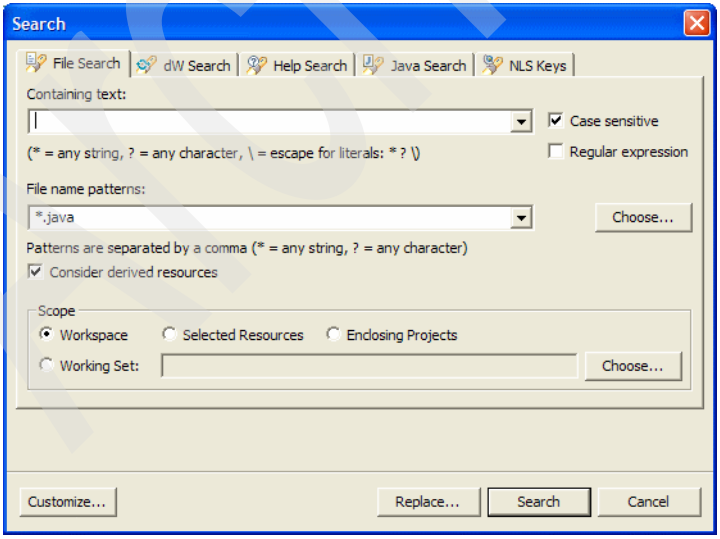


Figure 7-58 Java editor - Search

The following types of searches are supported in Rational Application Developer:

- File search:
  - Files can be search for the specified text.
  - Regular expressions can now be specified in the search text.
  - You can specify the scope of the search that includes the workspace, a select set of resources, enclosing projects (you can select specific projects in your workspace in the Package Explorer; selecting this option will search only through the selected project), or a working set. Working sets are discussed later in this section.
  - The results of this search are displayed in the Search view.
- dW search: Rational Application Developer introduces searches of the developer works Web site from within the Workbench. The results of this search are displayed in the Classic Search view.
- Help search: You can search the contents of help. The scope can be set to all or a working set. Working sets are discussed later in this section.
- Java search: You can search for Java types, methods, packages, constructors, and fields.
- In Rational Application Developer, you have the option of searching through the JRE system libraries as well, for example, a Java search for ArrayList and its result set. The scope can be set to workspace, selected resources, or working sets. Workings sets are discussed later in this section.
- NLS keys:
  - You can search for unused keys in a property file or incorrectly used keys using the NLS key search.
  - You have to specify the accessor class that reads the property file and the name of the property file itself.
  - The scope of the search can be set to workspace, selected resources, enclosing projects, or working sets. Working sets are discussed later in this section.

7.4.9 Working sets

Working sets are used to filter resources by only including the specified resources. They are selected and defined using the view's filter selection dialog. We will use an example to demonstrate the creation and use of a *working set* as follows:

1. Open a search dialog by clicking the search icon( ) or pressing Ctrl+H.

Archived

2. When the Search dialog appears, enter `itso.bank.model.Credit` in the search string field.
3. Select **Working Set** under Scope and then click **Choose**.
4. From the Select Working Set dialog, click **New** to create a new working set.
5. When the New Working Set dialog appears, select **Java** to indicate that the working set will comprise of only Java resources and then click **Next**.
6. When the Java Working Set dialog appears, expand **BankJava** → **src**.
   a. Select **itso.bank.model**.
   b. Type `EntityPackage` for the name and click **Finish.**
7. Click **OK** in the Select Working Set dialog.
8. We have now created a working set containing Java resources comprised of all the Java files in the itso.bank.model package named EntityPackage.
9. Double-click the line in the search result to open the file in the source editor.

7.4.10 Quick Assist (Quick Fix)

Rational Application Developer introduces *Quick Assists* in the editor to provide suggestions to complete Java editor tasks quickly. When enabled, a green lightbulb ( ) is displayed.

Enable Quick Assists

By default, the display of the Quick Assists lightbulb is disabled. To enable the Quick Assists feature, do the following:

1. Select **Window** → **Preferences**.
2. Expand **Java** and select **Editor**.
3. Check **Light bulb for quick assists** on the Appearance tab.

Invoking Quick Assists

To use the Quick Assists feature once enabled, double-click the lightbulb ( ) icon. Alternatively, press Ctrl+1 to provide a list of intelligent suggestions, and selecting one completes the task.

7.4.11 Code Assist (content)

The *Code Assist* feature of the Rational Application Developer displays possible completions that are valid with the current context.

Archived

Code Assist preferences

The Code Assist preferences can be configured as follows:

1. Select **Window** → **Preferences**.
2. Expand **Java** → **Editor** → **Code Assist**.
3. Modify the settings as desired and then click **OK**.

Invoke Code Assist

Code Assist is invoked by pressing Ctrl+Space bar.

In the example shown in Figure 7-59, the Code Assist wizard provides the method completions that are valid for the transaction object by displaying all the public methods of the Transaction class. In the Code Assist window use the arrow keys, select the method, and press Enter to complete the method call. Code Assist can also be invoked to insert or to complete Javadoc block tags in the Javadoc comment block.

Archived

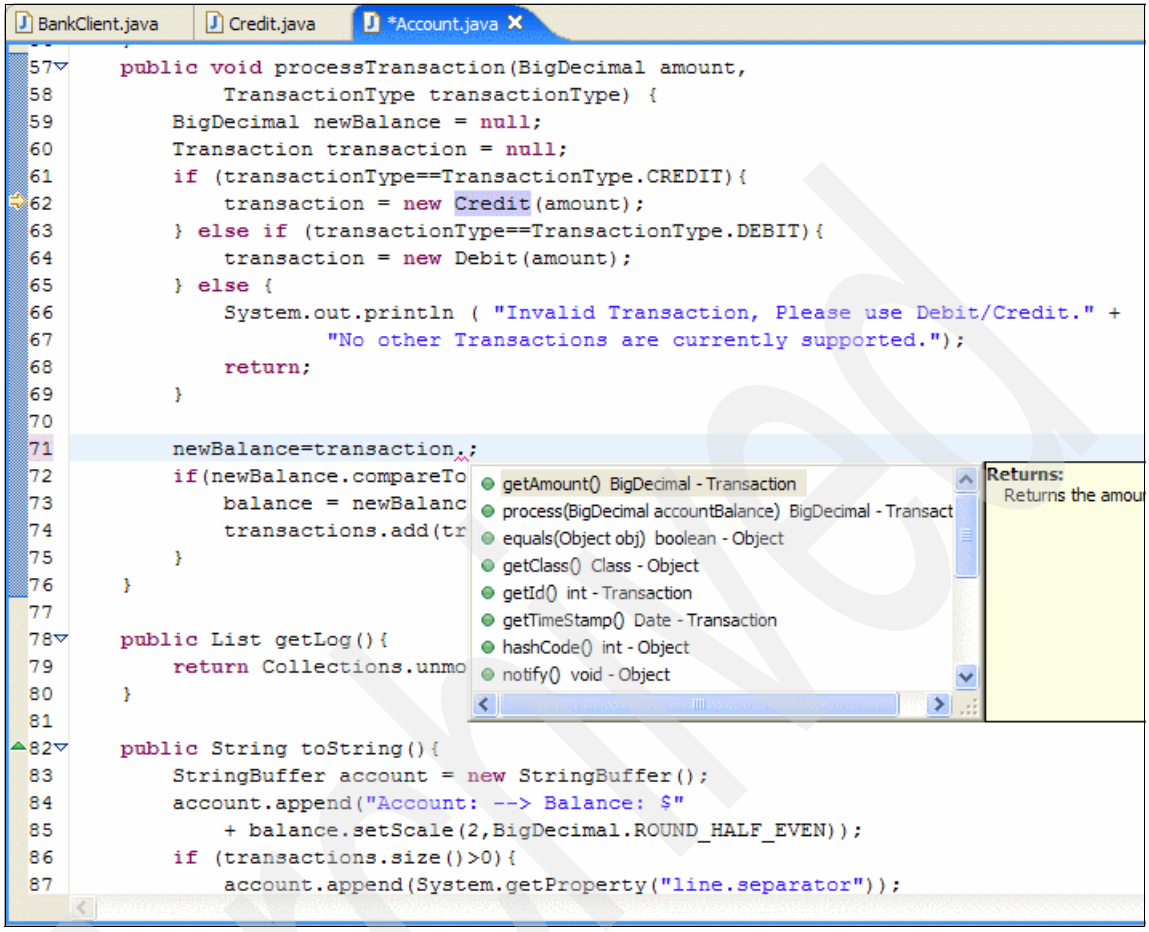


Figure 7-59 Java editor - Code/content assist

Code Assist also provides Javadoc in an information window that pops up when you hover over a java element. You can customize the code assists and provide your own templates that will pop up, by using Rational Application Developer's *Templates* feature.

7.4.12 Import generation

The Rational Application Developer Java editor simplifies the task of finding the correct import statements to use in your Java code.

Simply select the type name in the code and select **Source** → **Add Import** from the context menu or press Ctrl+Shift+M. If the type name is unambiguous, the import will be pasted at the correct place in the code. If the type exists in more

than one package, a window with all the types is displayed and you can choose the correct type for the import statement.

Figure 7-60 shows an example where the selected type (BigDecimal) exists in several packages. Once you have determined that the java.math package is what you want, double-click the entry in the list, and the import statement is generated in the code.

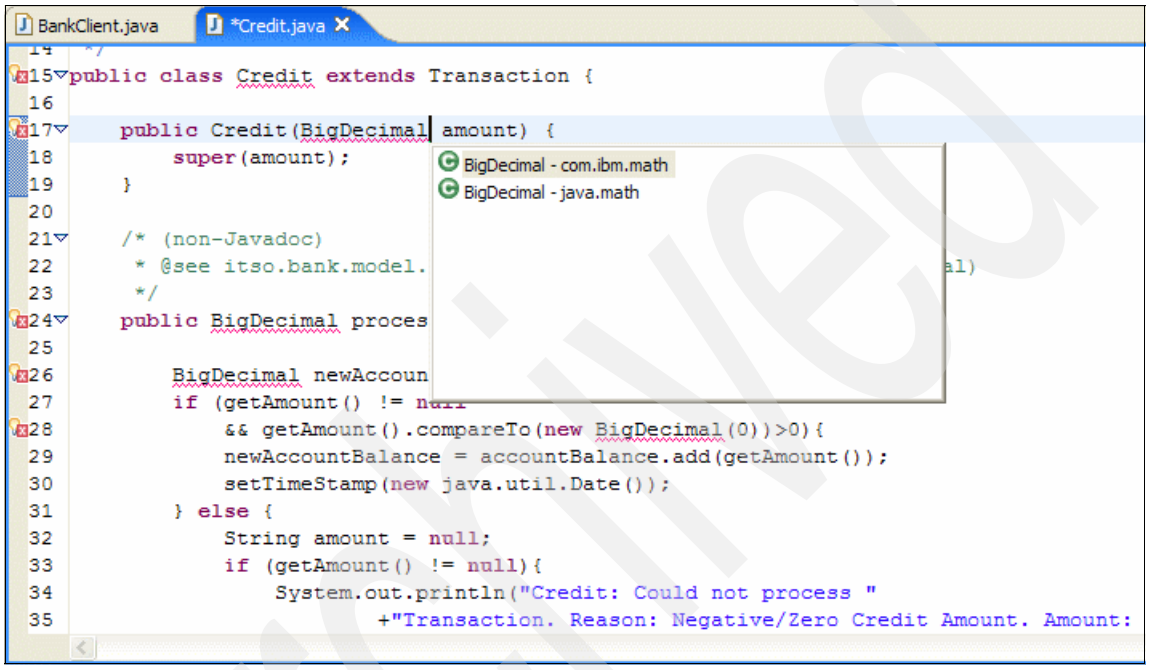


Figure 7-60 Java editor - Import generation

You can also add the required import statements for the whole compilation unit. Open the context menu somewhere in the Java source editor and select **Source → Organize Imports**. The code in the compilation unit is analyzed and the appropriate import statements are added.

You can control the order in which the imports are added and when package level imports should be used through the Preferences dialog.

Archived

7.4.13 Generate getters and setters

When working with the Java editor you can generate accesors (getters and setters) for the fields of a type inside a compilation unit. There are several ways to generate getters and setters for a field:

- Select **Source** → **Generate Getter and Setter...** from the context menu in the Java editor.
- Select **Source** → **Generate Getter and Setter...** from the context menu of the field in the Outline view.
- Select **Source** → **Generate Getter and Setter...** from the menu bar.

A dialog opens to let you select which methods you want to create. Select the methods and click **OK** (see Figure 7-61 on page 325).

Archived

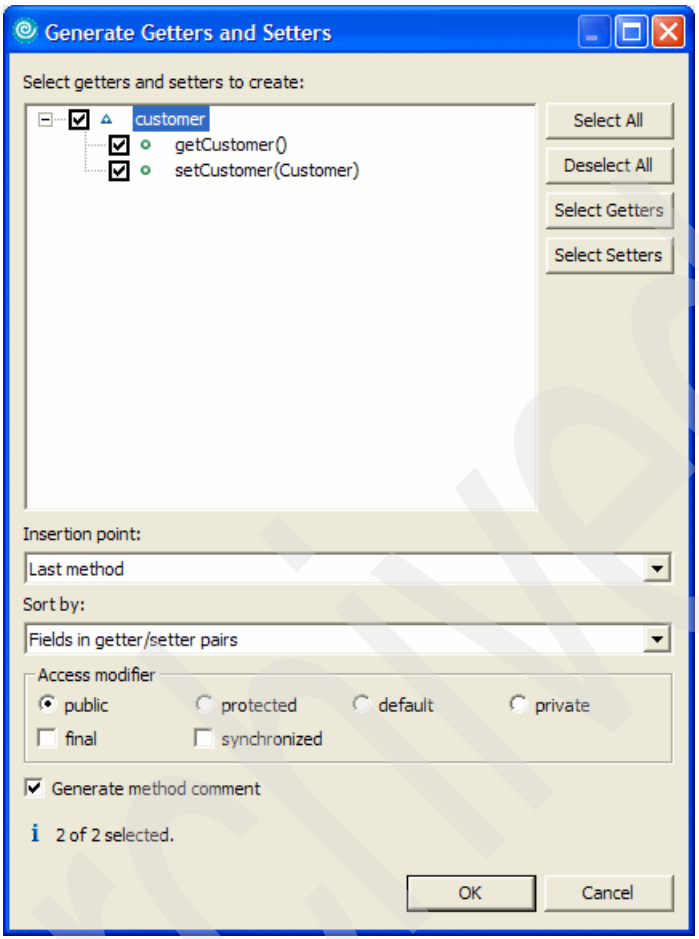


Figure 7-61 Generate Getter and Setter dialog

7.4.14 Override/implement methods

The override methods feature helps you to override methods from the superclass or implement methods from an interface implemented by the selected type. Select **Source** → **Override/Implement Methods**... from the menu or select **Override/Implement Methods**... in the context menu of a selected type in the editor or on a text selection in a type.

The Override/Implement Methods dialog (see Figure 7-62 on page 326) displays all methods that can be overridden from superclasses or implemented from interfaces. Abstract methods or not yet implemented methods are selected by default.

Archived

When clicking **OK**, method stubs for all selected methods are created.

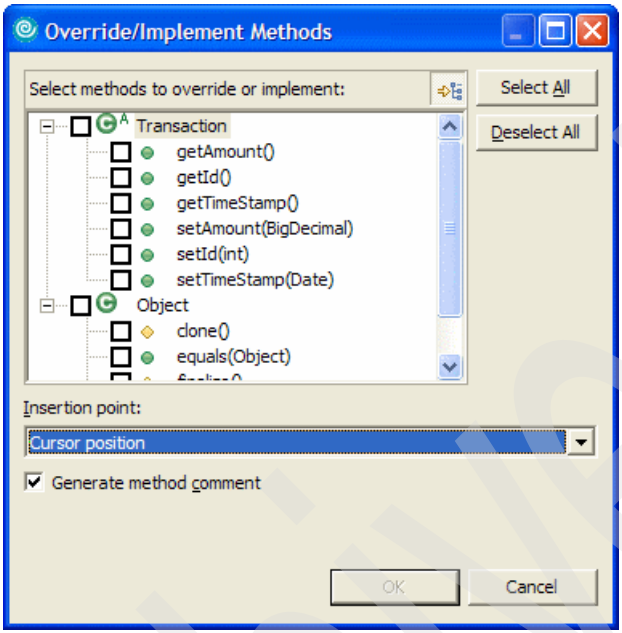


Figure 7-62 Java editor - Override/Implement Methods

7.4.15 Adding constructors

This feature allows you to automatically add any or all of the constructors defined in the superclass for the currently selected type. Open the context menu in the Java editor and select **Source** → **Add Constructors from Superclass**. Selecting any of the methods, as shown in Figure 7-63 on page 327 will add the selected constructor to the current type.

Archived

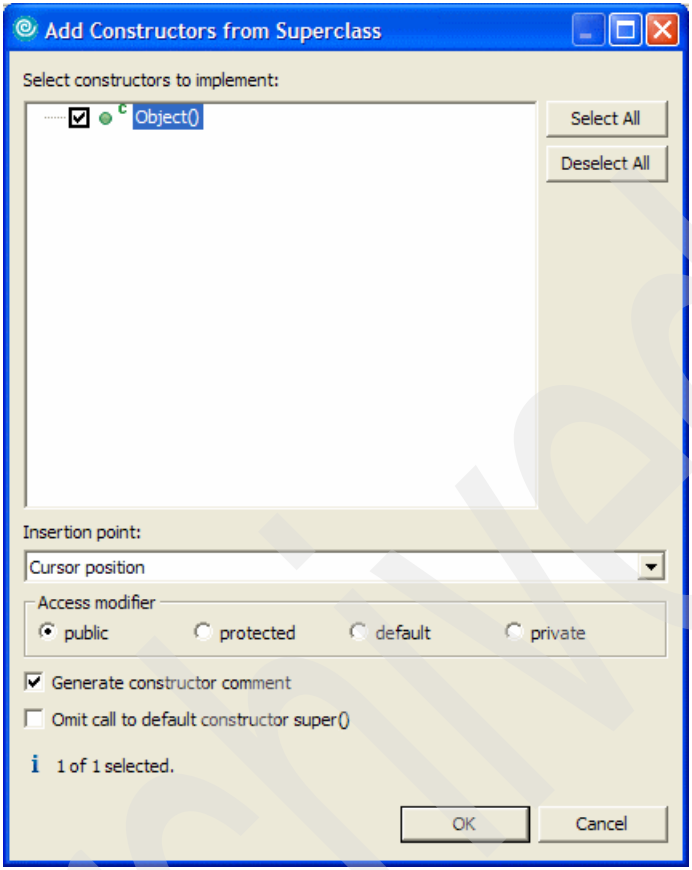


Figure 7-63 Java editor - Add Constructors from Superclass

The Delegate Method Generator feature allows you to delegate methods from one class to another. This feature is applicable to java files, classes, and fields in the Package Explorer.

To use this feature, do the following:

1. Select a Java file, a class, or a field in a class.
2. Right-click, and in the context menu choose **Source** → **Generate Delegate Methods**. For example, select **firstName** in the class Customer and choose **Source** → **Generate Delegate** from the context menu.
3. When the Delegate Method Generation dialog appears, click **OK**.

   This adds all String class (since firstName is of type String) methods to the Customer class, and code is added in the body of the method to delegate the method call to the String Class through the firstName attribute. Do not save

the changes to the Customer class, as this was just an exercise to demonstrate the Delegate Method Generator feature.

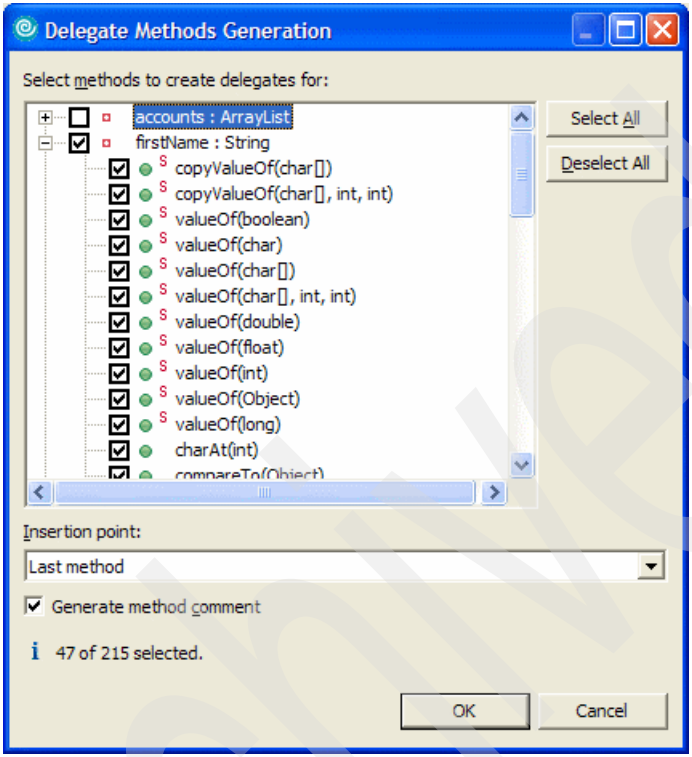


Figure 7-64 Java editor - Delegate Method Generator

7.4.16 Refactoring

When developing Java applications, it is often necessary to perform tasks such as renaming classes, moving classes between packages, and breaking out code into separate methods. In traditional programming environments such tasks are both time consuming and error prone, because it is up to the programmer to find and update each and every reference throughout the project code. Rational Application Developer provides an enhanced list of refactor actions compared to previously released products to automate the this process.

Note: We found that when renaming servlet classes or packages, the corresponding servlet configuration was not updated automatically. To ensure that such resources are updated, use the “Update fully qualified names in non-java files” option in the Rename Compilation Unit dialog and enter `*.xml` in the File name patterns field.

The Java development tools (JDT) of Application Developer provide assistance for managing refactoring. In the Refactoring wizard you can select:

- **Refactoring with preview**: Click **Next** in the dialog to bring up a second dialog panel where you are notified of potential problems and are given a detailed preview of what the refactoring action will do.
- **Refactoring without preview**: Click **Finish** in the dialog and have the refactoring performed. If a stop problem is detected, refactoring is halted and a list of problems is displayed.

Table 7-7 provides a summary of common refactoring actions. For an exhaustive list of refactoring actions available in Rational Application Developer, check the product help documentation, under **Developing Java Applications → Using the Java Integrated Development Environment → JDT Actions → Refactor**.

Table 7-7 Refactoring actions

Name	Function
Rename	Starts the Rename refactoring wizard. Renames the selected element and (if enabled) corrects all references to the elements (also in other files). Is available on methods, fields, local variables, method parameters, types, compilation units, packages, source folders, projects, and on a text selection resolving to one of these element types.
Move	Starts the Move refactoring wizard. Moves the selected elements and (if enabled) corrects all references to the elements (also in other files). Can be applied on one or more static methods, static fields, types, compilation units, packages, source folders and projects, and on a text selection resolving to one of these element types.
Change Method Signature	Starts the Change Method Signature refactoring wizard. You can change the visibility of the method; additionally, you can change parameter names, parameter order, parameter types, add parameters, and change return types. The wizard all references to the changed method.
Extract Interface	Starts the Extract Interface refactoring wizard. You can create an interface from a set of methods and make the selected class implement the newly created interface.
Push Down	Starts the Push Down refactoring wizard. Moves a field or method to its subclasses. Can be applied to one or more methods from the same type or on a text selection resolving to a field or method.

Archived

Name	Function
Pull Up	Starts the Pull Up refactoring wizard. Moves a field or method to its super class. Can be applied on one or more methods and fields from the same type or on a text selection resolving to a field or method.
Extract Method	Starts the Extract Method refactoring wizard. Creates a new method containing the statements or expressions currently selected, and replaces the selection with a reference to the new method.
Extract local Variable	Starts the Extract Variable refactoring wizard. Creates a new variable assigned to the expression currently selected and replaces the selection with a reference to the new variable.
Extract Constant	Starts the Extract Constant refactoring wizard. Creates a static final field from the selected expression and substitutes a field reference, and optionally replaces all other places where the same expression occurs.
Inline	Starts the Inline refactoring wizard. Inlines local variables, methods, or constants. This refactoring is available on methods, static final fields, and text selections that resolve to methods, static final fields, or local variables.
Encapsulate Field	Starts the Encapsulate Field refactoring wizard. Replaces all references to a field with getter and setter methods. Is applicable to a selected field or a text selection resolving to a field.
Undo	Does an Undo of the last refactoring.
Redo	Does a Redo of the last undone refactoring.

Refactoring example

The following example of a refactoring operation assumes that you want to rename a class *Debit* that has been misspelled as *Debti* in your Java project.

To initiate the refactoring to rename *Debti* to *Debit*, do the following:

1. Select the **Debti** class.
2. Right-click and select **Refactor** → **Rename** from the context menu. Alternatively, press the Alt+Shift+R hot key sequence.
3. When the Rename Compilation Unit wizard appears, select the appropriate refactoring settings. For example, we did the following (as seen in Figure 7-65), and then clicked **Preview**:
   - New name: `Debit`
   - Check **Update references** (default).

– Check **Update textual matches in comments and strings** (default).
– Check **Update fully qualified names in non-java files**.
  • File name patterns: `*.java`

These options force the wizard to display a preview indicating what changes will be made by the Rename Refactoring wizard.

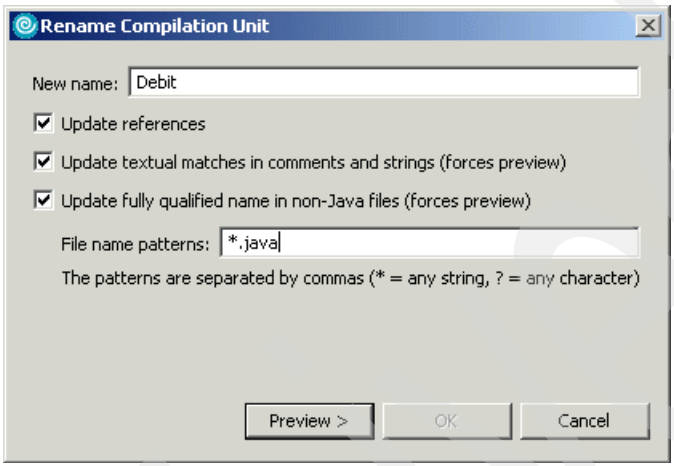


Figure 7-65 Refactoring - Rename Compilation Unit wizard

If there are any files with unsaved changes in the Workbench and you have not indicated in the preferences that the save should be done automatically, you are prompted to save these files before continuing the refactoring operation.

If there are problems or warnings, the wizard presents the user with the Found Problems window. If the problems are severe, the Continue button will be disabled and the refactoring must be aborted until the problems have been corrected.

4. After reviewing the changes that will be applied, you can again select whether to accept or cancel the refactoring operation.

   Click **OK** to perform the renaming operation of the class. If there are any problems detected, they will be displayed after the operation has been completed.

   Rational Application Developer provides one level of an undo operation for refactoring commands. If you want to undo the renaming changes at this point, select **Refactor** → **Undo** from the menu bar.

Archived

8

Develop Java database applications

In an enterprise environment, applications that use databases are very common. In this chapter we explore technologies that are used in developing Java database applications. Our focus is on highlighting the database tooling provided with IBM Rational Application Developer V6.0.

This chapter is organized into the following sections:

- Introduction to Java database programming
- Preparing for the sample
- Data perspective
- Create databases and tables from scripts
- Create and work with database objects
- UML visualization
- Create SQL statements
- Access a database from a Java application
- Java stored procedures

© Copyright IBM Corp. 2005. All rights reserved.

8.1 Introduction to Java database programming

This section includes an introduction to Java database technologies and highlights the key Rational Application Developer database tooling features.

8.1.1 JDBC overview

Java Database Connectivity (JDBC), like Open Database Connectivity (ODBC), is based on the X/Open SQL call-level interface specifications; but unlike ODBC, JDBC does not rely on various C features that do not fit well with the Java language. Using JDBC, you can make dynamic calls to databases from a Java application or Java applet.

JCBC is vendor neutral and provides access to a wide range of relational databases, as well as other tabular sources of data. It can even be used to get data from flat files or spreadsheets. JDBC is especially well suited for use in Web applications. Using the JDBC API you can connect to databases using standard network connections. Any modern Web browser is Java enabled, so you do not have to worry about whether the client can handle the application.

Figure 8-1 shows the basic components of JDBC access. The JDBC API sends the SQL commands from the application through a connection to the vendor-specific driver that provides access to the database. Connections can be established through a driver manager (JDBC 1.x) or a data source (JDBC 2.x).

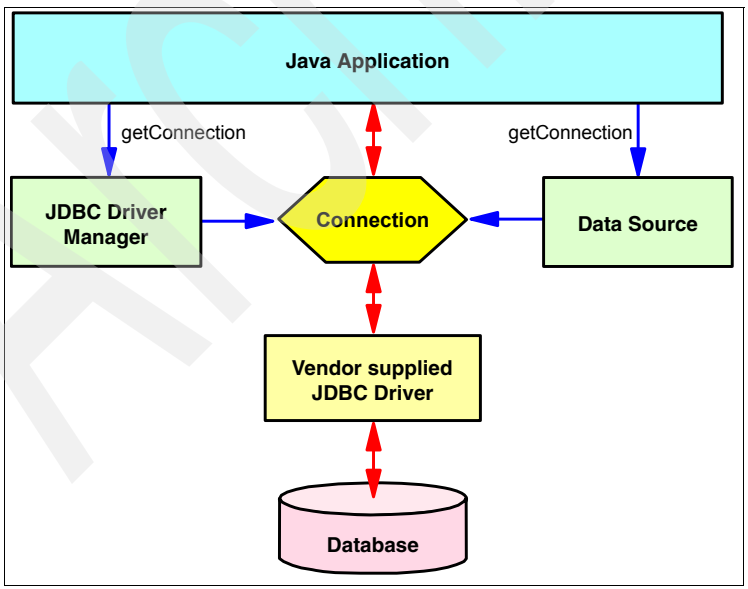


Figure 8-1 JDBC overview

8.1.2 Data source versus direct connection

In JDBC 1.x the only way of establishing a database connection was by using the *DriverManager* interface. This was expensive in terms of performance since a connection was created each time you had to access the database from your program, thereby incurring a substantial processing overhead. In the JDBC 2.x Standard Extension API, an alternative way of handling database connections was introduced.

Data source connection pooling advantages

By using data source objects you have access to a pool of connections to a data source. Using connection pooling gives you the following advantages:

- Improves performance

  Creating connections is expensive; a data source object creates a pool of connections as soon as it is instantiated.

- Simplifies resource allocation

  Resources are only allocated from the data source objects, and not at arbitrary places in the code.

Important: Because of the advantages of connection pooling, using data source objects is the preferred method of handling database connections in Web applications. The WebSphere Application Server has full support for connection pooling and for registering data sources through JNDI.

Data source objects at work

Data source objects work as follows:

- When a servlet or other client wants to use a connection, it looks up a data source object by name from a Java Naming and Directory Interface (JNDI) server.
- The servlet or client asks the data source object for a connection.
- If the data source object has no more connections, it may ask the database manager for more connections (as long as it has not exceeded the maximum number of connections).
- When the client has finished with the connection, it releases it.
- The data source object then returns the connection to the available pool.

8.1.3 XMI and DDL

XML Metadata Interchange (XMI) is an Object Management Group (OMG) standard format for exchanging metadata information. Rational Application

Archived

Developer uses the XMI format to store all local descriptors of databases, tables, and schemas. Rational Application Developer appends a prefix to the extension of the XMI files, for example, local descriptors for databases are stored in databasename.*dbxmi* files, schemas are stored in schemaname.*schxmi*, and tables are stored in a tablename.*tblxmi* file. The content of these XMI files can be viewed and edited using tailored editors. When you import an existing database model, it can be stored in XMI format.

Data definition language (DDL) is a format used by relational database systems to store information about how to create database objects. Rational Application Developer allows you to generate DDL from an XMI file and vice versa.

8.1.4 Rational Application Developer database features

Rational Application Developer provides a number of features that make it easier to work with relational databases in your projects.

- Built-in Cloudscape database for development use that allows one client connection. This is a useful feature for rapid development. There is no extra installation required to use this database. A database and database objects can be created in this database once Rational Application Developer is installed.
- Support for multiple versions of Cloudscape, DB2, Informix, SQL Server, and Sybase databases.

  Note: To obtain a more detailed listing of database types and versions supported by Rational Application Developer, search for `supported database types` in the Rational Application Developer online help.

- Ability to create new databases and database objects using class diagram, information engineering (IE) diagram, and integrated definition extended (IDEF1X) diagram logical modeling diagrams, and generate DDL for the target database.
- Ability to import and use existing database models.
- Ability to generate XML schemas from existing database models.
- Ability to interactively build and execute SQL queries from an imported database model or through an active connection, using the SQL Wizard and SQL Query Builder.
- Ability to generate Web pages and supporting Java classes based on existing or new SQL queries.
- Ability to access database API from JavaServer Pages, using either JavaBeans or JSP tags.

8.2 Preparing for the sample

In this chapter, we choose to provide the completed sample code up front and walk the reader through the Rational Application Developer database related tooling using the sample. This section describes how to import and set up the sample in Rational Application Developer.

Note: This section assumes that you have already downloaded and unpacked the redbook sample code 6449code.zip to the c:\6449code directory. For more detailed information, refer to Appendix B, “Additional material” on page 1395.

8.2.1 Import the BankDB sample project

To import the BankDB.zip project interchange file, do the following:

1. From the Workbench, select **File** → **Import**.
2. Select **Project Interchange** and click **Next**.
3. When the Import Projects dialog appears, we entered the following and then clicked **Finish**:
   - From zip file: `c:\6449code\database\BankDB.zip`
   - Project location root: `c:\workspace`
   - Check **BankDB**.

After importing the BankDB project interchange file you should see the BankDB project displayed (as seen in Figure 8-2) in the Data Definition view.

Archive

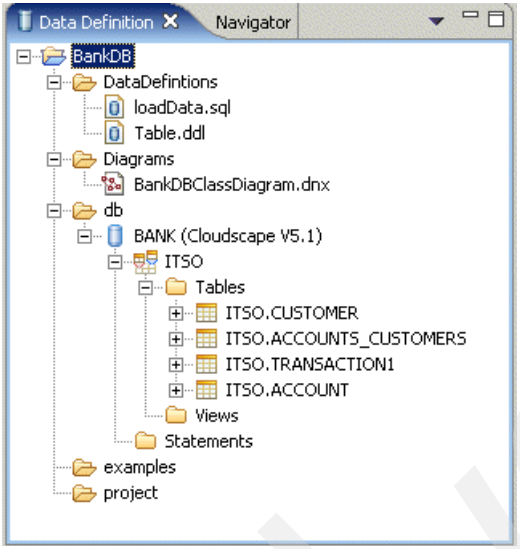


Figure 8-2 BankDB project displayed in the Data Definition view

8.2.2 Set up the BANK sample database

This section provides instructions for deploying the BANK sample database and populating the database with sample data. For simplicity we will use the built-in Cloudscape database.

Note: The database and tables can be created within Rational Application Developer, or externally using the native database utilities such as Cloudscape CView or DB2 UDB (DB2 Control Center, Command Window, etc.).

For detailed information on creating databases, creating connections, creating tables, and loading data in tables from within Rational Application Developer or externally via Cloudscape CView or DB2 UDB, refer to 8.4, “Create databases and tables from scripts” on page 343.

To create the database, create the connection, and create and populate the tables for the BANK sample, do the following:

1. Create the Cloudscape *BANK* database.

   For details refer to “Create Cloudscape database via Cloudscape CView” on page 344.

Archived

2. Create the connection to the Cloudscape BANK database from within Rational Application Developer.

   For details refer to “Create a database connection” on page 347.

3. Create the BANK database tables from within Rational Application Developer.

   For details refer to “Create database tables via Rational Application Developer” on page 350.

4. Populate the BANK database tables with sample data from within Rational Application Developer.

   For details refer to “Populate the tables within Rational Application Developer” on page 352.

8.3 Data perspective

The *Data perspective* is used to work with databases, database objects, and SQL statements. We will explore the following views of the Data perspective:

- Data Definition view
- Database Explorer view
- DB Output view
- Navigator view

Figure 8-3 shows the Data perspective and supporting views available within Rational Application Developer. IBM Rational Application Developer V6.0 introduces support for diagram tooling to design and deploy databases (local or remote). The tooling can also be used to generate DDL files that can be used later to generate databases and database objects.

Rational Application Developer introduces support for three diagrams used to design database diagrams and deploy them to local and remote databases or generate DDL files that can be later used to generate databases and database objects on a database server.

Archived

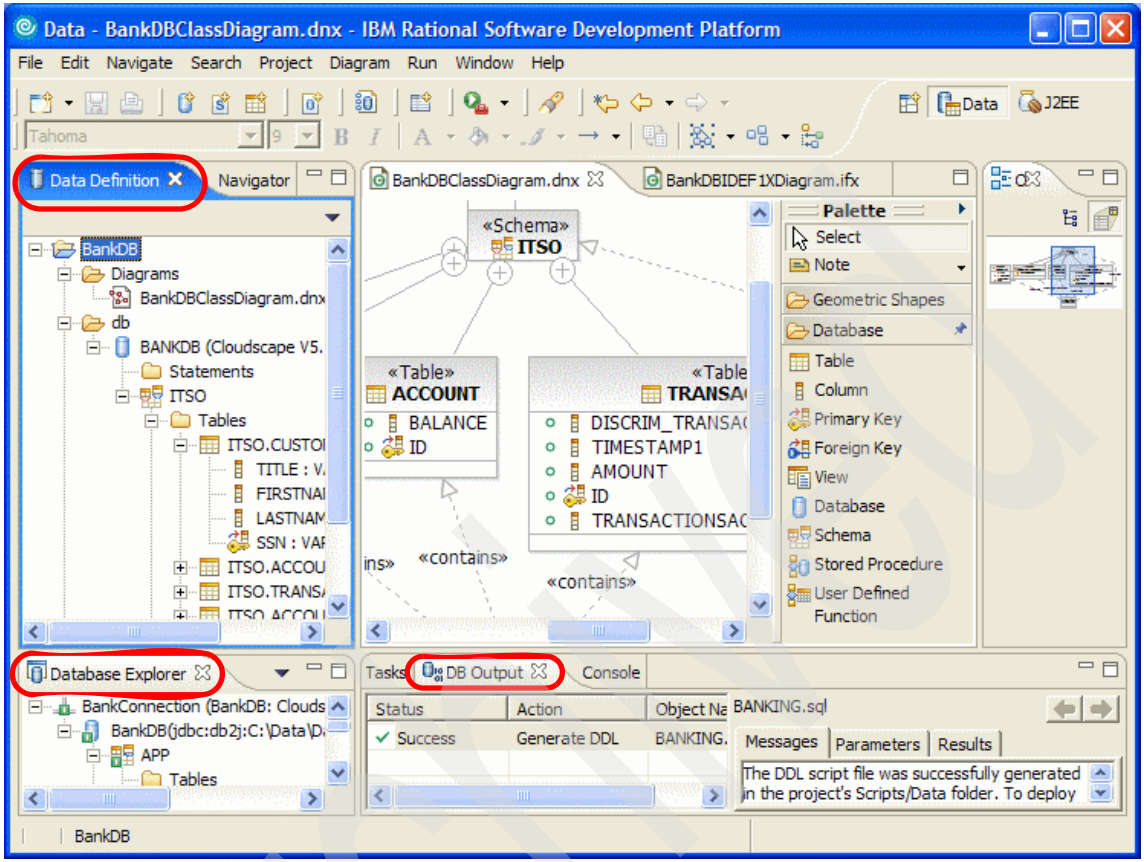


Figure 8-3 Data perspective and views

> **Note:** Additional information on the Data perspective can be found in “Data perspective” on page 145.

8.3.1 Data Definition view

The Data Definition view, as shown in Figure 8-4, contains copies of data definitions imported from an existing database, along with the designs that were created using the Data Definition view of Rational Application Developer.

Database models are stored within an Rational Application Developer project. This can be any type of project, such as a simple project, a Web Project with database access, or an EJB project with entity beans.

The Data Definition view is a hierarchical view of the database objects and does not display how these definitions are stored in actual files. To view the actual files you will need to switch to the Navigator view.

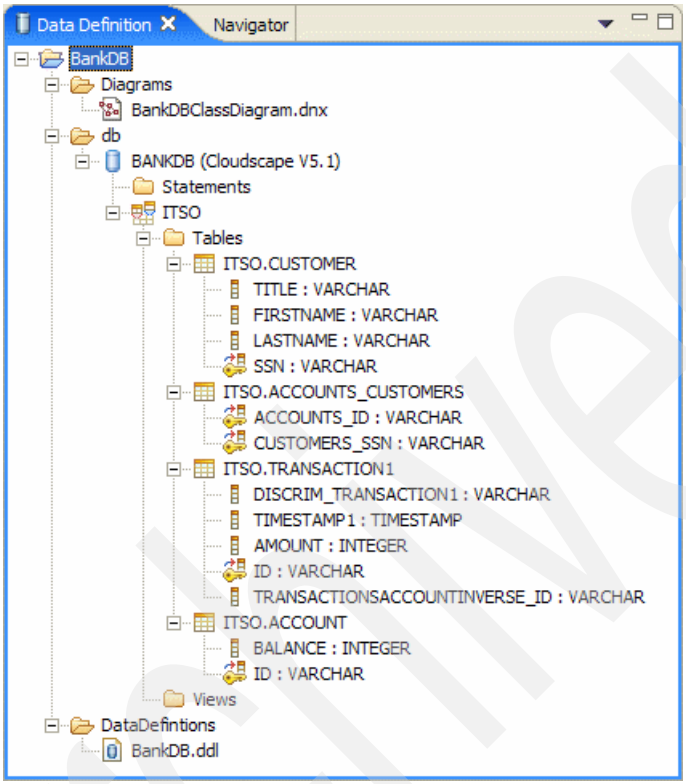


Figure 8-4 Data perspective - Data Definition view

8.3.2 Database Explorer view

The Database Explorer view shows active connections to databases and allows you to create new connections. In Figure 8-5 on page 342 you can see an active connection to the Cloudscape BANK database. We will create this connection in 8.4.2, “Create a database connection” on page 347.

Archived

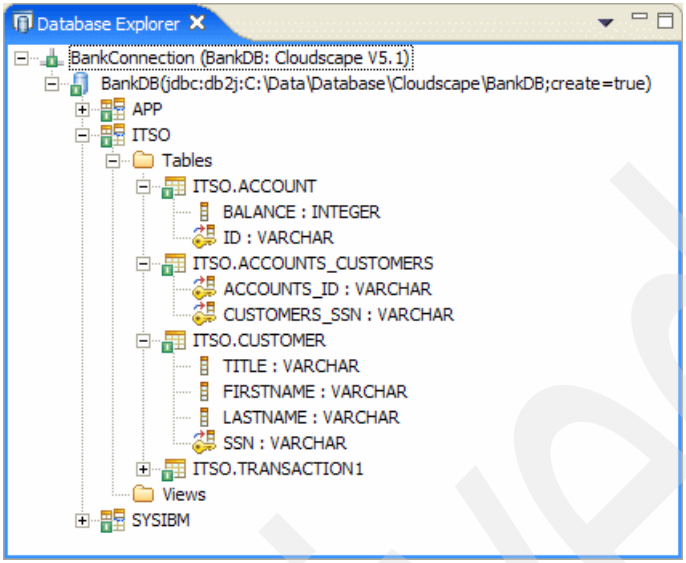


Figure 8-5 Data perspective - Database Explorer view

The Database Explorer view is independent of projects. It displays active connections to databases. An inactive connection can be activated by selecting the connection and then **Reconnect** from the context menu.

8.3.3 DB Output view

The DB Output view is used to display output information for operations performed on database objects or to display output when you run DDL or SQL files on connections, as described in 8.3.2, “Database Explorer view” on page 341. The DB Output view is shown in Figure 8-6 on page 342 after the execution of a DDL file on a local Cloudscape database.

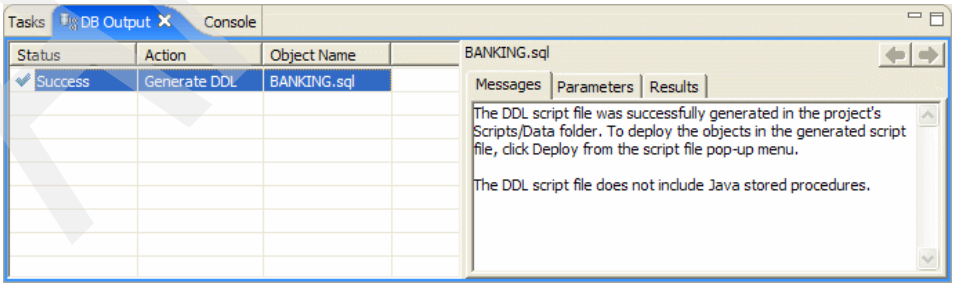


Figure 8-6 Data perspective - Database Output view

Archived

The DB Output has the Status, Action, and Object Name fields in the left side of the view, which display the status of the current operation, what action was performed, and the name of object that was used to perform the action, respectively.

The DB Output has the Messages, Parameters, and Results pages in the right side of the view, which display the progress of the action, any input to the operation, and any output returned by the operation.

8.3.4 Navigator view

The Navigator view is shown in Figure 8-7. In the Navigator view you can see the local descriptor files (.dbxmi, .schxmi, and .tblxmi files) that represent the various database objects. Each of the files has an editor associated with it. Double-clicking the file brings up the appropriate editor for the type of object that is described, which could be a database, a schema, or a table.

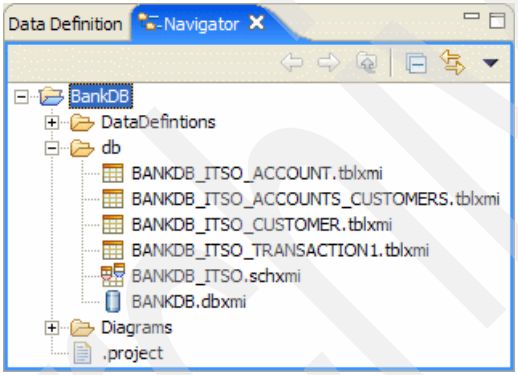


Figure 8-7 Data perspective - Navigator view

8.4 Create databases and tables from scripts

As part of the process of developing a database application, you will need to understand how to create databases, create a connection to the database, deploy tables, and populate the tables with data from scripts. This section describes how to create and work with databases and tables from within Rational Application Developer as well as externally for both Cloudscape and DB2 Universal Database (UDB) from a deployment perspective.

This section is organized into the following topics:

- Create a database.

Archived

  - Create Cloudscape database via Cloudscape CView.
  - Create DB2 UDB database via a DB2 command window.
- Create a database connection.
- Create the database tables from scripts.
  - Create database tables via Rational Application Developer.
  - Create Cloudscape database tables via Cloudscape CView.
  - Create DB2 UDB database tables via a DB2 command window.
- Populate database tables with data.
  - Populate the tables within Rational Application Developer.
  - Populate the tables via Cloudscape CView.
  - Populate the tables via a DB2 UDB command window.

8.4.1 Create a database

IBM Rational Application Developer V6.0 supports many database vender types (see “Supported databases” on page 9). In this section we focus on Cloudscape V5.1 included with Rational Application Developer, as well as IBM DB2 Universal Database V8.2. Many database types can be created from within Rational Application Developer using wizards, or externally using the native database utilities.

This section includes the following methods of creating a database:

- Create Cloudscape database via Cloudscape CView.
- Create DB2 UDB database via a DB2 command window.

Create Cloudscape database via Cloudscape CView

To create a Cloudscape database using the Cloudscape CView utility, do the following:

1. Start the Cloudscape CView utility by running cview.bat from one of the following directories:
   - Rational Application Developer Integrated WebSphere Application Server

     `<rad_home>\runtimes\base_v6\cloudscape\bin\embedded`

     Or:
   - WebSphere Application Server

     `<was_home>\cloudscape\bin\embedded`
2. From CView select **File** → **New** → **Database**, as seen in Figure 8-8.

Archived

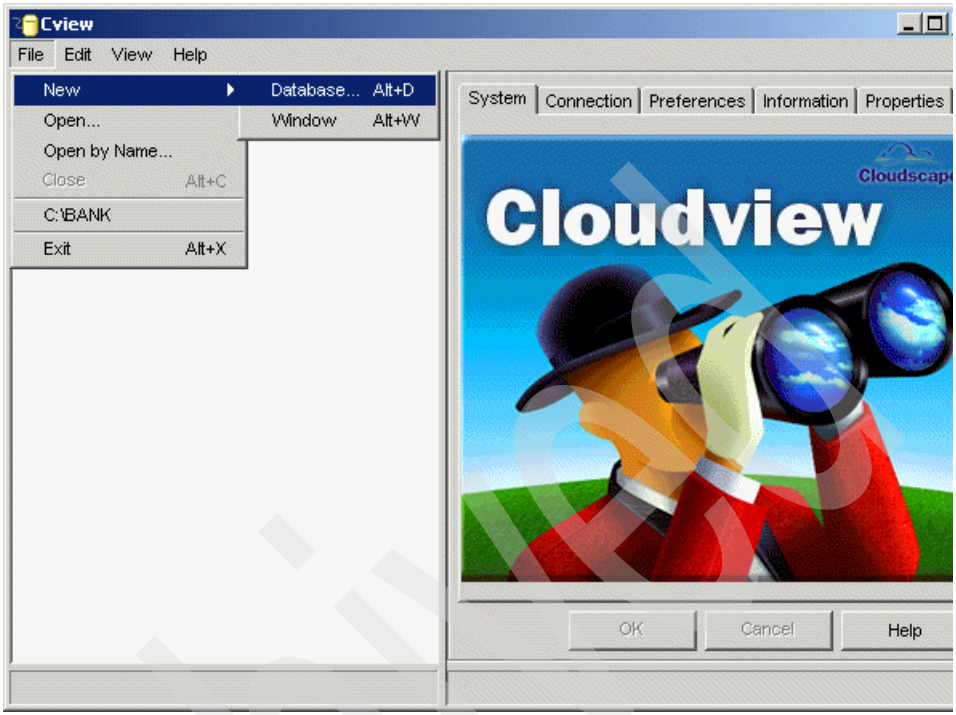


Figure 8-8 Cloudscape CView utility

3. When the New Database window appears, click the **Database** tab and enter the name <path_database_name>. For example, we entered c:\databases\BANK, as seen in Figure 8-9 on page 346, and then clicked **OK**.

Archived

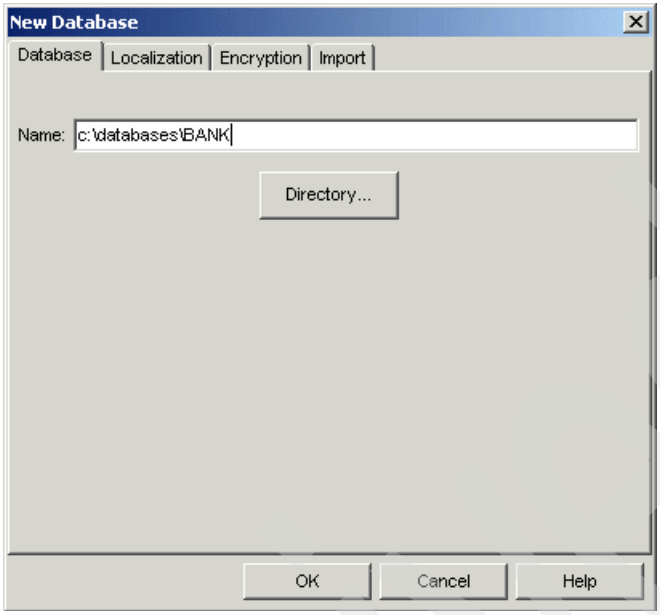


Figure 8-9 Cloudscape CView - Create BANK database

4. Close the Cloudscape CView utility.

Important: It is important that you close the CView utility after creating the database since it maintains a connection to the database.

The embedded version of Cloudscape only supports one connection from a JVM at a time. If the connection is open in the Database Explorer view, any applications we run (which will run in a separate JVM) will not be able to connect to the database. If you get errors in the Console view when you test a sample application, check that Bank Connection is disconnected or that CView is closed.

Create DB2 UDB database via a DB2 command window

To create a DB2 UDB database from a DB2 command window, do the following:

1. Open a DB2 UDB command window by selecting **Start** → **Programs** → **IBM DB2** → **Command Line Tools** → **Command Window**.
2. Enter the following in the DB2 command window to create a database:

   ```
   db2 create db <database_name>
   ```

   For example, create the following for the ITSO Bank sample:

   ```
   db2 create db BANK
   ```

Note: IBM DB2 Universal Database is not included with IBM Rational Application Developer V6.0. We used IBM DB2 Universal Database V8.2 for our testing.

8.4.2 Create a database connection

Once the database has been created (or if it existed previously) we need to create a database connection to the Cloudscape database (or other database type).

1. From the Data perspective, open the Database Explorer view.
2. Right-click in the Database Explorer view and select **New Connection...**.
3. When the New Database Connection dialog appears, enter the following (as seen in Figure 8-10) and then click **Next**:
   - Connection name: <connection name> (for example, Bank Connection).
   - Select **Choose a database manager and JDBC driver**.

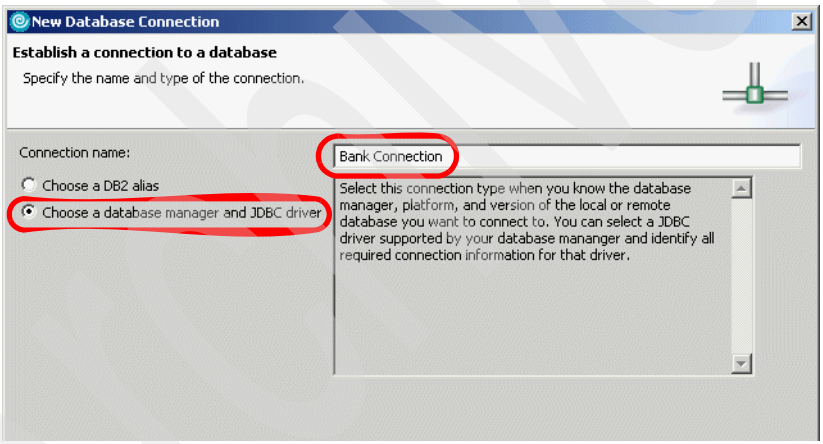


Figure 8-10 Database Connection - Name and type

4. When the Specify connection parameters dialog appears, enter the following (as seen in Figure 8-11 on page 349), and then click **Finish**:
   - Select database manager: Select **Cloudscape** → **V5.1**.

     Note: For DB2, select **DB2 Universal Database** → **Express V8.2**.

   - JDBC driver: Select **Cloudscape Embedded JDBC Driver**. The *JDBC driver* is a list of available drivers for the selected database manager. Typically you will use the default selected by Rational Application

Developer; if you have the need to use a different JDBC driver than the one specified, you will need to select this from the drop-down list.

Note: For DB2, select **IBM DB2 Application**.

– Database location: <database_directory> (for example, c:\databases\BANK).

Note: For DB2, enter BANK for the Alias.

– Check **Create the database if required**.
– Leave the Specify user information unchanged (not needed for Cloudscape).

Note: When using DB2 or Oracle you will have to provide the user information by providing the user ID and password. Also, you may need to update the class location. Once the user ID and password have been entered, you can click **Test Connection**.

Archived

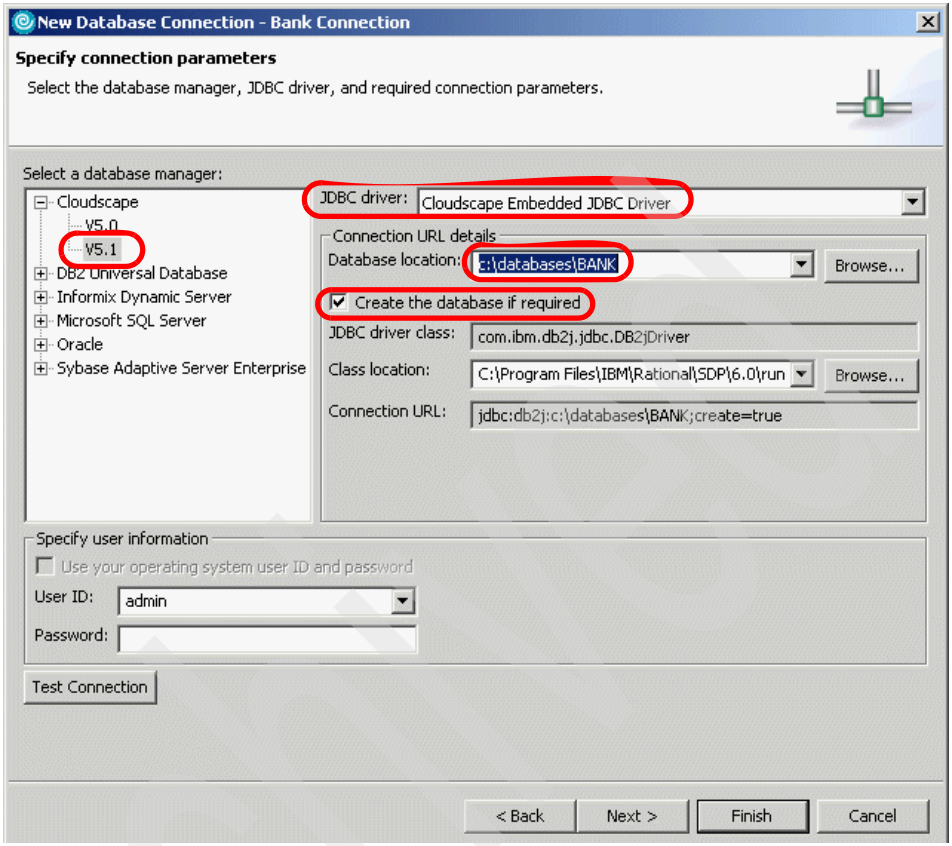


Figure 8-11 Database connection - Parameters

5. When prompted with the message `Do you want to copy the database metadata to a project folder?`, click **No**.

8.4.3 Create the database tables from scripts

This section describes how to create database tables from existing scripts using the following methods:

- Create database tables via Rational Application Developer.
  or
- Create Cloudscape database tables via Cloudscape CView.
  or
- Create DB2 UDB database tables via a DB2 command window.

Create database tables via Rational Application Developer

To create a database table from within Rational Application Developer using a wizard, do the following:

1. From the Navigator view of the Data perspective, select the project folder.
2. Create a folder named db to contain database files.
   a. Right-click the project and select **New** → **Folder**.
   b. Enter `db` for the folder name.
   c. Click **Finish**.
3. Right-click on the **db** folder and select **Import**.
4. When the Import window appears, select **File System** and click **Next**.
   a. In the From directory, click **Browse** to navigate to the following folder:

      `c:\6449code\database\cloudscape\Bank`

   b. Check Table.ddl and loadData.sql to import the table descriptors and sample customer data, respectively.
   c. Click **Finish**.
5. Right-click **Table.ddl** and select **Deploy...**.
6. When the Run Script - Statements dialog appears, accept the default (all statements checked), as seen in Figure 8-12 on page 350, and click **Next**.

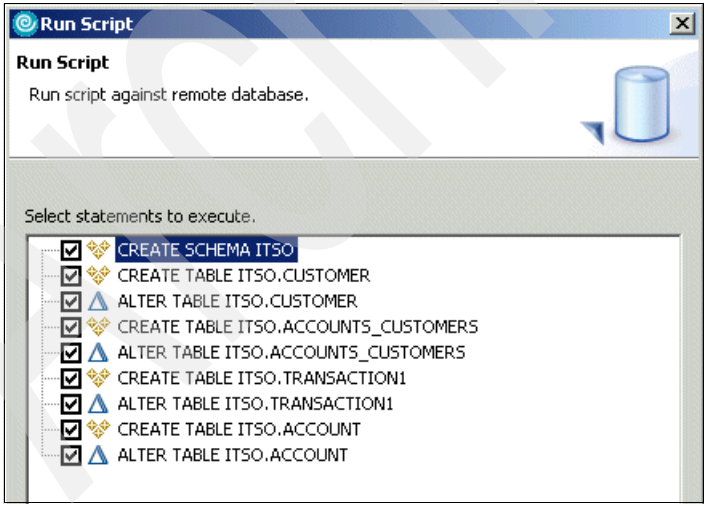


Figure 8-12 Create tables from the Table.ddl

7. When the Run Script - Options dialog appears, select **Commit changes only upon success** (default) and click **Next**.

Archived

8. When the Run Script - Database Connection dialog appears, do the following:
   - Check the **Use existing connection** check box.
   - Existing connection: Select the desired connection (for example, Bank Connection).
9. Click **Finish** to create the tables.

Create Cloudscape database tables via Cloudscape CView

To create a Cloudscape database table using the Cloudscape CView utility, do the following:

1. Ensure that a connection to the database is not active from within Rational Application Developer (only one connection permitted at a time to Cloudscape database). If a connection is active, select the connection in the Database Explorer view of the Data perspective, right-click and select Disconnect. Alternatively, close Rational Application Developer.
2. Launch the CView utility.
3. From CView, select the database (for example, c:\databases\BANK).
4. Click the **Database** tab.
5. Click the Script icon ( ) and select **Open**.
6. Navigate to the following directory and select the ddl file (for example, Table.ddl, found in the c:\6449code\database\cloudscape\BANK directory).
7. To execute the SQL to create the tables, click the Execute icon ( ).
8. To verify the tables where created, select and expand **Tables**. You should see the tables listed.

Create DB2 UDB database tables via a DB2 command window

To create a DB2 UDB database table from a DB2 command window, do the following:

1. Open a DB2 UDB command window by selecting **Start** → **Programs** → **IBM DB2** → **Command Line Tools** → **Command Window**.
2. From the DB2 command window, enter the following commands to create database tables:

```
db2 connect to <database_name>
db2 -tvf <path_ddl_file_name>
db2 connect reset
```

   For example, enter the following for the ITSO Bank sample:

```
db2 connect to BANK
db2 -tvf c:\6449code\database\DB2\BANK\Table.ddl
db2 connect reset
```

8.4.4 Populate database tables with data

This section describes how to populate database tables using SQL scripts containing data using the following methods:

- Populate the tables within Rational Application Developer.
- Populate the tables via Cloudscape CView.
- Populate the tables via a DB2 UDB command window.

Populate the tables within Rational Application Developer

To populate the tables within Rational Application Developer, do the following:

1. Import the loadData.sql as described in “Create database tables via Rational Application Developer” on page 350.
2. Right-click the **loadData.sql** file and select **Deploy...**.
3. When the Run Script - Statements dialog appears, accept the default (all checked) and click **Next**.
4. When the Run Script - Options dialog appears, select **Commit changes only upon success** (default) and click **Next**.
5. When the Database Connection dialog appears, do the following:
   - Check the **Use existing connection** check box.
   - Existing connection: Select **Bank Connection**.
6. Click **Finish** to populate the tables with data.
7. Verify the database was created and populated properly.
   a. In the Database Explorer view, right-click your database connection and select **Refresh**.
   b. Expand **Bank Connection** → **BANK** → **ITSO** → **Tables** as seen in Figure 8-13 on page 353.
   c. Right-click **ITSO.CUSTOMER** and select **Sample contents**. The contents of the table should be as shown in Figure 8-13 on page 353.
   d. Check that the other tables contain appropriate rows (check loadData.sql to find out what should be in the tables).
8. Right-click the connection (for example, Bank Connection) and select **Disconnect**.

Archived

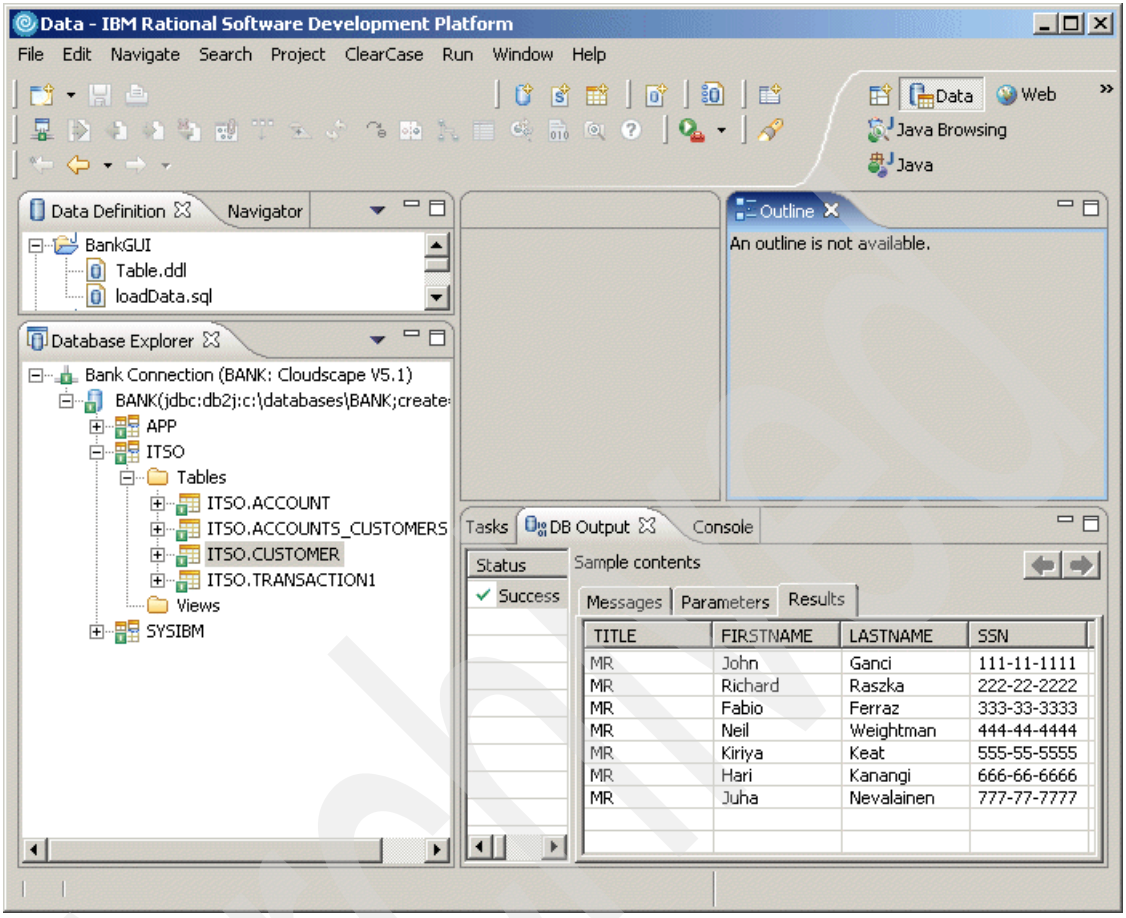


Figure 8-13 Bank connection working correctly

Populate the tables via Cloudscape CView

To populate the database tables with data from Cloudscape CView, do the following:

1. From CView, select the database (for example, c:\databases\BANK).
2. Click the **Database** tab.
3. Click the Script icon ( ) and select **Open**.
4. Navigate to the directory and select the sql file (for example, loaddata.sql, found in the c:\6449code\database\cloudscape\BANK directory).
5. To execute the SQL to create the tables, click the Execute icon ( ).

6. To verify that the tables where populated, select the **CUSTOMER** table, and then click the **Data** tab. You should see a list of seven customers.

Populate the tables via a DB2 UDB command window

To populate the tables via a DB2 UDB command window, do the following:

1. Open a DB2 UDB command window by selecting **Start** → **Programs** → **IBM DB2** → **Command Line Tools** → **Command Window**.
2. From the DB2 command window, enter the following commands to create database tables:

   ```
   db2 connect to <database_name>
   db2 -tvf <path_sql_filename>
   db2 connect reset
   ```

   For example, we did the following for the ITSO Bank sample:

   ```
   db2 connect to BANK
   db2 -tvf c:\6449code\database\DB2\BANK\loaddata.sql
   db2 connect reset
   ```

3. To verify that the sample data was added successfully, enter the following SQL command to query the CUSTOMER table:

   ```
   db2 connect to BANK
   db2 select * from ITSO.CUSTOMER
   ```

8.5 Create and work with database objects

This section describes how to use the Rational Application Developer database tooling to create database objects from scratch and work with the objects.

Rational Application Developer provides support for creating new databases, new schemas, and new tables. You can create database objects using either the context menu in the Navigator and Data Definition views or you could use the UML visualization with class diagram, IE, or IDEF1X notation to create database objects. We cover UML visualization in 8.6, “UML visualization” on page 370.

You can use the DB Explorer view to connect to existing databases and view their objects. The objects can be imported into Rational Application Developer and used in your applications. The DB Explorer view allows you to filter the designs that are returned to only show a subset of schemas or tables. You can also use the DB Explorer view to generate DDL files.

Archived

Important: The DB Explorer view is read-only. Before you can edit any database objects, you have to import them into an Rational Application Developer project.

This section is organized into the following topics:

- Create a database.
- Create a database connection.
- Create a schema.
- Create a table.
- Generate a DDL file.
- Deploy DDL from the workspace to a database.
- Copy database objects from a DDL file to a workspace.
- Generate DDL and XSD files for database objects.

8.5.1 Create a database

To create a database from scratch within Rational Application Developer, do the following:

1. To create a new database you have to have a project. If you have not already done so, you should now create a new simple project called BankDB and a folder called db.
2. Switch to the Data perspective and open the Data Definition view.
3. Select **BankDB** → **db**.
4. Right-click and select **New** → **Database Definition** from the context menu.
5. When the Database Definition wizard appears, enter the following, as seen in Figure 8-14 on page 356:
   - Folder: `/Bank/db`
   - Database name: `BankDBClone`
   - Database vender type: Select **Cloudscape V5.1**. When you later generate the database DDL, it will conform to the database type that you select.
6. Click **Finish** to create the new database definition.

Archived

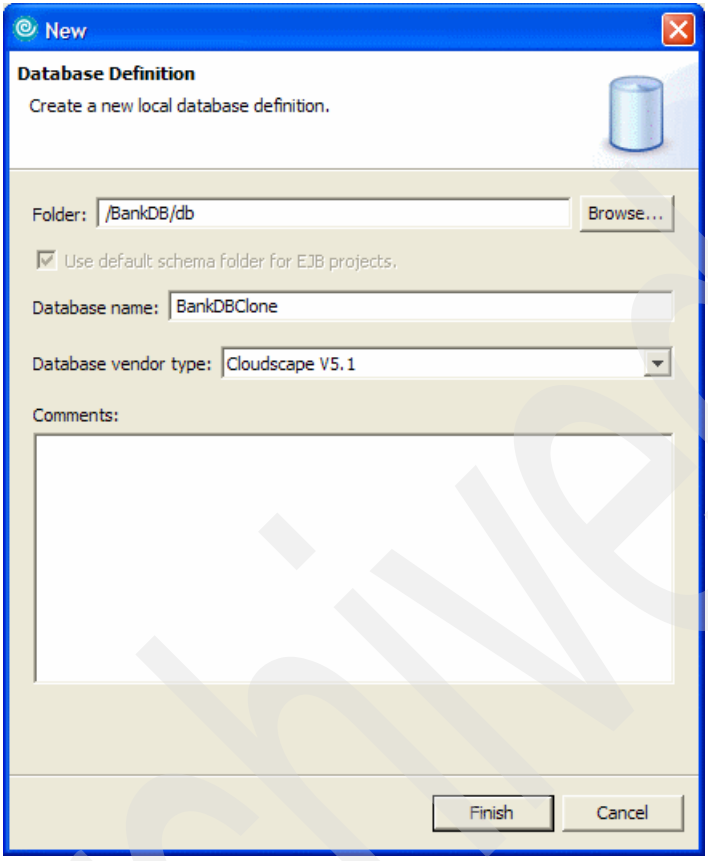


Figure 8-14 Database Definition wizard - New database definition

Important: Database definitions created within Rational Application Developer are not automatically created in the database system. You have to deploy the DDL as specified in 8.5.6, “Deploy DDL from the workspace to a database” on page 362, or use the appropriate database tool to create the objects from the DDL.

8.5.2 Create a database connection

To view the definition of an existing database, you first have to create a JDBC connection to the database. For details on how to create a connection refer to 8.4.2, “Create a database connection” on page 347.

For more information regarding JDBC drivers, refer to “JDBC drivers” on page 391.

Archived

8.5.3 Create a schema

Database schemas provide a method of logical classification of objects for a database. Some of the objects that a schema may contain include tables, views, aliases, indexes, triggers, and structured types.

The support for schemas varies between database types; some require them, and some have no support for them. The schema options available to you depend on the database vender type that you chose when the database was created. If the database type does not support schemas at all, this option will not be available, and the tables and other objects will be created directly under the database node.

To add a schema to a database, do the following:

1. Select the **BankDBClone** database definition previously created in 8.5.1, “Create a database” on page 355.
2. Right-click and select **New** → **Schema Definition**.
3. When the Schema Definition wizard appears, enter the Schema name (for example, ITSO) and click **Finish**.

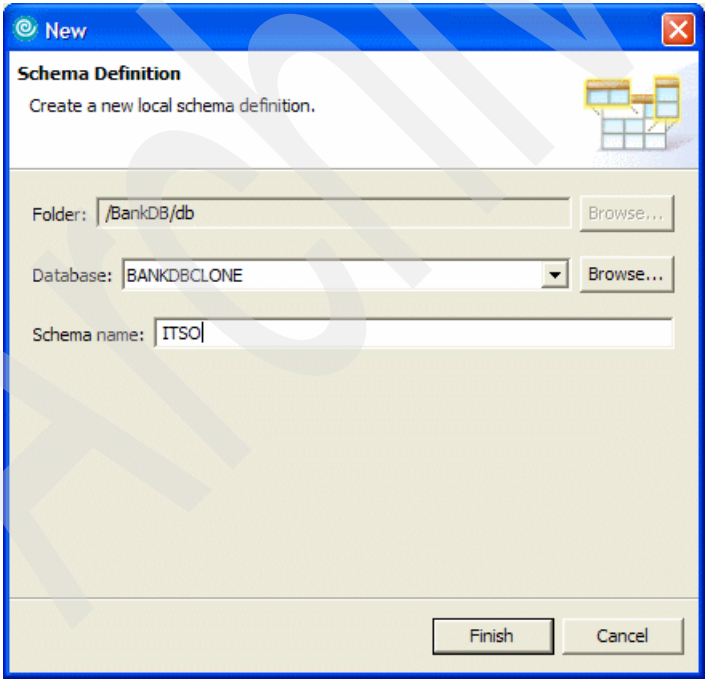


Figure 8-15 Schema Definition wizard - Schema Definition

4. Expand the new schema in the Data Definition view and you will see the following types of objects that can be added to the schema.
   - Tables
   - Views
   - Stored procedures
   - User-defined functions

Note: In IBM Rational Application Developer V6.0, tables and views can be created for databases. The creation of stored procedures and user-defined functions is supported only when the Database Manager is one of the DB2 UDB family of database types.

8.5.4 Create a table

This section describes how to create a new table in a schema. Rational Application Developer provides a wizard for defining table columns as well as primary and foreign keys.

To create a table in a schema, do the following:

1. Select the **ITSO** schema created in 8.5.3, “Create a schema” on page 357.
2. Right-click and select **New** → **Table Definition**.
3. When the New Table Definition dialog appears, enter the Table name (for example, Accounts), as seen in Figure 8-16, and then click **Next**.

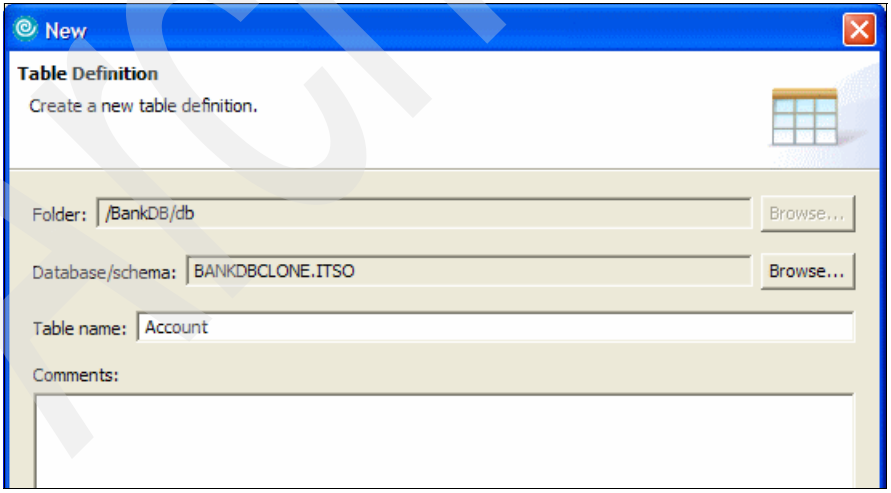


Figure 8-16 Table definition wizard -Table Definition

4. When the Table Columns dialog appears, click **Add Another** to add a column to the table and define the column properties.

   The exact properties available depend on the database manager selected. For more information about the properties available, you can consult the documentation provided by the database vendor.

5. On the next page you define the columns of the table. Figure 8-17 displays the ID column values. Click **Add Another** to add each of the columns listed in Table 8-1 for the ITSO example. When done adding columns click **Next**.

Table 8-1 Accounts table

Column name	Column type	Default	Nullable/Key column	String length
ID	VARCHAR		Key Column (primary key)	250 Note: For VARCHAR type
BALANCE	INTEGER			

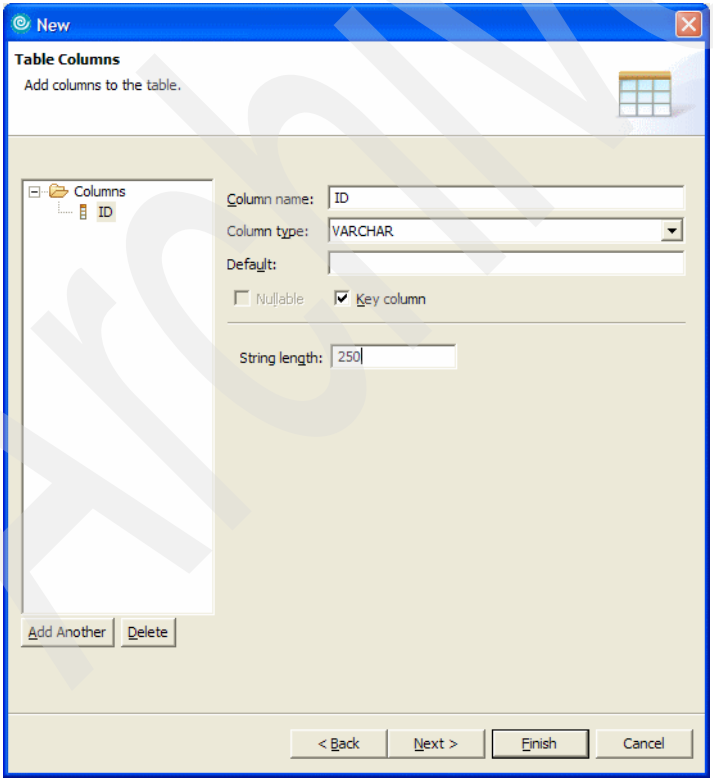


Figure 8-17 Table Definition wizard -Table Columns

Archived

6. When the Primary Key dialog appears (as seen in Figure 8-18), you can modify the primary key (optional) by selecting the items you want from the Source Columns, and add them to the primary key by clicking >.

   In our example, we accepted the default (ID) and clicked **Next**.

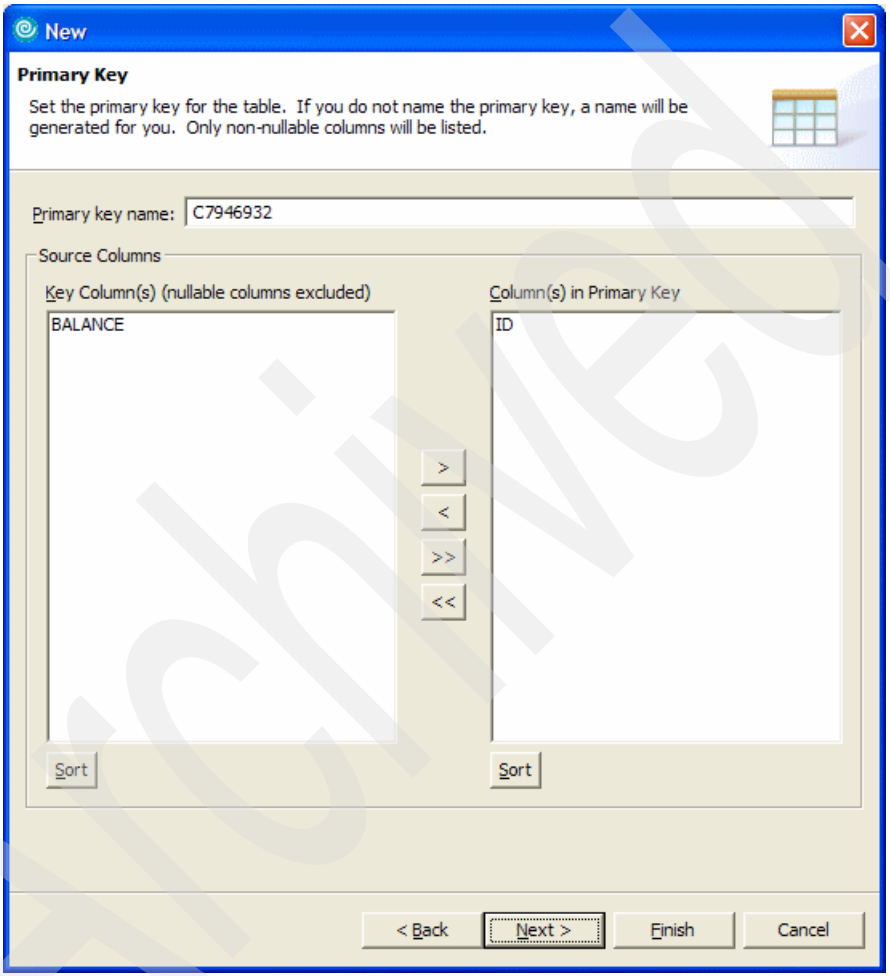


Figure 8-18 Table Definition wizard - Primary Key

7. When the Foreign Key dialog appears, define any foreign key constraints that you want to apply and then click **Finish**.

   In our example, we do not have another table defined so we do not add a foreign key.

Archived

8.5.5 Generate a DDL file

This section describes how to generate a DDL file from the data definition of the database table object (ACCOUNT), which can be used to create the database table (ACCOUNT) at the time of deployment in a database of the type defined (Cloudscape, DB2 UDB, Oracle, etc.).

To generate a DDL file for the table, do the following:

1. From the Data Definition view, select the **ITSO.ACCOUNTS** table.
2. Right-click and select **Generate DDL**.
3. When the Generate SQL DDL dialog appears, enter the DDL file name, select the desired project, specify the options, and then click **Finish**. For example, Figure 8-19 displays the options for the ACCOUT DDL file to be generated to the BankDB project.

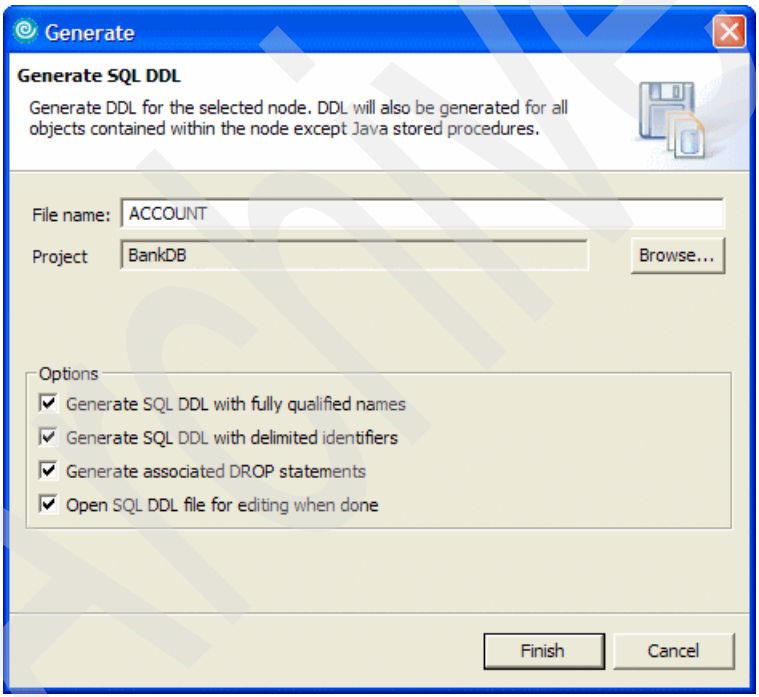


Figure 8-19 Generate SQL DDL wizard - Generate DDL

The options available are to create the DDL with or without the schema name, whether to place delimiters around identifiers, whether to generate DROP statements, and whether to open an editor on the generated file.

Example 8-1 Sample generated ACCOUNT.ddl file

```
DROP TABLE "ITSO"."ACCOUNT" CASCADE;
CREATE TABLE "ITSO"."ACCOUNT"
  ("ID" VARCHAR(250) NOT NULL,
   "BALANCE" INTEGER NOT NULL);
ALTER TABLE "ITSO"."ACCOUNT"
```

The generated DDL file is shown in Example 8-1. You can use the generated DDL to create the table in the database system by deploying the DDL as described in 8.5.6, “Deploy DDL from the workspace to a database” on page 362, or use the appropriate tool provided by the database vendor.

8.5.6 Deploy DDL from the workspace to a database

Once the DDL file exists in the workspace, it can be deployed to a database. For details refer to “Create database tables via Rational Application Developer” on page 350.

8.5.7 Copy database objects from a DDL file to a workspace

In the Database Explorer view you can browse the tables and columns, but before you can actually use them in your application (read only), you will need to copy the database object into a project folder. In the previous section we looked at how we can deploy a DDL file from the workspace to a database. In this section we cover copying existing database objects from a DDL file to the workspace.

To copy database objects from a DDL file to the workspace, do the following:

1. Open the Data perspective Database Explorer view.
2. Select the database for which a connect has been defined (for example, BANK).
3. Right-click and select **Copy to Project**.
4. When the Copy to Project dialog appears, enter the folder you wish to copy the project to and then click **Finish**. For example, we entered /BankDB/db in the Folder field, as seen in Figure 8-20.

Archive

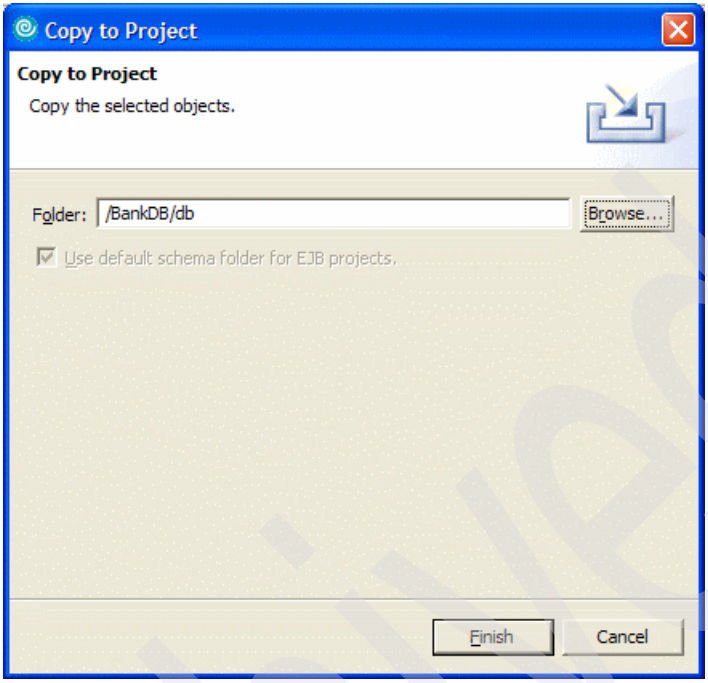


Figure 8-20 Copying database objects - Copy selected objects to project

Now all the schemas and tables under the BankDB database have been created in the BankDB project in the db folder.

5. Verify the database objects.

   To verify that the database objects have been copied over from the database to the project, do the following:

   a. Open the Data Definition view and expand the db folder under BankDB project.

   b. You will see all the database objects including the BANK database, the ITSO schema, and the tables under the schema, as shown in Figure 8-21 on page 364.

Archived

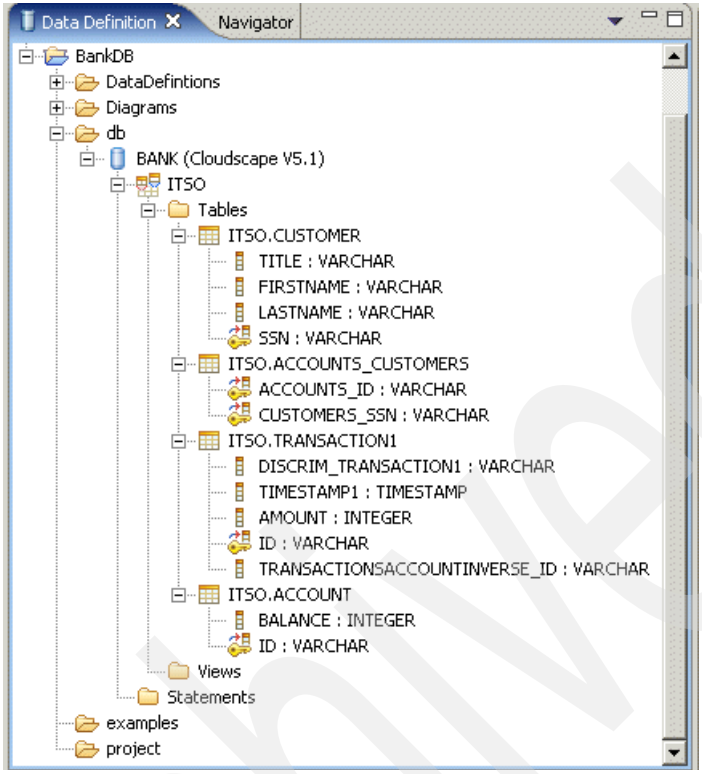


Figure 8-21 Copying database objects - Data Definition view

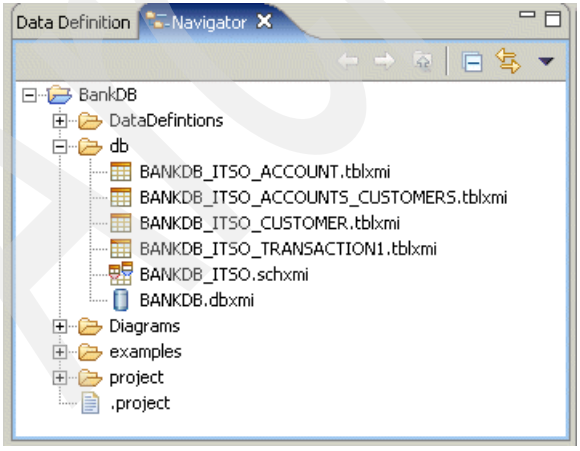


Figure 8-22 Copying database objects - Navigator view

Archived

In the Navigator view you will also notice that a number of XMI format files have been created for the database objects as shown in Figure 8-22 on page 364. XMI is an open information interchange model that allows developers who work with object technology to exchange programming data over the Internet.

If you double-click one of these files, the appropriate database object editor opens. If you want to see the XMI source, you can right-click any of the files and select **Open With** → **Text Editor**.

8.5.8 Generate DDL and XSD files for database objects

Rational Application Developer allows you to generate DDL files and XML schemas for database objects.

Note: To enable generation of XML Schemas options in the context menus, make sure the XML development capability is enabled.

To enable the XML development capability, do the following:

1. Select **Window** → **Preferences**.
2. From the Preferences dialog, expand **Workbench** and select **Capabilities**.
3. In the Capabilities list check the **XML Development** capability under XML Developer.

Generate DDL file for database objects

To generate DDL files for database objects, do the following:

1. Select the database object (for example, BANK) in the Data Definition view.
2. Right-click and select **Generate DDL...** from the context menu.
3. When the Generate SQL DDL wizard appears, enter the file name and project you wish to output the DDL file, and select the desired options. For example, we entered Table for the file name, clicked **Browse** to select the BankDB project, and checked the options displayed in Figure 8-23 on page 366.

Archived

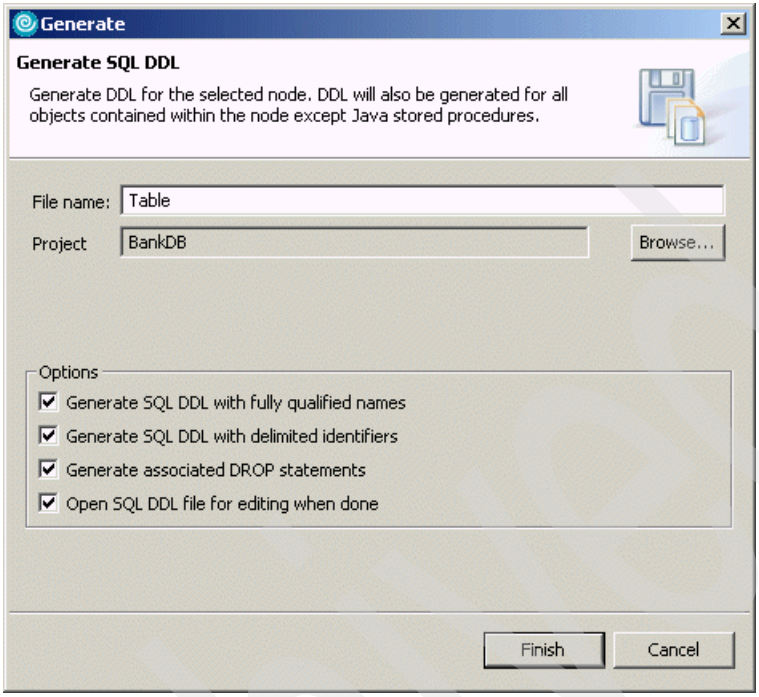


Figure 8-23 Generate DDL wizard - Generate SQL DDL for the BankDB database

The generated DDL file is shown in Example 8-2.

Example 8-2 Sample generated Table.sql file

```
DROP TABLE "ITSO"."CUSTOMER" CASCADE;
DROP TABLE "ITSO"."ACCOUNTS_CUSTOMERS" CASCADE;
DROP TABLE "ITSO"."TRANSACTION1" CASCADE;
DROP TABLE "ITSO"."ACCOUNT" CASCADE;
DROP SCHEMA "ITSO";

CREATE SCHEMA "ITSO";

CREATE TABLE "ITSO"."CUSTOMER"
  ("TITLE" VARCHAR(250) NULL,
   "FIRSTNAME" VARCHAR(250) NULL,
   "LASTNAME" VARCHAR(250) NULL,
   "SSN" VARCHAR(250) NOT NULL);

ALTER TABLE "ITSO"."CUSTOMER"
  ADD CONSTRAINT "PK_CUSTOMER" PRIMARY KEY ("SSN");

CREATE TABLE "ITSO"."ACCOUNTS_CUSTOMERS"
```

```
  ("ACCOUNTS_ID" VARCHAR(250) NOT NULL,
   "CUSTOMERS_SSN" VARCHAR(250) NOT NULL);

ALTER TABLE "ITSO"."ACCOUNTS_CUSTOMERS"
  ADD CONSTRAINT "PK_ACCOUNTS_CUSTO2" PRIMARY KEY ("ACCOUNTS_ID",
"CUSTOMERS_SSN");

CREATE TABLE "ITSO"."TRANSACTION1"
  ("DISCRIM_TRANSACTION1" VARCHAR(32) NOT NULL,
   "TIMESTAMP1" TIMESTAMP NULL,
   "AMOUNT" INTEGER NOT NULL,
   "ID" VARCHAR(250) NOT NULL,
   "TRANSACTIONSACCOUNTINVERSE_ID" VARCHAR(250) NULL);

ALTER TABLE "ITSO"."TRANSACTION1"
  ADD CONSTRAINT "PK_TRANSACTION1" PRIMARY KEY ("ID");

CREATE TABLE "ITSO"."ACCOUNT"
  ("BALANCE" INTEGER NOT NULL,
   "ID" VARCHAR(250) NOT NULL);

ALTER TABLE "ITSO"."ACCOUNT"
  ADD CONSTRAINT "PK_ACCOUNT" PRIMARY KEY ("ID");
```

Generate an XML schema for a table

XML schemas can be generated for tables. To generate an XML schema for a table, you must already have imported it into a folder and be in the Data Definition view.

1. Select the **CUSTOMER** table.
2. Right-click and select **Generate XML Schema** from the context menu.
3. When the Create XML Schema dialog appears, select the directory into which to put the XML schema file, and enter a file name, as seen in Figure 8-24 on page 368.

Archived

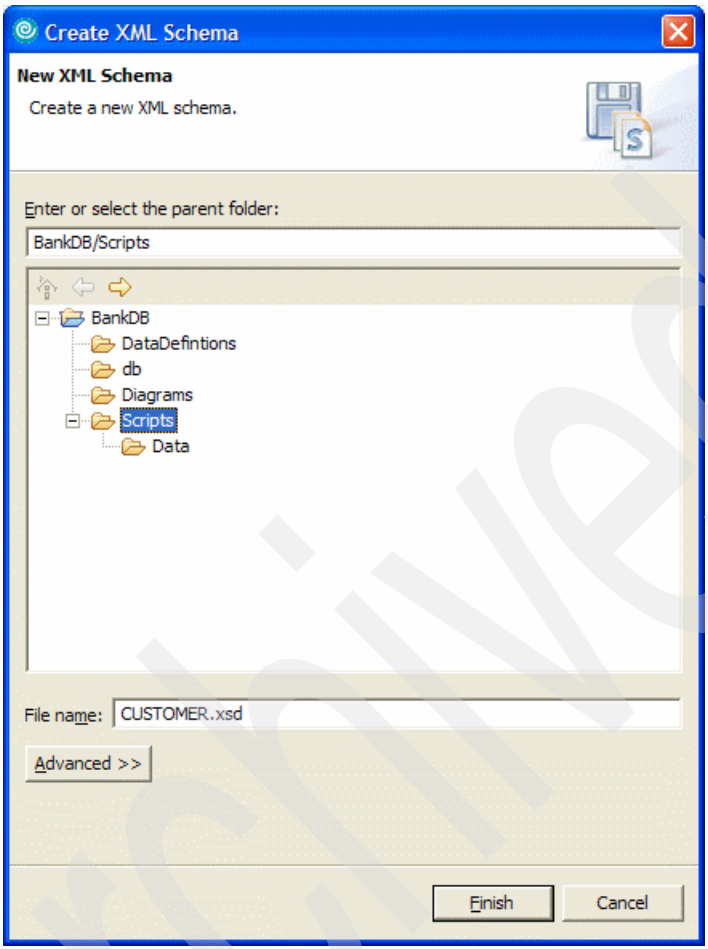


Figure 8-24 Generate XSD wizard - New XML Schema generation

4. Click **Finish** and the schema file (with extension .xsd) is created and opened in the XML schema editor.

   The content of the customer XSD file (visible in the Source tab of the editor) is shown in Example 8-3.

Example 8-3 Generated XML schema

```
<?xml version="1.0" encoding="UTF-8"?>
   <xsd:schema xmlns:BankDBTestITSO="http://www.ibm.com/BankDBTest/ITSO"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
targetNamespace="http://www.ibm.com/BankDBTest/ITSO">
     <xsd:element name="CUSTOMER_TABLE">
       <xsd:complexType>
```

```
      <xsd:sequence>
        <xsd:element maxOccurs="unbounded" minOccurs="0" ref="BankDBTestITSO:CUSTOMER"/>
      </xsd:sequence>
    </xsd:complexType>
    <xsd:key name="PK_CUSTOMER_SSN">
      <xsd:selector xpath="BankDBTestITSO:CUSTOMER"/>
      <xsd:field xpath="SSN"/>
    </xsd:key>
  </xsd:element>
  <xsd:element name="CUSTOMER">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="TITLE">
          <xsd:simpleType>
            <xsd:restriction base="xsd:string">
              <xsd:length value="250"/>
            </xsd:restriction>
          </xsd:simpleType>
        </xsd:element>
        <xsd:element name="FIRSTNAME">
          <xsd:simpleType>
            <xsd:restriction base="xsd:string">
              <xsd:length value="250"/>
            </xsd:restriction>
          </xsd:simpleType>
        </xsd:element>
        <xsd:element name="LASTNAME">
          <xsd:simpleType>
            <xsd:restriction base="xsd:string">
              <xsd:length value="250"/>
            </xsd:restriction>
          </xsd:simpleType>
        </xsd:element>
        <xsd:element name="SSN">
          <xsd:simpleType>
            <xsd:restriction base="xsd:string">
              <xsd:length value="250"/>
            </xsd:restriction>
          </xsd:simpleType>
        </xsd:element>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>
```

Notice the Graph page of the XML schema editor, as shown in Figure 8-25 on page 370. Expand the boxes by clicking the plus icon (**+**).

Archived

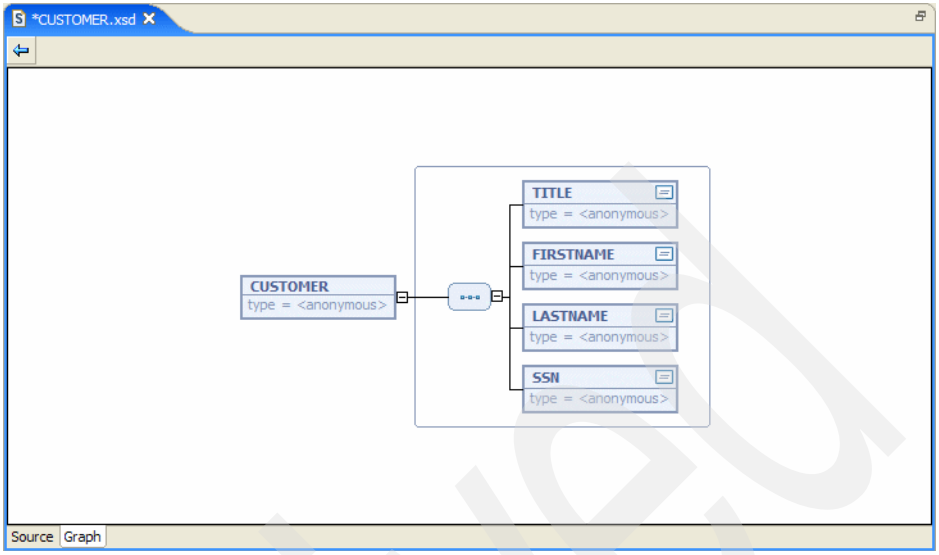


Figure 8-25 XML Schema Editor - Customer table graph

Study the Outline view as well. It shows the structure of the XML schema file. If you want, you can make changes to the XML file and generate a new DDL file by selecting **Generate** → **DDL** from the context menu.

8.6 UML visualization

You can create database objects including databases using either the wizards or UML visualization. In this chapter up until now we described how to create Tables using the wizard. In this section we look at creating database objects using visualization models. Rational Application Developer provides the ability to create class diagrams, IE diagrams and IDEFIX diagrams to support the design and development of database objects.

8.6.1 Class diagrams

To demonstrate Rational Application Developer capabilities we will create a database (BankDBUML), a schema (ITSOUML) and three tables (Customer, Accounts_Customer) using the UML visualization model.

Create a class diagram

To create a class diagram, do the following:

1. Select the **BankDB** project.

2. Create a new folder named diagrams if one does not already exist.
3. Select the **diagrams** folder.
4. Right-click and select **New** → **Class Diagram** from the context menu.
5. When the Create Class Diagram dialog appears, enter `BankDBClassDiagram` for the File Name and click **Finish**.

   A new class diagram is created and is opened in the class diagram editor, as shown in Figure 8-26. The Database Drawer in the palette contains all the database objects that you can create using the class diagram model.

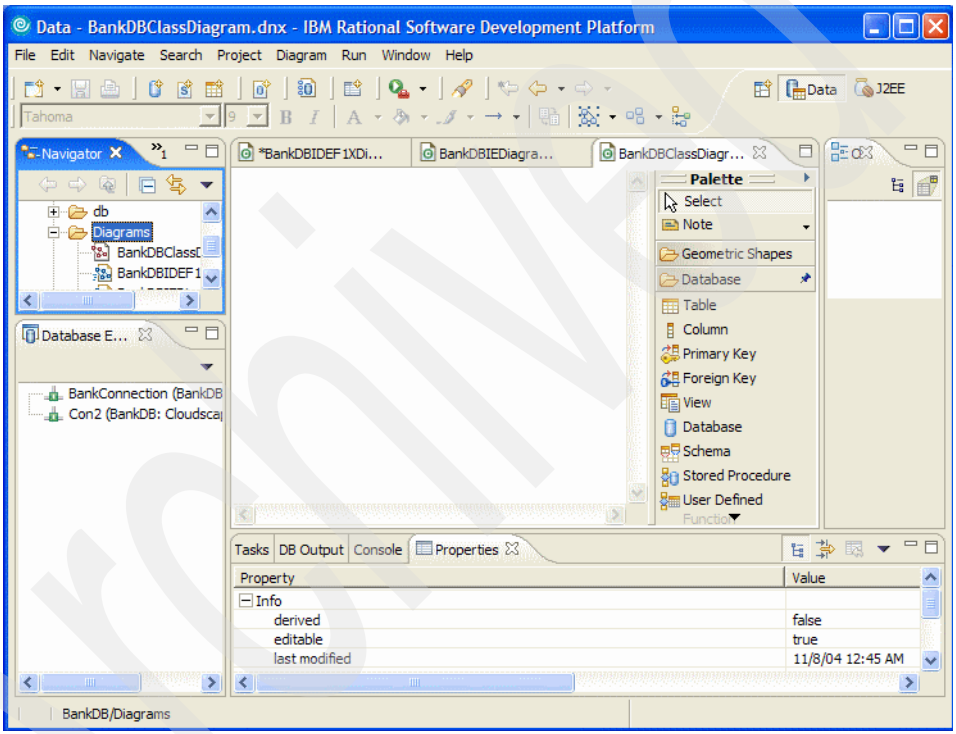


Figure 8-26 UML Visualization - Class diagram

Create a database

To create a database, do the following:

1. Click the **database** element in the Database Drawer in the palette and click anywhere on the class diagram editor.
2. When the Database Definition dialog appears, enter `BankDBUML` for the database name. Ensure that **Cloudscape V5.1** is selected for Database Vendor Type. Click **Finish**. At this point the database is created in the workspace under the db directory of the BankDB project.

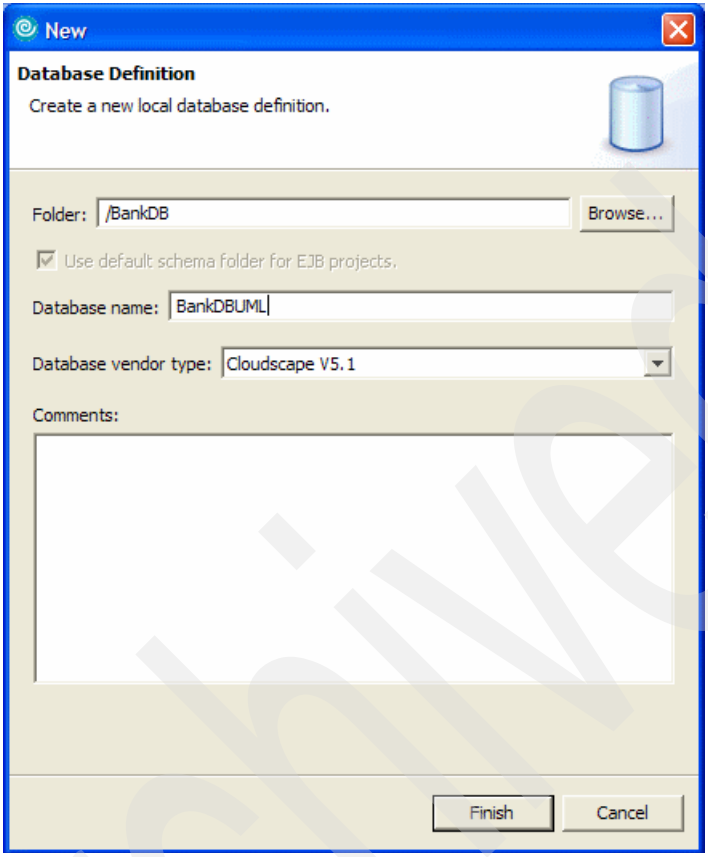


Figure 8-27 UML visualization - Database definition

Create a schema

To create a schema, do the following:

1. Click the **Schema** element in the Database Drawer in the palette.
2. Click the **BankDBUML** to create the schema within the BankDBUML database. Enter `ITSOUML` for the Schema Name, as shown in Figure 8-28 on page 373.
3. Click **Finish**.

Archived

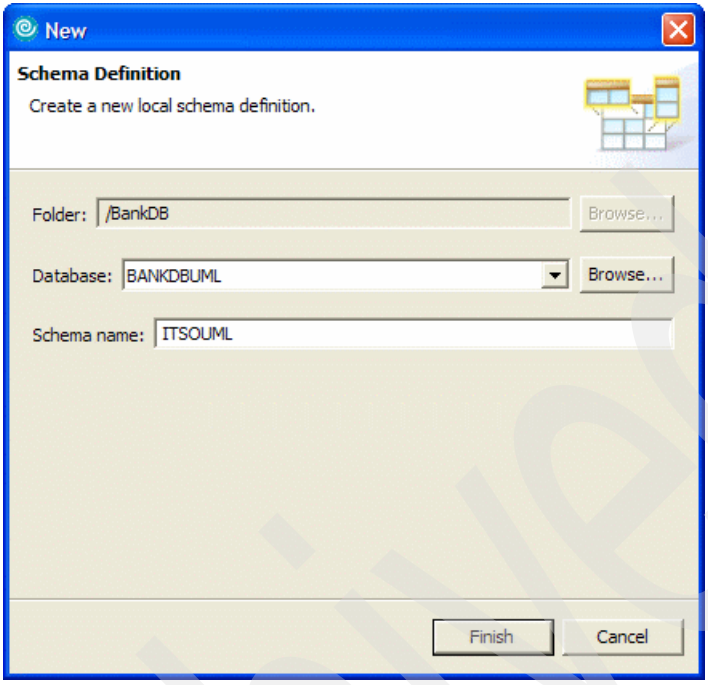


Figure 8-28 UML visualization - Schema Definition

Create two tables (Account, Accounts_Customer)

Create two tables (Account and Accounts_Customer) by clicking the table element in the Database Drawer in the palette and clicking the **ITSOUML** schema to make sure that the tables are created under the ITSOUML schema. Follow the directions in 8.5.4, “Create a table” on page 358, to create the two tables.

The resulting class diagram is shown in Figure 8-29 on page 374.

Archived

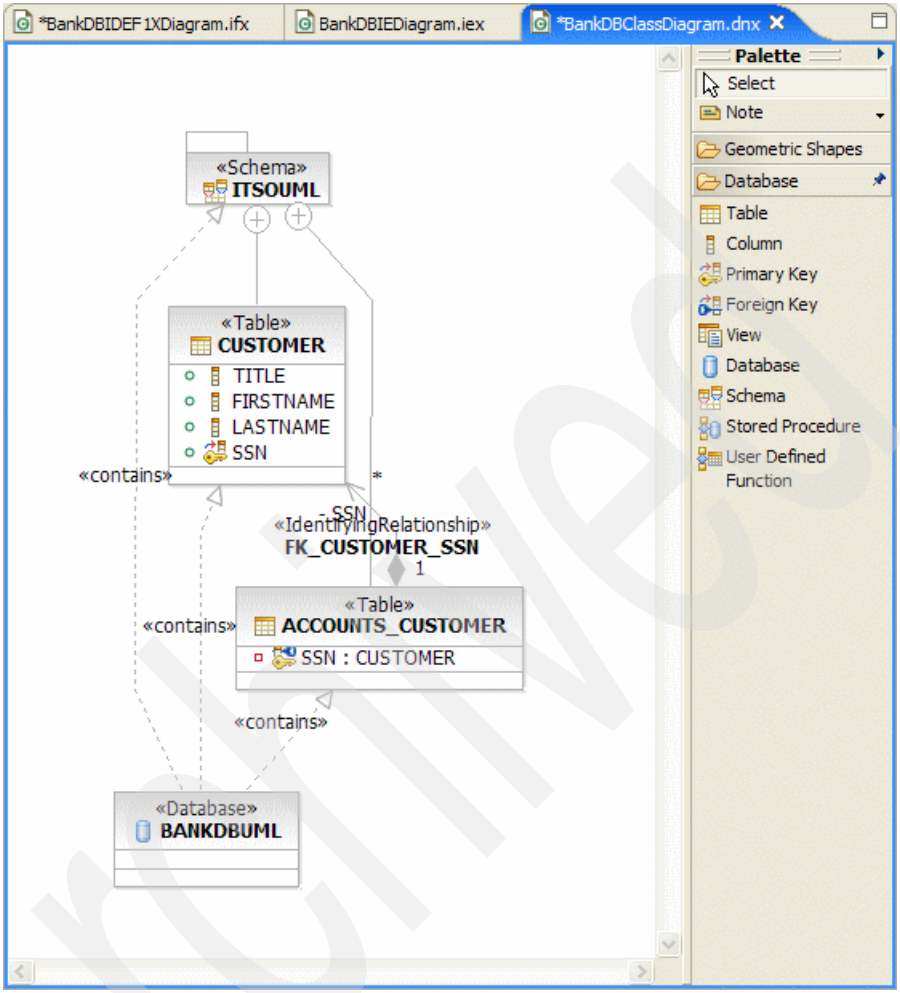


Figure 8-29 UML visualization - Class diagram

At this point the database objects are created in the workspace. If you want to deploy the objects to a database, follow the instructions in 8.5.6, “Deploy DDL from the workspace to a database” on page 362, to deploy the database objects to a local or remote database.

8.6.2 Information engineering (IE) diagrams

Using the ID diagram you can visualize tables, columns, primary keys, and foreign keys. The IE diagram is useful for engineers who are comfortable with the IE notation. These diagrams are stored as .ifx files and can be opened from the Navigator view. You can drag and drop existing database objects from the DB

Explorer or Data Definition view into the IE diagram. Figure 8-30 shows the IE diagram.

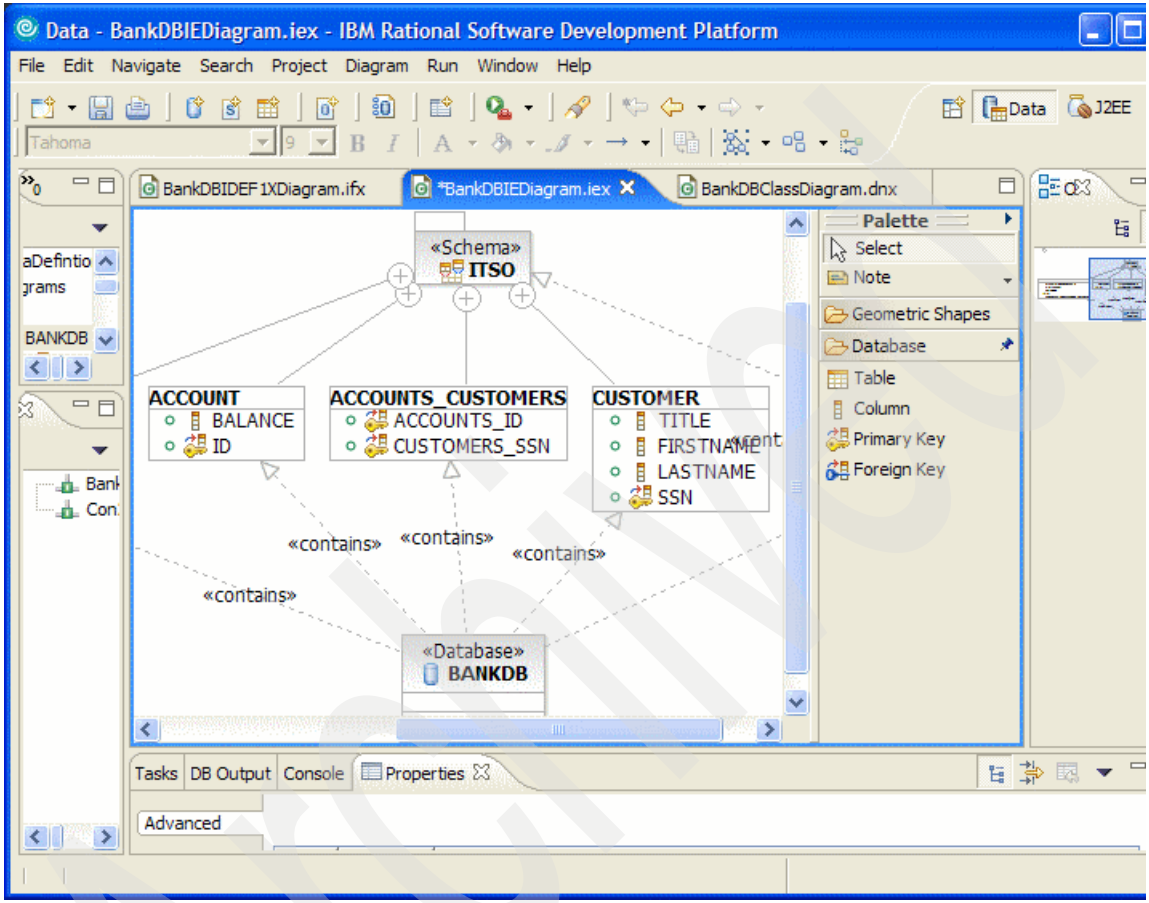


Figure 8-30 UML visualization - IE diagram

8.6.3 IDEF1X (Integrated Definition Extended) diagrams

With Rational Application Developer supports IDEF1X models, you can visualize existing databases, schemas, and other database objects to an IDEF1X model, and you can create tables, columns, and primary and foreign keys directly in an IDEF1X model. IDEF1X models are stored as .ifx files and can be opened and modified from the Navigator view.

8.7 Create SQL statements

There are two alternative methods of creating an SQL statement in Rational Application Developer:

- SQL Statement Wizard - Guided wizard resulting in SQL statements
- SQL Query Builder - Provides an editor for an advanced user

Both tools can be used to build an SQL statement. After using the SQL Statement Wizard, you can use the SQL Query Builder to updates the SQL.

Note: This section requires that you have already created the BANK database and tables, as well as populated the sample data. See 8.2, “Preparing for the sample” on page 337, for details.

8.7.1 Using the SQL Statement wizard

In this section we demonstrate how create an SQL statement using the SQL Statement Wizard.

To create an SQL statement using the SQL Statement Wizard, do the following:

1. Create an sql folder in which to store the scripts.
   a. Select the **BankDB** project in the Navigator view.
   b. Right-click sand elect **New** → **Folder**.
   c. When the New Folder dialog appears, enter `sql` as the folder name and then click **Finish**.
2. Select the **BankDB** project in the Navigator view.
3. Right-click and select **File** → **New** → **Other**.
4. When the Select a Wizard dialog appears, select **Data** → **SQL Statement**, and then click **Next**.
5. When the Specify SQL Statement Information dialog appears, we entered the following, as seen in Figure 8-31 on page 377, and clicked **Next**:
   – SQL statement: Select **SELECT** (default).

     How would you like to create your SQL statement? Select **Be guided through creating an SQL statement** (default).
   – Uncheck **Create a new database connection**.

Archive

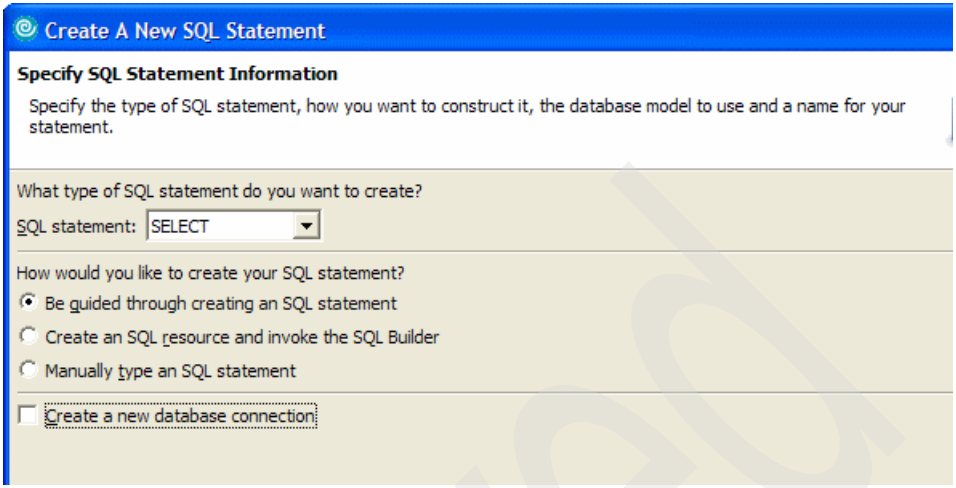


Figure 8-31 SQL Statement wizard - Specify SQL Statement Information

6. When the Choose and Existing Database Model dialog appears, we clicked **Browse**, selected **BankDB** → **db** → **BANK**, and the clicked **OK**. Click **Next**.
7. When the Database and Statement Location dialog appears, we entered `Select1` in the SQL statement name field and then clicked **Next**.

Select tables and columns

Complete the following steps to add tables and columns from the SQL Statement Wizard - Construct an SQL Statement dialog.

1. Select the tables to include in the SQL statement.
   a. Click the **Tables** tab.
   b. Click the **>** button from the list of available tables to add to the list of selected tables.
   c. When done click **Next**.

   In our example, we added the following tables, as seen in Figure 8-32 on page 378:

   – ITSO.ACCOUNT
   – ITSO.ACCOUNTS_CUSTOMERS
   – ITSO.CUSTOMER

Archived

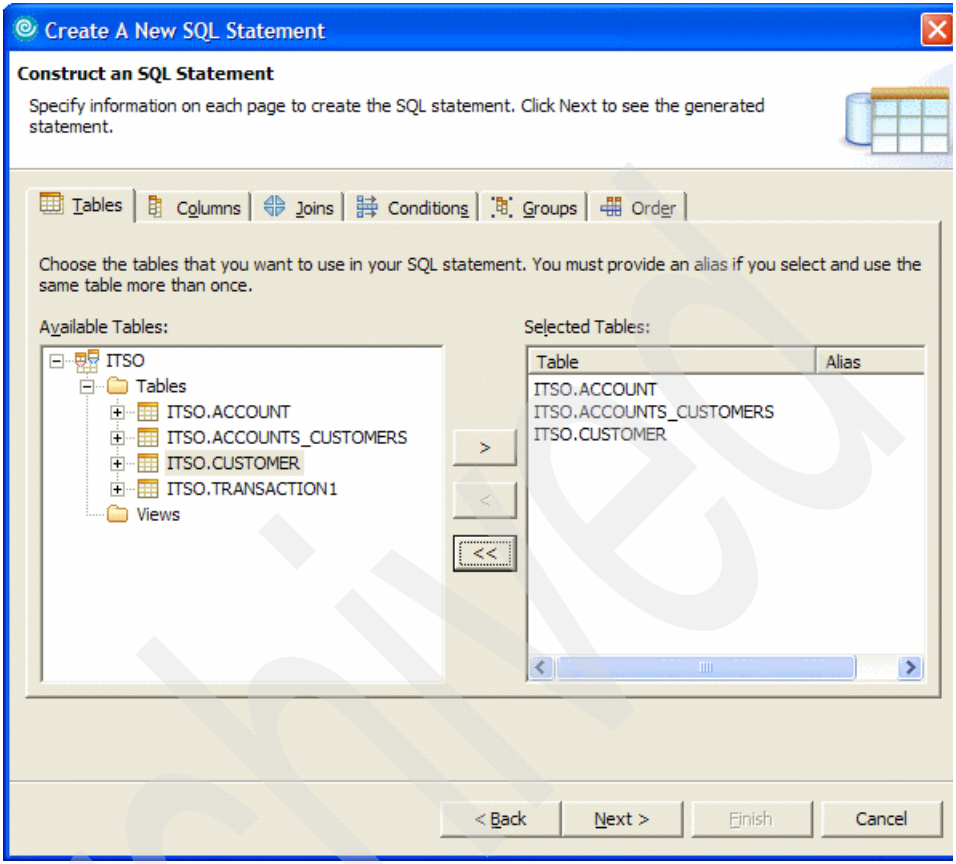


Figure 8-32 SQL Statement wizard - Construct an SQL Statement - Table selection

2. Select the columns to include in SQL statement.
   a. Click the **Columns** tab.
   b. Expand the desired table and select the desired columns by clicking the **>** button to add columns from the Available Columns list to the Selected Columns list. Order the output columns using the Move Up and Move Down buttons. When done click **Next**.

      In our example, we selected the columns listed in Table 8-2.

Table 8-2 Selected columns for each table

Table	Columns
ITSO.CUSTOMER	TITLE, FIRSTNAME, LASTNAME, SSN
ITSO.ACCOUNT	ID, BALANCE

Archive

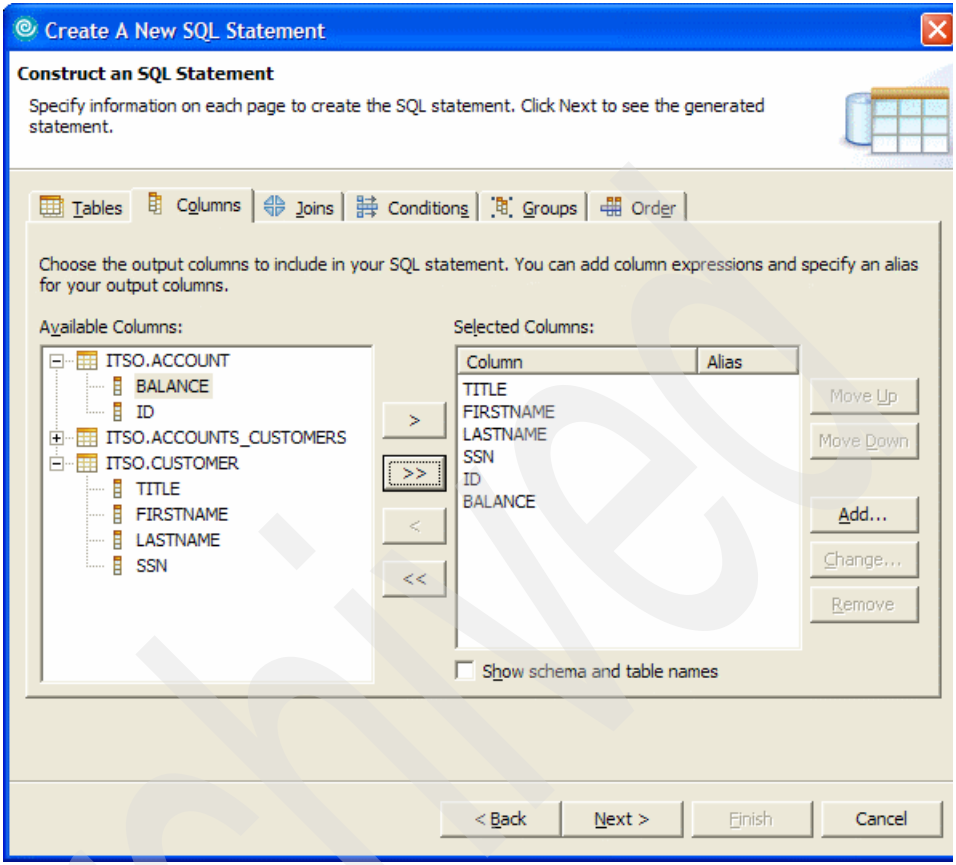


Figure 8-33 SQL Statement wizard - Construct an SQL Statement - Choose Columns

Define table joins

Complete the following steps to add tables and columns from the SQL Statement Wizard - Construct an SQL Statement dialog.

1. Define the columns to join between the tables.

   This is done by selecting the column from one table and dragging it to the corresponding column of the other table.

   In our example, we did the following, as seen in Figure 8-34 on page 380, and then clicked **Next**:

   a. Click the **Joins** tab.

   b. We linked ITSO.CUSTOMER.SSN to ITSO.ACCOUNTS_CUSTOMERS.SSN and ITSO.ACCOUNT.ID to

Archive

ITSO.ACCOUNTS_CUSTOMER.ID. When the joins are complete, connection symbols are displayed, as shown in Figure 8-34.

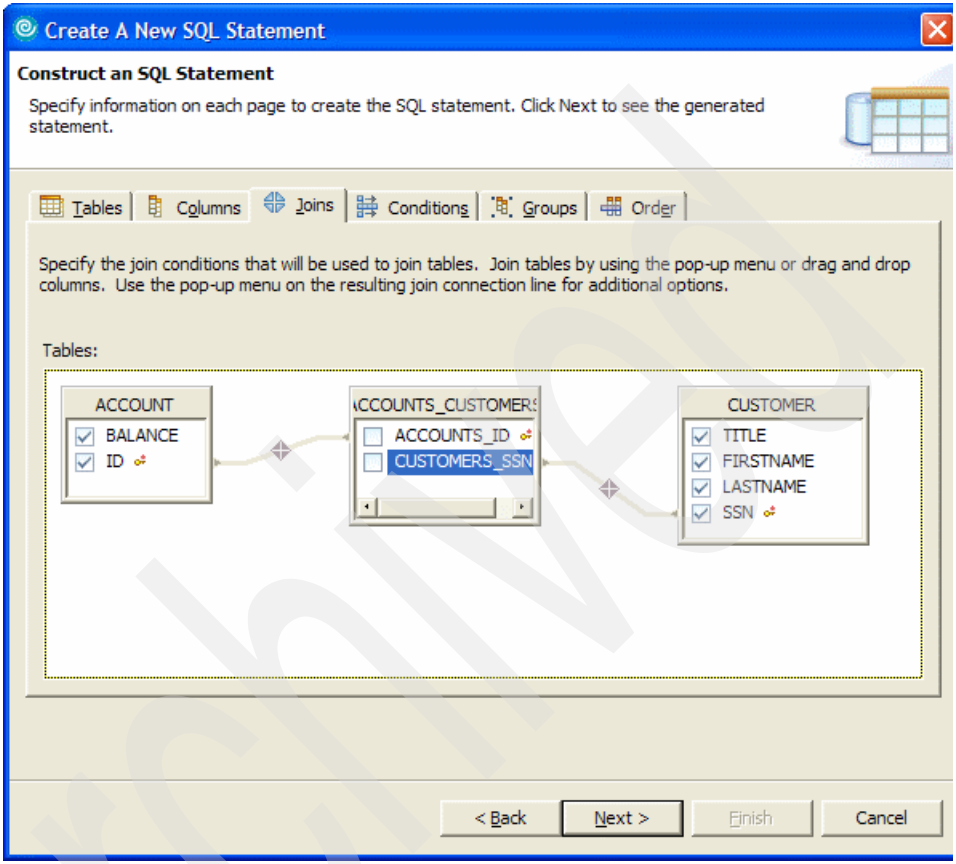


Figure 8-34 SQL Statement wizard - Construct an SQL Statement - Create joins

Tip: You can rearrange the tables by dragging them on the pane. You can enlarge a table by dragging the sides. You can also select the columns in this dialog step, or make changes to the selection from the previous step.

Define the conditions for the WHERE clause

The Conditions tab is used to define the restrictions on the SELECT statement. Each condition is added to the WHERE clause, as shown in Figure 8-35 on page 381.

1. Define the conditions for the WHERE clause.

Enter the appropriate values in the Column, Operator, Value, and And/Or fields of the SQL Statement wizard to define the conditions of the WHERE clause. Enter the value by typing in the field. When done click **Next**.

In our example, we did the following, as seen in Figure 8-35:

a. Click the **Conditions** tab.

b. Click in the cell for Column and then selected **CUSTOMER.FIRSTNAME**.

c. Click in the cell for Operator and then selected **LIKE**.

d. Click in the cell for Value and select **firstNameVariable** (variable defined in next section).

Tip: If you have to enter more than one condition, you must put in the `AND` or the `OR` element before the next row in the table becomes editable.

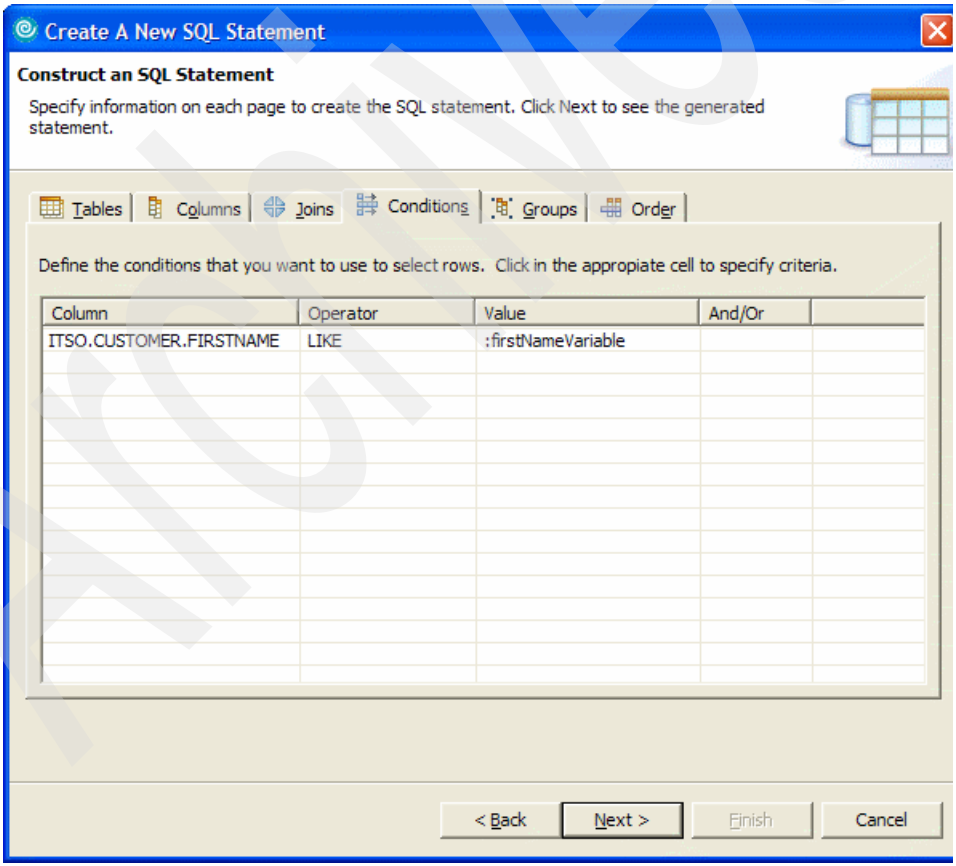


Figure 8-35 SQL Statement wizard - Construct an SQL Statement - Create conditions

Use a variable

The host variable can be used in an SQL statement to represent a value that will be substituted at execution time. For example, you might not want to hardcode the first name if it contains the letter j, but instead leave it as a host variable. Do not enter `'%j%'` in the Value column, rather enter a variable as `:firstNameVariable`.

Groups and order

On the next two tabs you can enter information regarding grouping (GROUP BY) and sorting of rows (ORDER BY).

View SQL statement

Once you have finished building the statement you can click **Next** to see the generated SQL statement, as shown in Figure 8-36.

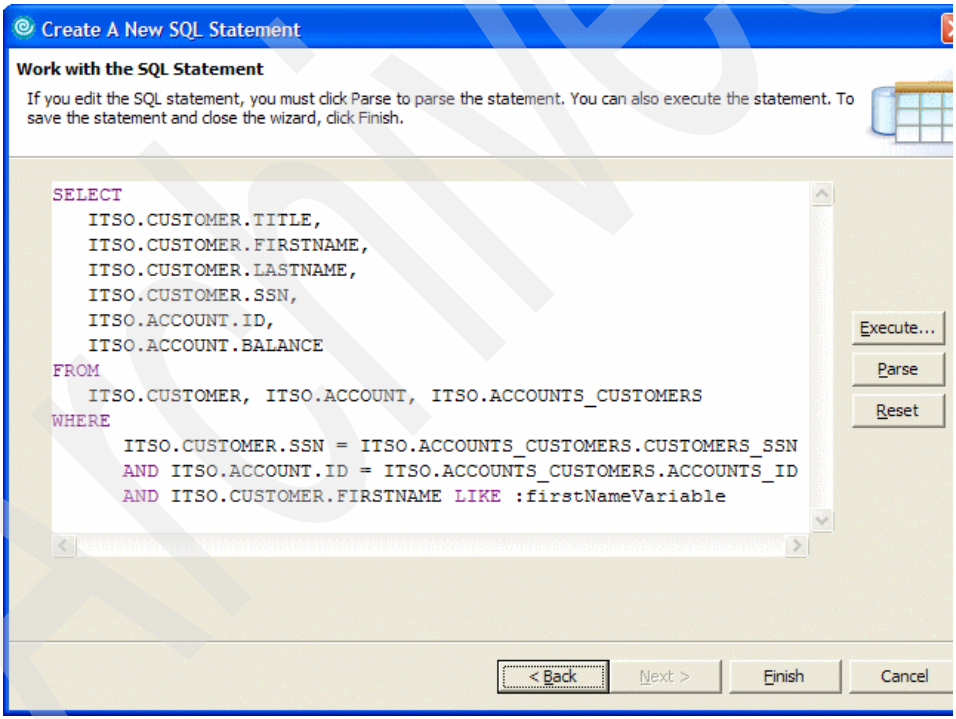


Figure 8-36 SQL Statement wizard - Construct an SQL Statement - Generated SQL

Parse the statement

If you want, you can edit the statement directly. When you are finished editing, you can click **Parse** to validate that the SQL statement is correct.

Archived

Execute an SQL statement

To execute the SQL statement, do the following:

1. Click **Execute**.
2. Click **Execute** again in the next window.
3. When prompted for the host variable, enter ‘J%’ (as seen in Figure 8-37), and click **Finish**.

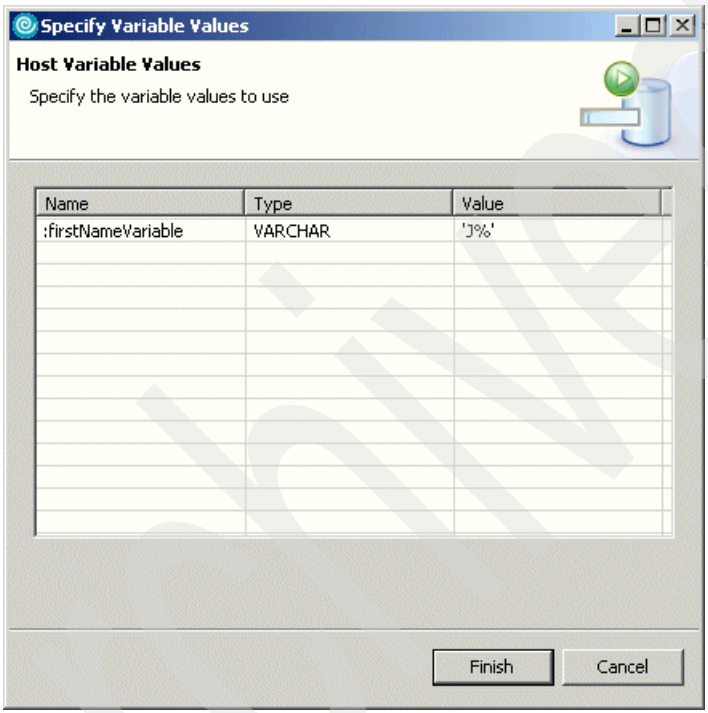


Figure 8-37 SQL Statement wizard - Execute - Enter the host variable value

The statement is executed and the results are displayed, as shown in Figure 8-38 on page 384.

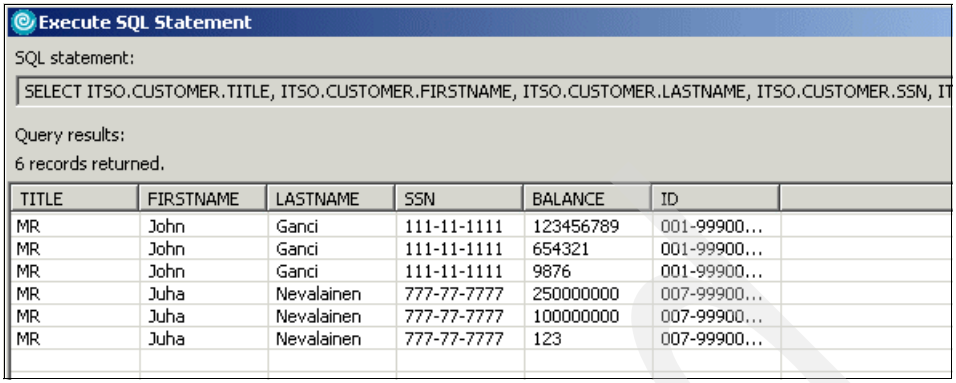


Figure 8-38 SQL Statement wizard - Construct an SQL Statement - SQL execution

4. Select **Close** to close the Execute SQL Statement window.
5. Select **Finish** to save the SQL Statement.
6. The SQL statement is opened in the SQL Query Builder editor. Close the editor.

The SQL statement appears as `BANK_Select1.sqx` in the Navigator view, and as `Select1` in the Data Definition view.

8.7.2 Using the SQL Query Builder

The other way of creating SQL statements in Rational Application Developer is to use the SQL Query Builder. This tool supports all the options of the SQL Statement Wizard, with the addition of WITH and FULLSELECT.

In this section we describe how to use the SQL Query Builder to build a similar SELECT statement as we did using the SQL Statement Wizard. We develop a SELECT statement against the BANK database. We would like to select all customers and their account balances whose first name is a variable.

To start the SQL Query Builder, do the following:

1. Expand the **BankDB** → **db** → **Statements** folder in the Data Definition view.
2. Right-click and select **New** → **Select Statement**.
3. When the New Select Statement dialog appears, enter the name of the statement to be displayed (for example, SelectCustomers) and click **OK**.

   The SQL Query Builder editor is displayed as shown in Figure 8-39 on page 385.

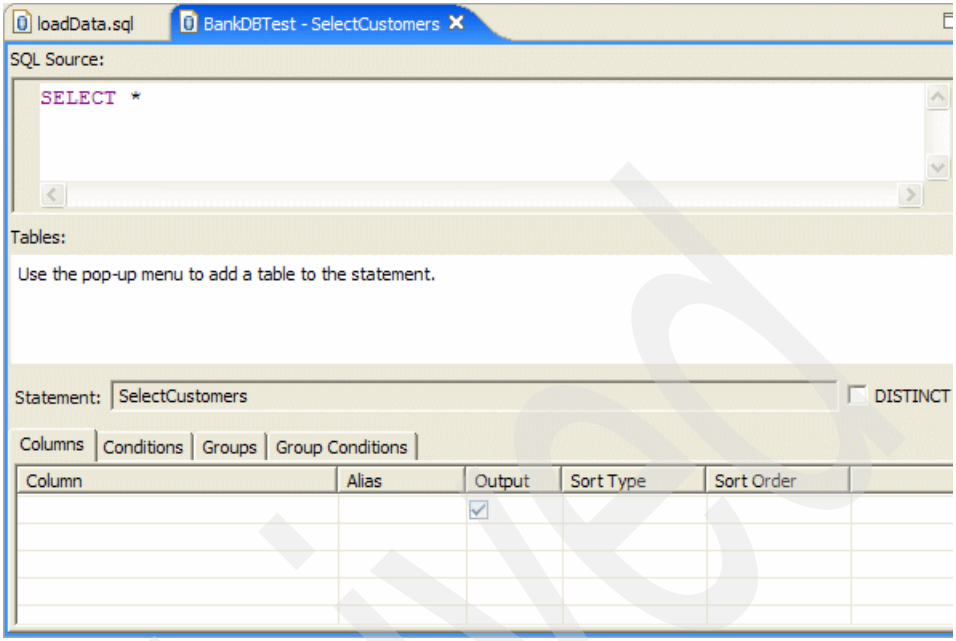


Figure 8-39 SQL Query Builder - Select statement

Define tables and columns

To define tables and columns:

1. Add the tables to the SQL statement.

   To add them, simply drag them from the Navigator or Data Definition view and drop them in the middle pane of the SQL Query Builder screen.

   In our example, we added the ITSO.CUSTOMER, ITSO.ACCOUNT, and ITSO.ACCOUNTS_CUSTOMER tables.

   The result is shown in Figure 8-40 on page 386. As you can see, the tables have been added to the SELECT statement in the top pane.

Archived

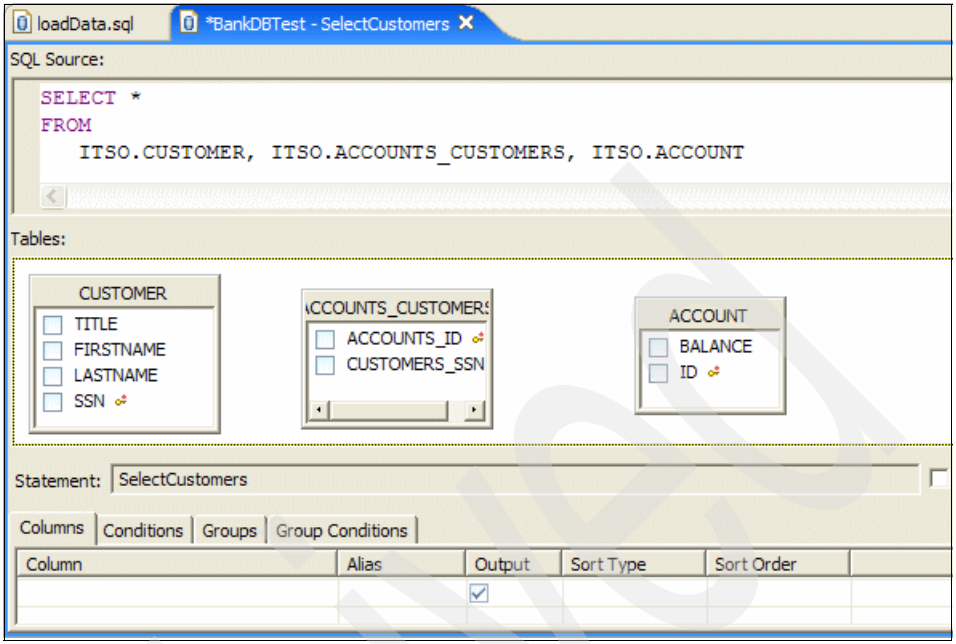


Figure 8-40 SQL Query Builder - Select tables

2. Add columns to the SQL statement.

   To select a column, check the box next to its name. For both the CUSTOMER and ACCOUNT tables, select **all columns**. Do not select any columns of the ACCOUNTS_CUSTOMER table (they are duplicates anyway). As you select the columns, the SELECT statement is updated in the top pane and the columns are added in the bottom pane.

Define table joins

Define the table joins. To define the tables to join, do the following:

1. Select the **CUSTOMER.SSN** column in the CUSTOMER table and drag it across to the corresponding column in the ACCOUNTS_CUSTOMER table.
2. Select the **ID** column in the ACCOUNTS_CUSTOMER table and drag it across to the corresponding column in the ACCOUNT table.

   A link symbol is shown between the tables, and the SELECT statement is updated with the corresponding WHERE clauses, as shown in Figure 8-41 on page 387.

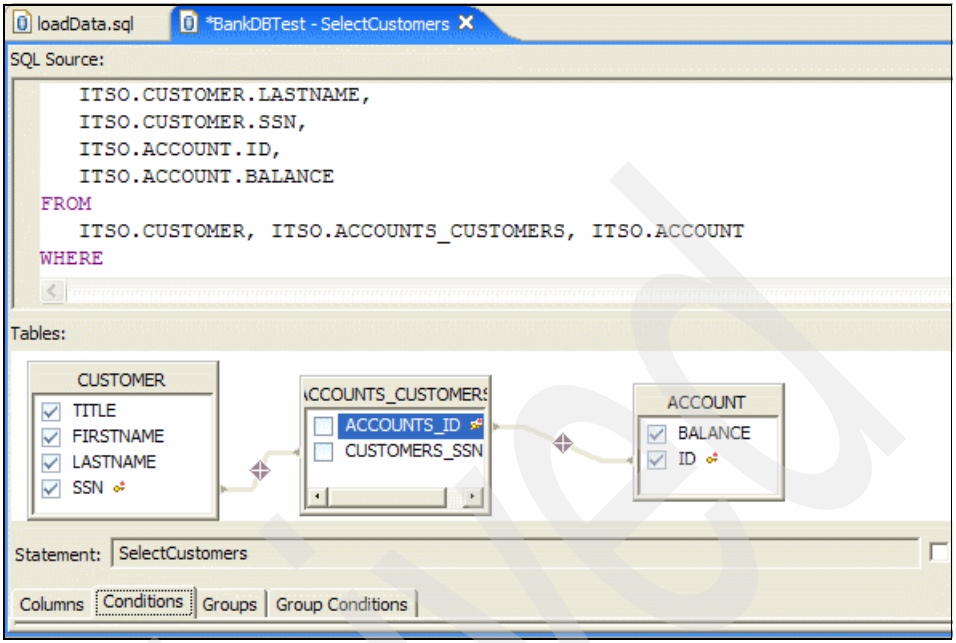


Figure 8-41 SQL Query Builder - Creating joins

Define conditions

Define the conditions for the SQL statement.

1. Add a condition that specifies the CUSTOMER.FIRSTNAME LIKE :firstNameVariable clause.
2. Use the Conditions tab in the bottom pane to add the conditions using the drop-down menus, or type them directly into the SQL statement, and the Conditions tab will be updated, as shown in Figure 8-42.

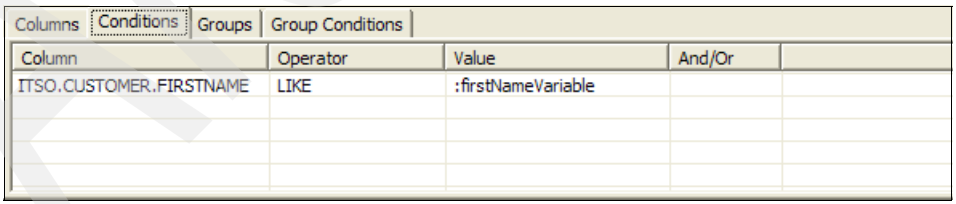


Figure 8-42 SQL Query Builder - Creating conditions

3. The complete statement is created and the workspace is as shown in Figure 8-43 on page 388. Save the statement.

Archived

4. You are prompted for the host variables; just click **Cancel** to dismiss the dialog.

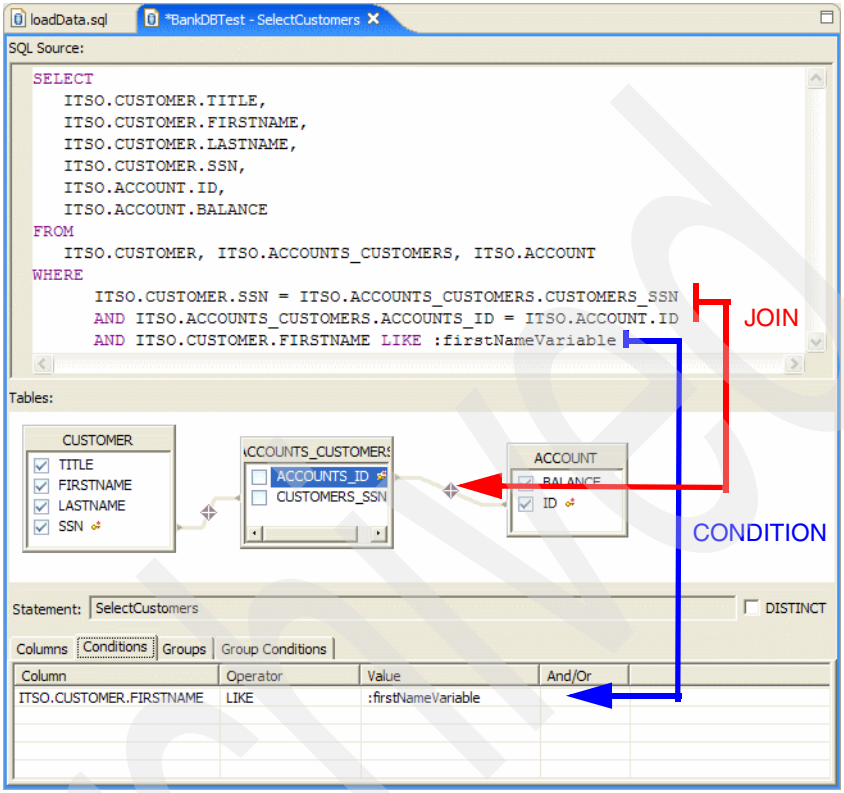


Figure 8-43 SQL Query Builder - SQL select statement

Execute the SQL statement

To execute the SQL statement:

1. To test the statement, select it in the Statements folder and select **Execute** from the context menu or select **SQL** → **Execute** in the menu bar.
2. Enter ‘J%’ as the value for the :firstNameVariable variable in the Host Variable Values window.
3. Click **Finish** to execute the query.

8.8 Access a database from a Java application

This section demonstrates by example how to access a database from a Java application.

Note: For information on accessing databases from Web and J2EE applications, refer to the following chapters:

- Chapter 11, “Develop Web applications using JSPs and servlets” on page 499
- Chapter 13, “Develop Web applications using JSF and SDO” on page 673
- Chapter 15, “Develop Web applications using EJBs” on page 827
- Chapter 16, “Develop J2EE application clients” on page 925

To demonstrate how to access a database from a Java application, we will create a simple Java class named CustomerList, which will perform an SQL query to the BANK database and display a listing of customers. We will leverage the Java code created in Chapter 7, “Develop Java applications” on page 221, as a starting point for this sample. Our sample will describe how to access the database using the DriverManager and data source.

This section is organized as follows:

- Prepare for the sample.
- Access the database using the DriverManager.
- Access using a data source.

8.8.1 Prepare for the sample

This section describes the steps required to prepare for the sample to access a database from a Java application.

1. If you have not already done so, import the BankJava.zip project interchange file, which is described in detail in Chapter 7, “Develop Java applications” on page 221.

   For details on importing the BankJava.zip project interchange file refer to Appendix B, “Additional material” on page 1395.

2. If you have not already done so, create the BANK database.

   For details refer to 8.2.2, “Set up the BANK sample database” on page 338.

3. Add the JDBC driver for Cloudscape to the BankJava project.

   a. Select the **BankJava** project in the Package Explorer view of the Java perspective.

b. Right-click and select **Properties** → **Java Build Path**.

c. Click the **Libraries** tab at the top of the dialog and click **Add Variable...**.

A further dialog appears, allowing you to select from a list of predefined variables. By default, there is no variable defined for the JAR file we need, so we will have to create one.

d. Click **Configure Variables...** and in the resulting dialog click **New...**.

e. Enter `CLOUDSCAPE_DRIVER_JAR` in the Name field and click **File...**.

f. Find the appropriate JAR file, which is in <rad_home>\runtimes\base_v6\cloudscape\lib and is called db2j.jar.

g. Click **Open**, **OK**, and **OK** and you will be back at the New Variable Classpath Entry dialog.

h. Select the **CLOUDSCAPE_DRIVER_JAR** variable you just created and click **OK**.

4. Import Java source into itso.bank.client.

a. Select the **itso.bank.client** package.

b. Right-click and select **Import** → **File system** and then click **Next**.

c. When the File System dialog appears, enter `c:\6449code\database` in the From directory field, and check BankClientCustomerList.java.

d. Click **Finish**.

5. Run the BankClientCustomerList.

8.8.2 Access the database using the DriverManager

This section describes how to use the driver manager class when using JDBC 1.x to manage the connection to the database.

Note: The BankClientCustomerList.java contains a completed sample to access the database using the DriverManager approach.

This section is organized as follows:

- Load JDBC driver and connect to the database.
- JDBC drivers.
- Execute SQL statements.

Load JDBC driver and connect to the database

The first task is to load the JDBC driver. The Class.forName() call loads the JDBC driver, as seen in Example 8-4 on page 391 (JDBC driver for Cloudscape).

The JDBC driver name is dependent on which database type (Cloudscape, DB2 UDB, Oracle, etc.) you are connecting to.

Example 8-4 Load JDBC driver and connect to the database

```
protected static Connection connect() {
   Connection con = null;
   try {
      Class.forName("com.ibm.db2j.jdbc.DB2jDriver");
      con = DriverManager.getConnection(
         "jdbc:db2j:C:/databases/BANK");
   } catch(Exception e) {...}
   return con;
}
```

After loading the JDBC driver, we need to establish a connection to the database. The class that handles the connection is called the DriverManager.

The URL string that is passed in to the getConnection method is dependent on which Database Manager you are using. In the example listed in Example 8-4, we are connecting to a Cloudscape database called BANK.

In this example we are not passing a user ID and password, but if that was required, they would be the second and third parameters of the getConnection call (needed for DB2 UDB and Oracle).

Note: You do not have to create an instance of the driver or register it. This is done automatically for you by the DriverManager class.

JDBC drivers

In this section we discuss the JDBC drivers provided with Cloudscape and DB2 UDB. If you are using a database other than Cloudscape and DB2 UDB, refer to the Rational Application Developer online help and the documentation provided with the database for the driver class and client usage information.

Cloudscape JDBC driver

Cloudscape provides the com.ibm.db2j.jdbc.DB2jDriver JDBC driver. This is a JDBC type 4 driver for an embedded Cloudscape V5.1 database that clients use to connect to the database that runs within the JVM (embedded).

The database URL has the format:

`jdbc:db2j:`*`database`*

If the database is within the Cloudscape system directory, no path information is needed; if the database is external to the system directory, as is the case in our

example, you will need to specify the entire path (or relative path if part of the path is in the classpath) as used in the getConnection call.

DB2 UDB JDBC drivers

DB2 UDB includes two JDBC drivers:

- COM.ibm.db2.jdbc.app.DB2Driver

  This is a JDBC type 2 driver that uses a DB2 client installed on the machine where the application runs. You would use this driver when accessing a local database or a remote database through a local DB2 client.

  The database URL has the format:

  `jdbc:db2:`*`databasename`*

- COM.ibm.db2.jdbc.net.DB2Driver

  This is a JDBC type 3 driver. It is a Java driver that is designed to enable Java applets access to DB2 data sources. Using this driver your application will talk to another machine where the DB2 client is installed.

  The database URL has the format:

  `jdbc:db2://`*`hostname`*`:`*`port`*`/`*`databasename`*

  The standard port of the DB2 JDBC applet server service is 6789. This DB2 service must be started in your machine.

  To connect to a database you have to supply user ID and password:

```
con = DriverManager.getConnection("jdbc:db2://localhost:6789/BANK",
                                  "db2admin","password");
```

  The classes required when connecting to a DB2 database from Java are found in .\sqllib\java\db2java.zip. You would make this available in Rational Application Developer by creating a classpath variable for it and adding that to the project build path.

Execute SQL statements

Once you have loaded the JDBC driver and established a connection, you are now ready to perform operations on the database. Example 8-5 includes a basic SQL statement that will retrieve all customers from the CUSTOMER table.

Example 8-5 Execute SQL statement

```
stmt = con.createStatement();
ResultSet rs = stmt.executeQuery("SELECT * FROM ITSO.CUSTOMER");
```

You create a statement using the connection obtained from the DriverManager and then you execute the query passing the select statement. The result set from the query is returned as a ResultSet object.

Next, you have to process the result set from the query. The ResultSet class provides a number of get methods for various data types, as shown in Example 8-6.

Example 8-6 Process the result set from the SQL query

```
while (rs.next()) {
   String title = rs.getString("title");
   String firstName = rs.getString("firstName");
   String lastName = rs.getString("lastName");
   String ssn = rs.getString("SSN");
   System.out.println(title + " " + firstName + " " + lastName + " " + ssn);
}
```

Finally, JDBC objects must be closed to release the resources and keep the database in a healthy state, as shown in Example 8-7. The best place is a *finally* clause that is executed even in the case of exceptions.

Example 8-7 Release the JDBC resources

```
} finally {
   try { if (rs != null) rs.close(); }
      catch (SQLException e) {e.printStackTrace();}
   try { if (stmt != null) stmt.close(); }
      catch (SQLException e) {e.printStackTrace();}
   try { if (con  != null) con.close(); }
      catch (SQLException e) {e.printStackTrace();}
}
```

8.8.3 Access using a data source

JDBC access using a data source is not well suited for stand-alone applications. It is, however, the preferred way to access databases from Web applications where multiple clients use the same servlet for database access.

Note: The following chapters include examples for using the data source approach:

- Chapter 11, “Develop Web applications using JSPs and servlets” on page 499
- Chapter 15, “Develop Web applications using EJBs” on page 827

When using a data source with WebSphere Application Server, you configure the data source in the Administration Console. For information on how to configure a data source, refer to 23.4.1, “Configure the data source in WebSphere Application Server” on page 1225.

Example 8-8 shows the basic code sequence to get a connection through a data source in the Java application.

Example 8-8 Get a connection through a data source

```
try {
   javax.naming.InitialContext ctx = new javax.naming.InitialContext();
   javax.sql.DataSource ds = (javax.sql.DataSource)
                                ctx.lookup("jdbc/bankDS");
   con = ds.getConnection();
} catch (javax.naming.NamingException e) {
   System.err.println("Naming-Exception: " + e.getMessage());
} catch (java.sql.SQLException e) {
   System.err.println("SQL-Exception: " + e.getMessage());
}
```

The data source is retrieved using a the lookup method of the InitialContext. The data source must be registered in the JNDI server. In our example we use a JNDI name of jdbc/bankDS, which points to a connection pool that in turn contains connections to the BankDB database. Once a connection is obtained, the rest of the code is the same.

Retrieving the data source is expensive. Good coding practice for Web applications is to retrieve the data source only once in the init method of a servlet, and to get and release a connection in the doGet or doPost method for each client request.

8.9 Java stored procedures

A stored procedure is a block of procedural constructs and embedded SQL statements that are stored in a database and can be called by name. Stored procedures allow an application program to be run in two parts, one on the client and the other on the server. One client-server call can produce several accesses to the database. This is good for performance because the traffic between the client and the server is minimized.

Stored procedures can be written as SQL procedures, or as C, COBOL, PL/I, or Java programs. In this section we look at how to write and use Java stored procedures.

Note: IBM Rational Application Developer V6.0 only supports creating SQL Stored Procedures, Java SQL Stored Procedures, SQL user-defined functions, and WebSphere MQ user-defined functions when the database type is IBM DB2 UDB.

This section is organized as follows:

- Prepare for the sample.
- Create a Java stored procedure.
- Build a stored procedure (deploy to database).
- Java DriverManager access to a Java stored procedure.
- JavaBean access to Java stored procedure.

8.9.1 Prepare for the sample

Complete the following steps to prepare the environment for the Java store procedures example:

1. Install DB2 UDB.
2. Create a DB2 UDB database.
3. Create a connection.
4. Create the database tables.
5. Populate the database tables.
6. Create a Web Project.
7. Copy the database to the Web Project.

Install DB2 UDB

Java stored procedures require that IBM DB2 Universal Database be installed.

For details on installing DB2 UDB refer to “IBM DB2 Universal Database V8.2 installation” on page 1387.

Create a DB2 UDB database

Create a DB2 UDB database named BANK.

For details refer to “Create DB2 UDB database via a DB2 command window” on page 346.

Create a connection

Create a connection to the BANK database from within Rational Application Developer. Figure 8-44 on page 396 displays the settings we entered to create the DB2 Bank Connection.

For details refer to “Create a database connection” on page 347. When using DB2 UDB you will need to specify the user ID and password.

Archived

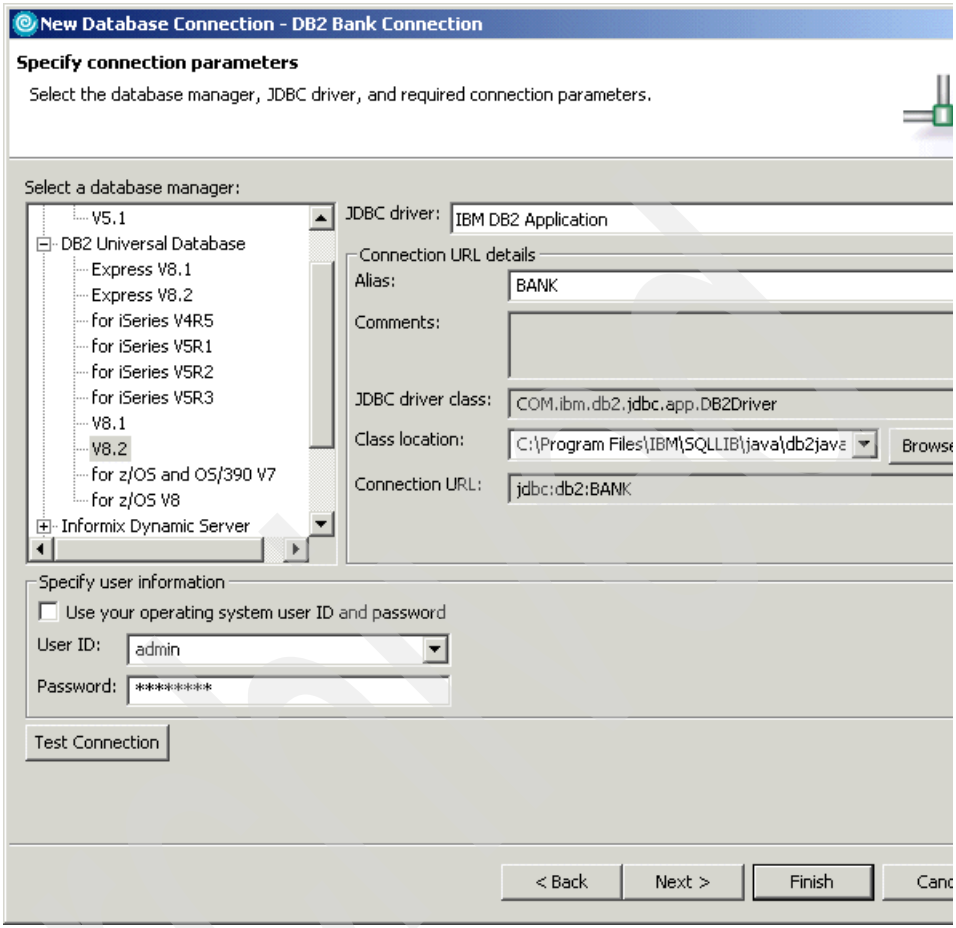


Figure 8-44 DB2 UDB database connection

Create the database tables

Create the BANK database tables.

For details refer to “Create DB2 UDB database tables via a DB2 command window” on page 351.

Populate the database tables

Populate the BANK database tables with sample data.

For details refer to “Populate the tables via a DB2 UDB command window” on page 354.

Create a Web Project

Create a new Web Project named BankDBWeb for the Java stored procedure examples.

For details on creating a Web Project refer to 11.3.2, “Create a Web Project” on page 517.

Copy the database to the Web Project

To copy the database definition to the Web Project, do the following:

1. Open the Data perspective and change to the Database Explorer view.
2. Create a folder named `databases` in the /BankDBWeb/WebContent/WEB-INF directory.
3. Right-click **BANK** and select **Copy to Project**.
4. When the Copy to Project window appears, we entered `/BankDBWeb/WebContent/WEB-INF/databases` and then clicked **OK**.
5. Click **Finish**.

   Figure 8-45 on page 397 shows the Data Definition view in the Data perspective after the DB2 UDB Bank database has been copied to the BankDBWeb Project.

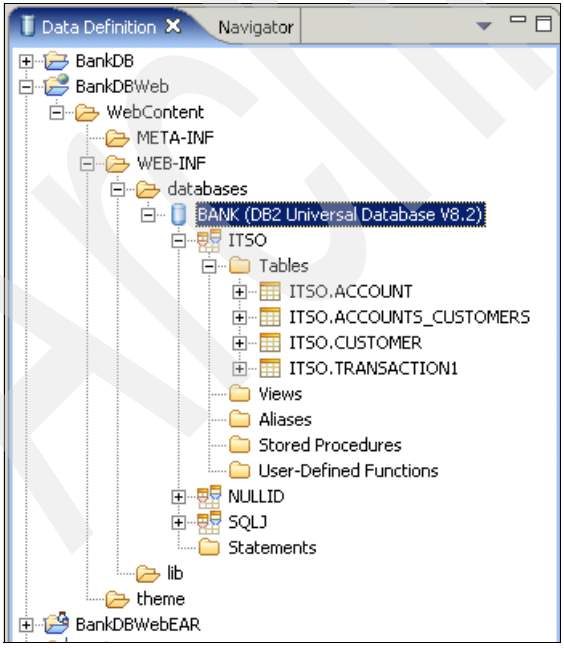


Figure 8-45 Data Definition view - Copy the BANK DB2 database to BankDBWeb

Archive

8.9.2 Create a Java stored procedure

In this section we demonstrate how create a simple stored procedure that takes a customer SSN and returns all customer information. Later we create a Web application that uses the stored procedure.

Enable Java stored procedures

Prior to using Java stored procedures, the Workbench capability must be enabled.

1. Select **Window** → **Preferences** from the main menu.
2. Select **Workbench** → **Capabilities**.
3. Check the **Database Developer** check box to enable all database capabilities.
4. Check **Stored Procedure and User-Defined Function Development**, and click **OK**.

Using the Stored Procedure Wizard

To create a stored procedure using the Stored Procedure Wizard, do the following:

1. Open the Data Definition view in the Data perspective.
2. Select **BankDBWeb** → **WebContent** → **WEB-INF** → **databases** → **BANK** → **ITSO**.
3. Right-click and select **New** → **Java Stored Procedure**.

   Alternatively, select **File** → **New** → **Other** → **Data** → **Java Stored Procedure**.
4. When the Specify name for stored procedure dialog appears, we entered the following (as seen in Figure 8-48 on page 401) and then click **Next**:
   - Name: `getCustomer`
   - Build: Leave unchecked, as we will build later.

Archived

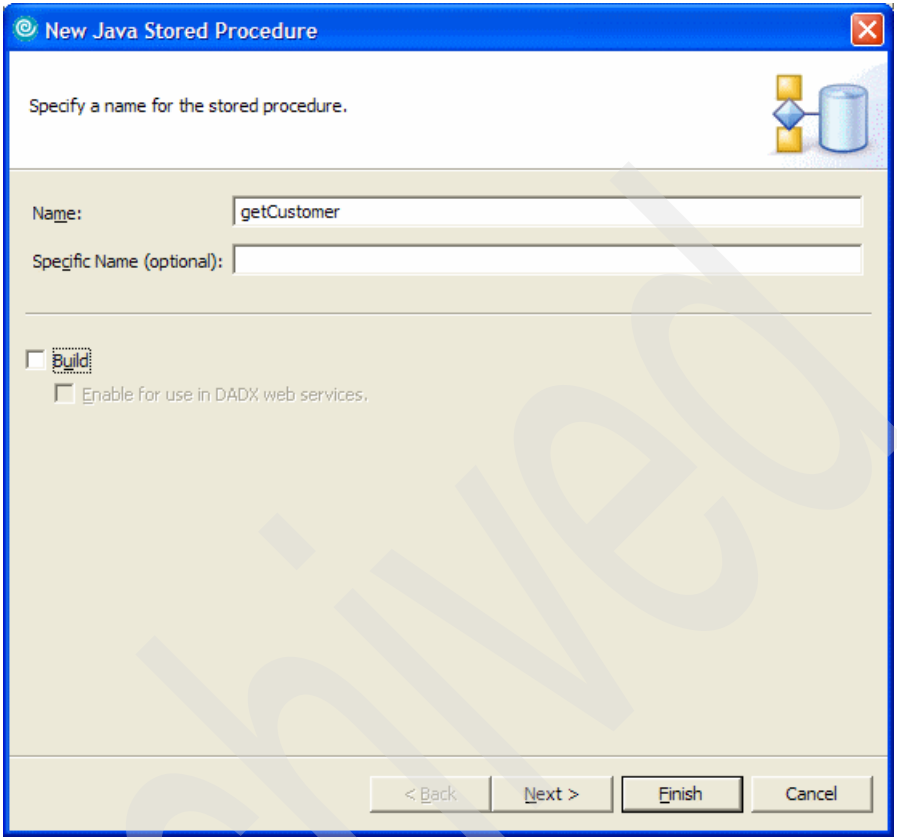


Figure 8-46 Java stored procedure - Specify name for stored procedure

5. When the Create new SQL statements dialog appears, do one of the following (as seen in Figure 8-48 on page 401) to update the SQL. When done click **Next**.
   - SQL Statement Wizard.

     We recommended that you use the SQL Statement Wizard by clicking **SQL Assist**. Refer to 8.7.1, “Using the SQL Statement wizard” on page 376, for more information on using SQL Assist. The desired SQL for the example is displayed in Example 8-9. Or do the following.
   - Manually enter SQL.

     You can enter the statement manually by typing the statement as seen in Example 8-9. This statement gets all fields from the customer table given a customer ID.

Archived

Example 8-9 getCustomer SQL

```
SELECT
   ITSO.CUSTOMER.TITLE,
   ITSO.CUSTOMER.FIRSTNAME,
   ITSO.CUSTOMER.LASTNAME,
   ITSO.CUSTOMER.SSN
FROM
   ITSO.CUSTOMER
WHERE
   ITSO.CUSTOMER.SSN = :ssn
```

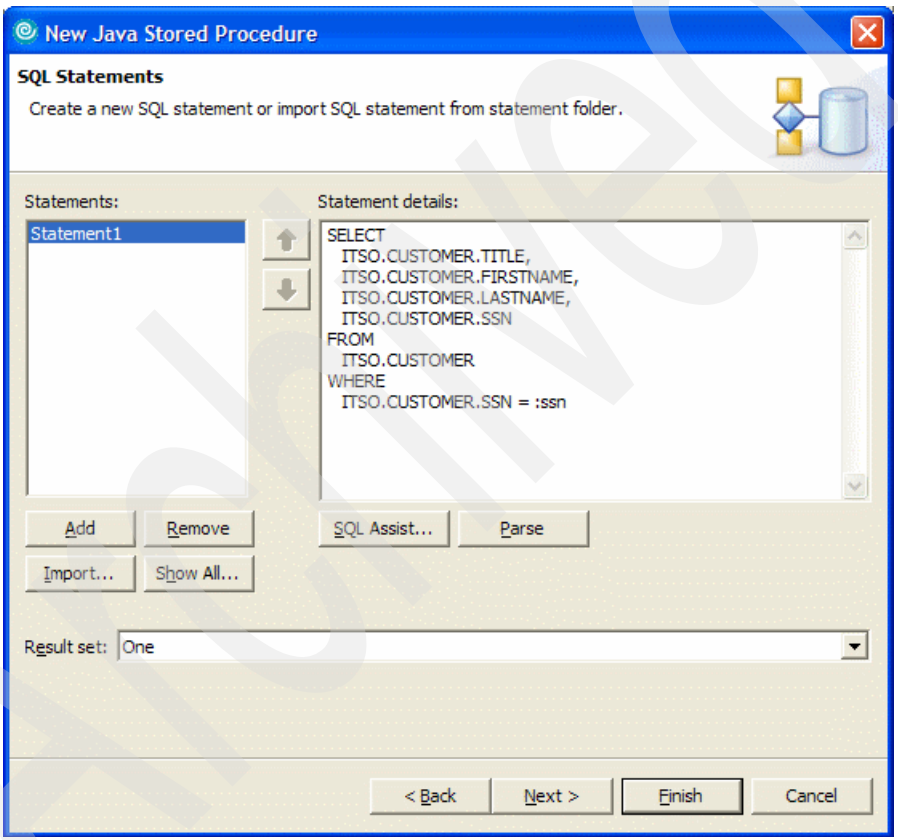


Figure 8-47 Java stored procedure - SQL Statements

Back in the wizard, the new SQL statement is shown in the statement text box. In the remainder of this page of the wizard, you can add code fragments that you wish to include in the generated stored procedure.

6. When the Specify parameters dialog appears, specify the parameters for the stored procedure (as seen in Figure 8-48 on page 401) and then click **Next**.

– If you used SQL Assist to create the SQL statement, the parameter is already filled in.
– If you typed in the SQL statement, you have to create the parameter manually by clicking **Add** and specifying *In* for the mode field and ssn for the name field, and VARCHAR for the SQL Type field.

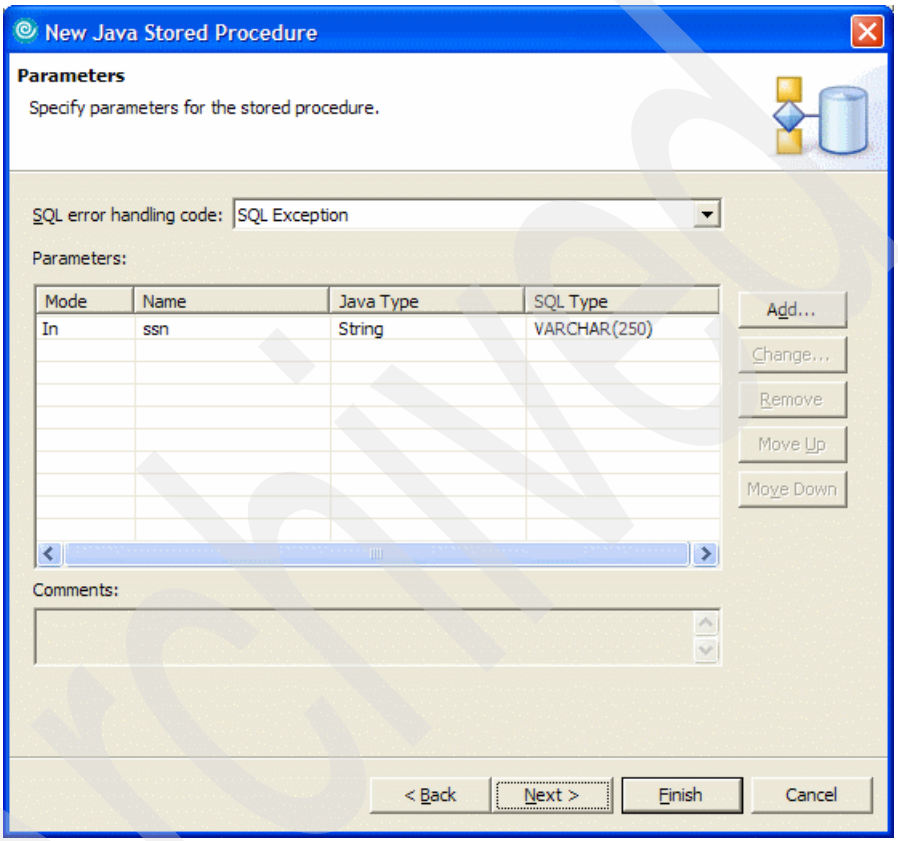


Figure 8-48 Java stored procedure - Specify parameters for the stored procedure

7. When the Specify options dialog appears, we entered the following (as seen in Figure 8-49 on page 402) and then clicked **Next**:
   – Jar ID: Accept default (value contains current time stamp).
   – Java package: `itso.bank.example.db`
   – We accepted the default values in the remaining fields.

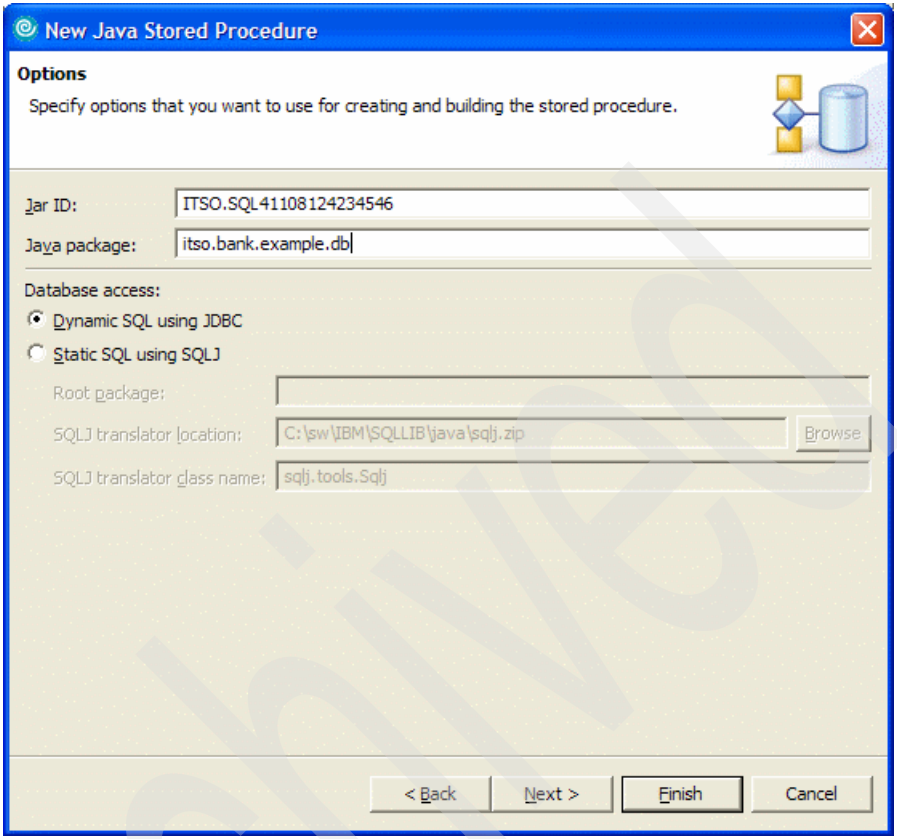


Figure 8-49 Java stored procedure - Specify options

8. When the Code Fragments dialog appears, customize the generated code.

   This page provides a means to customize the generated code by specifying header, import, data, and method code fragment file that will be inserted at the appropriate place in the generated code (see Figure 8-50 on page 403).

   In our example, we accepted the default generated code and clicked **Next**.

Archived

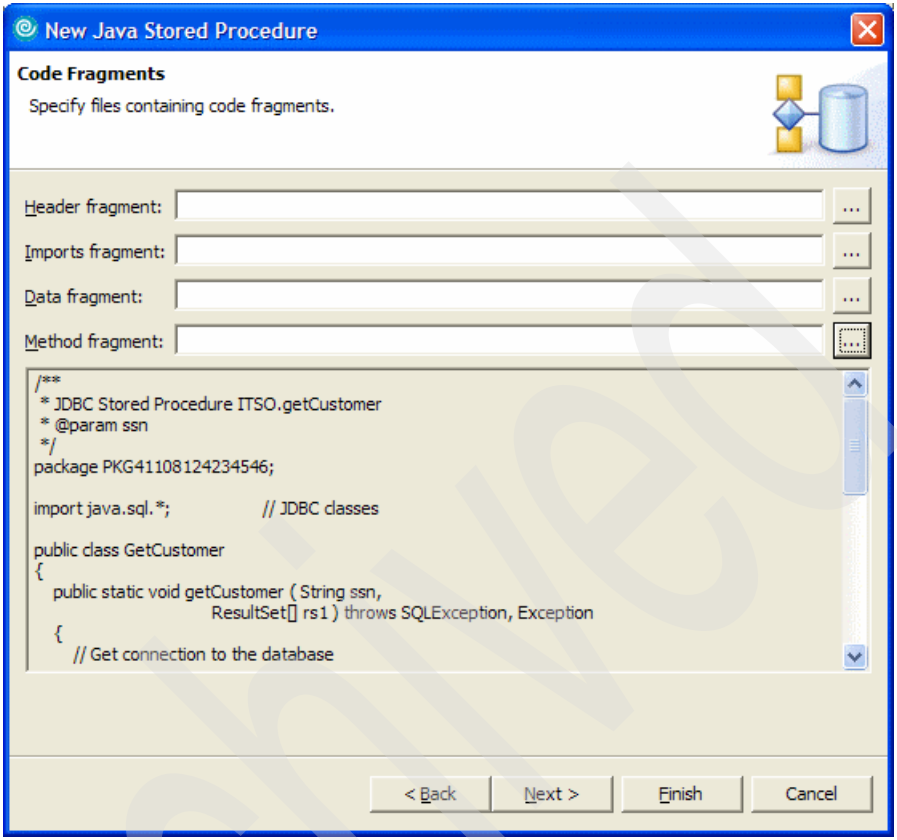


Figure 8-50 Java stored procedure - Code fragments

9. When the Summary of the stored procedure dialog is displayed, as shown in Figure 8-51 on page 404, review the settings and then click **Finish**.

Archived

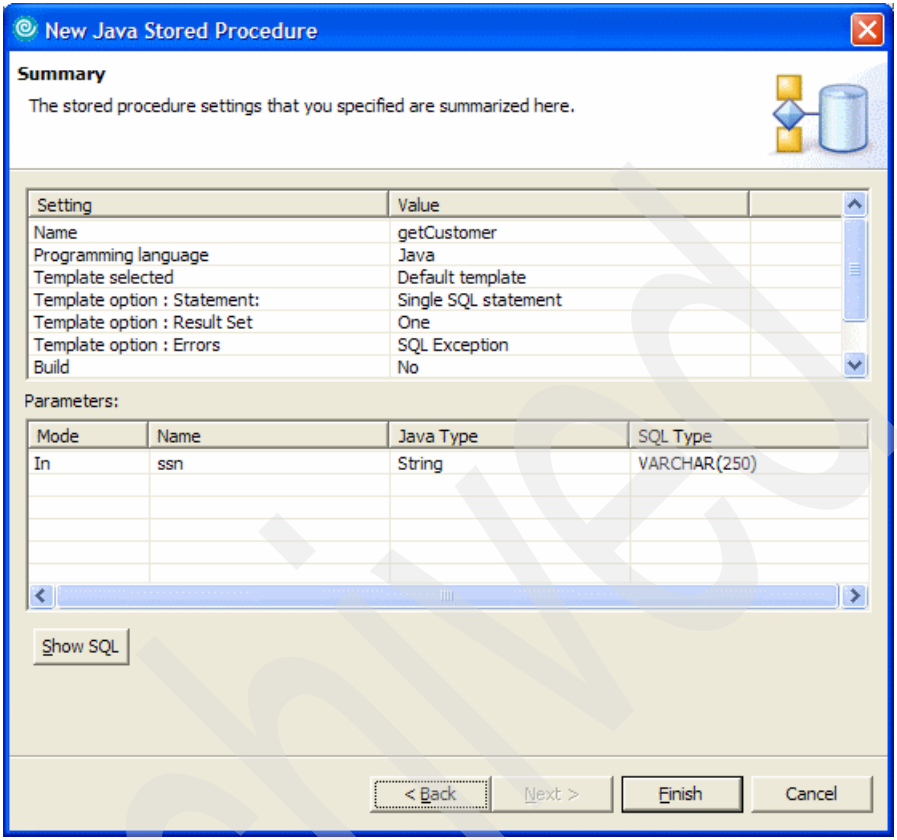


Figure 8-51 Summary of the new stored procedure before it is created

10. The generated Java stored procedure code is shown in Example 8-10. You can double-click the **getCustomer** Java stored procedure to open the Java editor.

    The getCustomer Java stored procedure can be found in the **BankDBWeb** → **WebContent** → **WEB-INF** → **databases** → **BANK** → **ITSO** → **Stored Procedures** folder.

Example 8-10 getCustomer Java stored procedure

```
/**
 * JDBC Stored Procedure ITSO.getCustomer
 * @param ssn
 */
package itso.bank.example.db;

import java.sql.*;                  // JDBC classes
```

```
public class GetCustomer
{
    public static void getCustomer ( String ssn,
                                      ResultSet[] rs1 ) throws SQLException,
Exception
    {
        // Get connection to the database
        Connection con =
DriverManager.getConnection("jdbc:default:connection");
        PreparedStatement stmt = null;
        boolean bFlag;
        String sql;

        sql = "SELECT "
            + "    ITSO.CUSTOMER.TITLE, "
            + "    ITSO.CUSTOMER.FIRSTNAME, "
            + "    ITSO.CUSTOMER.LASTNAME, "
            + "    ITSO.CUSTOMER.SSN"
            + " FROM"
            + "    ITSO.CUSTOMER"
            + " WHERE"
            + "    ITSO.CUSTOMER.SSN =  ?";
        stmt = con.prepareStatement( sql );
        stmt.setString( 1, ssn );
        bFlag = stmt.execute();
        rs1[0] = stmt.getResultSet();
    }
}
```

8.9.3 Build a stored procedure (deploy to database)

After a stored procedure is written locally, you must build it (deploy to database) on the database server. The build procedure uploads and compiles the Java stored procedure to the database server.

1. Expand **BankDBWeb** → **WebContent** → **WEB-INF** → **databases** → **BANK** → **ITSO** → **Stored Procedures** from the Data Definition view.
2. Select the **getCustomer** stored procedure.
3. Right-click and select **Build** from the context menu.
4. If you are prompted to create a new connection, select the Use Existing Connection check box, select the appropriate connection (for example, DB2 Bank Connection) and click **Finish**.

   Figure 8-52 on page 406 displays the DB Output view, which contains the result of the build.

Archived

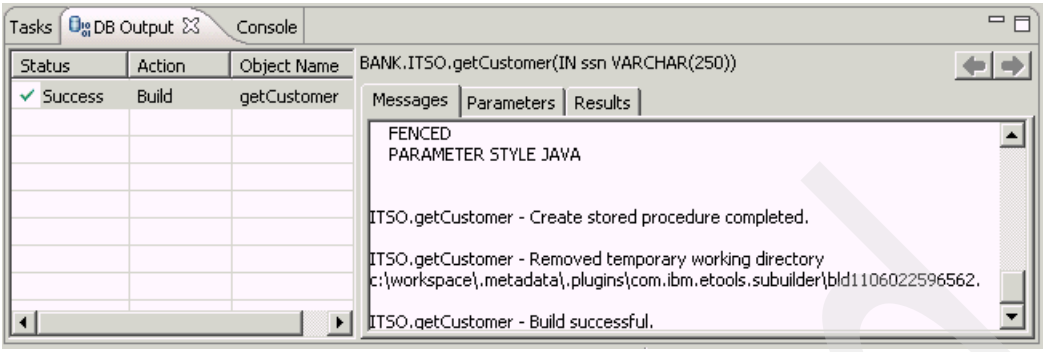


Figure 8-52 Java stored procedure - DB Output view with build output of stored procedure

In Figure 8-52 we see that the build was successful. The GetCustomer class is compiled and placed into a JAR file, and the JAR file is installed in the target database.

Tip: You can see stored procedures in the DB2 UDB database system from the DB2 Control Center.

Execute and test the stored procedure

IBM Rational Application Developer V6.0 provides a test facility for testing Java stored procedures.

1. Select the **getCustomer** stored procedure in the Data Definition view.
2. Right-click and select **Run** from the context menu.
3. When the Run Settings dialog appears, enter a customer ssn (for example, 111-11-1111) as seen in Figure 8-53 on page 407, and then click **OK**.

Archived

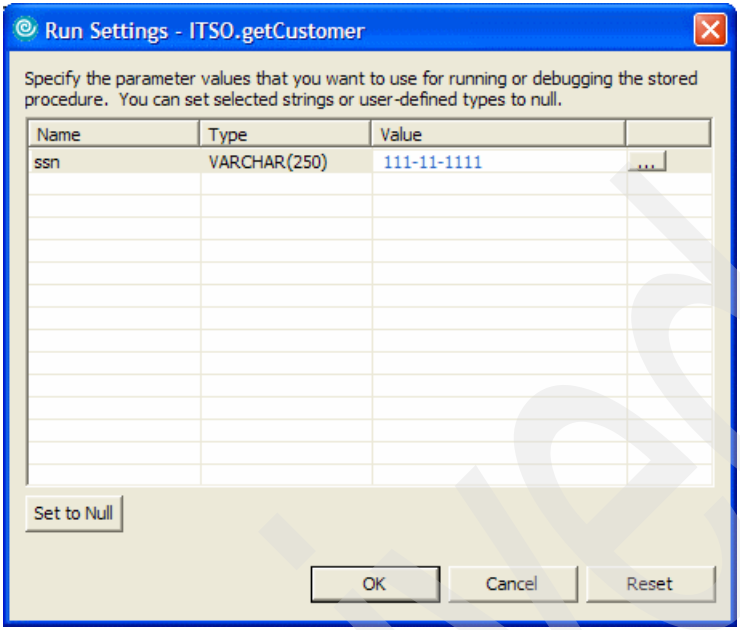


Figure 8-53 Java stored procedure - Run Settings

The output of the Java stored procedure is shown in the DB Output view, as shown in Figure 8-54.

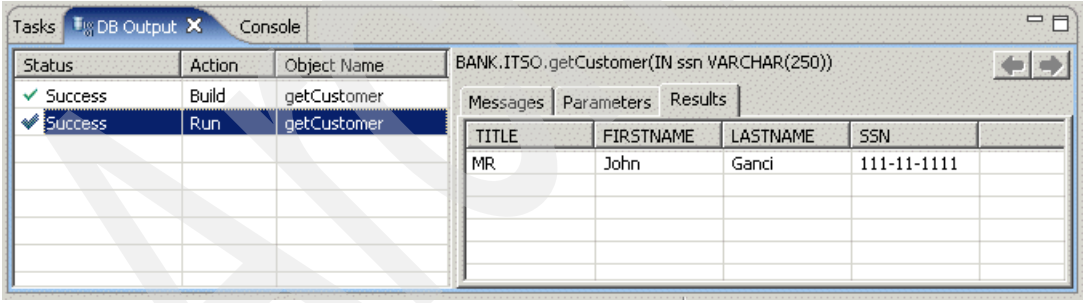


Figure 8-54 Java stored procedure - DB Output view from running the getCustomer stored procedure

We have now created a simple procedure that takes an input argument (the customer ssn) and returns the details about that customer from the database.

Tip: You can make changes to the stored procedure by selecting **Properties** from the context menu.

8.9.4 Java DriverManager access to a Java stored procedure

Once you have loaded a stored procedure into the database, it can be accessed from an application via a DriverManager connection. Example 8-11 displays a code fragment illustrating how to call the Java stored procedure from an application.

Example 8-11 Code snippet from getCustomerMain.java on how to access a Java store procedure

```
public static void main(String[] args) throws SQLException {
   String ssn = "111-11-1111";
   if (args != null && args.length > 0) ssn = args[0];

   GetCustomerMain custTest = new GetCustomerMain();

   // connect to database
   Connection con = custTest.getConnection();

   // prepare statement that calls stored procedure
   CallableStatement cs = con.prepareCall("{call ITSO.GetCustomer(?)} ");
   cs.setString(1, ssn);

   // execute
   ResultSet rs = cs.executeQuery();

   while (rs.next()) {
      // get the data from the row
      System.out.println("CUSTOMER SSN: " + rs.getString("ssn"));
      System.out.println("TITLE:      " + rs.getString("title"));
      System.out.println("FIRST NAME:  " + rs.getString("firstname"));
      System.out.println("LAST NAME:   " + rs.getString("lastname"));
   }
}
```

The question mark (?) is a place holder for a parameter. The second line sets 111-11-1111 as the value for the parameter. Following that, the stored procedure is executed and the results are obtained from the result set.

We provide a main program called GetCustomerMain.java that you can import into the BankDBWeb Project.

1. Import GetCustomerMain.java into a new itso.bank.example.db package from the following directory of the ITSO sample code:

   `c:\6449code\database`

2. Add the db2java.zip to the Java build path of the BankDBWeb Project. In our example, the db2java.zip was found in the <db2_home>\java directory.
3. Right-click **GetCustomerMain.java** and select **Run** → **Java Application**.

The Console output should look like Figure 8-55.

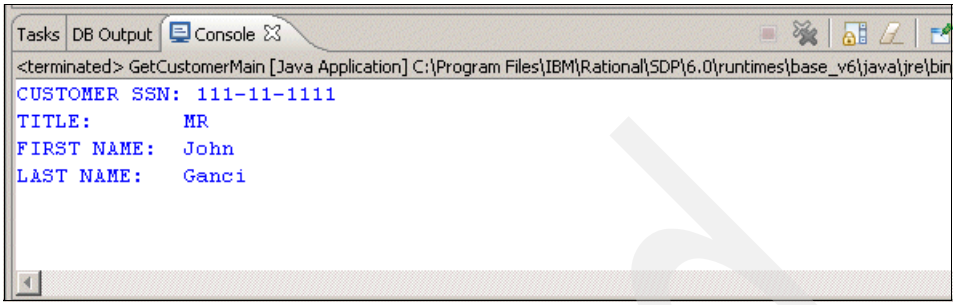


Figure 8-55 Console output from Java application accessing a stored procedure

Note: If you get a SQL Exception indicating no suitable driver was found, ensure that the db2java.zip file is in the Java Build Path.

8.9.5 JavaBean access to Java stored procedure

In this section we describe how to generate a JavaBean using a wizard provided by Rational Application Developer, and then use the JavaBean to access a Java stored procedure.

Generate a JavaBean to access the stored procedure

Rational Application Developer provides a wizard to generate a JavaBean that accesses a Java stored procedure.

To generate a JavaBean, do the following:

1. Open the Package Explorer view in the Java perspective.
2. Expand **BankDBWeb** → **JavaSource**.
3. Right-click and select **New** → **Package**.
4. When the New Package dialog appears, enter `itso.bank.example.db.bean` in the Name field and then click **OK**.
5. Select the **getCustomer** stored procedure in the Data Definition view.
6. Right-click and select **Generate JavaBean** from the context menu.
7. When the Java Class Specification dialog appears, we entered the following as seen in Figure 8-56 on page 410, and then click **Next**:
   - Source Folder: Click **Browse** to locate the BankDBWeb/JavaSourceJavaSource folder.

- Package: Click **Browse** to locate the itso.bank.example.db.bean package.
- Name: `GetCustomerBean`
- Select **Stored procedure returns a result set**.
- Check **Generate a helper class with methods to access each column in the result set**.

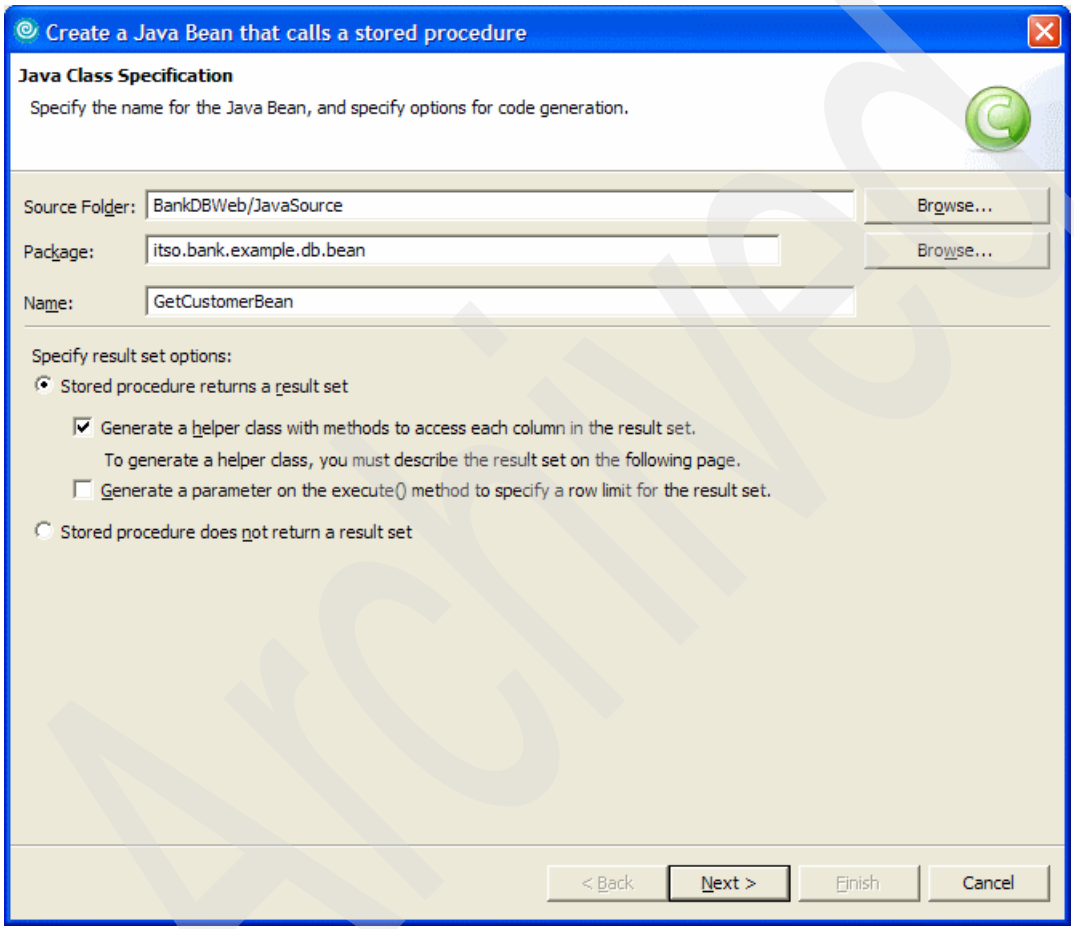


Figure 8-56 Generate a JavaBean - Java class specification

8. When the Describe the result set dialog appears, select the **ITSO.CUSTOMER** table from the Tables tab, click **>** to move to the Selected Tables, and then click **Next**.
9. Click the **Columns** tab, select each of the columns (TITLE, FIRSTNAME, LASTNAME, SSN), click **>** to move to the Selected Columns (as seen in Figure 8-57 on page 411), and then click **Next**.

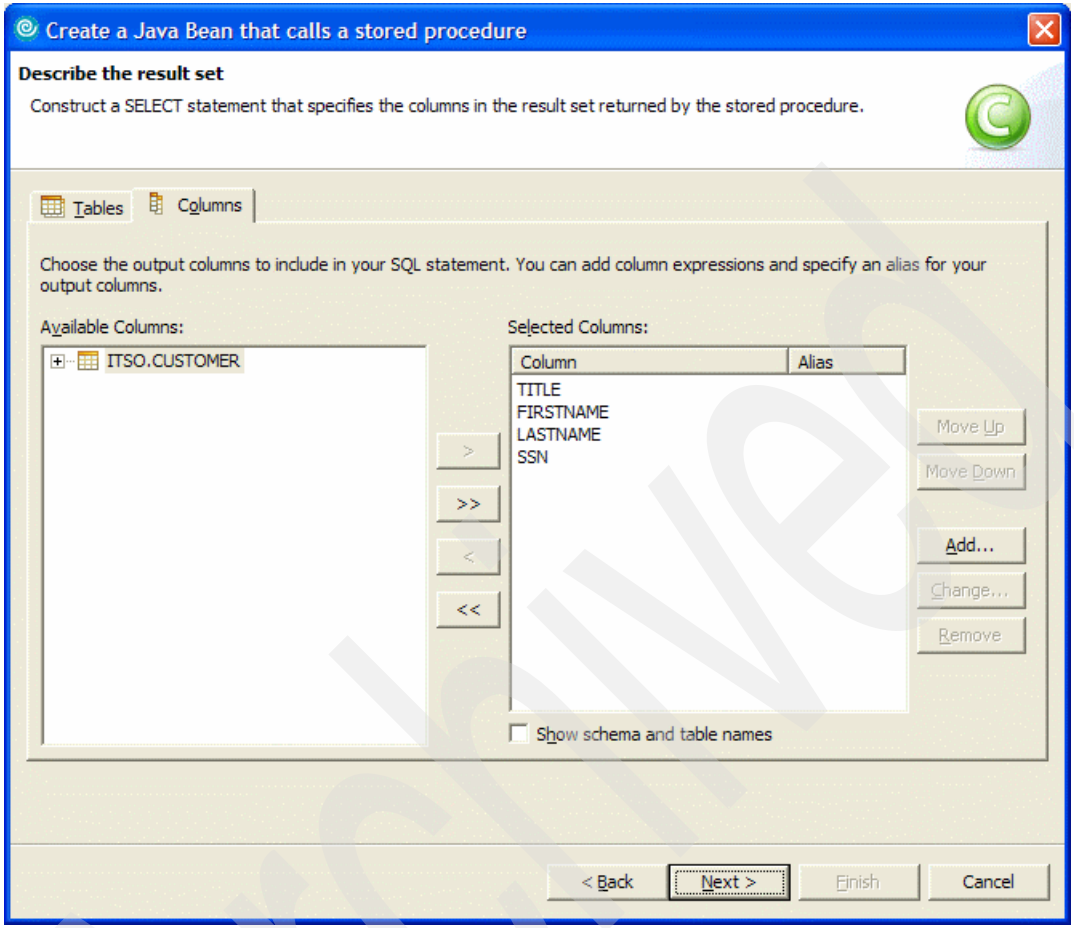


Figure 8-57 Generate a JavaBean - Describe the result set

10. When the Specify Runtime Database Connection Information dialog appears, we entered the following (as seen in Figure 8-58 on page 412), and then click **Next**:
    - Select **Use DriverManager connection**.
    - Driver name: `COM.ibm.db2.jdbc.app.DB2Driver`
    - URL: `jdbc:db2:BANK`
    - Select **Inside the execute () method**.
    - User ID: `<db2_admin>`
    - Password: `<db2_admin_password>`

Archive

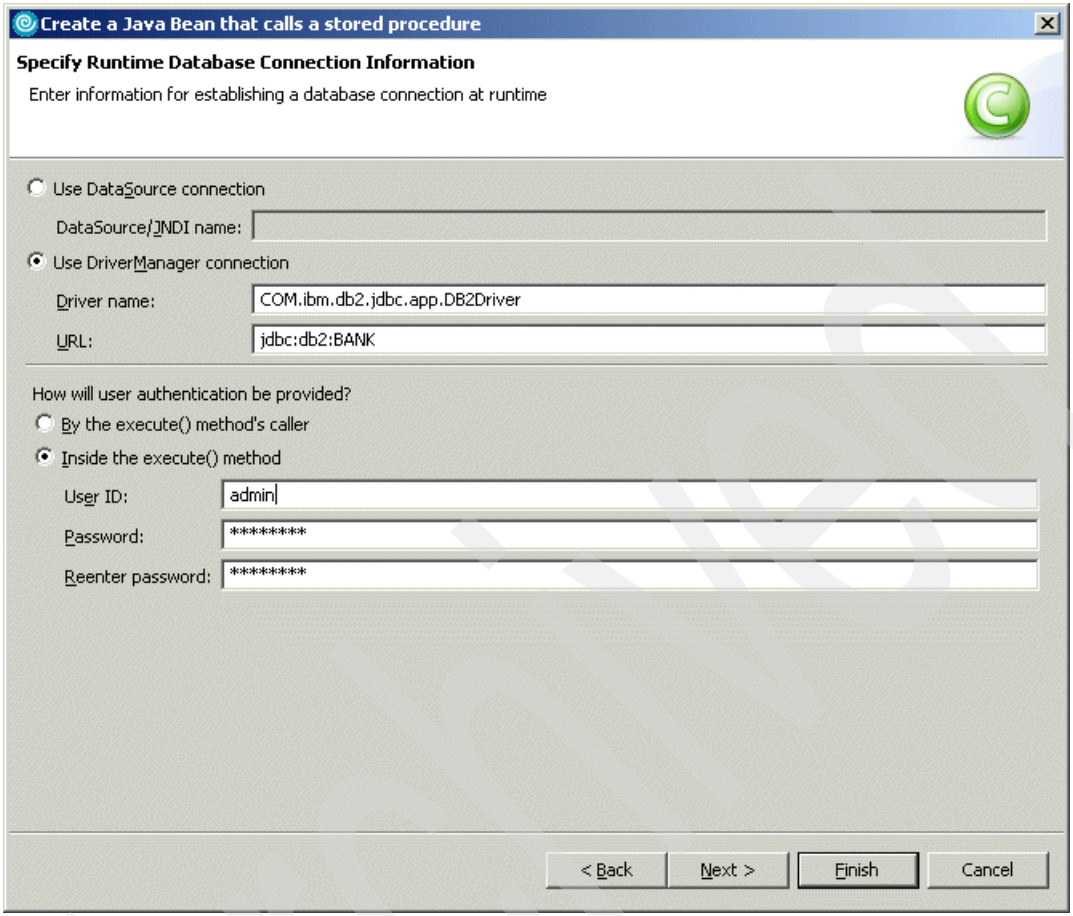


Figure 8-58 Generate a JavaBean - Database connection properties

11. When the Review the specification dialog appears, review the methods that generated (as seen in Figure 8-59 on page 413), and then click **Finish**.

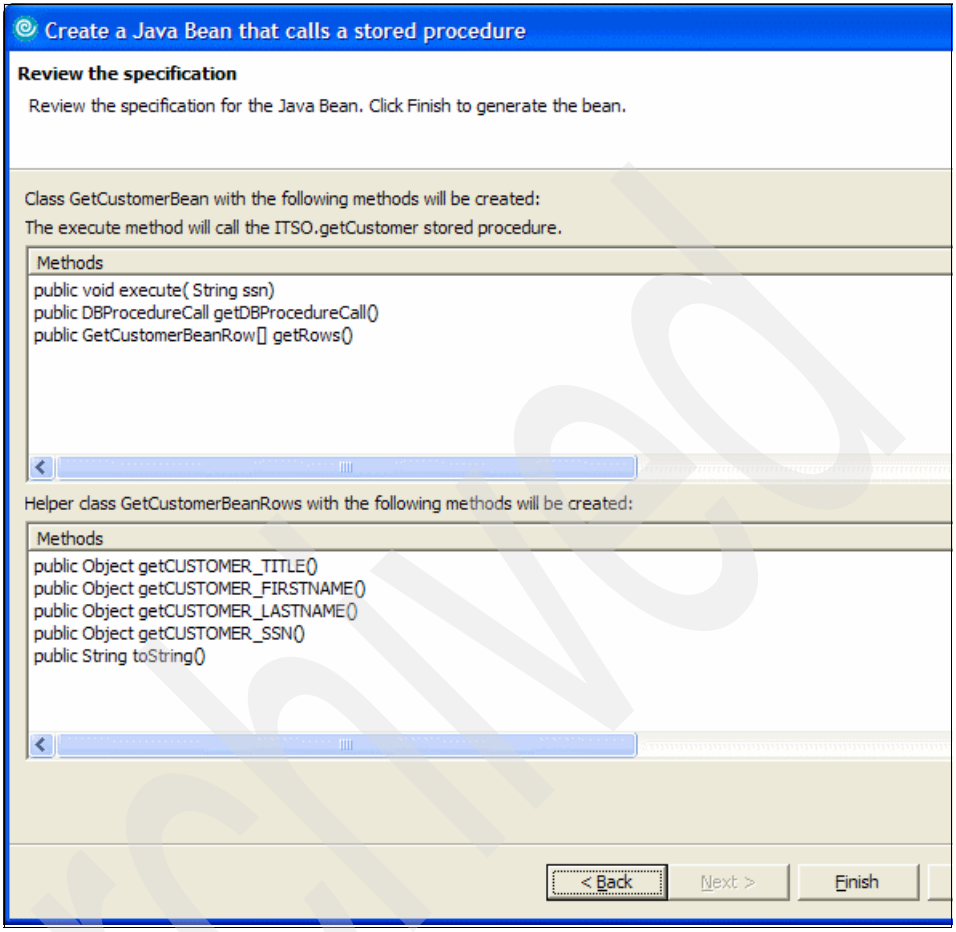


Figure 8-59 Generate a JavaBean - Review the specification

The following two classes are generated:

- GetCustomerBeanRow

  Provides access to one row of the result set.

- GetCustomerBean

  Executes the stored procedure and provides a method to retrieve an array of GetCustomerBeanRow objects.

Using the JavaBean

The generated GetCustomerBean JavaBean can be used in a servlet or JSP to execute the stored procedure and access the result set.

Archive

A simple JSP to execute the JavaBean is shown in Example 8-12 on page 414.

Example 8-12 JSP used to execute the JavaBean with the Java stored procedure

```
<BODY>
<H1>JSP -> JavaBean -> Stored Procedure</H1>
<jsp:useBean id="getCustomer"
   class="itso.bank.example.db.bean.GetCustomerBean"></jsp:useBean>

<% getCustomer.execute( new String("111-11-1111") ); %>

<% itso.bank.example.db.bean.GetCustomerBeanRow row = getCustomer.getRows()[0];
%>
<TABLE border="1">
   <TR><TH align="left">TITLE</TH>
      <TD><%= row.getCUSTOMER_TITLE() %></TD></TR>
   <TR><TH align="left">FIRSTNAME</TH>
      <TD><%= row.getCUSTOMER_FIRSTNAME() %></TD></TR>
   <TR><TH align="left">LASTNAME</TH>
      <TD><%= row.getCUSTOMER_LASTNAME() %></TD></TR>
   ......
</TABLE>
</BODY>
```

Import the RunGetCustomerBean.jsp provided in the following directory of the ITSO provided sample code:

```
c:\6449code\database\RunGetCustomerBean.jsp
```

The GetCustomerBean JavaBean is instantiated using a <useBean> tag. The stored procedure is executed and the first row of the result set is retrieved and displayed. Note that the customer ssn is passed as a constant, and no error checking is performed. In a real application the code would be more complex.

Note: Java stored procedures can also be executed through the jspsql tag library. For information, refer to the Rational Application Developer online help

Archived

9

Develop GUI applications

Rational Application Developer provides a Visual Editor for building Java graphical user interfaces (GUIs).

In this chapter we introduce the Visual Editor and develop a sample GUI, which interacts with back-end business logic. This GUI is runnable as a JavaBean and as a Java application.

The chapter is organized into the following sections:

- Introduction to the Visual Editor
- Prepare for the sample
- Launching the Visual Editor
- Visual Editor overview
- Work with the Visual Editor

Note: The sample code described in this chapter can be completed by following along in the procedures documented. Alternatively, you can import the sample code provided in the c:\6449code\gui\BankGUI.zip Project Interchange file. For details refer to Appendix B, “Additional material” on page 1395.

© Copyright IBM Corp. 2005. All rights reserved.

9.1 Introduction to the Visual Editor

The Visual Editor is used to design applications containing a graphical user interface (GUI) based on the JavaBeans component model. It supports visual construction using either the Abstract Window Toolkit (AWT), Swing, or the Standard Widget Toolkit (SWT).

More information concerning Swing and AWT can be found at:

```
http://java.sun.com/j2se/1.4.2/docs/guide/swing/
http://java.sun.com/j2se/1.4.2/docs/guide/awt/
```

The SWT is part of Eclipse. More information can be found at:

```
http://www.eclipse.org/platform/
```

The Visual Editor allows you to design GUIs visually. Using the Visual Editor, you can drag beans from different palettes, manipulate them in the Design view, and edit their properties in the Properties view. The Visual Editor also includes a Source view where you can both see and modify the generated Java code. You can make changes in either the Source view or in the Design view—the Visual Editor uses a process known as round-tripping to synchronize the two views.

9.2 Prepare for the sample

The sample GUI we develop in this chapter is shown in Figure 9-1 on page 417. It allows a user to search for a specific social security number and view full information about the customer and the accounts held by the customer. We use Swing components for our sample.

Archive

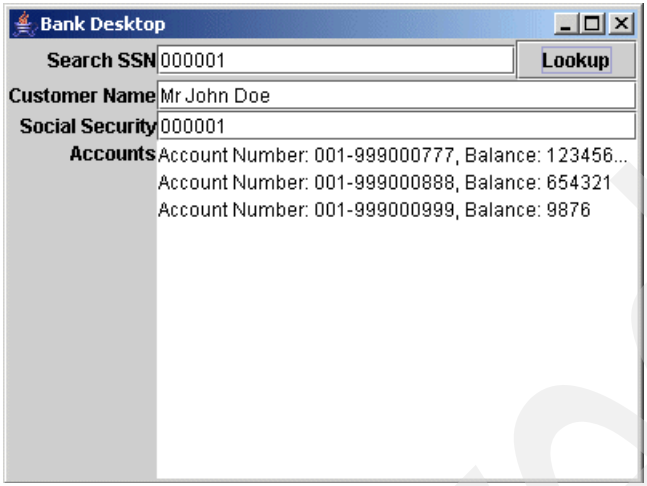


Figure 9-1 Sample GUI

By creating the sample GUI, you should learn how to work with the Visual Editor and how to compose and add visual components, change their properties, add event handling code, and run the GUI.

To set up your development environment for the GUI sample application, you need to create the database that the GUI will use and import some database access classes.

9.2.1 Create the project for the sample

To create the BankGUI project for our sample with Cloudscape database support, do the following:

1. From the Workbench select **File** → **New** → **Project...**.
2. From the first page of the New Project wizard, select **Java Project** and click **Next >**.
3. On the second page of the wizard, enter the following and then click **Next**:
   - Project name: `BankGUI`
   - Select **Create project in workspace** (default).
   - Select **Use project folder as root for sources and class files** (default).

9.2.2 Add JDBC driver for Cloudscape to project

To add the Cloudscape JDBC driver to the project, do the following:

1. Select the project, right-click, and select **Properties**.

Archive

2. Select **Java Build Path**.
3. Add the JAR file that contains the JDBC drivers for the Cloudscape database we will be using in the sample.
   a. Select the **Libraries** tab at the top of the dialog and click **Add Variable...**.

      A further dialog appears, allowing you to select from a list of predefined variables. By default, there is no variable defined for the JAR file we need, so we will have to create one.

   b. Click **Configure Variables...**, and in the resulting dialog click **New...**.
   c. Enter `CLOUDSCAPE_DRIVER_JAR` in the Name field and click **File...**.
   d. Find the appropriate JAR file, which is in <rad_home>\runtimes\base_v6\cloudscape\lib and is called db2j.jar.
   e. Click **Open**, **OK**, and **OK** and you will be back at the New Variable Classpath Entry dialog.
   f. Select the **CLOUDSCAPE_DRIVER_JAR** variable you just created and click **OK**.
4. Click **Finish** on the New Java Project wizard.
5. Unless you have previously turned this feature off, Rational Application Developer will display a Confirm Perspective Switch dialog asking whether you want to switch to the Java perspective. Click **Yes**. If you have turned this feature off, you will need to open the Java perspective now.

We will not be using the J2EE perspective, so it can be closed now to save memory.

Note: Creating a variable is a good practice for this kind of situation. The alternative is to select **Add External JARs...**, which adds the full path to the JAR file into the project build settings. A variable can be shared among all the projects in the workspace, saving time for JAR files that are likely to be needed elsewhere, like this one.

9.2.3 Set up the sample database

If the Cloudscape database has already been configured for another sample in this book, you can skip the next step and go straight to 9.2.4, “Import the model classes for the sample” on page 419.

This section provides instructions for deploying the BANK sample database and populating the database with sample data. For simplicity we use the built-in Cloudscape database.

To create the database, create the connection, and create and populate the tables for the BANK sample from within Rational Application Developer, do the following:

1. Create the database and connection to the Cloudscape BANK database from within Rational Application Developer.

   For details refer to “Create a database connection” on page 347.

2. Create the BANK database tables from within Rational Application Developer.

   For details refer to “Create database tables via Rational Application Developer” on page 350.

3. Populate the BANK database tables with sample data from within Rational Application Developer.

   For details refer to “Populate the tables within Rational Application Developer” on page 352.

9.2.4 Import the model classes for the sample

To import the model classes for the GUI sample, do the following:

1. Switch to the Java perspective.
2. Right-click **BankGUI** project and select **Import...**.
3. Choose **Zip file** from the list (this also covers JAR files) and click **Next**.
4. Select **model.jar** from the c:\6449code\gui directory and click **Finish**.

9.3 Launching the Visual Editor

The Visual Editor allows you to create and modify GUIs by manipulating JavaBeans in a WYSIWYG (what you see is what you get) editor. There are two ways to launch the Visual Editor:

- Create a visual class.

  Create a visual class in which case the class is automatically associated with the Visual Editor and opened with it.

- Open an existing class with the Visual Editor.

  Create a Java class and open it with the Visual Editor.

In this example we create a visual class.

Archived

9.3.1 Create a visual class

To create a visual class, do the following:

1. Select **File** → **New** → **Other** → **Java** → **Visual Class**, and then click **Next**.

   Alternatively, right-click the project or package in which you wish to create the visual class and select **New** → **Visual Class** from the context menu.

2. When the Create a new Java class dialog appears, enter the following (as seen in Figure 9-2 on page 421):
   - Source Folder: `BankGUI` (generate by default)
   - Package: `itso.bank.gui`
   - Name: `BankDesktop`
   - Style: Expand **Swing** and select **Frame**. (Note that this enters the correct superclass, javax.swing.JFrame, into the Superclass field.)
   - Interfaces: Click the **Add...** button next to the Interfaces box, type `ActionListener` (to select java.awt.event.ActionListener), and click **OK**.
   - Which method stubs would you like to create? Check **public static void main (String[] args)** and **Inherited abstract methods**. Leave Constructors from superclass unchecked.

   The dialog should look like Figure 9-2 on page 421.

3. Click **Finish**.

Archived

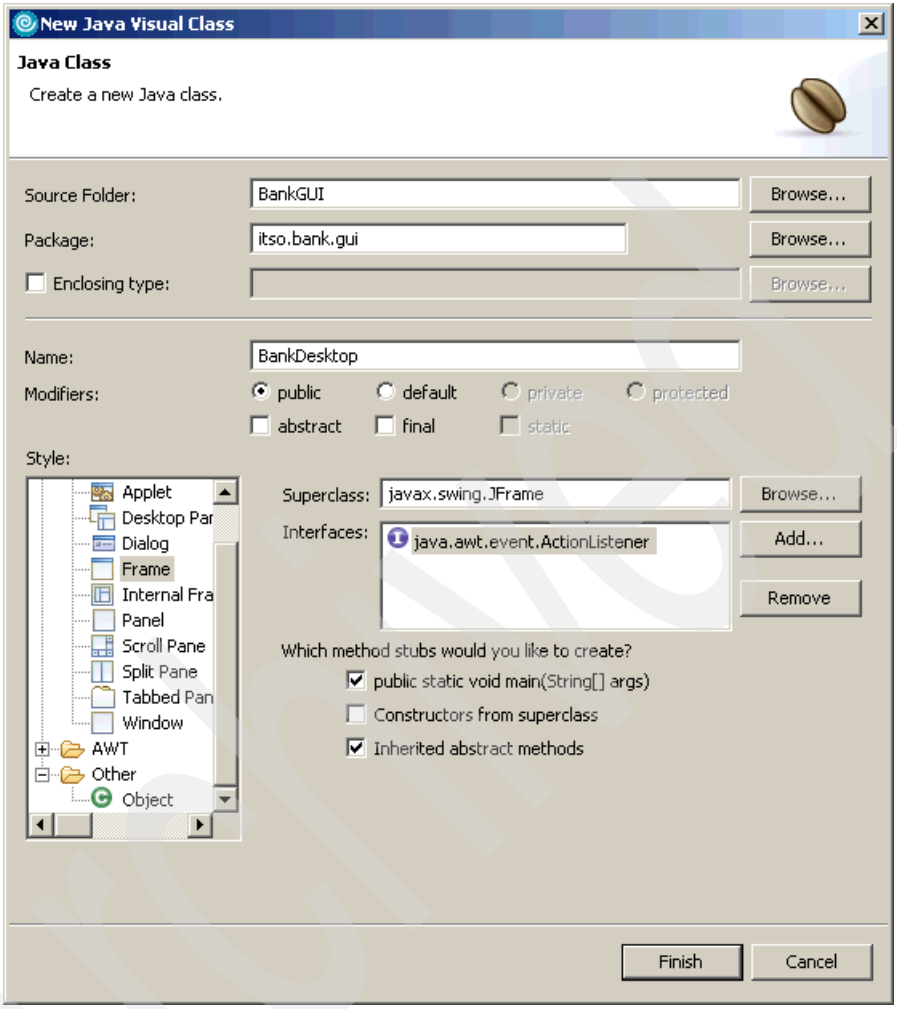


Figure 9-2 New Java Visual Class

The new BankDesktop GUI class opens the Visual Editor. This can take a while, since Rational Application Developer has to start a new JVM to display the GUI visually in the Display view. Figure 9-3 on page 422 shows the new class CustomerGUI in the Visual Editor.

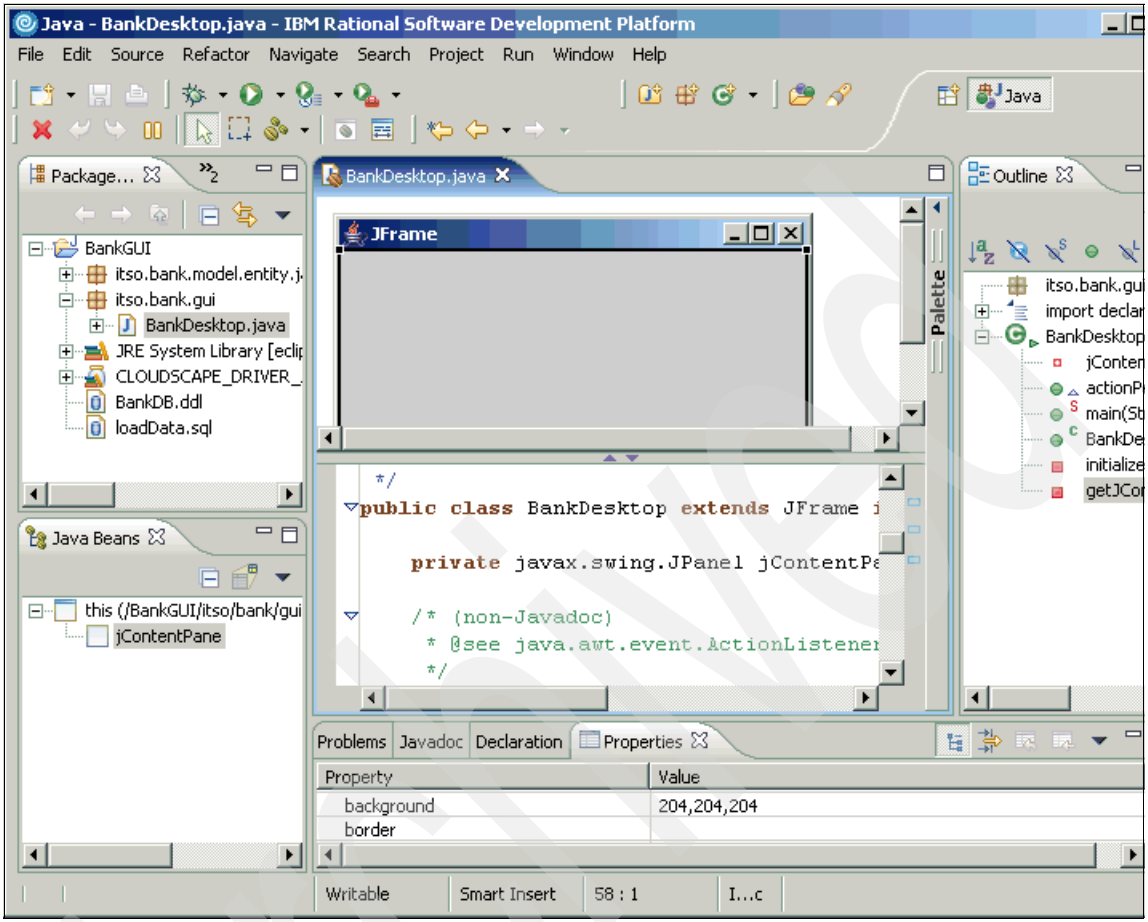


Figure 9-3 New Java Class open in the Visual Editor

9.3.2 Open an existing class with the Visual Editor

Alternatively, you can open any other Java class with the Visual Editor by using the context menu. Right-click the class and select **Open With** → **Visual Editor** (Figure 9-4 on page 423).

Note: The big dot in front of the Visual Editor menu item of the context menu indicates that the last editor used was the Visual Editor.

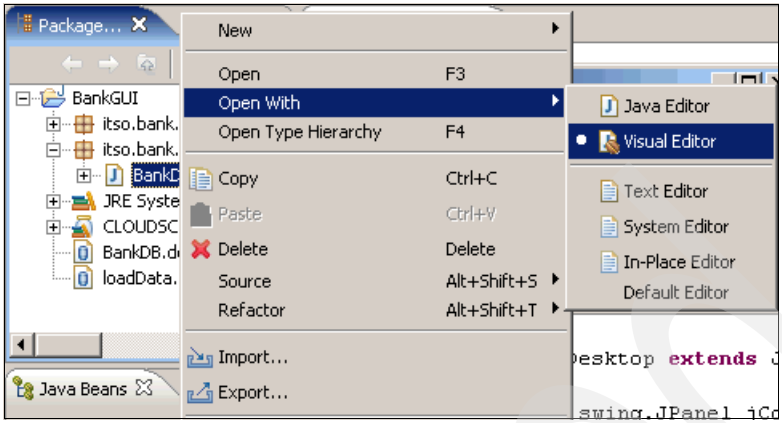


Figure 9-4 Open with context menu

9.4 Visual Editor overview

This section describes the overall layout of the Visual Editor and some of the options for changing the appearance and operation of the editor.

9.4.1 Visual Editor layout

The Visual Editor consists of two panes and a palette bar, shown in Figure 9-5 on page 424. In this screen the editor has been maximized within Rational Application Developer to make it easier to see (double-click the title bar to maximize or click the Maximize button at the top-right of the pane):

- A graphical canvas is located in the top section of the Visual Editor—this is the Design view, where you compose the GUI.
- The source file is displayed beneath it in the Source view, which has all the normal functionality of a Java source editor in Rational Application Developer.
- A palette of common JavaBeans is available on the right:
  - The palette allows you to select JavaBeans for building your GUI.
  - It is divided into drawers, which group components by type.
  - The palette is initially just a gray bar along the right-hand side of the Visual Editor pane. Move the mouse pointer over this bar and the palette appears. (Figure 9-5 on page 424 shows the palette revealed.)
  - The appearance of the palette can be modified as explained in 9.4.2, “Customizing the appearance of the Visual Editor” on page 424.

Archived

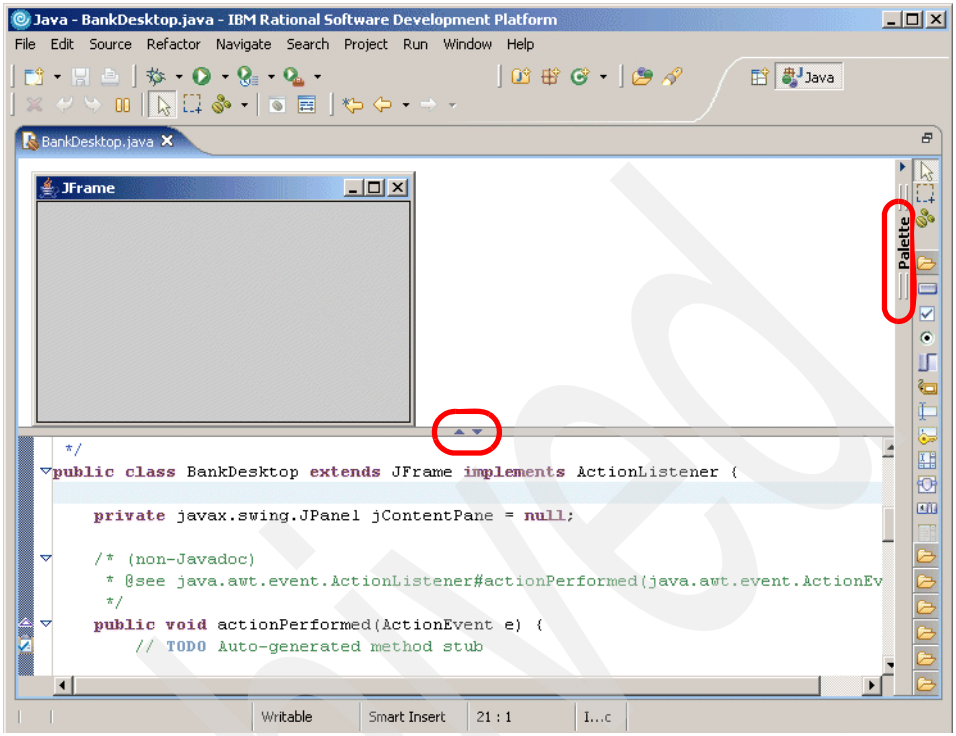


Figure 9-5 Visual Editor layout

When the Visual Editor is opened, two additional views open automatically:

- The Java Beans view, in the bottom-left corner of the workspace, displays the structure of the JavaBeans used in your GUI in a tree view.
- The Properties view, at the bottom-right, lets you view and modify attribute settings.

9.4.2 Customizing the appearance of the Visual Editor

Various aspects of the Visual Editor can be customized.

Click the up arrow, located in the center-left of the separator between the Design and Source views (see Figure 9-5), to maximize the Source view or the down arrow, located to the right of the up arrow, to maximize the Design view.

The palette will automatically hide when not in use, but the appearance of the palette can be modified. Right-click the palette and use the context menu to change the **Layout** (options: Columns, List, Icons Only, Details) and choose whether large or small icons should be displayed. Select **Settings...** from this

menu to open a preferences dialog for the palette. In addition to the options described above, you can also configure the behavior of the drawers and set the font for the palette; this dialog is shown in Figure 9-6. Drawers can be fixed open within the palette by clicking the pin icon next to the drawer name. The width of the palette can also be changed by dragging the left edge, and it can be docked on the left-hand side of the editor if preferred.

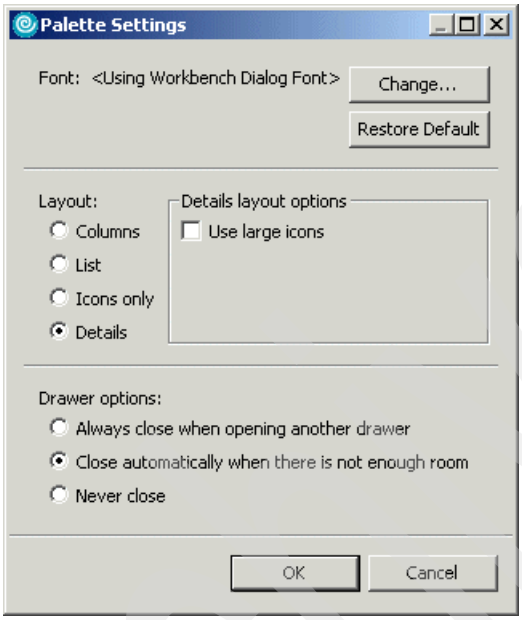


Figure 9-6 JavaBeans Palette Settings

To modify the preferences for the Visual Editor as a whole, select **Window** → **Preferences** and choose **Java** → **Visual Editor**. The preferences page is shown in Figure 9-7 on page 426.

Archived

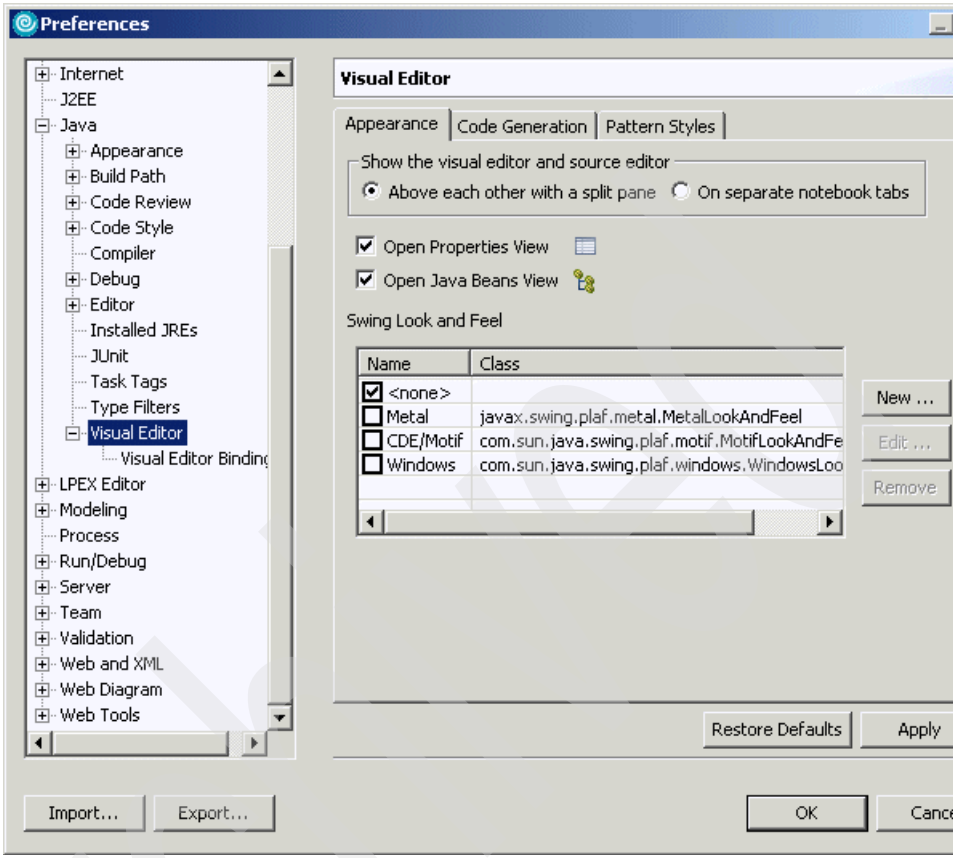


Figure 9-7 Visual Editor preferences

The options are as follows:

- Split or notebook view: The editor is shown as a split pane by default, but a tabbed view is also available.
- Views to open: By default, the Java Beans and Properties views will open; this can be turned off.
- Swing look and feel: The default appearance for Swing components can be set using this list. Check the desired look and feel.
- Code generation: Specify whether try/catch blocks and comments should be generated and set the interval for synchronization between the code and visual views of your GUI.
- Pattern styles: Define how the code generation will use visual beans as the GUI is displayed in the Design view.

The Visual Editor Bindings preferences page allows you to specify the package name for generated code that will bind component properties with other Java components. This is covered later in this chapter in 9.5.11, “Visual Editor binding” on page 438.

9.5 Work with the Visual Editor

We will continue to develop the BankDesktop GUI application to explore the main GUI design features of the Visual Editor.

9.5.1 Resize a JavaBean component

To resize a component, select it (either by clicking the component in the Design view or in the Java Beans view). Control handles appear around the component (small black squares at the corners and in the middles of the sides). Moving these will resize the component.

Use this approach to resize the JFrame. Notice that the generated *initialize* method is modified as a result of this change, so it will now look something like Figure 9-1 (the numbers in the setSize method call will probably be different).

Example 9-1 Resize JavaBean component

```
private void initialize() {
    this.setSize(361, 265);
    this.setContentPane(getJContentPane());
    this.setTitle("JFrame");
}
```

This code will be generated automatically when the visual and code views of your GUI are synchronized (by default this takes place every second).

9.5.2 Code synchronization

We have seen that changes made in the Design view cause changes to the code. The reverse is also true. Change the setSize method call in the initialize method (400, 300), as seen in Figure 9-2.

Example 9-2 Resize JavaBean component

```
private void initialize() {
    this.setSize(400, 300);
    this.setContentPane(getJContentPane());
    this.setTitle("JFrame");
```

```
}
```

After you overtype the first number (the width), replacing it with `400`, wait for around a second and you will see the visual representation of your GUI update even without saving the code changes. During the short interval before the views are synchronized, the status bar displays ***`Out of synch`*** instead of the usual `In synch`.

You can also disable synchronization by pressing the pause button in the toolbar. This is a good idea when you wish to make a number of large changes to the source without incurring the overhead of the round-tripping for synchronizing the code and visual representations.

9.5.3 Changing the properties of a component

We are going to use the GridBagLayout Swing layout manager to position the components in our GUI. By default, the content pane that is created within our JFrame uses BorderLayout.

1. Select the content pane (not frame) by clicking in the grey area within the frame.

   Alternatively, select the component from the list of objects in the **J**ava Beans view (the name is jContentPane).

   Tip: Once the content pane is selected, if you accidentally press Delete the content pane will be deleted. To add it back, right-click the frame and select **Create Content Pane**.

2. Select the **Properties** view and scroll down until you can see the layout property. It should have a value of BorderLayout.
3. Click the value to select it, then click it again to display a list of the available values.
4. Select **GridBagLayout** from the list.
5. Select the **JFrame** title bar. Change the title of the JFrame by clicking the title value (**JFrame**) and change it to `Bank Desktop`.
6. Press Ctrl+S to save.

9.5.4 Add JavaBeans to a visual class

Now we need to add the various GUI components (or JavaBeans) to our GUI.

1. Move the cursor over the palette to display it (or click the small left arrow **Show Palette** button at the top of the palette bar if hidden).
2. Open the **Swing components** drawer.
3. Click the **JLabel** button and move the mouse pointer over to the content pane.

   The mouse pointer will show a set of coordinates `(0,0)` to indicate which row and column within the GridBagLayout the component will be placed (see Figure 9-8).
4. Click in the content pane and the JLabel will appear in the center of the pane.

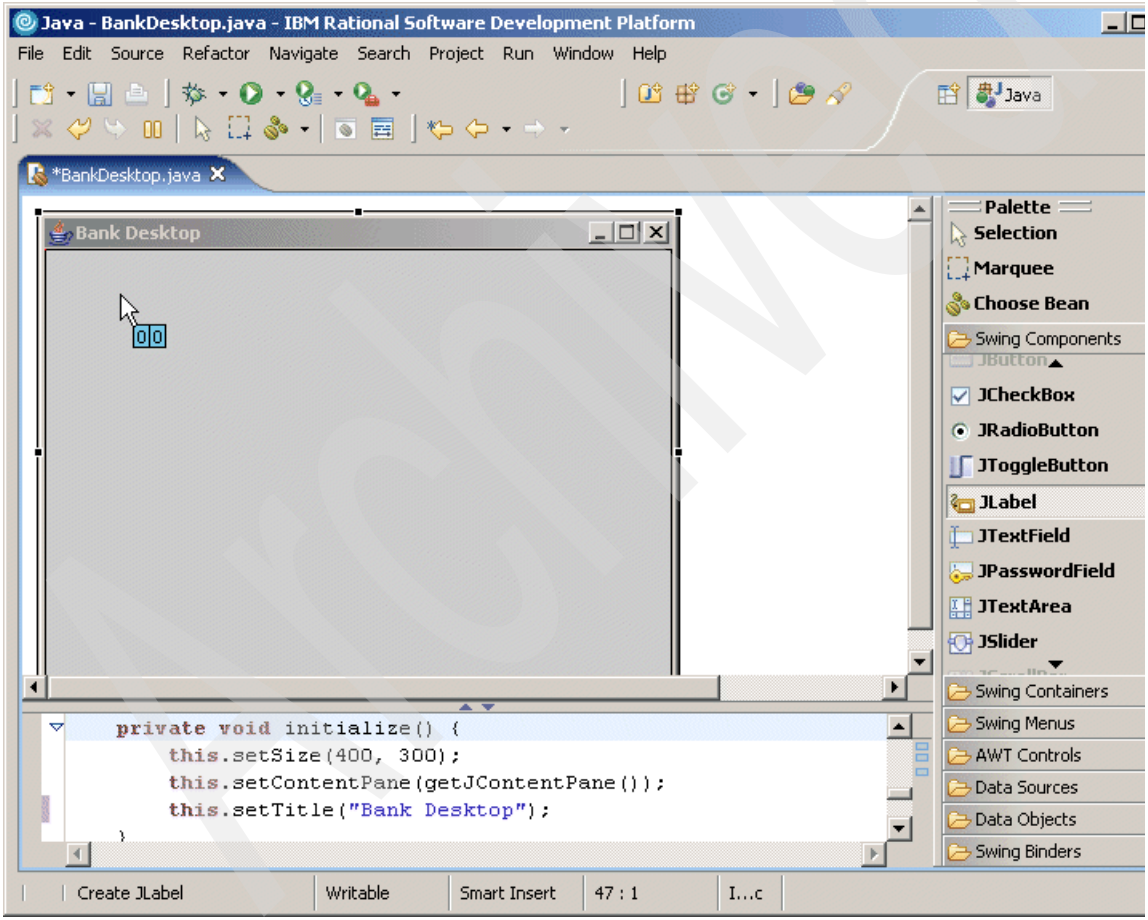


Figure 9-8 Positioning Swing components

5. Select **JTextField** from the palette and drop it to the right of the JLabel.

When the mouse pointer is in the correct position the coordinates should be `(1,0)`—indicating column 1, row 0.

6. Add a JButton to the right of the JTextField at (2,0). A yellow bar will appear in the correct position when you have the mouse pointer placed correctly.
7. Add three additional JLabels below the first one at (0,1), (0,2), and (0,3).
8. Add `JTextFields` at (1,1) and (1,2)—to the right of the first two new JLabels, and a `JList` at (1,3).

   The interface should now appear as shown in Figure 9-9.

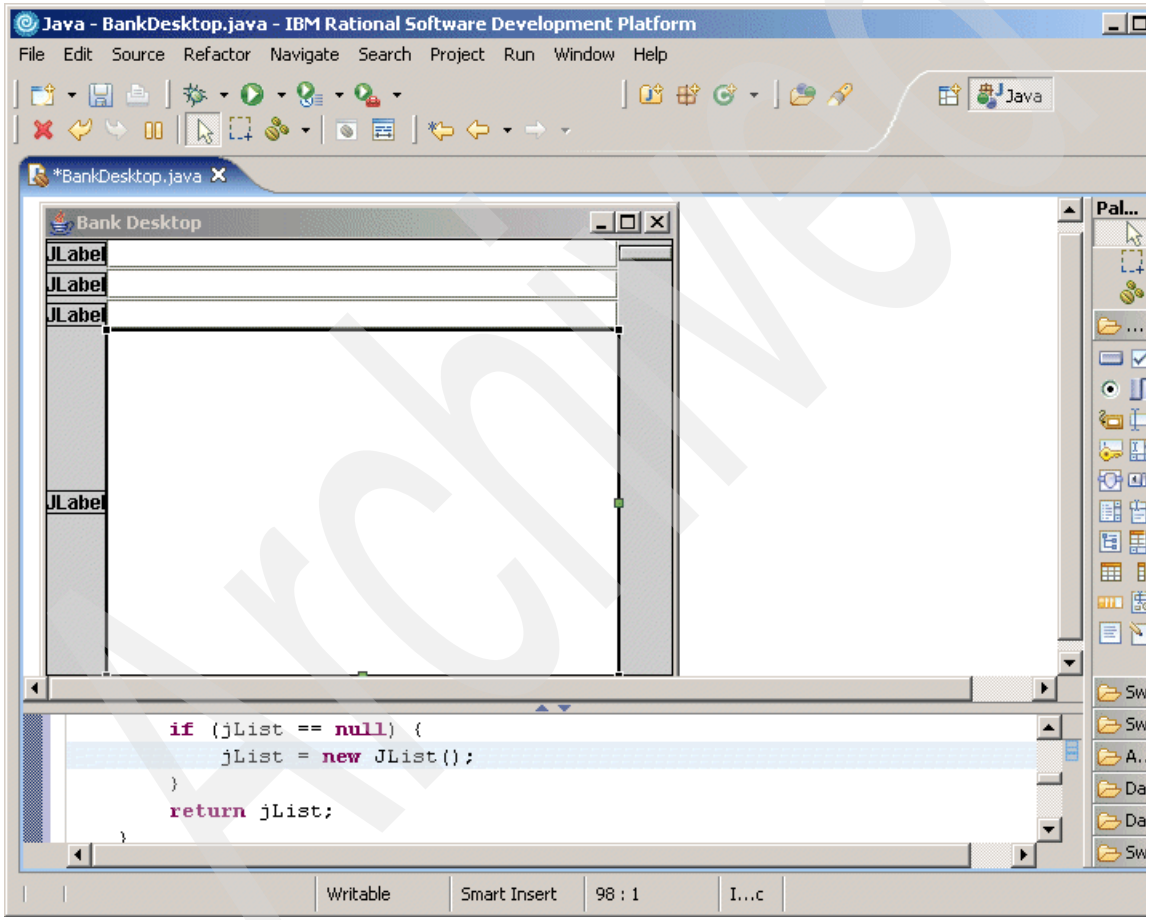


Figure 9-9 Basic GUI layout

At the moment we have an unused space at the bottom-right corner of our GUI. We need to change the width of some of the other components to fill this space.

9. Select the **JTextField** at (1,1) (look at constraint property to see the x and y values in the Property view) and drag the green control handle to the right so that the field spans two columns, as shown in Figure 9-10.
10. Repeat this process with the JTextField at (1,2) and the JList at (1,3).

    The empty space in the GUI should now be filled up.

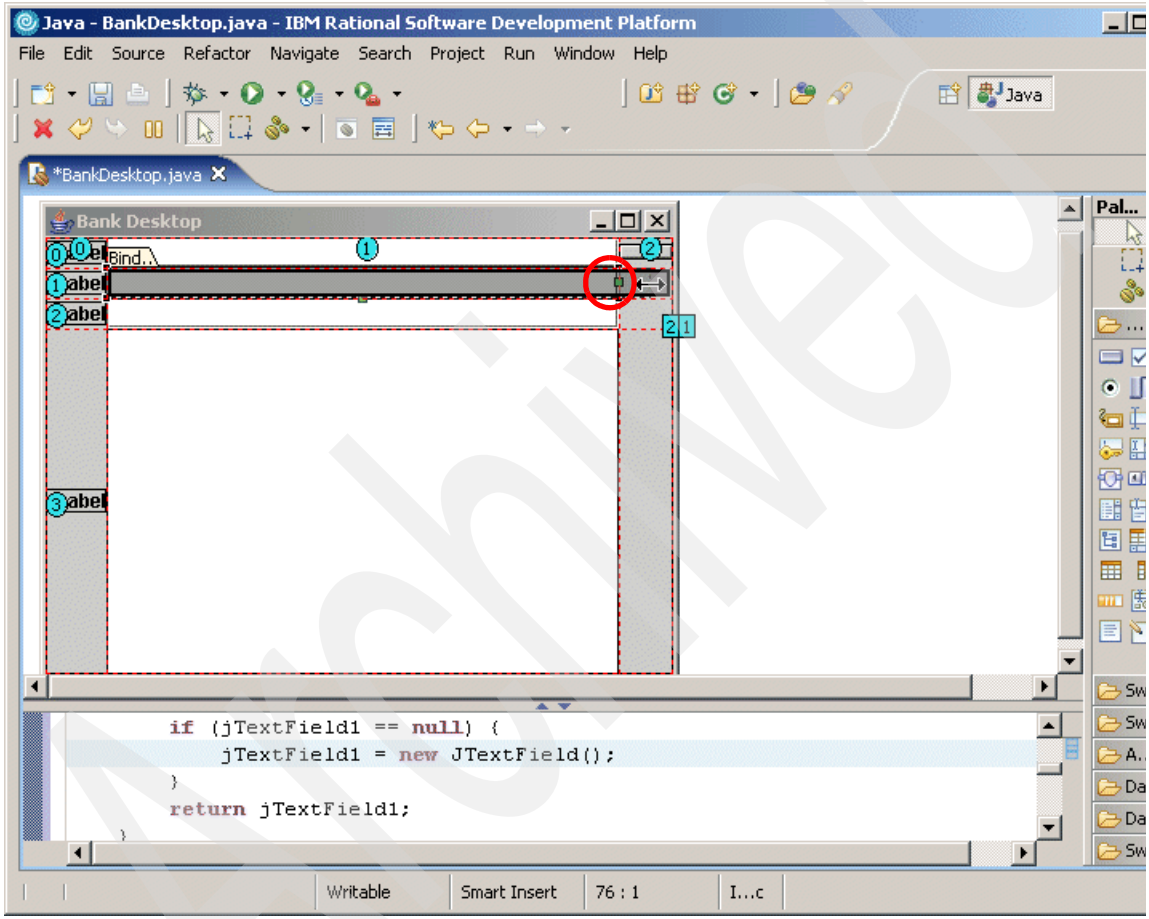


Figure 9-10 Resizing components within the layout

11. Click the top **JLabel** to select it, and then click it again to make the text editable. Change it to `Search SSN`.
12. Change the other JLabel text entries in the same way to `Customer Name`, `Social Security`, and `Accounts`.
13. Add some text to the JButton in the same way. Click once to select, then a second time to edit the text. Enter `Lookup` as the name of the JButton.

At this stage the GUI should look like Figure 9-11.

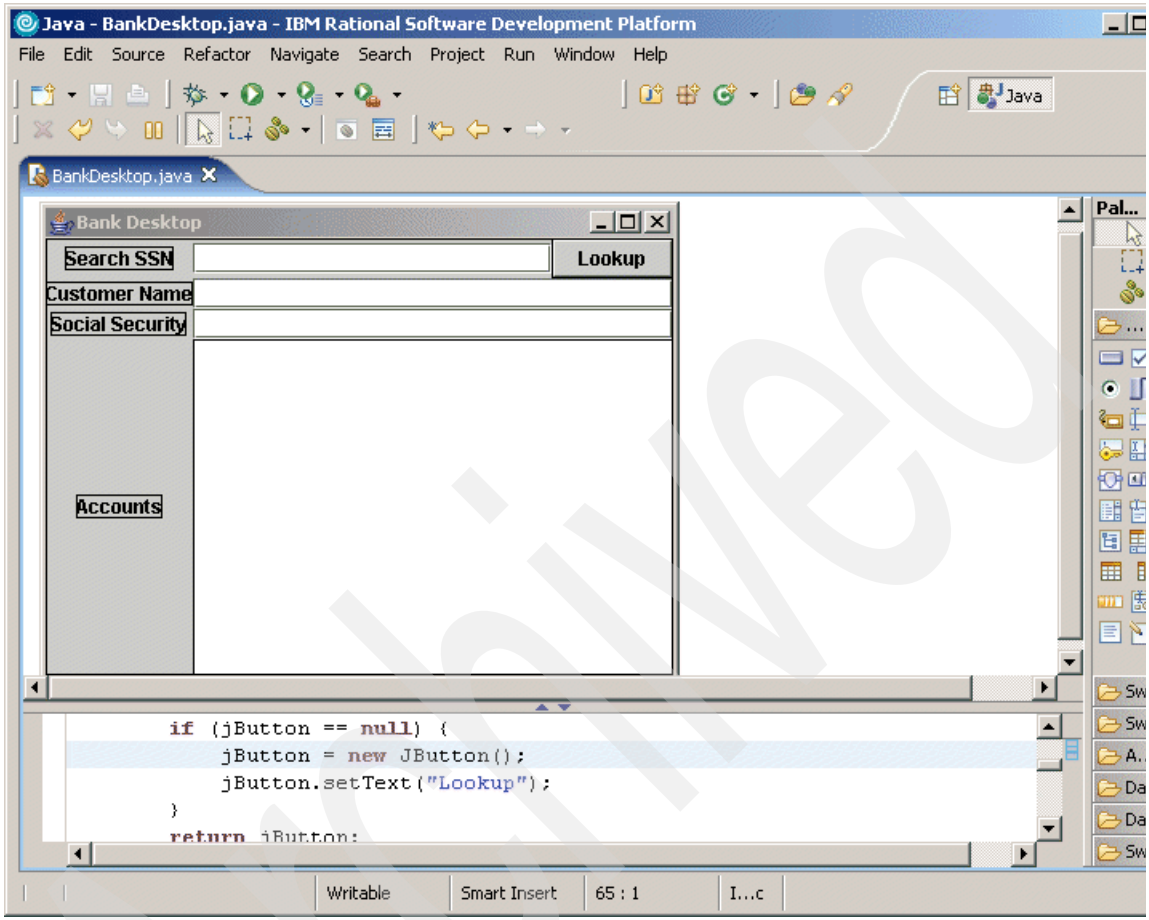


Figure 9-11 Changed text labels

9.5.5 Work with the Properties view

It would be better if the JLabels were all lined up along one edge. We are going to line them all up along their right edges using the properties view.

1. Select the **Search SSN** JLabel and in the Properties view find the constraint property.
2. Expand the property and change the anchor value to `EAST`.

   Figure 9-12 on page 433 shows this.

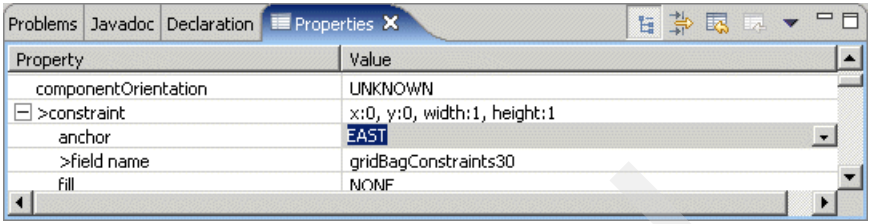


Figure 9-12 Changing the anchor property

3. Do the same thing with the Customer Name and Social Security JLabels.
4. Change the anchor property for the Accounts JLabel to `NORTHEAST`.

   Although it is not noticeable at this stage, the font of the JList is different from the font for the JTextFields.
5. Select the **JList**, select the **font** property, and click the **...** button to the right of the value field.
6. In the pop-up Java Property Editor, change the Style from bold to plain and then click **OK**.
7. Save your changes (Ctrl+S or **File** → **Save**).

9.5.6 Testing the appearance of the GUI

As a quick test of the appearance of the GUI, we can run the JFrame as it is. Select **Run** → **Run As** → **Java Bean**. The GUI should start up and appear as a separate window, looking something like Figure 9-13.

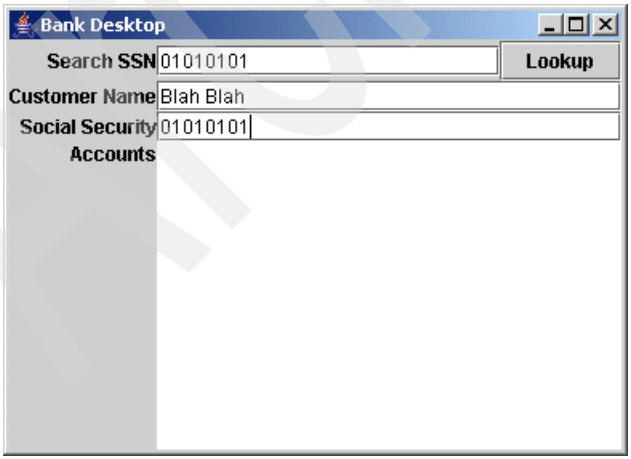


Figure 9-13 First GUI test

Archived

The GUI should work up to a point—you should be able to type into the text fields and click the Lookup button—but we have not yet coded any event handling behavior.

Close the GUI by clicking the close box at the top-right.

9.5.7 Add event handling to GUI

For our example, we want the user to be able to enter a search term into the Search SSN field (a social security number for a customer) and click the **Lookup** button. When the button is clicked, we need to search the database for the customer details, find the accounts for the customer, and put the account information in the list at the bottom of our GUI.

Add event handling

To add event handling to the GUI, do the following:

1. Register the JFrame as an ActionListener for the JButton.

   The first stage is to register our JFrame as an ActionListener for our JButton so that it will receive messages when the button is pressed.

   The best place to do this is in the initialize method of BankDesktop.java, as seen in Example 9-3.

Example 9-3 Add addActionListener sample

```
private void initialize() {
   this.setSize(400, 300);
   this.setContentPane(getJContentPane());
   this.setTitle("Bank Desktop");
   this.getJButton().addActionListener(this);
}
```

2. Add code to the actionPerformed method.

   Since we chose to implement the ActionListener interface when we created this class, the required method is already present, although at the moment it does not do anything—that is the next step.

   The code for the actionPerformed method in BankDesktop should look like Example 9-4.

Example 9-4 Add actionPerformed method

```
public void actionPerformed(ActionEvent e) {
   String searchSsn = getJTextField().getText();
   CustomerFactory factory = new CustomerFactory();
   try {
      Customer customer = factory.getCustomer(searchSsn);
```

```
            getJTextField1().setText(customer.getFullName());
            getJTextField2().setText(customer.getSsn());
            getJList().setListData(customer.getAccounts());
        } catch (SQLException e1) {
            getJTextField1().setText("<SSN not found>");
            getJTextField2().setText("");
            getJList().setListData(new Vector());
            e1.printStackTrace();
        }
    }
```

3. Add the appropriate import statements (Ctrl+Shift+O).
4. Save the file (Ctrl+S).

Note: The generated code will use simple names for the components you add, based on the type name with an incrementing suffix (jTextField, jTextField1, and so on). Components can be renamed to give them a more meaningful name by right-clicking the component in the Design or Java Beans view and selecting **Rename Field**, or by changing the field name property for the component. The accessor method for the component will automatically be renamed and any references will automatically be updated.

Update the imported model class database location

The event handling code described in this section uses the back-end code we imported in 9.2.4, “Import the model classes for the sample” on page 419. We will need to modify the DatabaseManager class so that it points to the correct database.

1. From the Package Explorer, expand **BankGUI** → **itso.bank.model.entity.java** → **DatabaseManager.java**.
2. Select the **DatabaseManager** class from the Project Explorer.
3. Change the URL string for the database in the getConnection method to reflect the actual location of the database. For example, "jdbc:db2j:C:\\databases\\BANK" if you have used the location suggested in 9.2.3, “Set up the sample database” on page 418.
4. Save the file (Ctrl+S).

9.5.8 Verify the Java GUI application

To verify that the Java GUI application is working properly, do the following:

1. Ensure that no connections to the BANK database are active. If they are, disconnect prior to running the Java GUI application.

2. Click the **Run** button in the toolbar (or select **BankDesktop.java**, right-click **Run** → **Java application**). If you hover the mouse pointer over it you will see that the tooltip text reads `Run BankDesktop`.
3. Enter a valid social security number into the Search SSN field. For example, we entered 111-11-1111, which is valid in our sample data set. For more information on the sample data, either query the BANK database or look at the loadData.sql used to populate the sample data.
4. Click **Lookup** and the program will search the database for the requested value.

   The results are shown in Figure 9-14.

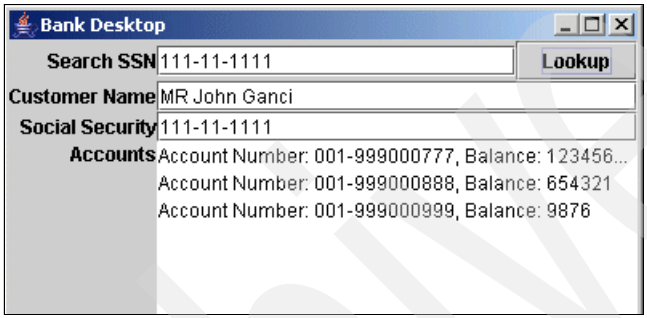


Figure 9-14 Testing the working GUI

5. Try entering an invalid value. In the Console view, you will see an error message for the invalid value entered as well as an exception stack trace.

9.5.9 Run the sample GUI as a Java application

To run the sample GUI as a Java application, we need to add code to the main method.

1. Change the main method in BankDesktop, which we asked to be generated when we created the class, as seen in Example 9-5.

Example 9-5 BankDesktop method

```
public static void main(String[] args) {
   BankDesktop instance = new BankDesktop();
   instance.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);
   instance.setVisible(true);
}
```

2. Run the BankDesktop as a Java application by selecting **Run** → **Run As** → **Java Application**. The results should be the same as in 9.5.8, “Verify the Java GUI application” on page 435.

9.5.10 Automatically add event handling

The Visual Editor provides a mechanism for generating the code necessary for handling component events.

To demonstrate the automatic adding of event handling, we will first remove the existing event handling code.

1. Select all the code in the existing actionPerformed method and cut it (right-click and select **Cut**, or, Ctrl+X, or **Edit** → **Cut**).
2. Delete the actionPerformed method from BankDesktop (this will produce a syntax error, which we will now fix).
3. Delete implements ActionListener from the class definition. It should now read:

```
public class BankDesktop extends JFrame {
```

4. Remove the line from the initialize method that registers BankDesktop as a button listener. The initialize method should now be like Example 9-6.

Example 9-6 Initialize method

```
private void initialize() {
   this.setSize(400, 300);
   this.setContentPane(getJContentPane());
   this.setTitle("Bank Desktop");
}
```

All the errors should now have disappeared.

5. In the Design view, right-click the **Lookup JButton** and select **Events** → **actionPerformed**.

   This adds the addActionListener method call to the getJButton method and codes an anonymous inner class to implement the ActionListener interface. We must now add code to this anonymous inner class to make it work.

6. Delete the line of code that is automatically generated in the actionPerformed method (a System.out.println call).
7. Paste the code you previously cut from the actionPerformed method into this new actionPerformed method. The getJButton method should now look like Example 9-7.

Example 9-7 getJButton method

```
private JButton getJButton() {
   if (jButton == null) {
      jButton = new JButton();
      jButton.setText("Lookup");
      jButton.addActionListener(new java.awt.event.ActionListener() {
```

Archived

```
                public void actionPerformed(java.awt.event.ActionEvent e) {
                    String searchSsn = getJTextField().getText();
                    CustomerFactory factory = new CustomerFactory();
                    try {
                        Customer customer = factory.getCustomer(searchSsn);
                        getJTextField1().setText(customer.getFullName());
                        getJTextField2().setText(customer.getSsn());
                        getJList().setListData(customer.getAccounts());
                    } catch (SQLException e1) {
                        getJTextField1().setText("<SSN not found>");
                        getJTextField2().setText("");
                        getJList().setListData(new Vector());
                        e1.printStackTrace();
                    }
                }
            });
        }
        return jButton;
    }
```

8. Test the GUI as described in 9.5.8, “Verify the Java GUI application” on page 435.

9.5.11 Visual Editor binding

The Visual Editor can perform some automatic code generation for binding the properties of certain components to data source objects. The data source objects can be Enterprise JavaBeans (EJBs), Web services, or ordinary JavaBeans.

1. Select the **Customer Name** text field.
2. Click the yellow **Bind..** button above the field, as seen in Figure 9-15.

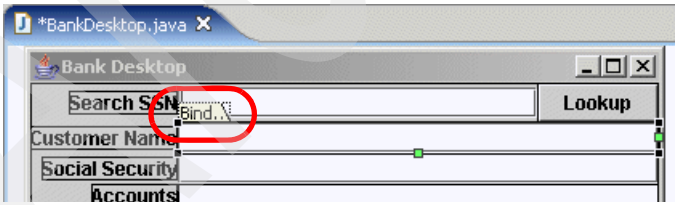


Figure 9-15 The Bind.. button

3. Click the **New Data Source Data Object** button.
4. When the New Data Source Data Object window appears, enter the following (as seen in Figure 9-16), and then click **OK**:
   - Name: `CustomerDataObject`

- Source type: Select **Java Bean Factory** (default).
- Data source: Click **New...** next to Data Source, enter `CustomerFactory` into the search text field, and click **OK**.
- Source service: Select **getCustomer**.
- Argument: Select **jTextField**.
- Property: Select **text**.

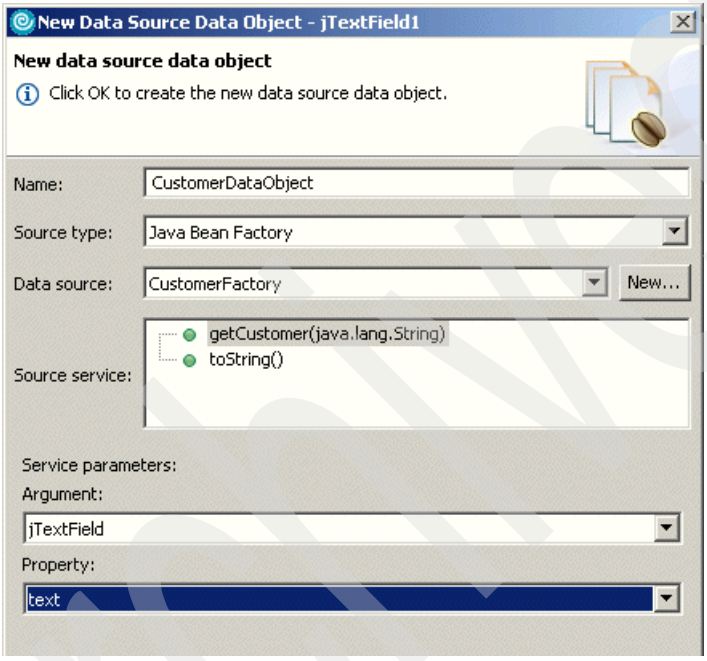


Figure 9-16 New Data Source Data Object dialog

5. Select **fullName** from the Data object properties list, as shown in Figure 9-17 on page 440.
6. Click **OK**.

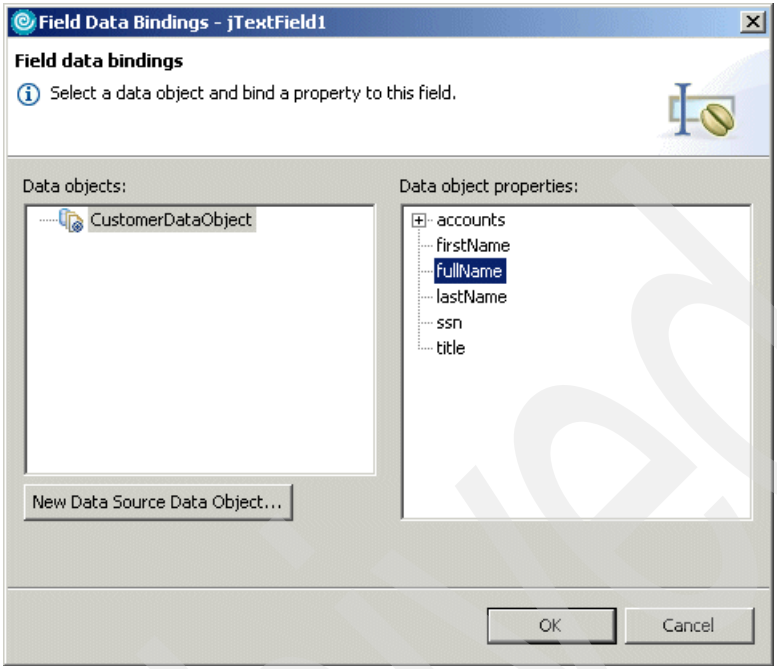


Figure 9-17 Field Data Bindings dialog

7. Select the **Social Security** text field and click its **Bind..** button.
8. Select **ssn** from the Data object properties list.
9. Click **OK**.
10. Unfortunately, only JTextField, JTable, JTextArea, or JButton can be bound to data source objects in this way, so we cannot use this technique for our JList.

    The getJButton method can now be changed as seen in Example 9-8.

Example 9-8 getJButton method modifications

```
private JButton getJButton() {
   if (jButton == null) {
      jButton = new JButton();
      jButton.setText("Lookup");
      jButton.addActionListener(new java.awt.event.ActionListener() {
         public void actionPerformed(java.awt.event.ActionEvent e) {
            getCustomerDataObject().refresh();
            String searchSsn = getJTextField().getText();
            CustomerFactory factory = new CustomerFactory();
            try {
               Customer customer = factory.getCustomer(searchSsn);
```

Archived

```
                    getJList().setListData(customer.getAccounts());
                } catch (SQLException e1) {
                    getJList().setListData(new Vector());
                    e1.printStackTrace();
                }
            }
        });
    }
    return jButton;
}
```

The main change is highlighted in bold. The code needed to set the text of the name and social security fields is no longer required—it is essentially replaced by the call to getCustomerDataObject().refresh(), which now handles the synchronization for us.

Archived

Archived

10

Develop XML applications

This chapter introduces the XML capabilities provided by Rational Application Developer and describes how to use the XML tooling.

The chapter is organized into the following topics:

- XML overview and technologies
- Rational Application Developer XML tools
- Where to find more information

Note: The sample code described in this chapter can be completed by following along in the procedures documented. Alternatively, you can import the sample Java code provided in the c:\6449code\xml\BankXMLWeb.zip Project Interchange file. For details refer to Appendix B, “Additional material” on page 1395.

© Copyright IBM Corp. 2005. All rights reserved.

10.1 XML overview and technologies

eXtensible Markup Language (XML) is a subset of Standard Generalized Markup Language (SGML). Both XML and SGML are meta languages, as they allow the user to define their own tags for elements and attributes.

XML is a key part of the software infrastructure. It provides a simple and flexible means of defining, creating, and storing data. XML is used for a variety of purposes ranging from configuration to messaging to data storage.

By allowing the user to define the structure of the XML document, the rules that define the structure can also be used to validate a document to ensure they conform.

Note: For more detailed information and examples on developing XML/XSL applications using Application Developer, refer to the redbook *The XML Files: Development of XML/XSL Applications Using WebSphere Studio Version 5*, SG24-6586.

10.1.1 XML and XML processor

XML is tag-based; however, XML tags are not predefined in XML. You have to define your own tags. XML uses a document type definition (DTD) or an XML schema to describe the data.

XML documents follow strict syntax rules. For more information regarding XML consult the W3C web site:

`http://www.w3.org/XML`

See the following web site for XML syntax rules:

`http://www.w3.org/TR/rdf-syntax-grammar/`

To create, read, and update XML documents, you need an XML processor or parser. At the heart of an XML application is an XML processor that parses an XML document, so that the document elements can be retrieved and transformed into a presentation understood by the target client. The other responsibility of the parser is to check the syntax and structure of the XML document.

Archive

Where to find more information

Refer to the following for more information on XML and XSLT parsers:

- Xerces (XML parser - Apache)

  http://xml.apache.org/xerces2-j

- Xalan (XSLT processor - Apache)

  http://xml.apache.org/xalan-j

- Crimson (XML parser)

  http://xml.apache.org/crimson/

Refer to the following for more information on XML parser APIs:

- JAXP (XML parser - Sun)

  http://java.sun.com/xml/jaxp

- SAX2 (XML API)

  http://sax.sourceforge.net

10.1.2 DTD and XML schema

Document Type Definitions (DTDs) and XML schemas are both used to describe structured information; however, in recent years the acceptance of XML schemas has gained momentum. Both DTDs and schemas are building blocks for XML documents and consist of elements, tags, attributes, and entities. Both define the rules by which an XML document must conform.

An XML schema is more powerful than DTD. Here are some advantages of XML schemas over DTDs:

- They define data types for elements and attributes, and their default and fixed values. The data types can be of string, decimal, integer, boolean, date, time, or duration.
- They apply restrictions to elements, by stating minimum and maximum values (for example, on age from 1 to 90 years), or restrictions of certain values (for example, redbooks, residencies, redpieces with no other values accepted, such as in a drop-down list). Restrictions can also be applied to types of characters and their patterns (for example, only accepting values ‘a’ to ‘z’ and also specifying that only three letters can be accepted). The length of the data can also be specified (for example, passwords must be between 4 and 8 characters).
- They specify complex element types. Complex types may contain simple elements and other complex types. Restrictions can be applied to the sequence and the frequency of their occurrences. These complex types can then be used in other complex type elements.

Archived

- Since schemas are written in XML, they are also extensible. This also implies that the learning curve for learning another language has been eliminated. The available parsers need not be enhanced. Transformation can be carried out using XSLT, and its manipulation can be carried out using XML DOM.
- With XML schemas being extensible, they can be re-used in other schemas. Multiple schemas can be referenced from the same document. In addition, they have the ability to create new data types from standard data types.

For more information on XML schema requirements:

http://www.w3.org/TR/NOTE-xml-schema-req

Note: DTDs consists of elements such as text strings, text strings with other child elements, and a set of child elements. DTDs offer limited support for types and namespaces, and the syntax in DTDs is not XML.

10.1.3 XSL and XSLT

The eXtensible Style Language (XSL) is a language defined by the W3C for expressing style sheets.

XSL has the following three parts:

- XSL transformations (XSLT): Used for transforming XML documents
- XML path language (XPath): Language used to access or refer to parts of an XML document
- XSL-FO: Vocabulary for specifying formatting semantics

A transformation in XSLT must be a well-formed document and must conform to the namespaces in XML, which can contain elements that may or may not be defined by XSLT. XSLT-defined elements belong to a specific XML namespace. A transformation in XSLT is called a style sheet.

XSL uses an XML notation and works on two principles—pattern matching and templates. XSL operates on an XML source document and parses it into a source tree. It applies the transformation of the source tree to a result tree, and then it outputs the result tree to a specified format. In constructing the result tree, the elements can be reordered or filtered, and other structures can be added. The result tree can be completely different from the source tree.

10.1.4 XML namespaces

Namespaces are used when there is a need for elements and attributes of the same name to take on a different meaning depending on the context in which

they are used. For instance, a tag called <TITLE> takes on a different meaning, depending on whether it is applied to a person or a book.

If both entities (a person and a book) need to be defined in the same document (for example, in a library entry that associates a book with its author), we need some mechanism to distinguish between the two and apply the correct semantic description to the <TITLE> tag whenever it is used in the document.

Namespaces provide the mechanism that allows us to write XML documents that contain information relevant to many software modules.

10.1.5 XPath

The XML path language (XPath) is used to address parts of an XML document. An XPath expression can be used to search through an XML document, and extract information from the nodes (any part of the document, such as an element or attribute) in it.

There are four different kinds of XPath expressions:

- Boolean: Expression type with two possible values
- Node set: Collection of nodes that match an expression's criteria, usually derived with a location path
- Number: Numeric value, useful for counting nodes and for performing simple arithmetic
- String: Text fragment that may come from the input tree, processed or augmented with general text

An XPath expression returns a certain node set, which is a collection of nodes. The following is a sample XPath expression:

```
/ACCOUNT_TABLE/ACCOUNT/ACCID
```

This expression selects any elements named ACCID (account ID), which are children of ACCOUNT elements, which are children of ACCOUNT_TABLE elements, which are children of the document root.

10.2 Rational Application Developer XML tools

Rational Application Developer provides a comprehensive visual XML development environment. The tool set includes components for building DTDs, XML schemas, XML, and XSL files.

Rational Application Developer includes the following XML development tools:

- DTD editor
- XML schema editor
- XSL editor
- XML editor
- XPath expression wizard
- XML to XML mapping editor
- XSL debugging and transformation
- XML and relational data
- Relational database to XML mapping (visual DAD builder)

This chapter covers only a few XML tools of Rational Application Developer. We demonstrate how to create XML files, and we introduce you to some of Rational Application Developer's XML generators.

The following topics are discussed:

- Create a project for XML sample.
- Work with DTD files.
- Work with XML schema files.
- Work with XML files.
- Work with XSL files.
- Transform an XML file.
- Java code generation.

Note: We do not create a fully XML-enabled application here. This chapter only shows some XML capabilities of Rational Application Developer where we create and work with XML files.

Refer to Rational Application Developer online help for more detailed information regarding XML tools and editors.

10.2.1 Create a project for XML sample

To demonstrate some of Rational Application Developer's XML tools, we set up a new project, create a new package, and add a new folder.

1. Create a Dynamic Web Project named BankXMLWeb.

   We created a new Dynamic Web Project named BankXMLWeb without adding to an EAR project (delete BankXMLWebEAR manually after creating the project).

   Refer to Chapter 11, "Develop Web applications using JSPs and servlets" on page 499, for details on creating a new Web Project.

2. Create a Java package named itso.xml.

Once you have created a Web Project, create a Java package and name it itso.xml.

Refer to 7.2.4, “Create Java packages” on page 246, for details on creating a package in Rational Application Developer.

3. Create a new folder in WEB-INF named xml (WEB-INF\xml).

   We also create a new folder in the WEB-INF folder and named the folder xml. The xml folder will be used to store an XML schema file.

   The new Web Project skeleton is displayed in the J2EE Navigator view, and should now look like Figure 10-1.

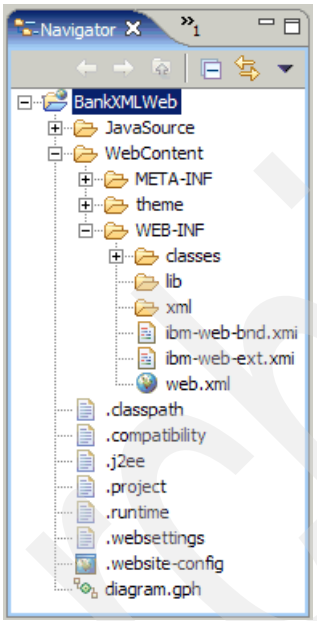


Figure 10-1 Sample project skeleton

10.2.2 Work with DTD files

In this section we demonstrate how to use the DTD editor by completing the following tasks:

- Create a DTD file.
- DTD editor features.
- Validate DTD files.

Archived

Create a DTD file

To create a new DTD file, use the new DTD wizard as follows:

1. Switch to the Web perspective, Package Explorer view.
2. Select the **BankXMLWeb** project.
3. Select **File** → **New** → **Other** (or press Ctrl+N).
4. When the New wizard appears, do the following (as seen in Figure 10-2):
   a. Check **Show All Wizards**.
   b. Expand **XML**.
   c. Select **DTD**.
   d. Click **Next**.

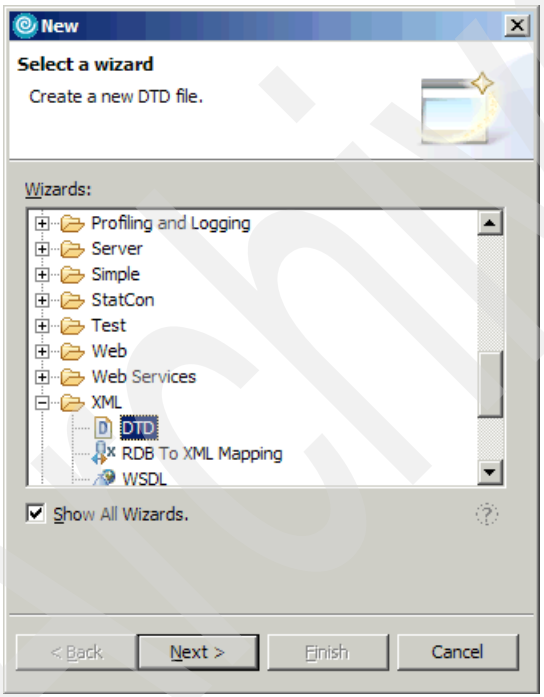


Figure 10-2 Create new dialog

5. When prompted with the message seen in Figure 10-3, check **Always enable capabilities and do not ask again**, and then click **OK**.

Archived

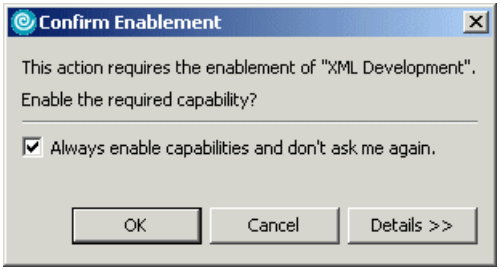


Figure 10-3 Enable XML development

6. When prompted, enter the file name `AccountTable.dtd`, select the **xml** folder (as seen in Figure 10-4), and then click **Finish**.

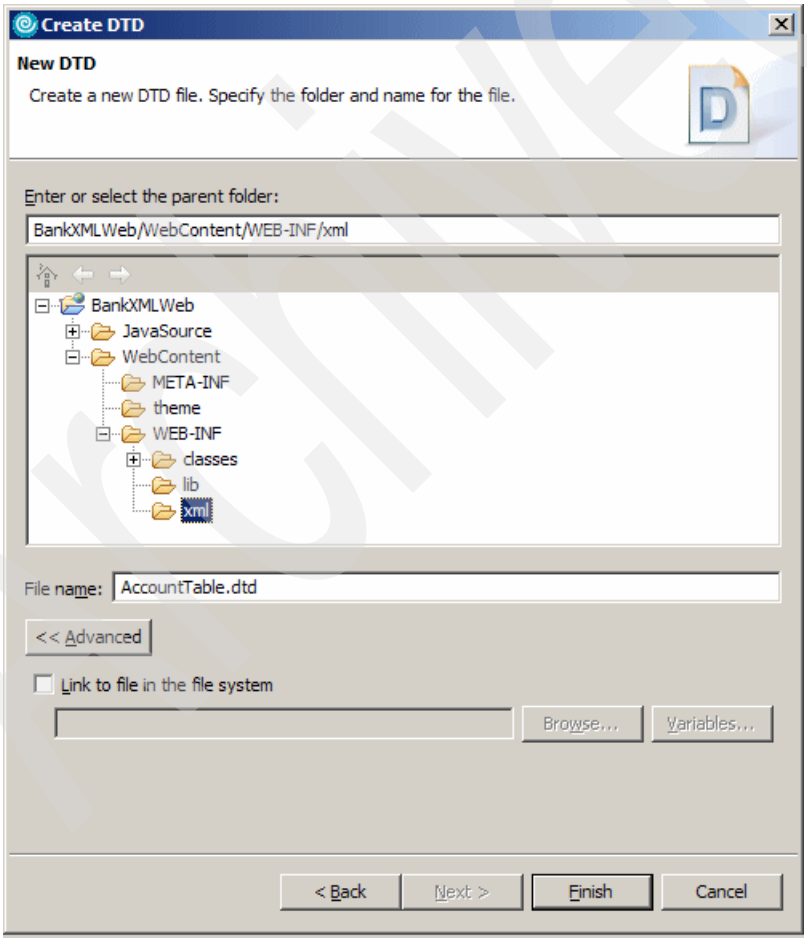


Figure 10-4 Create new DTD dialog

Archived

Note: IBM Rational Application Developer V6.0 now has the ability to link to external files on the file system. The Advance button on the Create DTD wizard provides the ability to specify the location on the file system that the DTD is linked to.

DTD editor features

The section highlights the following DTD editor features:

- Syntax highlighting preferences.
- Add items to a DTD file.
- Change the content type.
- Edit Model Group or Occurrence of and Element.
- Add entities.
- Add notations.
- Add comments.

Syntax highlighting preferences

To change the syntax highlighting preferences in the DTD editor, right-click the source editor, and select **Preferences ...** from the context menu. The relevant preference options will be displayed.

Views that are useful for editing DTD files are as follows:

- Outline view (Figure 10-5)

  The outline view has useful feature buttons on the top right of the view. The first button allows you to sort the items in alphabetical order, and the second allows you to group like items into a logical grouping. The third allows you to collapse all the items, and the fourth links the outline to the editor.

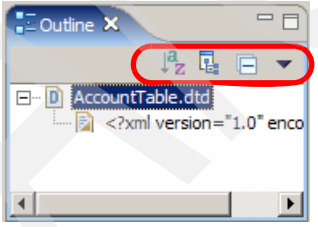


Figure 10-5 Outline view

- Properties view (Figure 10-6 on page 453)

  The properties view has two tabs—one for general information that is context sensitive; and the second for the description, which allows you to enter descriptive text about the item that you are editing. The description will

Archived

actually create a comment item above the item that the description is associated with.

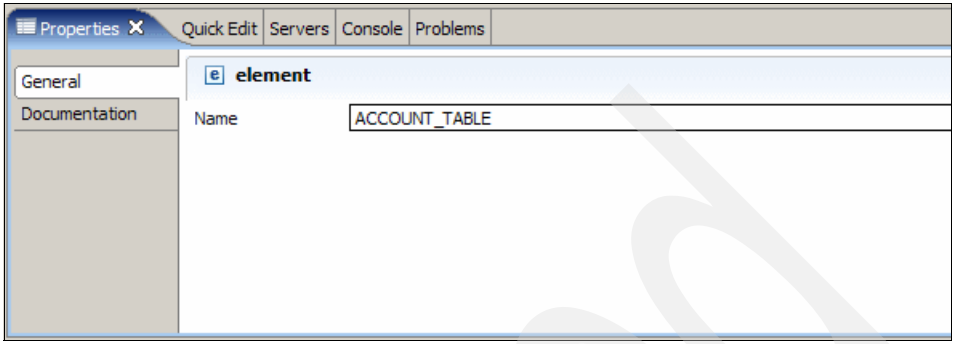


Figure 10-6 Properties view

Add items to a DTD file

To add new items such as Elements, Entities, Notations, Parameter Entity Reference, Comment, or Add to the attribute list to a DTD file, do the following:

1. Double-click **AccountTable.dtd** to open the file in the editor.
2. Add the item. Do one of the following:
   - Add the content by typing into the source editor.
   - Add via the context menu in the Outline view (see Figure 10-7).

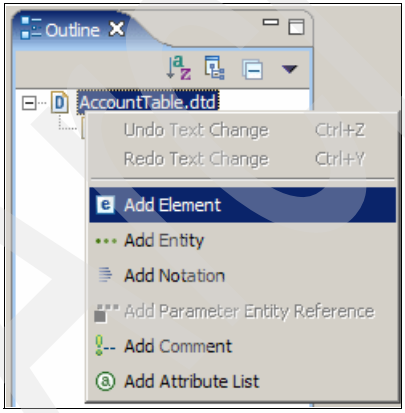


Figure 10-7 Adding items to a DTD file

When adding new elements, the Properties view will be presented with the new element selected. It will also create the new element with and EMPTY content type. The General tab on the Properties view allows you to change the name of the element.

Change the content type

To change the content type you must expand the element in the outline, and then select the EMPTY node or select EMPTY in the Source view.

The options to change the content type to are as follows:

- EMPTY
- PCDATA
- Children content
- Mixed content (First child will be PCDATA)
- Other elements

When you select the content type of either Children content or Mixed content, a grouping node is added to the Outline view and the Properties view will be changed to include Model Group and Occurrence.

- Model Group: Allows you to define the grouping of the Children content as follows:
  - Sequence: The editor will insert commas (,) between children.
  - Choice: The editor will insert pipe (|) between children.
- Occurrence: Allows you to define the number of times the child can occur in the element. The valid choices are as follows:
  - Just once - One and only one child.
  - One or more - One or more children.
  - Optional - This child may or may not exist.
  - Zero or more - Zero or more instances of this child.

Edit Model Group or Occurrence of and Element

To e**dit *Model Group* or *Occurrence* of a**n Element you must select the grouping node in the Outline view, as shown in Figure 10-8 on page 455.

Archived

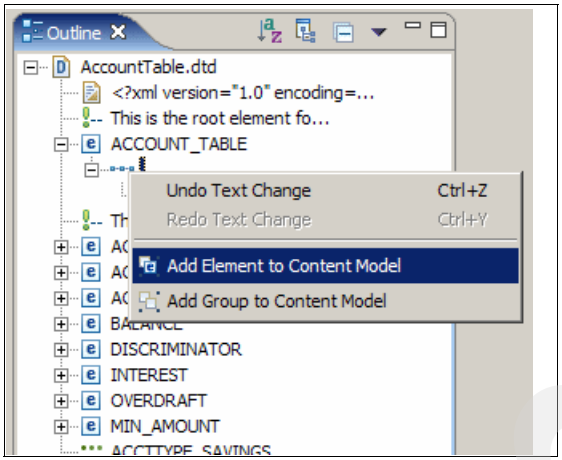


Figure 10-8 Select grouping node

Tip: When using the Context menu to add elements, it is easier to add all the elements first and then go back and change the content types.

To add more children to an element you can right-click the group node in the Outline view and select **Add Element to Content Model**, as shown in Figure 10-8. This will add a new child node, from which you can the select the appropriate child type.

Add entities

When adding new entities, the Properties view will be presented with the new entity selected. The Properties view will contain the following fields:

- Name - The name of the entity that must be unique
- Entity Type - General or Parameter (External or Internal)
- Entity Value - The value associated with the entity

Entities are used to define values that can be used in the XML file.

Add notations

When adding new notations, the Properties view will be presented with the new notation selected. The Properties view will contain the following fields:

- Name - The name of the notations that must be unique
- Public ID - The public identifier of the notations
- System ID - The system identifier of the notations

Archived

Notations are used to define additional processing instructions for unparsed entities.

Add comments

When adding new comments, the Properties view will be presented with a Comment window in the General tab. New comments will be added to the bottom of the source file.

A completed sample AccountTable.dtd file is listed in Example 10-1.

Note: Comments inserted above an element, entity, or notation will appear as the description for that item.

Example 10-1 Sample AccountTable.dtd

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- This is the root element for this document type.-->
<!ELEMENT ACCOUNT_TABLE (ACCOUNT)*>

<!-- This element defines a record in the account table.-->
<!ELEMENT ACCOUNT
(ACCT_ID,ACCT_TYPE,BALANCE,DISCRIMINATOR,INTEREST,OVERDRAFT,MIN_AMOUNT)>

<!ELEMENT ACCT_ID (#PCDATA)>
<!ELEMENT ACCT_TYPE (#PCDATA)>
<!ELEMENT BALANCE (#PCDATA)>
<!ELEMENT DISCRIMINATOR (#PCDATA)>
<!ELEMENT INTEREST (#PCDATA)>
<!ELEMENT OVERDRAFT (#PCDATA)>
<!ELEMENT MIN_AMOUNT (#PCDATA)>

<!ENTITY ACCTTYPE_SAVINGS "Savings">
<!ENTITY ACCTTYPE_LOAN "Loan">
<!ENTITY ACCTTYPE_FIXED "fixed">

<!--General comment for the document type definition - This document type is an
xml representation of the Account table.-->
```

Validate DTD files

To validate a DTD file, do the following:

1. Switch to the Web perspective.
2. Right-click the **AccountTable.dtd** file in the Navigator view.
3. In the Context menu select **Validate DTD file**.

- If validation was successful you will be presented with a pop-up window stating `Validation was successful.`
- If the file that you are trying to validate contains unsaved information, you will be prompted to save the file before validation. If you select **No** it will validate the previously saved version.
- If validation was not successful, you will be presented with a pop-up window stating that validation has failed. The source editor will also have a red cross next to the line that failed (see Figure 10-9), and the problem view will also have an entry with a reason why the validation failed (see Figure 10-10).

Figure 10-9 Error marker in the source editor

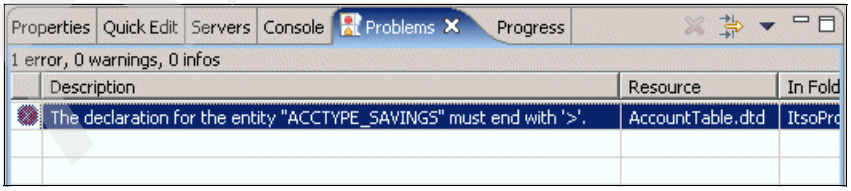


Figure 10-10 Problem view with description of problem in DTD file

10.2.3 Work with XML schema files

This section demonstrates how to use XML schema editor features to complete the following tasks:

- Generate an XML schema from an existing DTD file.
- Generate an XML schema from relational table definition.
- Generate an XML schema file from an XML file.
- Create a new XML schema.
- Edit an XML schema.
- Validate an XML schema.

Generate an XML schema from an existing DTD file

To generate an XML Schema file from an existing DTD file, do the following:

1. Switch to the Web perspective.
2. Right-click the DTD file in the Navigator view.

   In our example, we used the AccountTable.dtd supplied with the sample code and displayed in Example 10-1 on page 456.
3. Select **Generate** → **XML Schema**.
4. When the Generate XML Schema dialog appears, do the following (as seen in Figure 10-11 on page 459), and then click **Next**:
   - Select the folder in which you wish to place the new XML schema file (for example, xml).
   - File name: `AccountTable.xsd` (Note that XML schema files have an .xsd extension.)

Archived

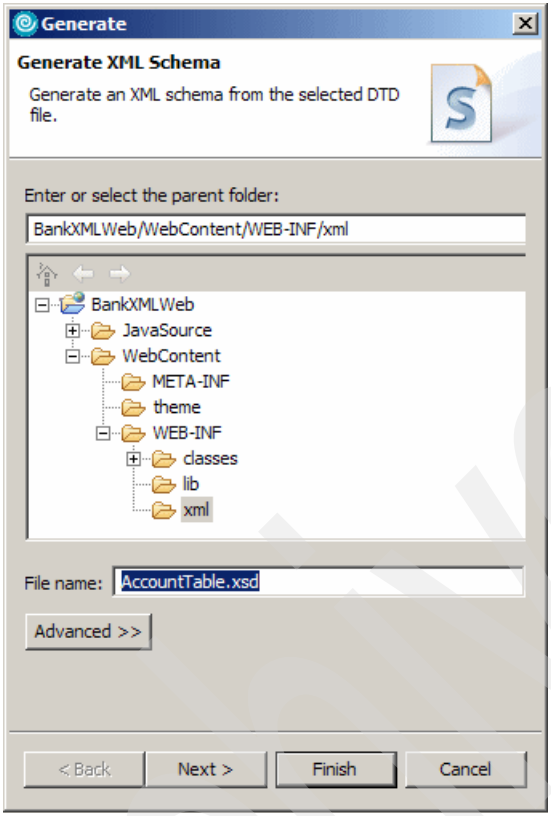


Figure 10-11 Generate new XML schema

5. When the XML Schema Generation Options dialog appears, select **Create one XML schema that includes all DTD files** (as seen in Figure 10-12 on page 460), and then click **Finish**.

   If you have a DTD file that references other DTD files, you can opt to create a single XML schema file or multiple XML schema files for each DTD file.

   You should see a message that the XML schema was successfully generated.

Archived

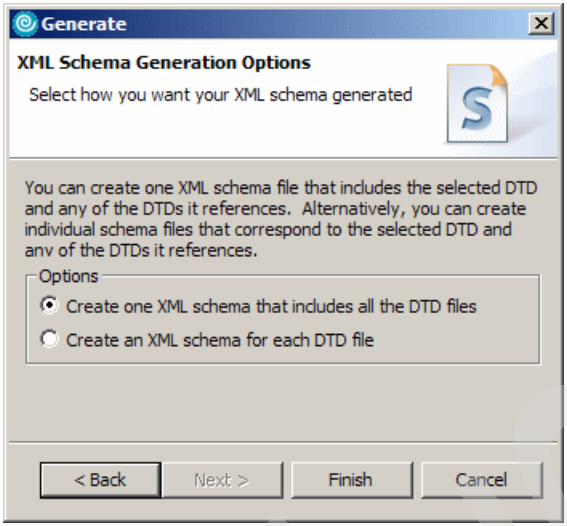


Figure 10-12 Creating multiple XML schema options

The XML schema AccountTable.xsd generated from the AccountTable.dtd can be seen in Example 10-2.

As you can see, the XML schema that is generated is suitable for simple validation; however, to actually take full advantage of XML schemas, the generated file should be modified to further refine the validation rules. In the previous version of Application Developer it was possible to generate DTD files from XSD files, but this was a backward step, as DTDs provide less functionality, so this feature has been removed.

Example 10-2 Sample XML schema AccountTable.xsd generated from AccountTable.dtd

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
    <xsd:element name="ACCOUNT_TABLE">
        <xsd:annotation>
            <xsd:documentation> This is the root element for this document
type.</xsd:documentation>
        </xsd:annotation>
        <xsd:complexType>
            <xsd:sequence>
                <xsd:element maxOccurs="unbounded" minOccurs="0" ref="ACCOUNT"/>
            </xsd:sequence>
        </xsd:complexType>
    </xsd:element>
    <xsd:element name="ACCOUNT">
        <xsd:annotation>
```

```
            <xsd:documentation> This element defines a record in the account
table.</xsd:documentation>
        </xsd:annotation>
        <xsd:complexType>
            <xsd:sequence>
                <xsd:element ref="ACCT_ID"/>
                <xsd:element ref="ACCT_TYPE"/>
                <xsd:element ref="BALANCE"/>
                <xsd:element ref="DISCRIMINATOR"/>
                <xsd:element ref="INTEREST"/>
                <xsd:element ref="OVERDRAFT"/>
                <xsd:element ref="MIN_AMOUNT"/>
            </xsd:sequence>
        </xsd:complexType>
    </xsd:element>
    <xsd:element name="ACCT_ID" type="xsd:string"/>
    <xsd:element name="ACCT_TYPE" type="xsd:string"/>
    <xsd:element name="BALANCE" type="xsd:string"/>
    <xsd:element name="DISCRIMINATOR" type="xsd:string"/>
    <xsd:element name="INTEREST" type="xsd:string"/>
    <xsd:element name="OVERDRAFT" type="xsd:string"/>
    <xsd:element name="MIN_AMOUNT" type="xsd:string"/>
</xsd:schema>
```

Generate an XML schema from relational table definition

To generate an XML schema file from a relational table definition, do the following:

1. Open the Data perspective.
2. Open the Data Definition view.
3. Right-click the table that you wish to create the XML schema base on.
4. Select **Generate XML Schema** from the context menu.
5. Select the folder in which you wish to place the new XML schema file.
6. Enter `generatedXMLSchemaFromTable.xml` in the Filename field.
7. Click **Finish**.

Generate an XML schema file from an XML file

To generate and XML schema file from a XML file, do the following:

1. Right-click the XML file and select **Generate** → **XML Schema ...** from the context menu.
2. Select the folder in which you wish to place the new XML schema file.
3. Enter a `generateXMLSchemaFromXML.xsd`.
4. Click **Finish**.

Archived

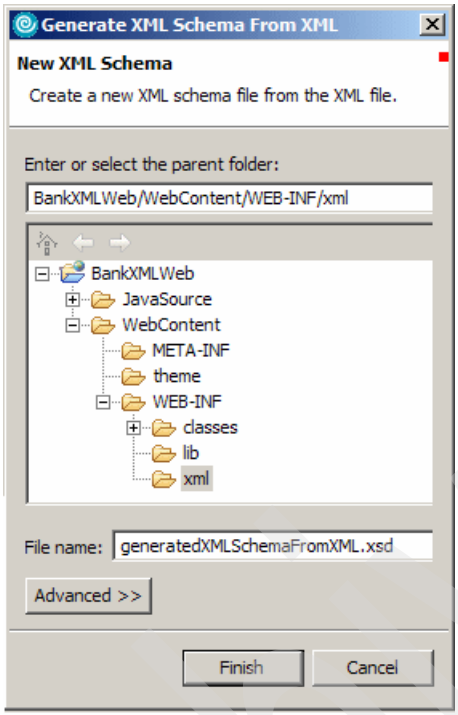


Figure 10-13 Generate an XML schema from an XML file

The sample generated file can be seen in Example 10-3.

Example 10-3 Sample generatedXMLSchemaFromXML.xsd file

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:tns="http://www.ibm.com/AccountTable"
   xmlns:xsd="http://www.w3.org/2001/XMLSchema"
   targetNamespace="http://www.ibm.com/AccountTable">
   <xsd:element name="ACCOUNT_TABLE">
      <xsd:complexType>
         <xsd:sequence>
            <xsd:element maxOccurs="unbounded" name="account">
               <xsd:complexType>
                  <xsd:sequence>
                     <xsd:element name="accountId"
                        type="xsd:string" />
                     <xsd:element name="accountType"
                        type="xsd:string" />
                     <xsd:element name="balance"
                        type="xsd:string" />
                     <xsd:element name="disclaiminator"
```

Archived

```
                        type="xsd:string" />
                    <xsd:element name="interest"
                        type="xsd:string" />
                    <xsd:element name="minAmount"
                        type="xsd:string" />
                    <xsd:element name="overdraft"
                        type="xsd:string" />
                </xsd:sequence>
            </xsd:complexType>
        </xsd:element>
    </xsd:sequence>
  </xsd:complexType>
 </xsd:element>
</xsd:schema>
```

Create a new XML schema

To create a new XML schema from scratch, use the new XML schema wizard, as follows:

1. Switch to the Web perspective.
2. Select **File** → **New** → **Other**.
3. When the Create new schema file wizard appears (as seen in Figure 10-14 on page 464), do the following:
   a. Check **Show All**.
   b. Expand the **XML** folder.
   c. Select **XML Schema** a
   d. Click **Next**.

Archived

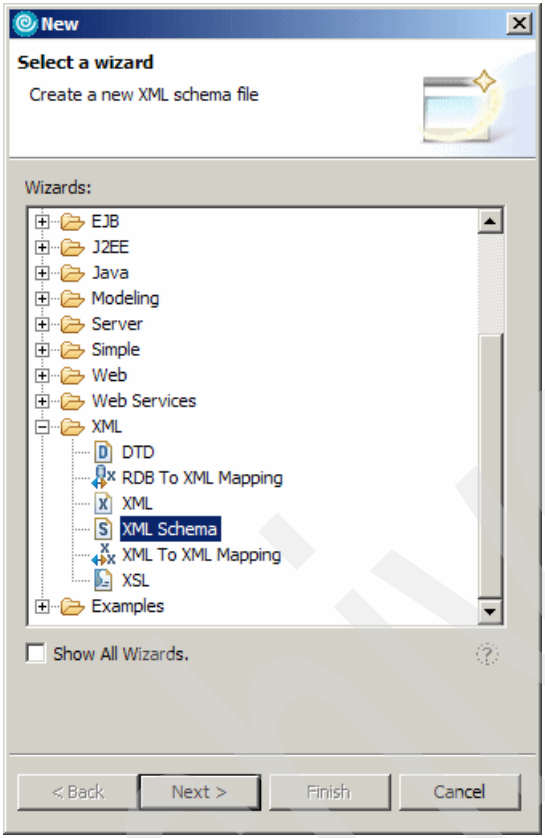


Figure 10-14 Create new XML schema file wizard

4. When the next XML schema dialog appears, do the following (as seen in Figure 10-15 on page 465):

   a. Select the folder in which you wish to place the new XML schema file.

   b. In the File name field enter `AccountTable.xsd`.

      If a file by this name already exists, either rename the existing file or change the file name to be a different name.

   c. Click **Finish**.

Archived

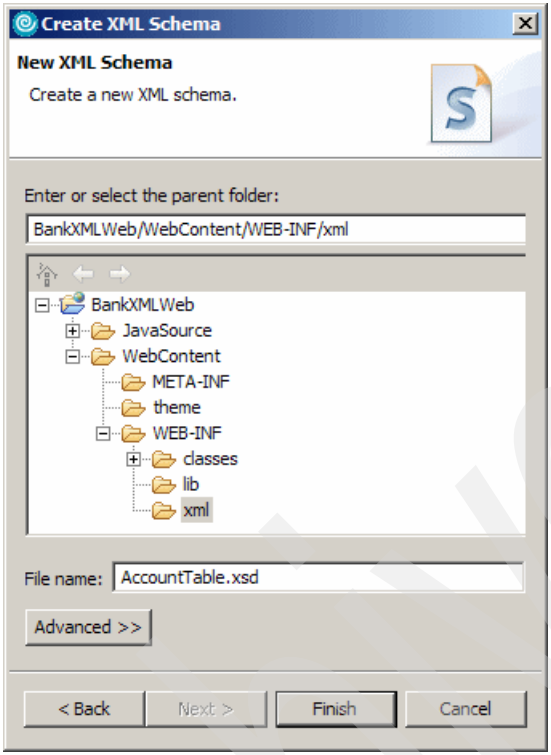


Figure 10-15 New XML schema wizard

Edit an XML schema

To add new elements, directives, simple types, complex types, and annotations to the XML schema, do the following after opening the file in the XML schema editor:

- Add content by typing in the source editor (see Figure 10-16 on page 466).
- Add via the context menu in the Outline view (see Figure 10-18 on page 468).
- Add via the context menu in the Graph pane of the editor (see Figure 10-17 on page 467).

The XML schema editor has two panes:

- XML Schema editor - Source pane
- XML Schema editor - Graph pane

XML Schema editor - Source pane

The source pane will allow you to edit the source directly (for example, see the source pane in Figure 10-16 on page 466).

Archived

The source editor provides for the following text editing features:

- Syntax highlighting
- Unlimited undo and redo of changes
- Node selection indicator
- Content Assist with Ctrl+Space

To change the syntax highlighting for the XML schema, do the following:

1. Right-click the source editor and select **Preferences**.
2. From the context menu, the associated preferences will be displayed. Expand the **Web and XML** folder, and expand the **XML Files** folder.
3. Select **XML Styles**.

   You can also change the formatting options within the XML Source options of the preferences.

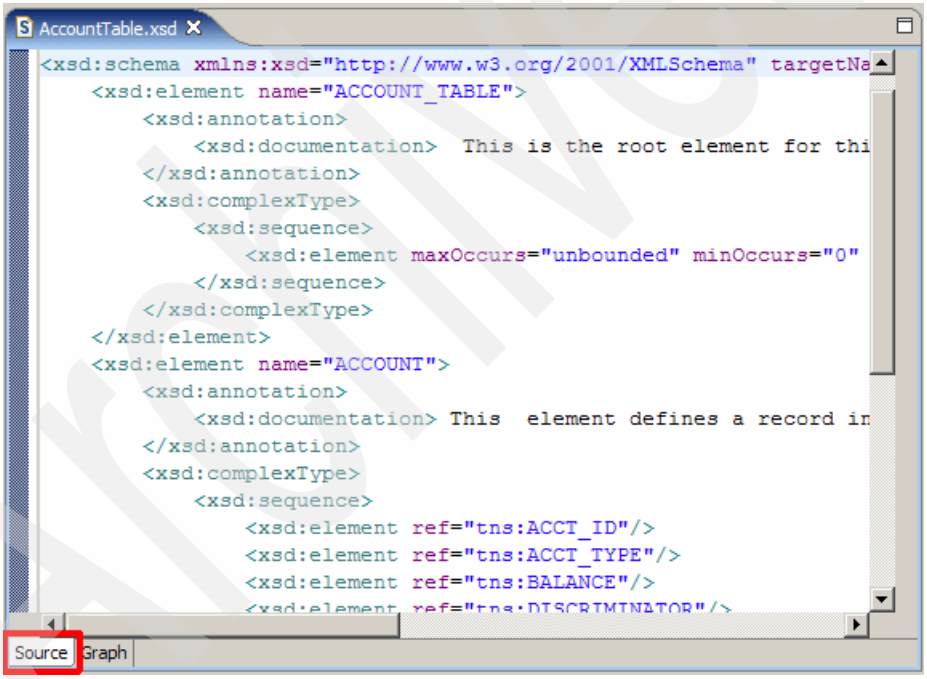


Figure 10-16 XML schema editor source pane

XML Schema editor - Graph pane

This pane will give you a graphical representation of the XML schema file. It also allows you to traverse the XML file and edit the items in the XML schema file. An example of the graph pane is shown in Figure 10-17 on page 467.

Archived

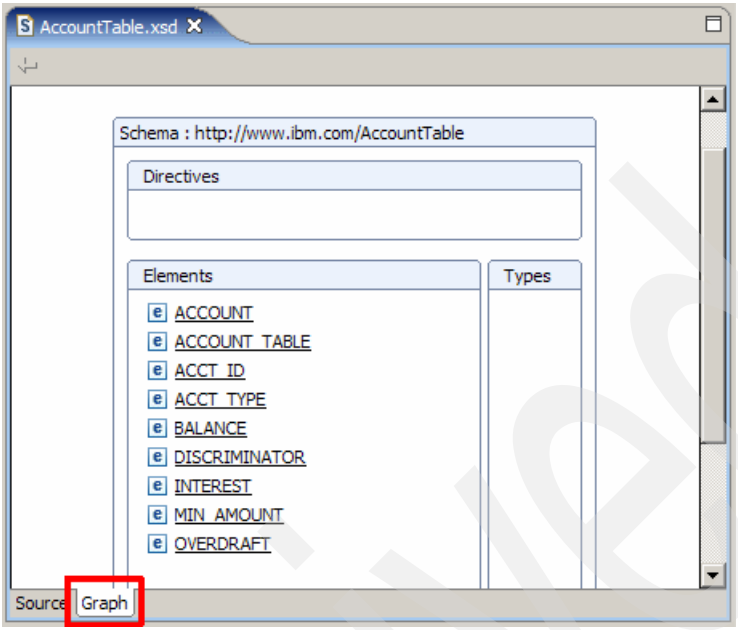


Figure 10-17 XML schema editor graph pane

XML Schema views

Views that are useful for editing XML schema file are:

- Outline view: This view presents a tree structure representation of all the categorized components of the XML schema, as shown in Figure 10-18 on page 468. This view contains:
  - Directive - Allows you to define include, import, and redefine statements
  - Elements - Allows you to define the elements in the schema
  - Attributes - Allows you to define global attributes
  - Attribute Groups - Allows you to define attribute groups
  - Type - Allows you to define *simple types* and *complex types* items in the schema
  - Groups - Allows you define groups of elements and element references
  - Notations - Allows you to define notations

Archived

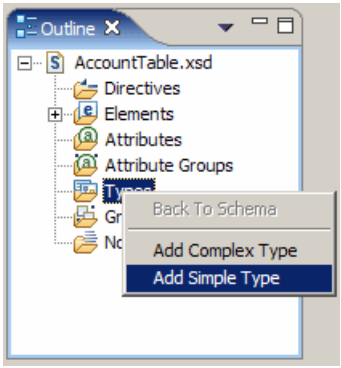


Figure 10-18 Outline view of XML schema

- Properties view - This view is used to edit the properties of the selected node.

  This view is context sensitive, and the tabs will change based on the type of node that you are currently editing.

When the XML schema node is selected, the following tabs are available:

- General - This tab allows you to define the prefix of the XML schema and the target namespace. The prefix and target namespace should be updated after generating an XML schema from a DTD.
- Other - This tab allows you to define various values for the XML schema file such as attributeFromDefault, blockDefault, elementFromDefault, version, and xml:lang.

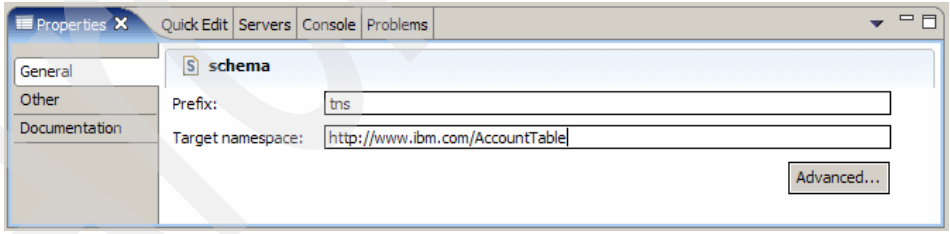


Figure 10-19 Changing the target namespace

While editing *simple types*, the following tabs are available:

- General - This tab allows you to change the common attributes of the simple type node such as Name, Variety, and Base type. The Variety field is limited to the list of valid values by a drop-down list box. To display more types to select from for the Base type you can click the More button to the right of the field; this will display a dialog of available types. The available types will only include built-in simple types and user-defined simple types.

Archived

- Enumeration - This tab allows you to define the list of possible values that this simple type node can possibly have.
- Documentation - This tab allows you to add documentation for the node. It will added the annotation and documentation tags into the node for you.
- Advance - This tab changes based on the base type of the node. It allows you to define various attributes for the node; as an example, if you define a base type of built-in simple type short, you will have the option to put restrictions on the totalDigits, fractionDigits, whiteSpace, maxInclusive, maxExclusive, minInclusive, and minExclusive.

We will change the elements of the account into user-defined simple type items rather than elements to take advantage of XML schema restrictions.

Add user-defined simple type

To add a user-define simple type, do the following:

1. Double-click **AccountTable.xsd** to open the file in the editor.
   a. Add the content by typing into the source editor.
   b. Add via the context menu in the source editor, as shown in Figure 10-18 on page 468.
   c. Add via the context menu in the graph editor, as shown in Figure 10-20 on page 470.
2. The new simple type item will be added and selected.
3. In the Properties view select the **General** tab, and in the Name field enter `accountId`.
4. Select the **Advanced** tab and enter a length of 6, and in the whiteSpace field select **collapse**, as shown in Figure 10-21 on page 470.

   This ensures that any XML files that are validated against this schema must have an accoutId of 6 non-white space characters.

Archived

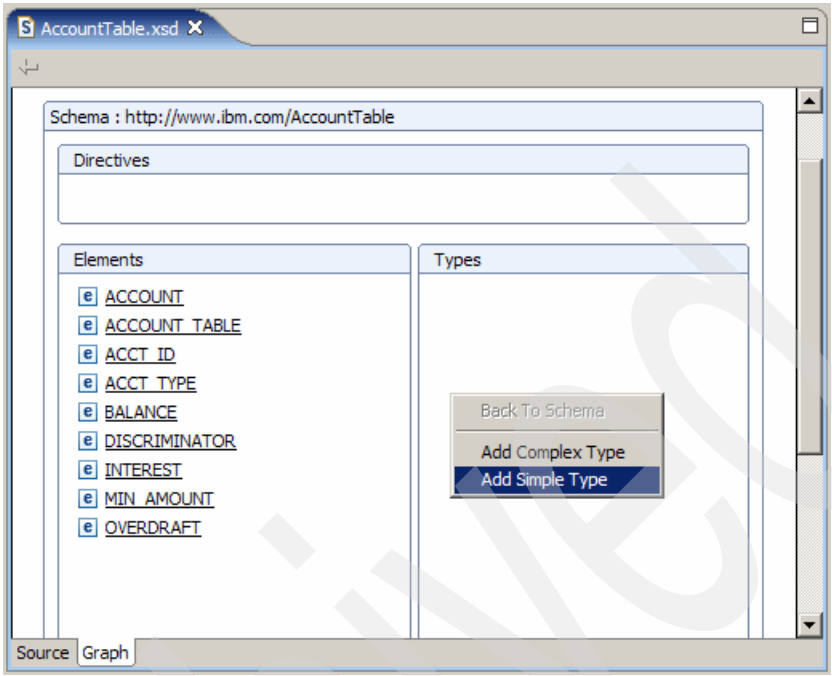


Figure 10-20 Adding a user-defined simple type in the graph editor

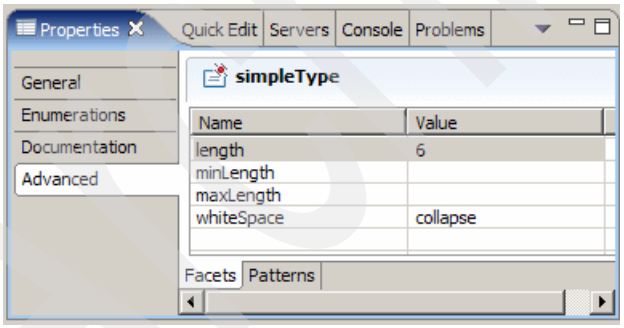


Figure 10-21 Advance tab for user-defined simple type, with a base type of string

While editing, elements the following tabs are available:

- General - This tab allows you to edit the common attributes of the selected node, such as the name and type. To select the element type you can click the **More** button, which will display a dialog with the types that you can choose from, as shown in Figure 10-23 on page 472. You can also choose to display user-defined simple types, as well as user-defined complex types, by selecting the appropriate radio button.

- Other - This tab allows you to define various values for the element such as abstract, block, final, fixed/default, nillable, and substitutionGroup.
- Attributes - This tab allows you to manage the attributes for the element.
- Documentation - This tab allows you to add documentation for the node. It will added the annotation and documentation tags into the node for you.

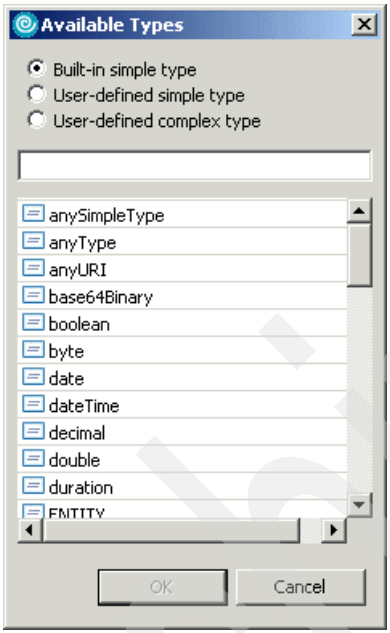


Figure 10-22 Available types for dialog

While editing complex types:

- General - This tab allows you to change the common attributes of the complex type such as the Name, Base type, and Derived by. The base type can be a build-in simple type, user-defined simple type, or a user-defined complex type. To select the base type, select the more **Button** to the right of the field.
- Other - This tab allows you to define various values for the element such as abstract, block, final, and mixed.
- Attributes - This tab allows you to manage the attributes associated with the complex type.
- Documentation - This tab allows you to add documentation for the node. It will add the annotation and documentation tags into the node for you.

Archived

While editing groups:

- General - This tab allows you to define the common attributes of the group, such as Kind, minOccurs, maxOccurs.
- Documentation - This tab allows you to add documentation for the node. It will add the annotation and documentation tags into the node for you.

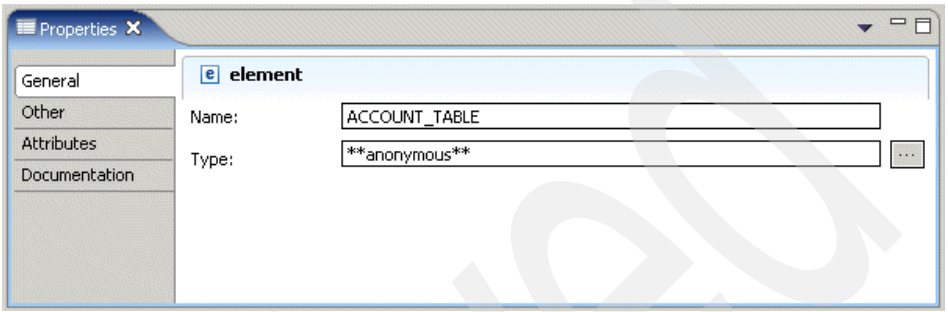


Figure 10-23 Properties view for XML schema

The updated AccountTable.xsd file is displayed in Example 10-23.

Example 10-4 The updated AccountTable.xsd file

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
targetNamespace="http://www.ibm.com/AccountTable"
xmlns:tns="http://www.ibm.com/AccountTable">
    <xsd:element name="ACCOUNT_TABLE">
    <xsd:annotation>
      <xsd:documentation>
         This is the root element for this document type.
      </xsd:documentation>
    </xsd:annotation>
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="account" type="tns:Account" minOccurs="0"
maxOccurs="unbounded"></xsd:element>
      </xsd:sequence>
    </xsd:complexType>
    </xsd:element>
    <xsd:simpleType name="AccountId">
    <xsd:restriction base="xsd:string">
      <xsd:length value="6"></xsd:length>
      <xsd:whiteSpace value="collapse"></xsd:whiteSpace>
    </xsd:restriction>
    </xsd:simpleType>
```

Archived

```
    <xsd:simpleType name="AccountType">
    <xsd:restriction base="xsd:string">
      <xsd:enumeration value="Savings"></xsd:enumeration>
      <xsd:enumeration value="Loan"></xsd:enumeration>
      <xsd:enumeration value="Fixed"></xsd:enumeration>
    </xsd:restriction>
    </xsd:simpleType>

    <xsd:simpleType name="Balance">
    <xsd:restriction base="xsd:decimal">
      <xsd:totalDigits value="15"></xsd:totalDigits>
      <xsd:fractionDigits value="2"></xsd:fractionDigits>
      <xsd:whiteSpace value="collapse"></xsd:whiteSpace>
    </xsd:restriction>
    </xsd:simpleType>

    <xsd:simpleType name="Discriminator">
    <xsd:restriction base="xsd:string">
      <xsd:enumeration value="Yes"></xsd:enumeration>
      <xsd:enumeration value="No"></xsd:enumeration>
    </xsd:restriction>
    </xsd:simpleType>

    <xsd:simpleType name="Interest">
    <xsd:restriction base="xsd:decimal">
      <xsd:totalDigits value="6"></xsd:totalDigits>
      <xsd:fractionDigits value="2"></xsd:fractionDigits>
      <xsd:whiteSpace value="collapse"></xsd:whiteSpace>
      <xsd:maxInclusive value="100.00"></xsd:maxInclusive>
      <xsd:minInclusive value="000.00"></xsd:minInclusive>
    </xsd:restriction>
    </xsd:simpleType>

    <xsd:simpleType name="MinAmount">
    <xsd:restriction base="xsd:integer"></xsd:restriction>
    </xsd:simpleType>

    <xsd:simpleType name="Overdraft">
    <xsd:restriction base="xsd:string">
      <xsd:enumeration value="Yes"></xsd:enumeration>
      <xsd:enumeration value="No"></xsd:enumeration>
    </xsd:restriction>
    </xsd:simpleType>

    <xsd:complexType name="Account">
    <xsd:sequence minOccurs="1" maxOccurs="1">
      <xsd:element name="accountId" type="tns:AccountId" minOccurs="1"
maxOccurs="1"></xsd:element>
```

Archived

```
      <xsd:element name="accountType" type="tns:AccountType" minOccurs="1"
maxOccurs="1"></xsd:element>
      <xsd:element name="balance" type="tns:Balance" minOccurs="1"
maxOccurs="1"></xsd:element>
      <xsd:element name="disclaiminator" type="tns:Discriminator"
minOccurs="1" maxOccurs="1"></xsd:element>
      <xsd:element name="interest" type="tns:Interest" minOccurs="1"
maxOccurs="1"></xsd:element>
      <xsd:element name="minAmount" type="tns:MinAmount" minOccurs="1"
maxOccurs="1"></xsd:element>
      <xsd:element name="overdraft" type="tns:Overdraft" minOccurs="1"
maxOccurs="1"></xsd:element>
    </xsd:sequence>
    </xsd:complexType>

</xsd:schema>
```

Validate an XML schema

To validate the XML schema file perform the following:

1. Switch to the Web perspective.
2. Right-click the **AccountTable.xsd** file in the Navigator view.
3. In the context menu select **Validate XML Schema**.

   If the file that you are trying to validate contains unsaved information you will be prompted to save the file before validation. If you select **No** it will validate the previously saved version.

If validation was successful you will be presented with a pop-up window stating that validation was successful.

If validation was not successful you will be presented with a pop-up window stating that validation has failed. The source editor will also have a red cross next to the line that failed (see Figure 10-24 on page 475), and the problem view will also have an entry with a reason why the validation failed (see Figure 10-25 on page 475).

Archived

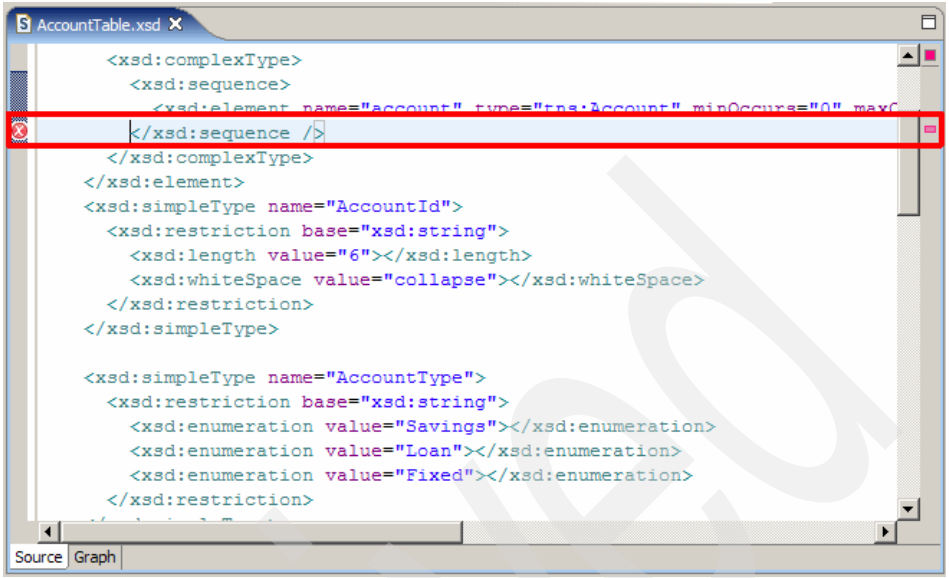


Figure 10-24 Error in the XML schema file

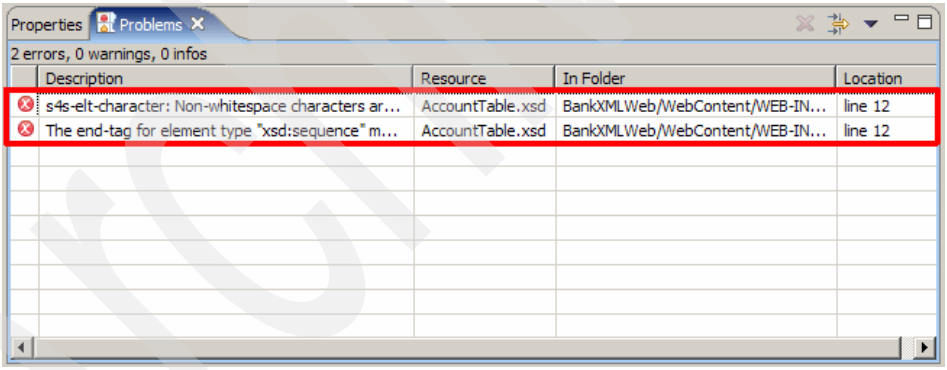


Figure 10-25 XML schema Problems view

10.2.4 Work with XML files

This section demonstrates the following capabilities of the XML editor:

- Generate XML file from an existing DTD file.
- Generate an XML file from an existing XML schema.
- Create a new XML file.
- Edit an XML file.
- Validate an XML file.

Archived

Generate XML file from an existing DTD file

To generate an XML file from an existing DTD file:

1. Switch to the Web perspective.
2. Right-click the DTD file in the Navigator view.
3. Select **Generate** → **XML File**.
4. When the New file dialog is displayed (as shown in Figure 10-26), select the folder in which you wish to place the new XML file.
5. Enter the file name `AccountTableFromDTD.xml` and then click **Next**.
6. Select the root element, which will be **ACCOUNT_TABLE**.
7. Select both **Create optional attributes** and **Create optional elements**.
8. Click **Finish**.

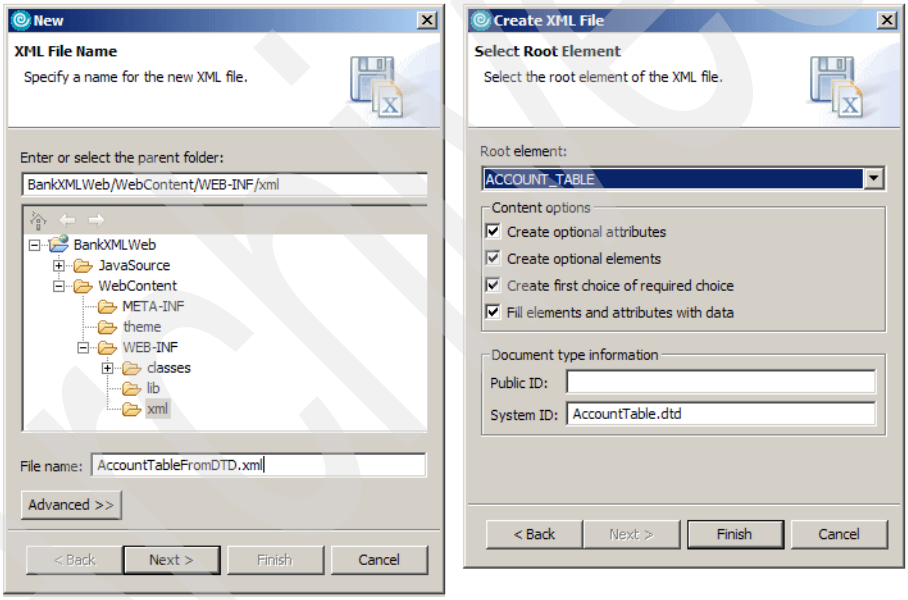


Figure 10-26 Generating a new XML file form DTD dialog

A sample of the generated file is displayed in Example 10-5. As you can see, the content of the XML file is valid according to the DTD file; however, this does not have much relevance to the actual content.

Example 10-5 The AccountTableFromDTD.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE ACCOUNT_TABLE SYSTEM "AccountTable.dtd" >
<ACCOUNT_TABLE>
```

```
  <ACCOUNT>
    <ACCT_ID>ACCT_ID</ACCT_ID>
    <ACCT_TYPE>ACCT_TYPE</ACCT_TYPE>
    <BALANCE>BALANCE</BALANCE>
    <DISCRIMINATOR>DISCRIMINATOR</DISCRIMINATOR>
    <INTEREST>INTEREST</INTEREST>
    <OVERDRAFT>OVERDRAFT</OVERDRAFT>
    <MIN_AMOUNT>MIN_AMOUNT</MIN_AMOUNT>
  </ACCOUNT>
</ACCOUNT_TABLE>
```

Generate an XML file from an existing XML schema

To generate an XML file from an existing XML schema:

1. Switch to the Web perspective.
2. Right-click XML schema in the Navigator view.
3. Select **Generate** → **XML File...**.
4. When the New file dialog is displayed (as shown in Figure 10-27 on page 478), select the folder in which you wish to place the new XML file.
5. Enter the file name file `AccountTableFromSchema.xml` and then click **Next**.
6. Select the root element, which will be **ACCOUNT_TABLE**.
7. Select both **Create optional attributes** and **Create optional elements**.
8. Click **Finish**.

Archived

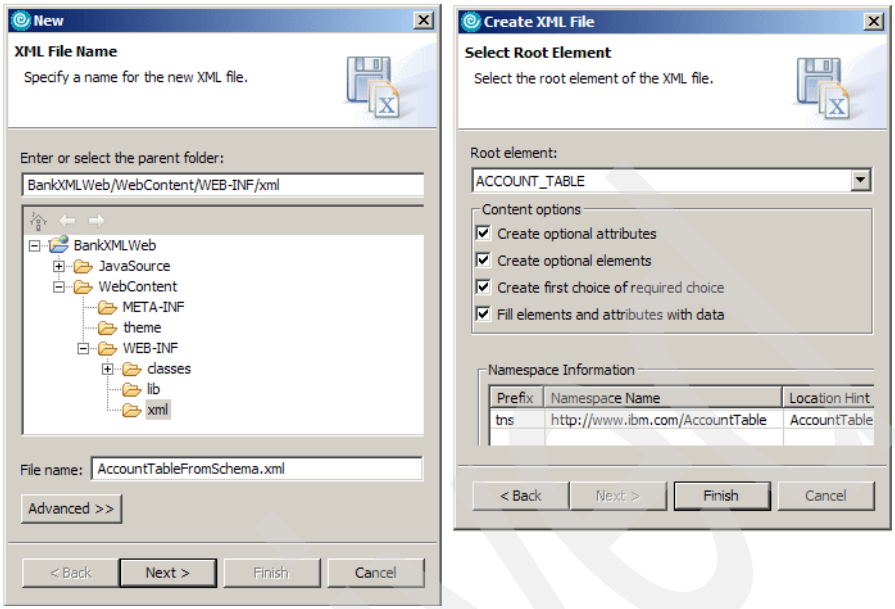


Figure 10-27 Generating an XML file from an XML schema

A sample of the generated XML file is displayed in Example 10-6. You can see in the source editor that there is an error in the generated xml file because the value accountId violates the schema's restrictions.

Example 10-6 The AccoutTableFromSchema.xml file

```
<?xml version="1.0" encoding="UTF-8"?>
<tns:ACCOUNT_TABLE xmlns:tns="http://www.ibm.com/AccountTable"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.ibm.com/AccountTable AccountTable.xsd ">
  <account>
    <accountId>accountId</accountId>
    <accountType>Savings</accountType>
    <balance>0.0</balance>
    <disclaiminator>Yes</disclaiminator>
    <interest>0.0</interest>
    <minAmount>0</minAmount>
    <overdraft>Yes</overdraft>
  </account>
</tns:ACCOUNT_TABLE>
```

Create a new XML file

To create a new XML file from scratch, use the new XML schema wizard.

1. Switch to the Web perspective
2. Select **File** → **New** → **Other**. The new file dialog will be displayed, as shown in Figure 10-28.
3. Check **Show All**.
4. Expand the XML folder.
5. Select **XML** and click **Next**.
6. When the Create XML file wizard is displayed (as shown in Figure 10-29 on page 480), select **Create XML file from scratch** and click **Next**.
7. Select the folder in which you wish to place the new XML file.
8. Enter the file name file `AccountTable.xml` and click **Finish**.

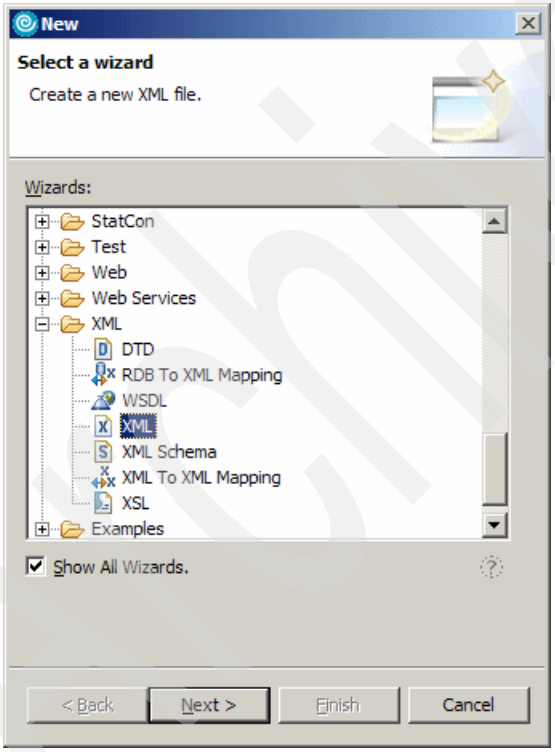


Figure 10-28 New XML file dialog

Archived

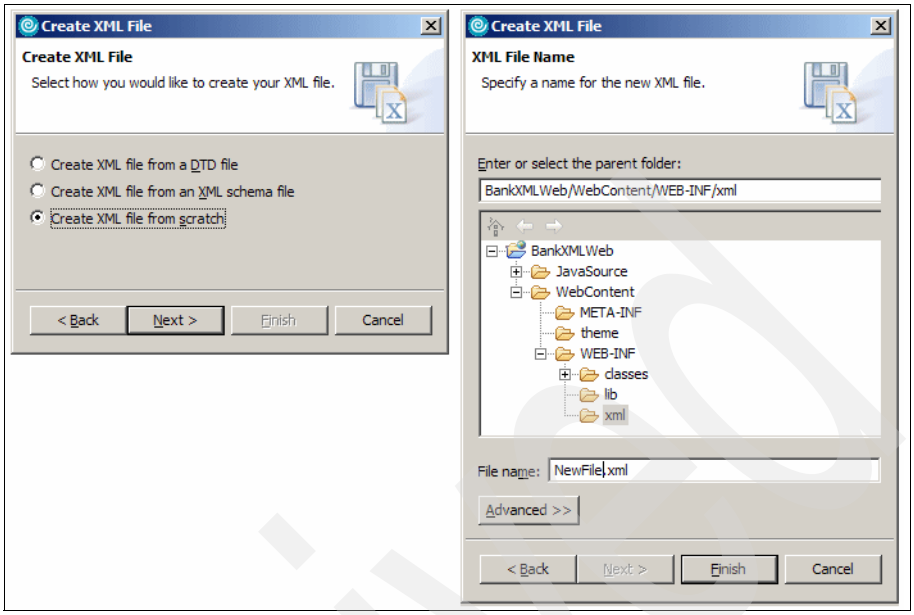


Figure 10-29 Create new XML file wizard

Edit an XML file

To add new elements, comments, and processing instructions, do the following after opening the file in the XML schema editor:

- Add content by typing in the source editor.
- Add via the context menu in the Outline view.

The XML editor has the following features:

- Syntax highlighting
- Unlimited undo and redo of changes
- Content assist with Ctrl+Space
- Node selection indicator

Views that are useful for editing XML files are:

- Outline view - This view presents a tree structure representation of the XML file. The outline view is shown in Figure 10-30 on page 481.
- Properties view - This view allows you to view and edit the attributes of the selected node.

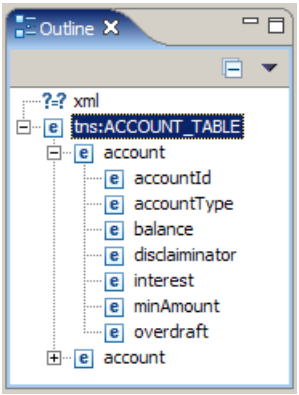


Figure 10-30 Outline view of XML file

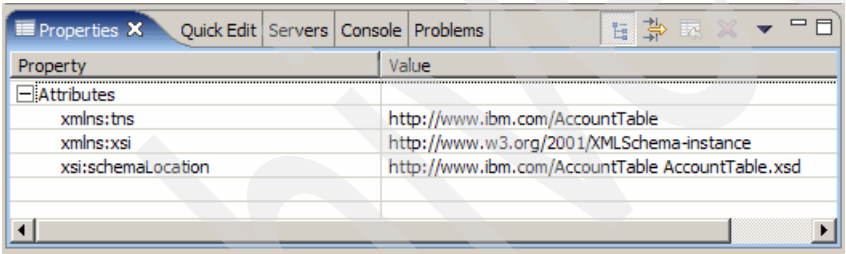


Figure 10-31 Properties view of the XML file

Validate an XML file

To validate the XML file perform the following:

1. Switch to the Web perspective.
2. Right-click **AccountTableFromSchema.xml** in the Navigator view.
3. In the context menu select **Validate XML Schema**.

   If the file that you are trying to validate contains unsaved information you will be prompted to save the file before validation. If you select **No** it will validate the previously saved version.

   If validation was successful you will be presented with a pop-up window stating that validation was successful.

   If validation was not successful you will be presented with a pop-up window stating that validation has failed. The source editor will also have a red cross next to the line that failed (see Figure 10-32 on page 482), and the problem view will also have an entry with a reason why the validation failed (see Figure 10-33 on page 482).

Archived

Figure 10-32 Error in XML file

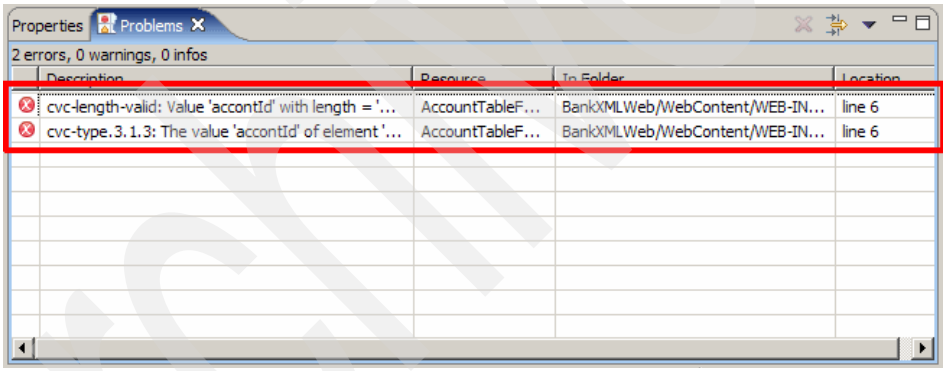


Figure 10-33 XML Problems view

10.2.5 Work with XSL files

This section demonstrates the following capabilities of working with XSL files with the editor:

- Create a new XSL file.
- Edit an XSL file.
- Validate an XSL file.

Archived

Create a new XSL file

To create a new XML schema from scratch using the new XML schema wizard, do the following:

1. Switch to the Web perspective.
2. Select **File** → **New** → **Other**.
3. Check **Show All**.
4. Expand the XML folder.
5. Select **XSL** and then click **Next** (see Figure 10-34 on page 484).
6. Select the **BankXMLWeb/WebContent/WEB-INF/xml** folder.
7. Enter the file name `BasicAccountTable.xsl` and then click **Next**.
8. Select the **AccountTableFromDTD.xml** file to associate with this XSL file.
9. Click **Finish**.

A sample of the generated XSL file is listed in Example 10-7.

Example 10-7 The generated BasicAccountTable.xsl

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    version="1.0"
    xmlns:xalan="http://xml.apache.org/xslt">

</xsl:stylesheet>
```

Archived

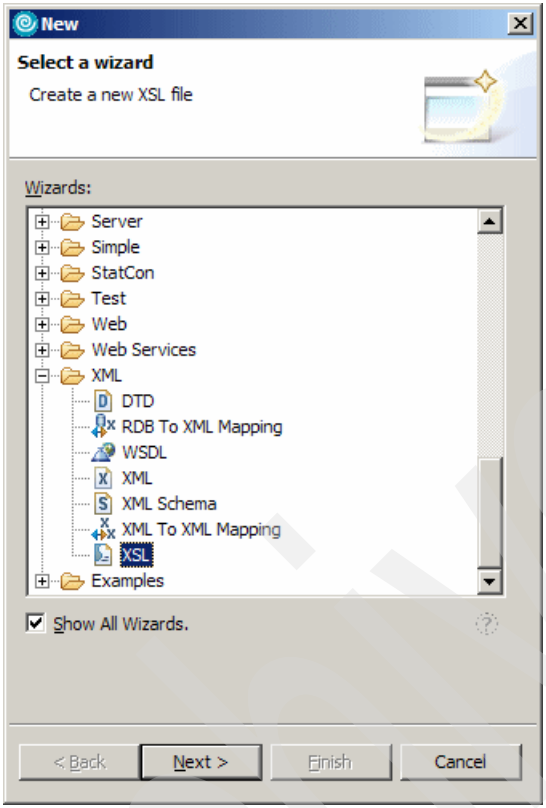


Figure 10-34 The new XSL file dialog

Archived

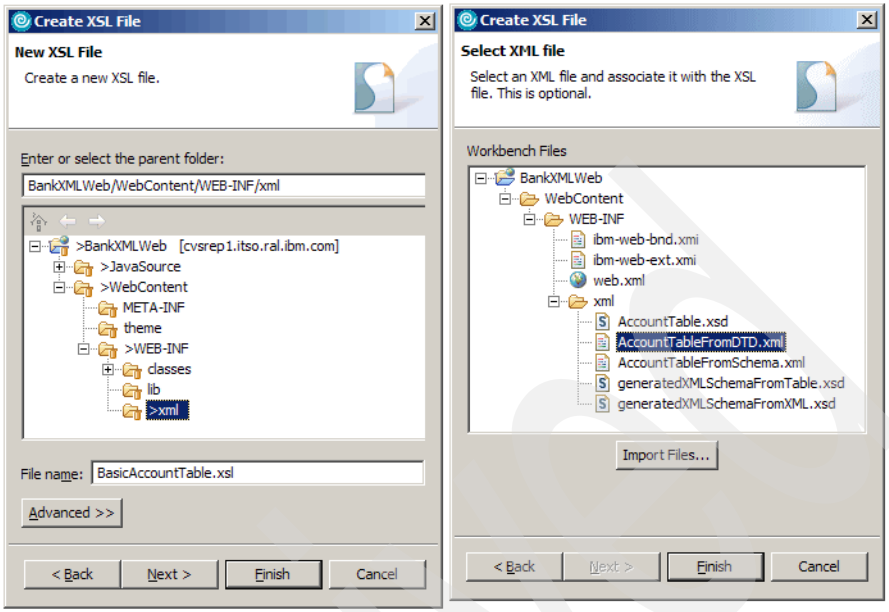


Figure 10-35 The create new XSL wizard

Edit an XSL file

Views that are useful for editing XSL files are:

- Outline view - This view presents the XSL file in a tree structure as shown in Figure 10-36 on page 486.
- Properties view - This view allows you to view and edit the attributes of the selected node, as shown in Figure 10-36 on page 486.
- Snippets view - This view allows you add in commonly used code. Different categorizations of code can be seen in Figure 10-37 on page 486.

Archive

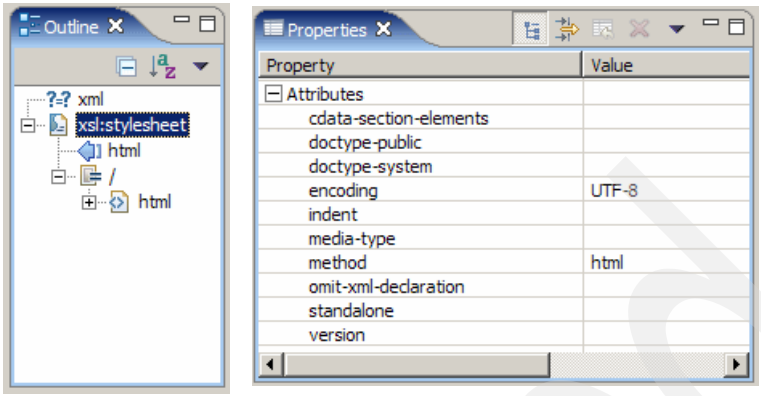


Figure 10-36 The Outline and Properties views of the XSL file

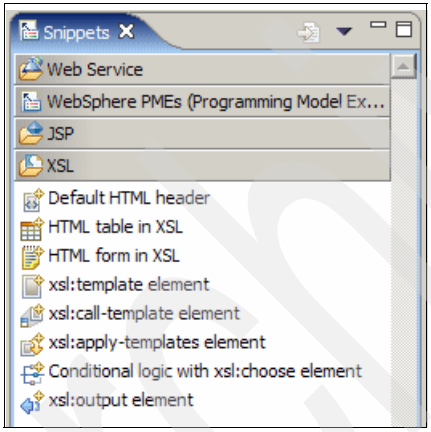


Figure 10-37 The Snippets view

To add code snippets to a selected XSL file, do the following:

1. Switch to the Web perspective.
2. Double-click **BasicAccountTable.xsl**.
3. In the editor view, place the cursor in between the stylesheet tags, as show in Figure 10-38 on page 487.
4. In the Snippets view, select the **XSL** bar to show the XSL snippets, as shown in Figure 10-37.
5. Double-click **Default HTML header**. This will add the default HTML header information into the BasicAccountTable.xsl file.
6. Remove `<xsl:apply-templates />`, and leave the cursor in its place.

Archived

7. Double-click **HTML table in XSL**. The XSL table wizard will be displayed, as shown in Figure 10-39.
8. Select the content node of **ACCOUNT**. Select the **Include header** check box and then click **Next**.
9. Enter 1 into the Border field and 1 into the Cellspacing field, and then click **Finish**.

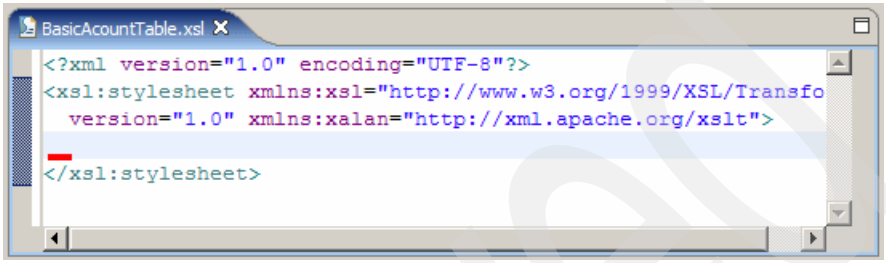


Figure 10-38 XSL editor with new xml file

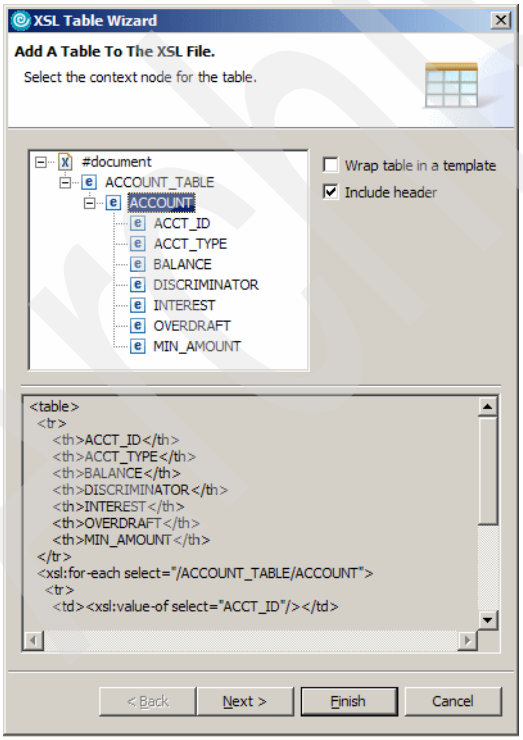


Figure 10-39 XSL table wizard

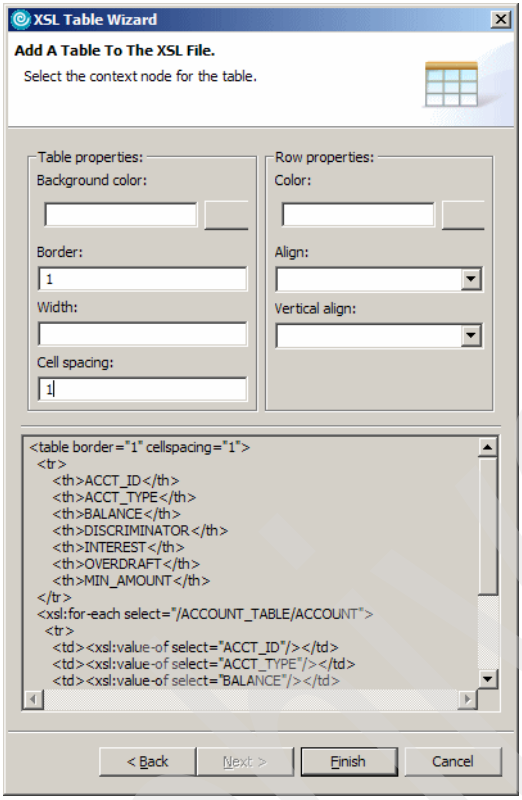


Figure 10-40 XSL Table wizard attributes

A sample of the generated snippet is shown in Example 10-8. The snippet may not be formatted correctly. To format the XSL file, right-click the source editor and select **Format** → **Document**.

Example 10-8 The BasicAccountTable.xsl file

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
   version="1.0" xmlns:xalan="http://xml.apache.org/xslt">
   <xsl:output method="html" encoding="UTF-8" />
   <xsl:template match="/">
      <html>
         <head>
            <title>Untitled</title>
         </head>
         <body>
            <table border="1" cellspacing="1">
               <tr>
                  <th>ACCT_ID</th>
```

Archived

```
            <th>ACCT_TYPE</th>
            <th>BALANCE</th>
            <th>DISCRIMINATOR</th>
            <th>INTEREST</th>
            <th>OVERDRAFT</th>
            <th>MIN_AMOUNT</th>
         </tr>
         <xsl:for-each select="/ACCOUNT_TABLE/ACCOUNT">
            <tr>
               <td>
                  <xsl:value-of select="ACCT_ID" />
               </td>
               <td>
                  <xsl:value-of select="ACCT_TYPE" />
               </td>
               <td>
                  <xsl:value-of select="BALANCE" />
               </td>
               <td>
                  <xsl:value-of select="DISCRIMINATOR" />
               </td>
               <td>
                  <xsl:value-of select="INTEREST" />
               </td>
               <td>
                  <xsl:value-of select="OVERDRAFT" />
               </td>
               <td>
                  <xsl:value-of select="MIN_AMOUNT" />
               </td>
            </tr>
         </xsl:for-each>
      </table>


      </body>
   </html>
</xsl:template>

</xsl:stylesheet>
```

Validate an XSL file

To validate the XMl schema file, do the following:

1. Switch to the Web perspective.
2. Right-click the **BasicAccountTable.xsl** file in the Navigator view.
3. In the context menu select **Validate XML Schema**.

Archived

If the file that you are trying to validate contains unsaved information you will be prompted to save the file before validation. If you select **No** it will validate the previously saved version.

If validation was successful you will be presented with a pop-up window stating that validation was successful.

If validation was not successful you will be presented with a pop-up window stating that validation has failed. The source editor will also have a red cross next to the line that failed (see Figure 10-41), and the problem view will also have an entry with a reason why the validation failed (see Figure 10-42 on page 491).

Figure 10-41 XSL error in the source editor

Archive

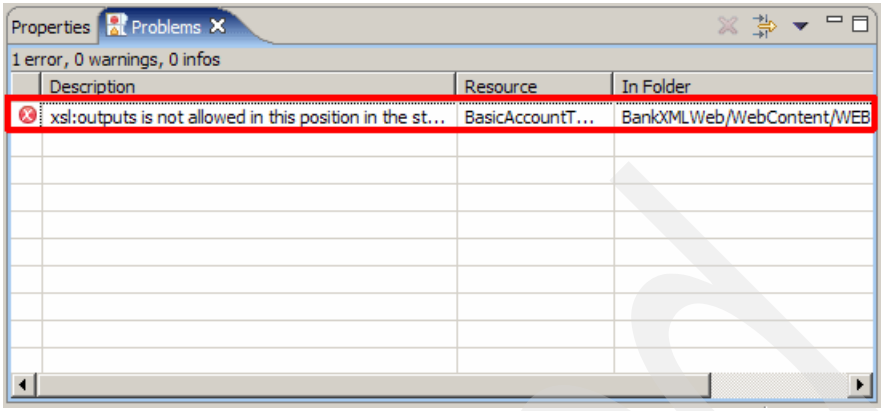


Figure 10-42 XSL Problem view

10.2.6 Transform an XML file

XML files can be transformed into other XML files, HTML files, or text files through the use of XSL files.

When moving from DTD to XML schema you will need a means of migrating your existing XML documents base on DTDs across to XML schemas. This can be done using XSLT.

IBM Rational Application Developer V6.0 offers an XML-to-XML mapping function that will perform this task for you.

Transform an XML file

To transform an XML file, do the following:

1. Switch to the Web perspective.
2. Select **File** → **New** → **Other**.
3. Expand the XML folder.
   a. If the XML folder is not listed, check the **Show All** check box.
   b. You may be asked to confirm enablement of XML Development. Click **OK**.
4. Select **XML To XML Mapping** and then click **Next**.
5. Select the folder in which you wish to place the new XML file.
6. Enter the file name `xmlmap.xmx`.
7. Select the **AccountTable.dtd** file and click the **>** button.
8. Click **Next**.
9. Select the **AccountTable.xsd** file and click **Next**.

Archived

10. Select the **ACCOUNT_TABLE** as the root element and click **Finish**.

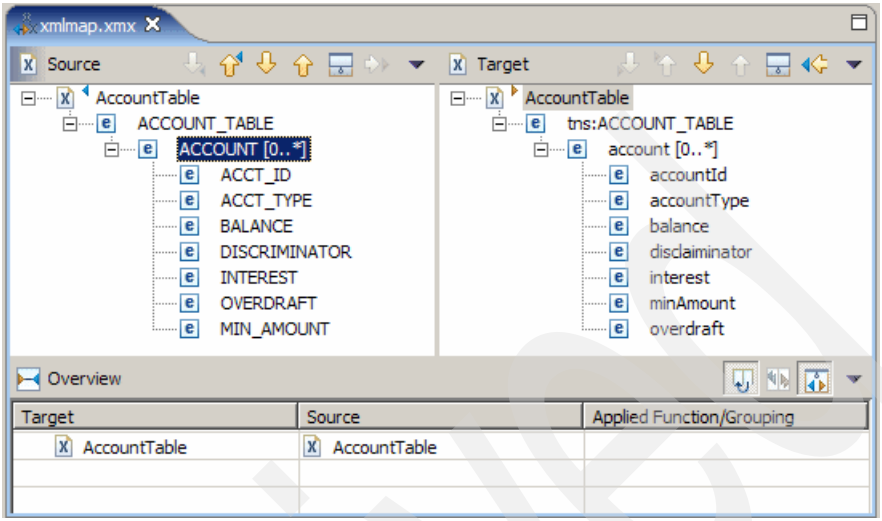


Figure 10-43 XML-to-XML Mapping view

To map the node in the DTD to the XML schema, select the un-mapped node and drag it from the left pane to the corresponding node in the right pane. Mapped nodes are indicated by the arrows next to the node; also, the node will appear in the table as shown in Figure 10-43.

After you have mapped the element from the DTD to the XSD file, you need to create an XSLT file.

Create an XSLT file

To create an XSLT file, do the following:

1. Right-click the **xmlmap.xmx** file and select **Generate XSLT ...**. This will display the Generate XSLT Script dialog, as shown in the Figure 10-44 on page 493.
2. Enter the file name `xmlmap.xsl`, and click the **Finish** button.

Archive

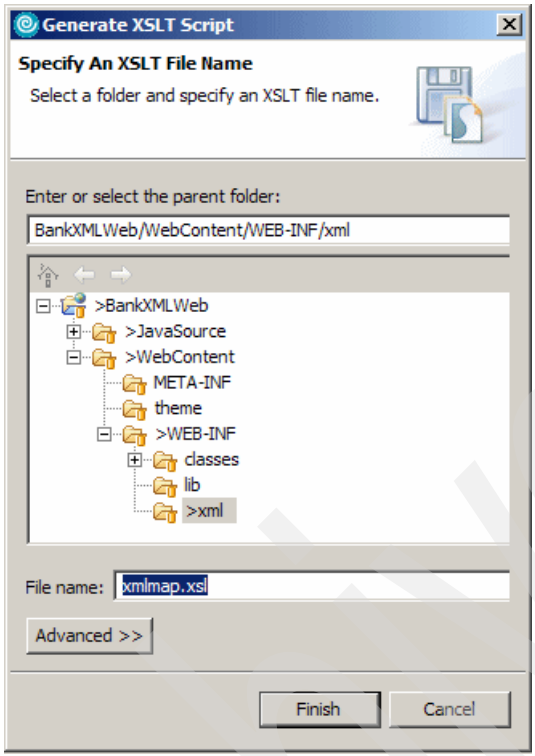


Figure 10-44 The Generate XSLT Script dialog

Debug an XSLT file

To debug the XSL file, do the following:

1. Switch to the Web perspective.
2. Right-click the **xmlmap.xmx** file.
3. Select **Debug** → **Debug**.
4. When the Debug dialog is displayed (as shown in Figure 10-45 on page 494), do the following:
   a. Enter `xmlmap` in the Name field. This will be used to identify this debugging session preferences so that you do not have to configure asession each time you want to debug the XSL file. You will see the name appear in the configuration list box.
   b. Enter `xmlmap.xsl` in the Source XSL file field, as this is the generated XSLT file for the XMX file.
   c. Enter `AccountTableFromDTD.xml` in the Source XML file. As you can see in Figure 10-45 on page 494, the XML input only has single enabled. This is

Archived

because you can only debug using one XML file at time; however, you can process more than one file when you run the XSL file.

d. Click the **Debug** button.

5. You may be asked to confirm a perspective switch, as shown inFigure 10-46 on page 495. Click the **Yes** button to proceed.
6. The Debug perspective will be displayed as shown in Figure 10-47 on page 496.

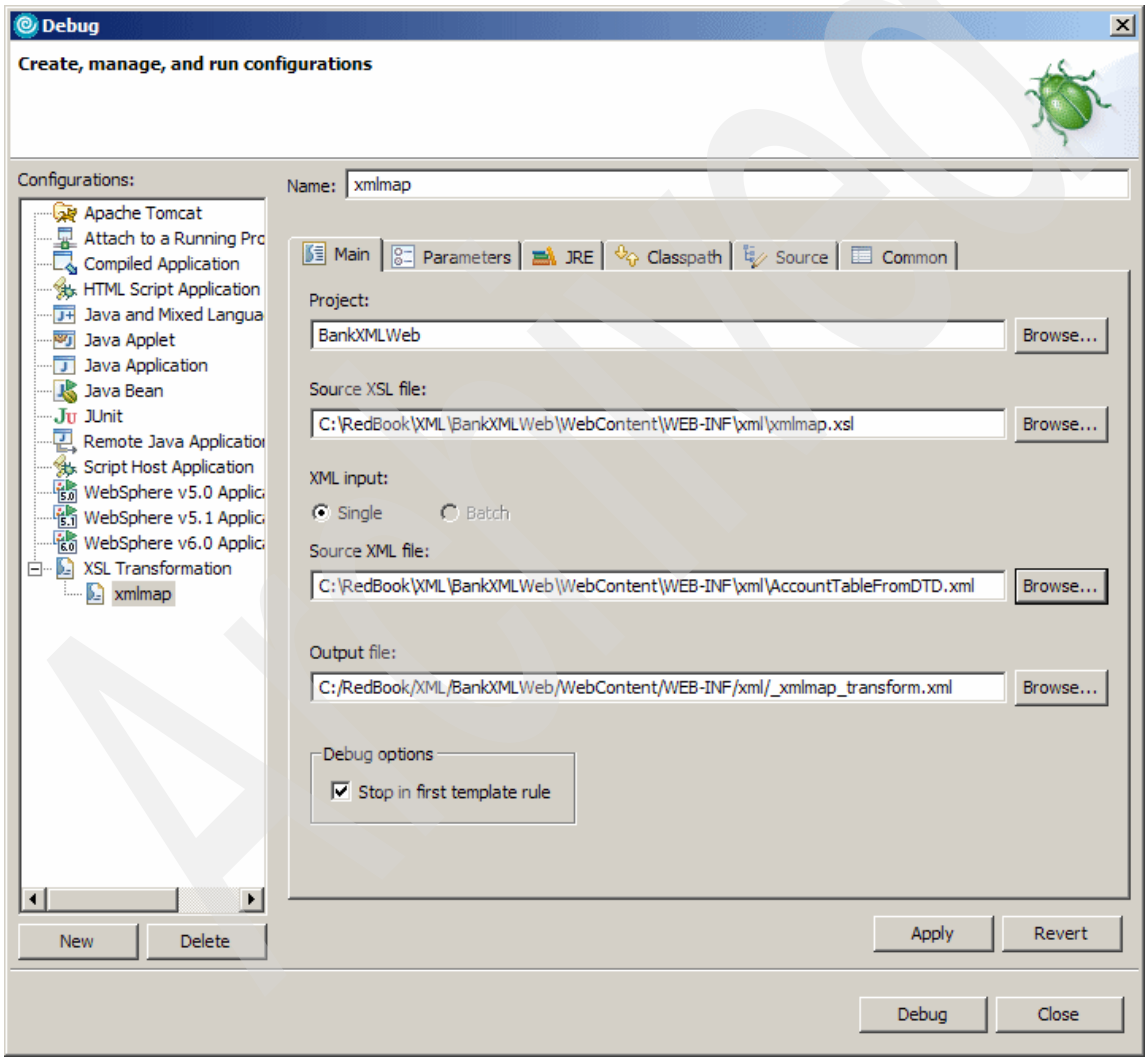


Figure 10-45 Debug dialog

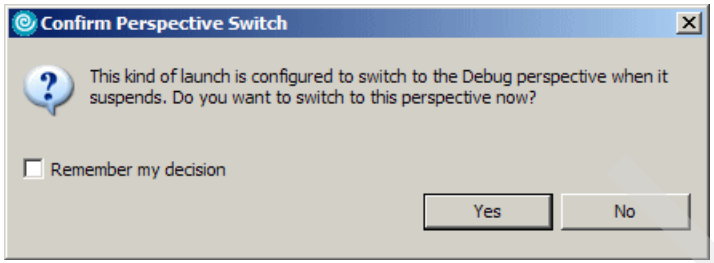


Figure 10-46 Confirm Perspective Switch dialog

The Debugging perspective is shown in Figure 10-47 on page 496. You can use the Debug view to step through the XSL file. The XSL file that is being debugged, as well as the source XML file that is being transformed, is also displayed. The end result is displayed in the output view.

Archived

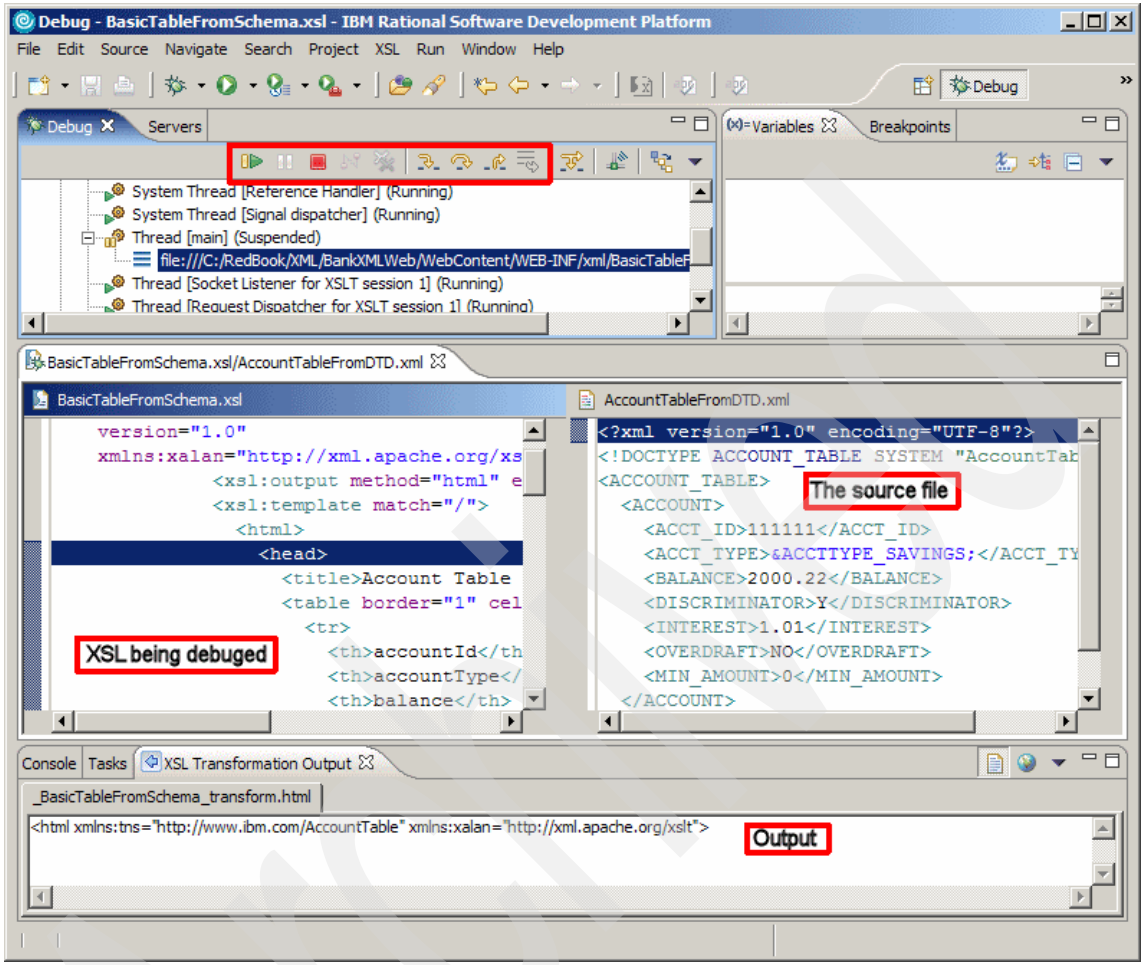


Figure 10-47 Debug perspective

10.2.7 Java code generation

IBM Rational Application Developer V6.0 includes some Java code generation tools. These tools allow you to create Java code based on an XML schema file.

Generate Java classes

IBM Rational Application Developer V6.0 allows you to generate Service Data Object classes from a XML schema file.

To generate SDO classes, do the following:

1. Switch to the Web perspective.

2. Right-click the **AccountTable.xsd** file in the Navigator view.
3. Select **Generate** → **Java**.
4. When the Generate Java dialog is displayed (as shown in Figure 10-48), select the **SDO Generator**, and in the Container field enter `/BankXMLWeb/JavaSource`.
5. Click **Finish**.
6. The Java code will be generated in the JavaSource folder with the package name as defined by the target namespace.

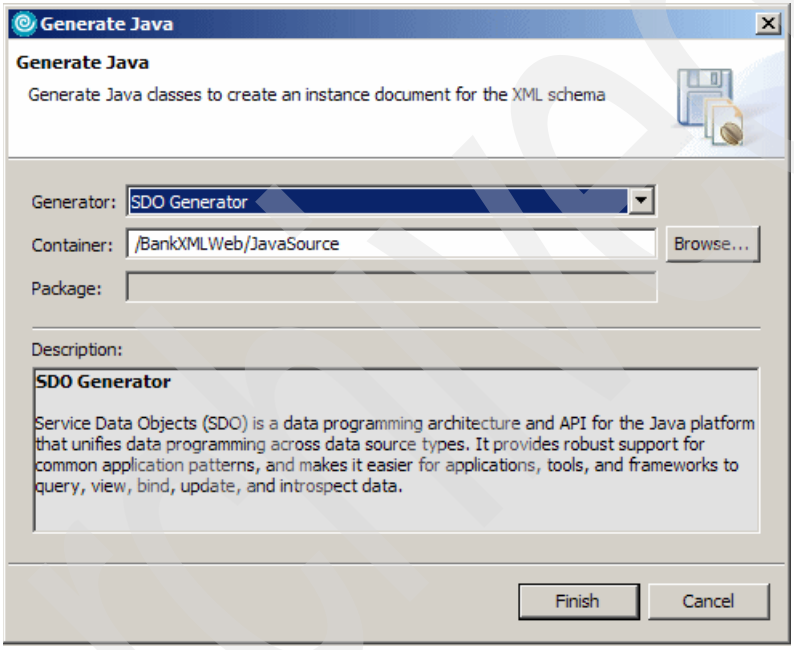


Figure 10-48 Java SDO classes generator

10.3 Where to find more information

In the IBM Redbook *The XML Files: Development of XML/XSL Applications Using WebSphere Studio Version 5*, SG24-6586, more detailed information is provided regarding developing XML/XSL applications using Application Developer. That book also contains examples using Application Developer Version 5.

For more information on XML schemas refer to:

http://www.w3.org/XML/Schema

For more information on XML refer to:

http://www.w3.org/XML/

For more information on XML parsers refer to:

- Xerces (XML parser - Apache)

  http://xml.apache.org/xerces2-j
- Xalan (XSLT processor - Apache)

  http://xml.apache.org/xalan-j
- JAXP (XML parser - Sun)

  http://java.sun.com/xml/jaxp
- SAX2 (XML API)

  http://sax.sourceforge.net

Archived

11

Develop Web applications using JSPs and servlets

There are many technologies available for developing Web applications. In this chapter, we focus on developing dynamic Web applications using JavaServer Pages (JSPs) and Java Servlet technology, and static Web sites using HTML.

The chapter is designed to guide the reader through the features of Application Developer used in the creation of Web applications. First, we introduce the ITSO Bank sample application and create a new Web Project to hold our example application. Next, we add static and dynamic content using wizards and tools such as Page Designer. Finally, we provide examples for working with filters and listeners, and creating a Web page from JavaBeans.

The chapter is organized into the following sections:

- Introduction to Web applications
- Web development tooling
- Prepare for the sample
- Define the site navigation and appearance
- Develop the static Web resources
- Develop the dynamic Web resources
- Test the application

© Copyright IBM Corp. 2005. All rights reserved.

11.1 Introduction to Web applications

There are many Web development technologies as well as tools included in Rational Application Developer. In this chapter we do not discuss Struts, JSF, EGL, or portals, since these Web application topics are covered in other chapters. Our focus in this chapter is on developing dynamic Web applications using JavaServer Pages (JSPs) and Java Servlet technology, and static Web sites using HTML with the tooling included with Rational Application Developer.

11.1.1 Concepts and technologies

This section provides an overview of the concepts and technolgy used by J2EE Web applications.

Static and dynamic Web application technologies

For an overview on the concepts and techologies used to develop static Web sites and dynamic Web applications refer to the following:

- “Static Web sites” on page 39
- “Dynamic Web applications” on page 43

Enterprise application

An enterprise application project contains the hierarchy of resources that are required to deploy an enterprise (J2EE) application. It can contain a combination of Web modules, EJB modules, JAR files, and application client modules. It includes a deployment descriptor and an IBM extension document, as well as files that are common to all J2EE modules that are defined in the deployment descriptor. It can contain a complete application that may be a combination of multiple modules. Enterprise applications make it easier to deploy and maintain code at the level of a complete application instead of as individual pieces.

There are a few methods to create an Enterprise Application:

- Create a New Enterprise Application wizard. This wizard can by started by selecting **File** → **New** → **Project**. Select **J2EE** → **Enterprise Application**.
- Create the enterprise application as part of creating a new Web Project. This is the method we will use for our sample.
- Create an enterprise application using the Import wizard. If you are importing a Web Project, you can create an enterprise application via the Import wizard.

Enterprise Application projects are exported as enterprise archive (EAR) files that include all files defined in the enterprise application project as well as the appropriate module archive file for each J2EE module project defined in the deployment descriptor, such as Web archive (WAR) files and EJB JAR files.

Archive

An enterprise application may contain JAR files to be used by the contained modules. This allows sharing of code at the application level by multiple Web or EJB modules.

The enterprise application deployment descriptor contains information about the components that make up the enterprise application. This deployment descriptor is called application.xml and is located under the META-INF directory.

Web application

The Java Servlet Specification 2.4 and the J2EE specification contain the concept of a Web application. A Web application contains JavaServer Pages, servlets, applets, Java classes, HTML files, graphics, and descriptive meta information that connects all these elements. The format is standardized and compatible between multiple vendors.

The specification also defines a hierarchical structure for the contents of a Web application that can be used for deployment and packaging purposes. Many servlet containers, including the one provided by Express Application Server, support this structure.

Any Web resource can be included in a Web application, including the following:

- Servlets and JavaServer Pages.
- Utility classes: Standard Java classes may be packaged in a Java archive (JAR) file. JAR is a standard platform-independent file format for aggregating files (mainly Java classes files).
- Static documents: HTML files, images, sounds, videos, etc. This term includes all the documents a Web server is able to handle and to provide to client requests.
- Client-side applets, beans, and classes.
- Descriptive meta information, which ties all of the above elements together.
- Custom Tag Libraries.
- Struts.
- XML files.
- Web Services.

The directory structure for a Web application requires the existence of a WEB-INF directory. This directory contains Java and support classes that contain application logic. Access to these resources is controlled through the servlet container, within the application server.

Archive

Figure 11-1 shows an example of a typical directory structure under the WEB-INF directory.

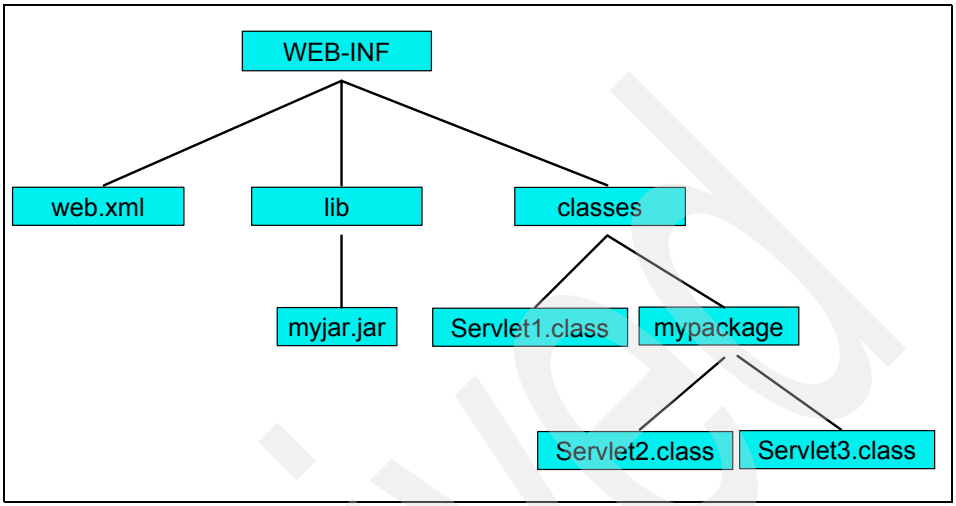


Figure 11-1 The WEB-INF directory: A sample structure

The required elements are:

- web.xml: This file is required and is the deployment descriptor for the Web application.
- lib: This directory is required and is used to store all the Java classes used in the application. This directory will typically contain JAR files, including tag libraries.
- classes: This directory is also required. Typically, servlet classes and utility classes for the servlets compose this directory. It is possible to keep a package structure in this directory and to put class files under several subdirectories of the classes directory (as it is done for the Servlet2.class file in the subdirectory mypackage in Figure 11-1).

Although there are no other requirements for the directory structure of a Web application, we recommend that you organize the resources in separate logical directories for easy management (for example, an images folder to contain all graphics).

As with the enterprise application, a deployment descriptor exists for the Web application. The Web deployment descriptor, web.xml, contains elements that describe how to deploy the Web application and its contents to the servlet container within the Web server. Note that JSPs execute as servlets and are treated as such in the Web deployment desciptor.

Archived

The deployment descriptor file enables the application configuration to be specified independently from the server. It clearly simplifies the deployment process because the same application can be deployed into different servers without having to review its content.

11.1.2 Model-view-controller (MVC) pattern

The model-view-controller architectural pattern was conceived in the mid-1980's. It has since then been extensively applied in most object-oriented user interfaces and has been improved to respond to specific platform requirements, such as J2EE. It has also been generalized as a pattern for implementing the separation of concerns among application layers in general, and not only the three originally proposed layers.

Following the MVC pattern, a software component (application or module) should separate its business logic (model) from its presentation (view). There are many reasons for this requirement, such as:

- More than one view of the same model: If both the business logic and its presentation were built together, adding an additional view would cause considerable disruptions and increase the component's complexity. A good example of a model with two views would be a banking application that can be accessed by traditional Web browser clients and mobile phones.
- Avoid model dependence on the view: You do not want to have to change the model every time you modify the view. The view is definitely dependent on the model, since it presents specific aspects of the model to the user. It makes no sense to have the model depend on the view. Building both together dramatically increases the chances of this happening, and the need to change the model every time you implement a small change to the view.

This separation can be achieved through the layering of the component into:

- Model layer: This layer is responsible for implementing the business logic.
- View layer: The view layer is responsible for rendering the user interface (be it graphical or not) to a specific client type and in a specific fashion.

With these two layers, we can implement the business logic and present it to the user. That solves only half of the problem. We would also like to be able to interact with the model. The implementation of this interaction is better left to a third layer, called the controller.

Model

The model layer manages the application domain’s concepts, both behavior and state. It responds to requests for information about its state and responds to instructions to change its state.

Just like any software component, the model should have a well-defined and an as simple as possible public interface. This is usually achieved through the use of a facade. The intent of facades is to provide a simple and unified interface to the otherwise complex model that lies behind it. By doing so, we reduce the dependencies between the model classes and its clients. Less dependencies mean more freedom to adapt to new requirements.

Figure 11-2 shows the model layer with its encapsulated business domain objects and the exposed facade object. Note that the model does not have any dependences on views or controllers.

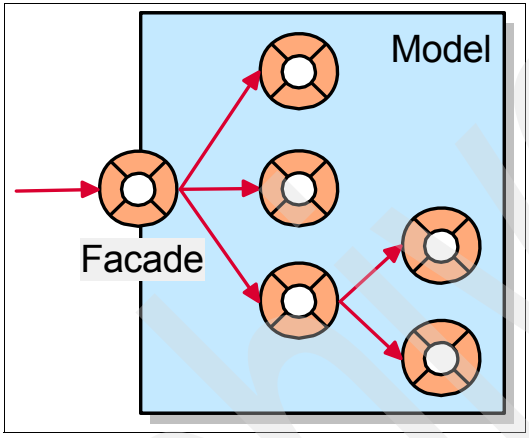


Figure 11-2 Model layer

Note: For more detailed information on the facade design pattern, we recommend reading *Design Patterns, Elements of Reusable Object-Oriented Software,* Eric Gamma, et al.

View

The view layer implements a rendering of the model. The responsibility of the view is to know what parts of the model's state are relevant for the user, and to query the model for that information. The view retrieves the data from the model or receives it from the controller, and displays it to the user in a way the user expects to see it.

Controller

The controller's responsibility is to capture user events and to determine which actions each of these events imply, depending on both the user's and the application's state. This usually involves verifying pre and post conditions. These

actions can then be translated to messages to the model and view layers, as appropriate.

Dependencies between MVC layers

Figure 11-3 shows the dependencies allowed in the MVC pattern. Note that the less dependencies your layers have, the easier it will be for the layers to respond to requirement changes.

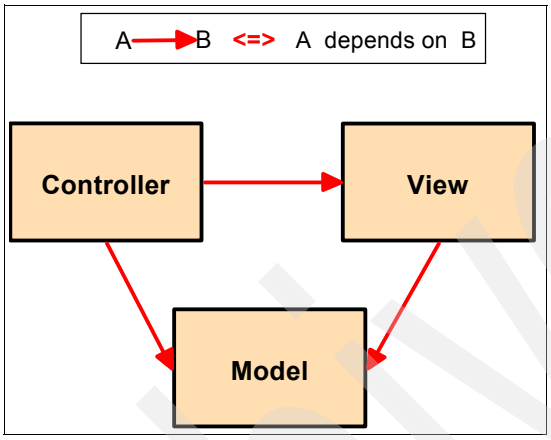


Figure 11-3 Dependencies allowed in the MVC pattern

As we go through this chapter and other Web application development chapters, we will explore how the MVC pattern is implemented by the various technologies.

11.2 Web development tooling

Rational Application Developer includes many Web development tools for buidling static and dynamic Web applications. In this section, we highlight the following tools and features:

- Web perspective and views
- Web Projects
- Web Site Designer
- Page Designer
- Page templates
- CSS Designer
- Javascript Editor
- WebArt Designer
- AnimatedGif Designer
- File creation wizards

Archived

These tools, as well as other tool,s will be further illustrated in the examples found throughout this chapter.

11.2.1 Web perspective and views

Web developers can use the Web perspective and supporting views within Rational Application Developer to build and edit Web resources, such as servlets, JSPs, HTML pages, style sheets, and images, as well as the deployment descriptor files.

The Web perspective can be opened by selecting **Window** → **Open Perspective** → **Web** from the Workbench. Figure 11-4 displays the default layout of the Web perspective with a simple home.html open.

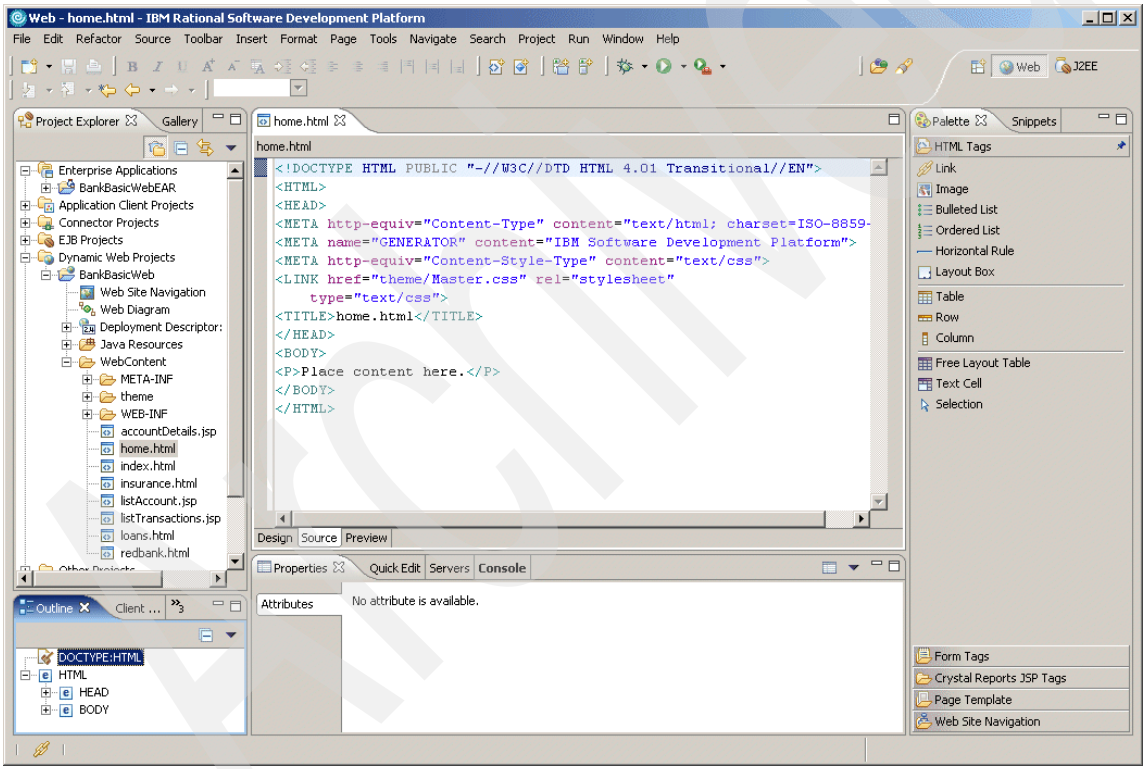


Figure 11-4 Web perspective

As you can see in Figure 11-5, there are many Web perspective views accessible by selecting **Window** → **Show View**, many of which are already open in the Web perspective default setting.

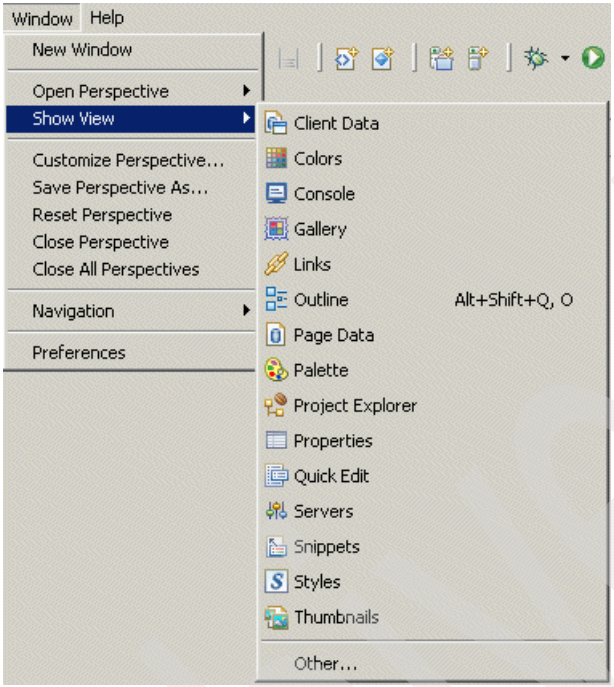


Figure 11-5 Web perspective views

Note: For more information on the Web perspective and views, refer to 4.2.14, "Web perspective" on page 162.

11.2.2 Web Projects

In Application Developer, you create and maintain Web resources in Web Projects. They provide an environment that enables you to perform activities such as link-checking, building, testing, and publishing. Within a Web Project, Web resources can be treated as a portable, cohesive unit.

Web Projects can be static or dynamic. Static Web Projects are comprised solely of static resources, which can be served by a traditional HTTP server (HTML files, images, etc.), and are useful for when you do not have to program any business logic. J2EE Web Projects, on the other hand, may deliver dynamic content as well, which gives them the ability to define Web applications.

A Web application contains components that work together to realize some business requirements. It might be self-contained, or access external data and

functions, as it is usually the case. It is comprised of one or more related servlets, JavaServer Pages, and regular static content, and managed as a unit.

Note: For detailed example on creating a Web Project refer to 11.3.2, “Create a Web Project” on page 517.

11.2.3 Web Site Designer

The Web Site Designer is provided to simplify and speed up the creation of the Web site navigation and creation of HTML and JSP pages. You can view the Web site in a Navigation view to add new pages, delete pages, and move pages in the site. The Web Site Designer is especially suited for building pages that use a page template.

The Web Site Designer is used to create the structure for your application in much the same way you would create a book outline to serve as the basis for writing a book. You use the Web Site Designer to visually lay out the flow of the application, rearranging the elements (JSPs, HTML pages) until it fits your needs. Then you continue by creating pages based on this design.

As you build your Web site design, the information is stored in the website-config.xml file so that navigation links and site maps can be generated automatically. This means that when the structure of a site changes, for example, when a new page is added, the navigation links are automatically regenerated to reflect the new Web site structure.

The Web Site Designer can be used with existing or new projects. To create a Web site configuration for a Web Project that does not currently have one, there is an option present on the context menu of the Web Project, Convert to Web Site in the Web perspective. The site can be easily constructed from site parts found in the Palette. The site parts represent parts for navigation bars, navigation menus, page trails (history), even a site map. Each site part will derive its layout from information present in the Web site configuration file.

To launch the Web Site Designer, double-click **Web Site Navigation** found in the root of your Web Project folder. Figure 11-6 on page 509 displays the sample ITSO Bank Web site navigation and pages in Web Site Designer.

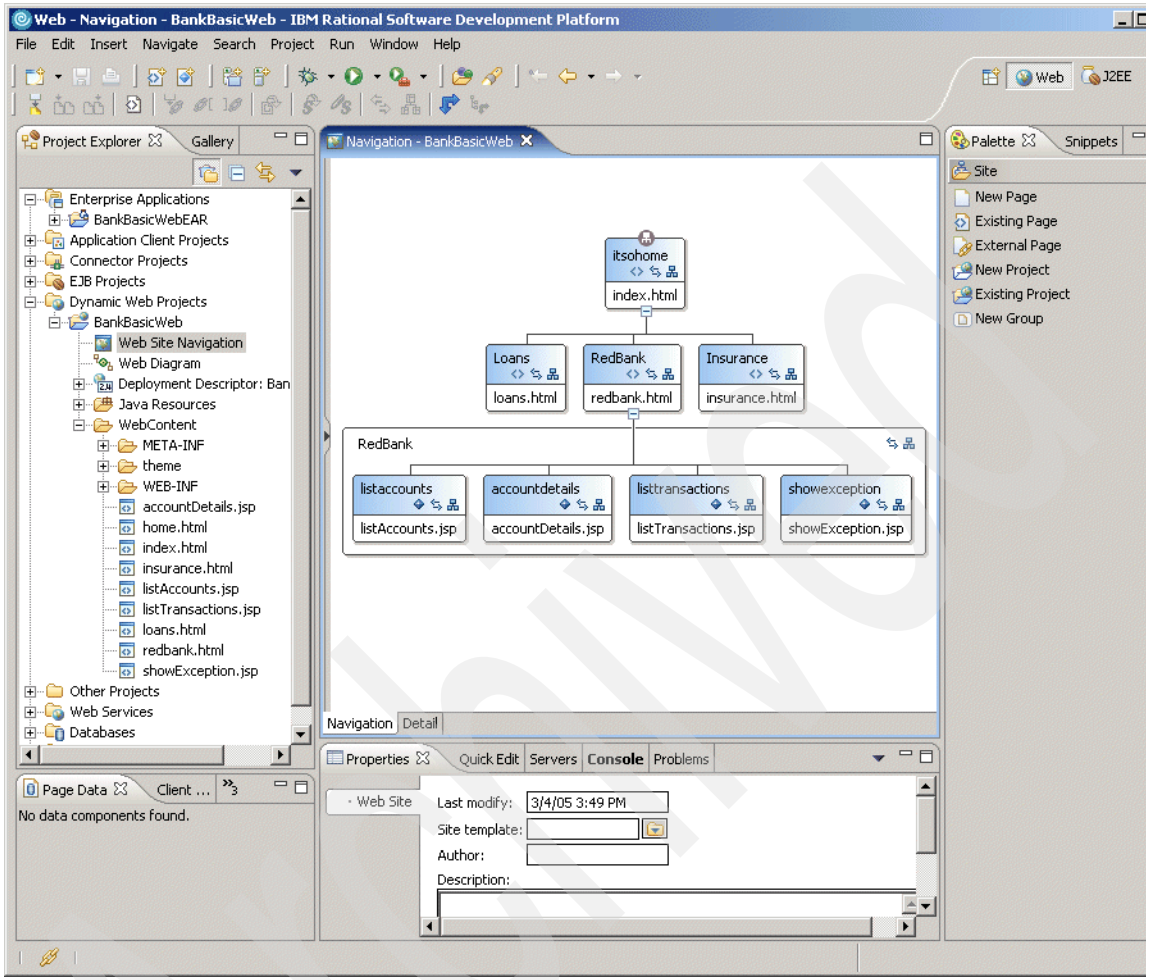


Figure 11-6 Web Site Designer

Note: For a detailed example of using the Web Site Designer refer to 11.4, “Define the site navigation and appearance” on page 524.

11.2.4 Page Designer

Page Designer is the primary editor for developing HTML, XHTML, JSPs, and Faces JSP source code. It has three representations of the page, including Design, Source, and Preview. The Design tab provides a WYSIWYG environment to visually design the contents of the page. As its name implies, the

Source tab provides access to the page source code. The Preview tab shows what the page would like if displayed in a Web browser.

We have listed the key new features of Page Designer in IBM Rational Application Developer V6.0:

- Bidirectional language support (full support on Windows, limited support on Linux)
- Usability Enhancements:
  - Embedded document (JSP Include) support
  - New Properties view
  - Changed visual cue for read only content areas
  - Improved “drag and drop” functionality in free layout mode
  - Improved cell editing functionality
  - Free layout mode improved

Note: For a detailed example of using the Page Designer refer to 11.6, “Develop the dynamic Web resources” on page 549.

11.2.5 Page templates

A page template contains common areas that you want to appear on all pages, and content areas that will be unique on the page. They are used to provide a common look and feel for a Web Project.

The Page Template File creation wizard is used to create the file. Once created the file is modified in the Page Designer. The page templates are stored as *.htpl for HTML pages and *.jtpl files for JSP pages. Changes to the page template will be reflected in pages that use that template. Templates can be applied to individual pages, groups of pages, or applied to an entire Web Project. Areas can be marked as read-only; thus Page Designer will not allow the user to modify those areas.

Note: For examples on creating, customizing, and applying page template, refer to the following:

- “Create a new page template” on page 526
- “Customize a page template” on page 531

Archived

11.2.6 CSS Designer

Style sheets can be created when you create the Web Project, or they can be added later. It is a good idea to decide on the overall theme (color, fonts, etc.) for your Web application in the beginning and create the style sheet at the start of the development effort. Then as you create the HTML and JSP files, you can select that style sheet to ensure that the look of the Web pages will be consistent. Style sheets are commonly kept in the WebContent/theme folder.

The CSS Designer is used to modify cascading style sheet *.css files. The changes are immediately applied to the Design page in Page Designer, if the HTML file is linked to the CSS file.

Note: An example of customizing style sheets used by a page template can be found in 11.4.4, “Customize a style sheet” on page 535.

11.2.7 Javascript Editor

The Javascript Editor provides a Source page and Preview page to enable you to work with source files and view them as if in a Web browser. The Snippets palette includes Javascript code that you can drag and drop into your Web pages.

11.2.8 WebArt Designer

WebArt Designer is a program that can be used to create and edit image files. Using WebArt Designer, you can create shape objects, draw a simple map, as well as create logos and buttons often seen on Web pages. The Page Designer also enables you to edit GIF and JPEG images, but WebArt Designer offers a much richer set of functionality for editing images.

11.2.9 AnimatedGif Designer

An animated GIF is a series of image files in GIF format, displayed sequentially to give the appearance of an animation. You can insert an animated GIF into a page in the same way as a regular GIF image file. Animated GIFs can be viewed on a regular Web browser without any special plug-ins.

The AnimatedGif Designer is a program for creating animated GIF files, and comes with a gallery of predefined animation files. With the AnimatedGif Designer, you can:

- Combine several images to create an animation.
- Apply an animation effect on a single image to create an animation.

Archived

- Apply an animation effect on text to create an animated banner.

The AnimatedGif Designer can be launched by clicking **Tools** → **AnimatedGif Designer** from the menu bar.

11.2.10 File creation wizards

Rational Application Developer provides many Web development file creation wizards by selecting **File** → **New** and then selecting the wizards, as seen in Figure 11-7.

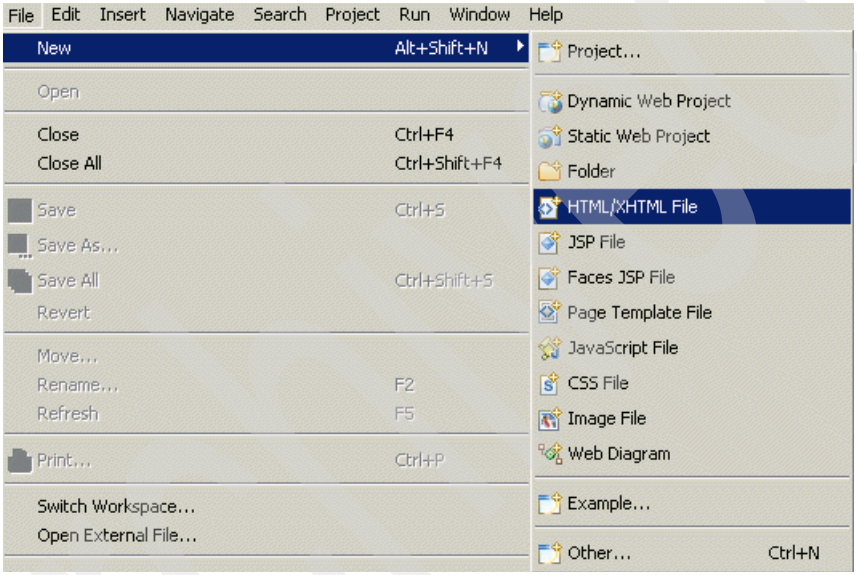


Figure 11-7 File creation wizard selection

HTML File wizard

The HTML File wizard allows you to create an HTML file in a specified folder, with the option to create from a page template. In addition, the markup language included can be defined as type, HTML, HTML Frameset, Compact HTML, XHTML, XHTML Frameset, and WML 1.3.

JSP File wizard

The JSP File wizard allows you to create a JSP file in a specified folder, with the option to create from a page template or create as a JSP Fragement, and define the markup language (HTML, HTML Frameset, Compact HTML, XHTML, XHTML Frameset, and WML 1.3). In addition, you optionally have the ability to select a model (none, JSP, Struts Portlet JSP, Struts JSP).

Faces JSP File wizard

This wizard is similar to the JSP File wizard in capability. It has a different set of models. This wizard will be covered in more detail in Chapter 13, “Develop Web applications using JSF and SDO” on page 673.

Page Template File wizard

The Page Template File wizard is used to create new page template files in a specified folder, with the option to create from a page template or create as a JSP Fragement, and define the markup language (HTML, HTML Frameset, Compact HTML, XHTML, XHTML Frameset, and WML 1.3). You can optionally create a new page template from an existing page template. In addition, you can select from one of the following models: Template contains Faces Components, Template containing only HTML, Template containing JSP.

Javascript File wizard

The Javascript File wizard is used to create a new Javascript file in a specified folder.

CSS File wizard

The CSS File wizard is used to create a new cascading style sheet (CSS) in a specified folder.

Image File wizard

The Image File wizard is used to create a new image files (bmp, mif, gif, png, jpg) in a specified folder.

11.3 Prepare for the sample

This section provides an overview of the ITSO Bank (RedBank) Web application, and describes the prepatory steps as a sample, such as creating a Web Project and importing supporting application classes.

The section is organized as as follows:

- ITSO Bank Web application overview.
- Create a Web Project.
- Import the ITSO Bank model.

Archive

11.3.1 ITSO Bank Web application overview

In this section we describe the architecture for our sample ITSO Bank Web application, also known as RedBank. Note that the intent of this chapter is to introduce you to the Application Developer's tools that make the development of Web applications possible. Together we will work on only a single HTML page, a single servlet, and a single JSP page. The rest of the application has already been developed and is made available to you so that you can explore it if you would like to.

The RedBank application was designed using the MVC architecture pattern introduced in 11.1.2, "Model-view-controller (MVC) pattern" on page 503. Since the same example is used throughout the book, you will have the opportunity to see how little it changes in the face of varying design constraints and evolving techniques. This is in fact the most important characteristic of the MVC pattern.

The following sections describe how the RedBank application implements each of the MVC layers, to gain a better understanding of its design and understand how the JSP and servlet technologies are used within the MVC pattern.

Model

The ITSO RedBank application's business model is depicted in Figure 11-8.

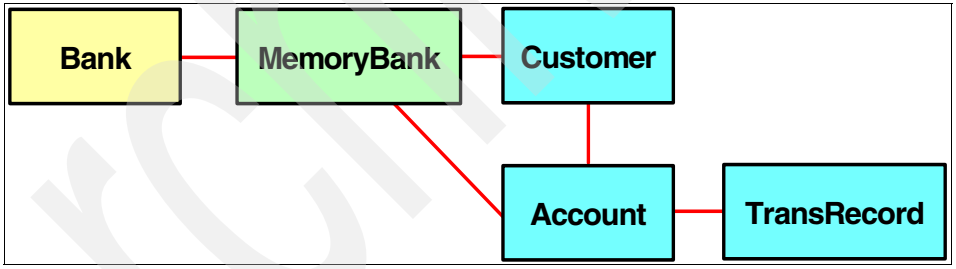


Figure 11-8 Banking model revisited

Controller

The control layer was implemented using two different strategies: One straightforward; and the other a little bit more complex, but more realistic. We did so to keep the discussion in the book simple, but still have a nice example.

The application has a total of four servlets:

- ListAccounts: Get the list of accounts for one customer.
- AccountDetails: Display the account balance and the selection of operations: List transactions, deposit, withdraw, and transfer.
- Logout: Invalidate the session data.

- PerformTransaction: Perform the selected operation by calling the appropriate control action: ListTransactions, Deposit, Withdraw, or Transfer.

Three of the servlets, including the ListAccounts servlet that you will implement, fall into the first category. They work as sole controllers, without any external collaboration. It is easier to implement and understand them this way.

The last of the four servlets, PerformTransaction, falls into the second category. It acts as a front controller, simply receiving the HTTP request and passing it to the appropriate control action object. These objects are responsible for carrying out the control of the application. For a more thorough explanation of this strategy, and the motivation behind it, please read Chapter 12, "Develop Web applications using Struts" on page 615.

Note: Action objects, or commands, are part of the command design pattern. For more information, refer to *Design Patterns: Elements of Reusable Object-Oriented Softwar*e.

View

The RedBank application's view layer is comprised of an HTML file and four JSP files. The application home page is the index.html. The home page allows you to type in the customer ID to access the customer services. There is no dynamic content in this page, so we use plain HTML. Note that security issues (logon and password) are not covered in this book.

Off of the home page are three HTML pages (Rates.html, Insurance.html, and RedBank.html). Both Rates.html and Insurance.html contain plain HTML content. The RedBank.html, when launched, will display the customer's accounts for selection.

The customer's name and the available accounts are processed dynamically, as they depend on the given customer ID, so we implemented this page as a JSP.

After selecting an account, the user can view the logged transactions or perform banking transactions, such as deposit, withdraw, and transfer.

The maintenance screen also shows the current account number and balance, both dynamic values. A simple JavaScript code controls whether the amount and destination account fields are available, depending on the option selected.

This is a mostly dynamic page. The user may check the transaction number, date, type, and amount. The color of the table rows alternate for readability reasons.

Finally, if anything goes wrong in the regular flow of events, the exception page is shown to inform the user of the error.

Facade

We will use a copy of the facade in the Web application. It is better to have the facade in the Web application to be able to access a different model that is implemented in another project (for example, as EJBs).

Application flow

The flow of the application is shown in Figure 11-9 on page 517:

- The view layer is comprised of four HTML files and four JSPs.
- The control layer is comprised of four servlets and four action classes. The PerformTransaction servlet passes control to one of the action classes.

  You will implement the ListAccounts servlet.
- The model layer is comprised of the facade and four model classes. All interactions from the servlets and actions classes go through the facade, the Bank class.

  The model is available in the ItsoProGuideJava project, which will be a utility project in the enterprise application.

Archived

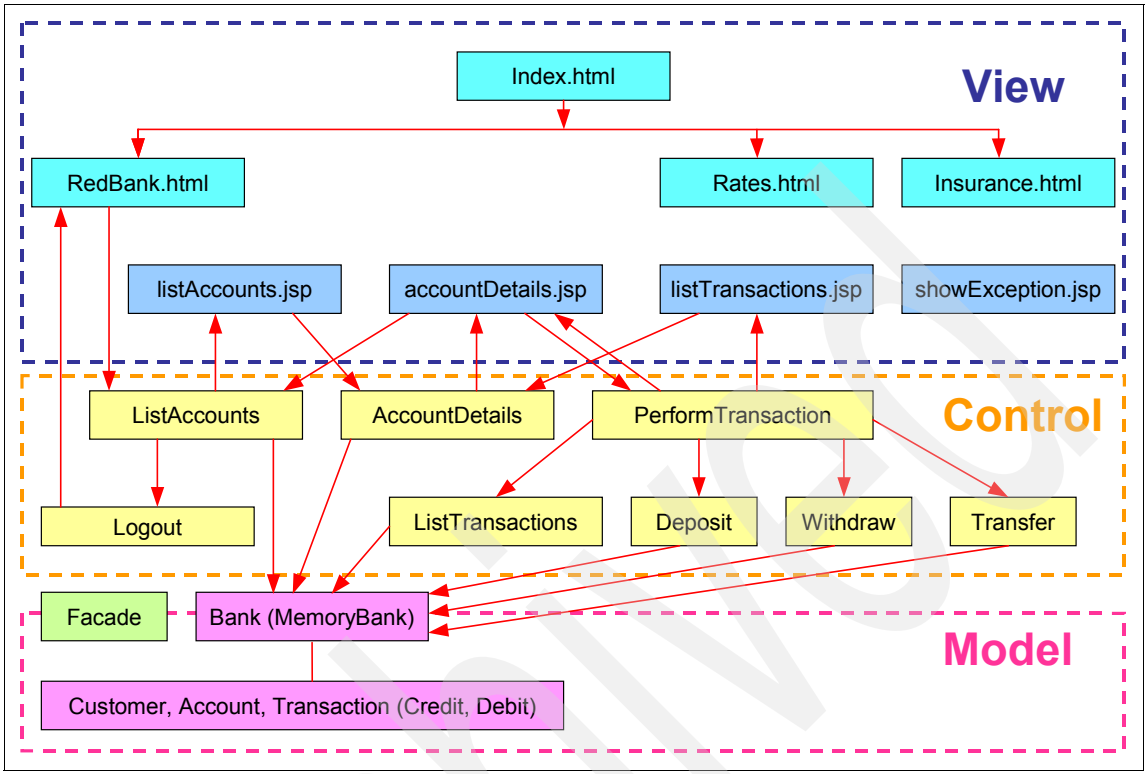


Figure 11-9 ITSO RedBank application flow

11.3.2 Create a Web Project

There are two types of Web Projects availabe in Rational Application Developer, namely Static and Dynamic. Throughout this chapter, we create both static and dynamic content. Thus, we will create a dynamic Web Project since it is capable of supporting tasks for static and dynamic content creation.

To create a new Web Project, do the following:

1. Start Rational Application Developer.
2. Open the Web perspective by selecting **Window** → **Open Perspective** → **Web**.
3. To create a new Web Project, select **File** → **New** → **Project**.
4. When the New Project dialog appears, select **Web** → **Dynamic Web Project** (as seen in Figure 11-10 on page 518), and then click **Next**.

Archived

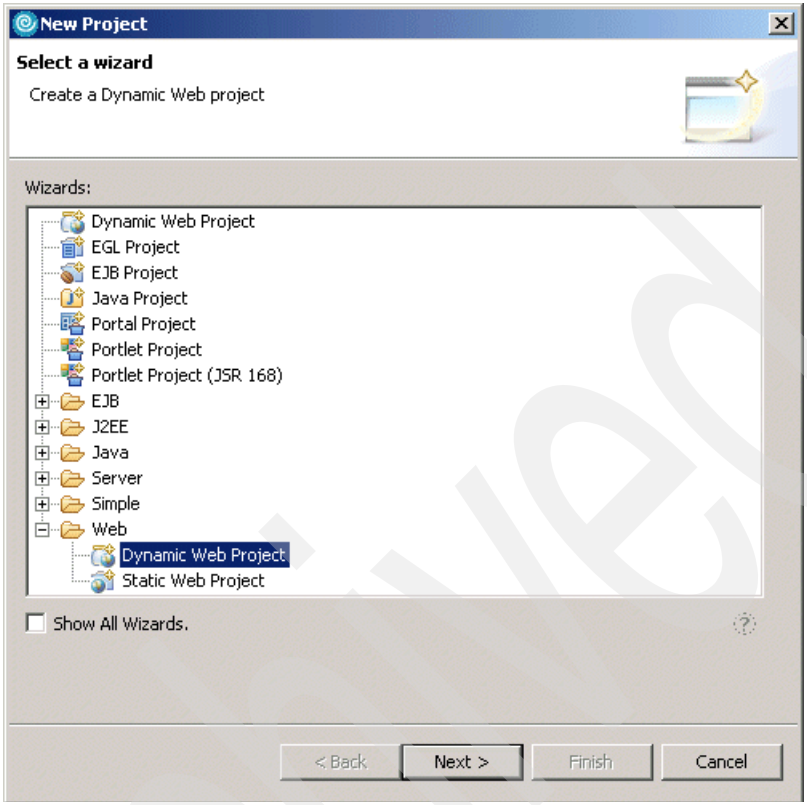


Figure 11-10 Create a Web Project

5. When the New Dynamic Web Project dialog appears, enter the following (as seen in Figure 11-11 on page 520), and then click **Next**:
   - Name: `BankBasicWeb`

     Most of the time, the only option you will enter is the project name. If the name is the only option you decide to supply, you can click **Finish** on this page.
   - Click **Show Advanced**.

     Advanced users may also want to change other options in this dialog.
   - Servlet version: Select **2.4** (default).

     Version 2.3 and 2.2 are also supported.
   - Target server: Select **WebSphere Application Server v6.0** (default).

Archived

This option will display the supported test environments that have been installed. In our case, we have only installed the integrated WebSphere Application Server V6.0 Test Environment.

– Check **Add module to an EAR project** (default).

  Dynamic Web Projects, such as the one we are creating, run exclusively within an enterprise application. For this reason, you have to either create a new EAR project or select an existing project.

– EAR project: `BankBasicWebEAR`

  Since we checked Add module to an EAR project, the wizard will create a new EAR project using the name value from above to create <name>EAR as the EAR project name.

– Context Root: `BankBasicWeb`

  The context root defines the Web application. The context root is the root part of the URI under which all the application resources are going to be placed, and by which they will be later referenced. It is also the top level directory for your Web application when it is deployed to an application server.

  Context roots are case-sensitive, as are all the Java URLs. Many developers like to make their context root all lowercase in order to facilitate the manual entering of URLs. The context root you select must be unique among all Web modules within the same application server cell. Application Developer's default is to use the project's name as the context root.

– Add support for annotated Java classes: In our example, we did not check this option.

  Annotation-based programming can be used to speed up application development by reducing the number of artifacts that you need to develop and manage on your own. By adding metadata tags to the Java code, the WebSphere Rapid Deployment tools can automatically create and manage the artifacts to build a J2EE-compliant module and application.

Archived

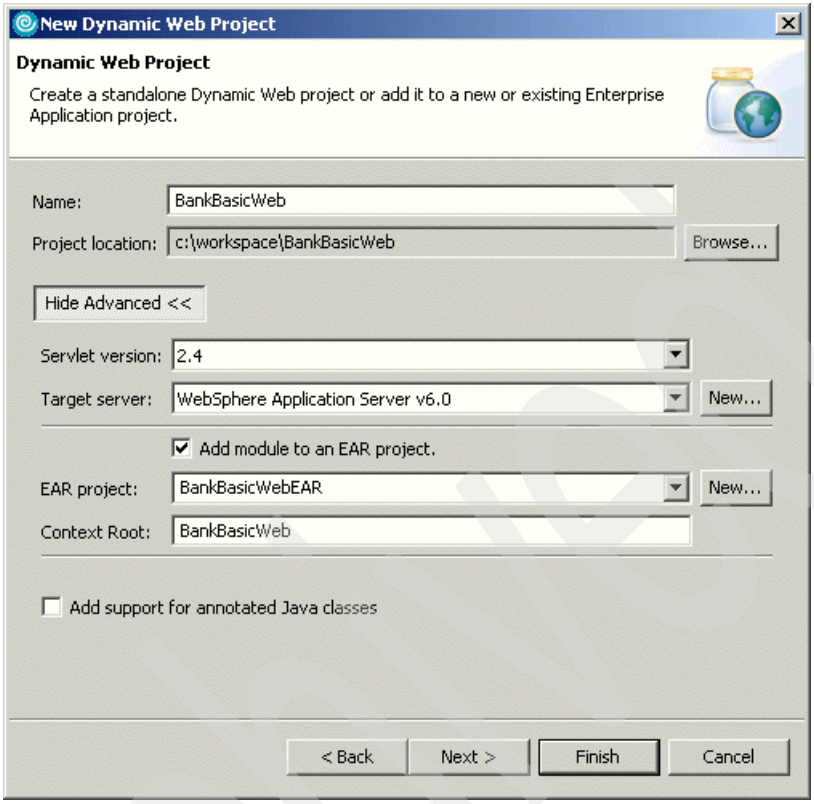


Figure 11-11 Create Dynamic Web Project settings

6. When the Dynamic Web Project - Features dialog appears, we accepted the default features seen in Figure 11-12 on page 521, and then clicked **Next**.

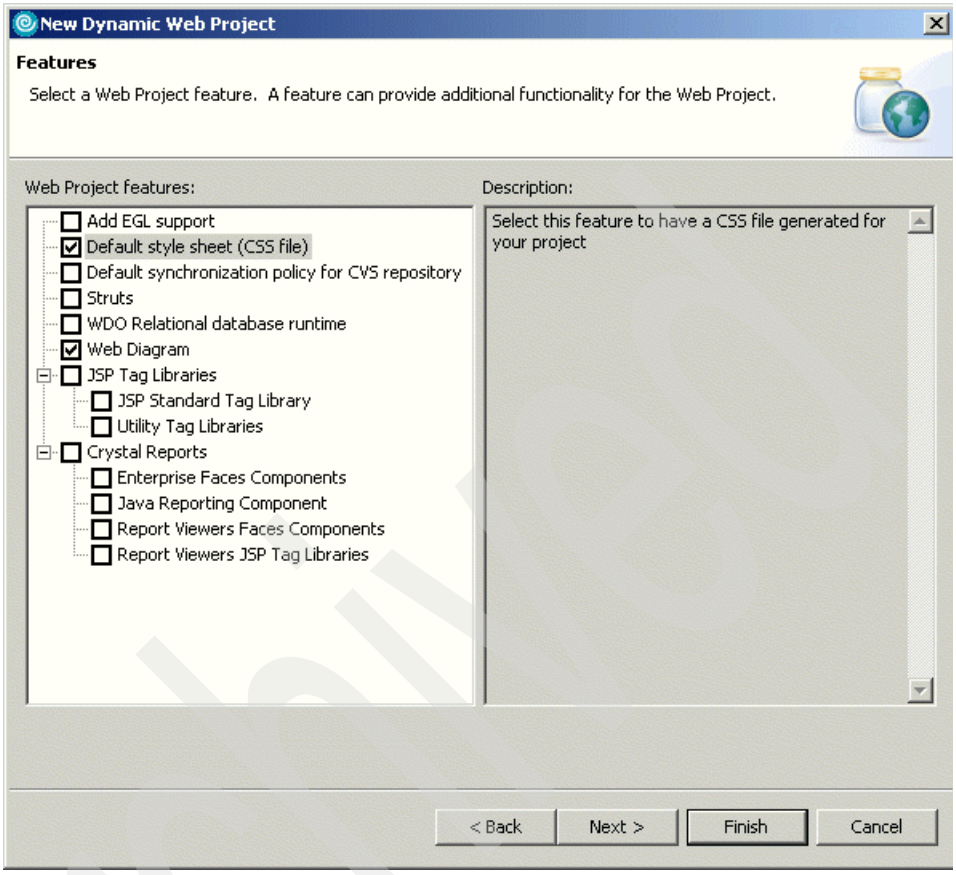


Figure 11-12 Dynamic Web Project Features selection

7. When the Select a Page Template for the Web Site dialog appears, we accepted the default, as seen in Figure 11-13 on page 522.

   This dialog allows you to select from Rational Application Developer supplied sample page templates, or you can select your own pre-defined page template.

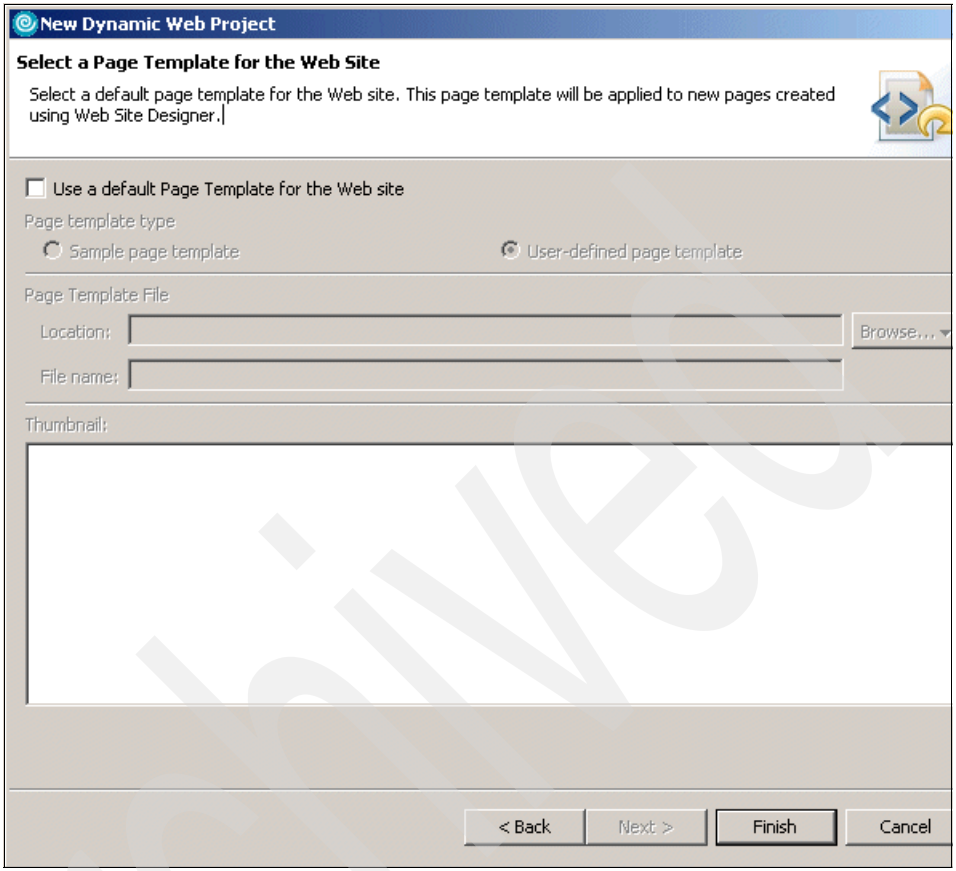


Figure 11-13 Select a Page Template for the Web Site

8. Click **Finish** and the Dynamic Web Project will be created.

11.3.3 Web Project directory structure

The Web Project directory structure for the newly created BankBasicWeb Project is displayed in Figure 11-14 on page 523.

Archive

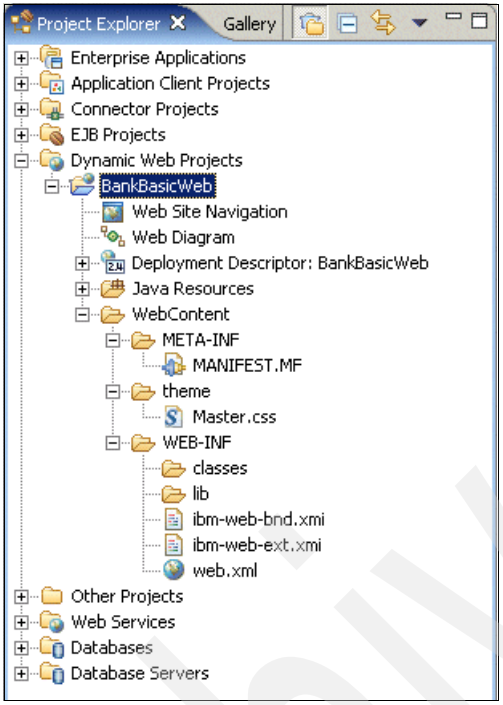


Figure 11-14 Web Project directory structure

- JavaSource: This folder contains the project's Java source code for regular classes, JavaBeans, and servlets. When resources are added to a Web Project, they are automatically compiled, and the generated files are added to the WebContent\WEB-INF\classes folder. By default, the contents of the source directory are not packaged in exported WAR files. If you want them to be, you have to select the appropriate option when exporting the WAR file.
- WebContent: This folder holds the contents of the WAR file that will be deployed to the server. It contains all the Web resources, including compiled Java classes and servlets, HTML files, JSPs, and graphics needed for the application.

  Important: Any files *not* under WebContent are considered design time resources (for example, .java and .sql files), and will not be deployed when the project is published. Make sure that you place everything that should be published under the WebContent folder.

  – WebContent\META-INF

Archived

This folder holds the MANIFEST.MF file, which describes the Web module's external dependencies.

– WebContent\theme

  Contains cascading style sheets and other style-related objects such as page templates.

– WebContent\WEB-INF

  This directory holds the supporting Web resources for the Web module, including the Web deployment descriptor (web.xml), IBM WebSphere extensions' descriptors (ibm-web-bnd.xmi and ibm-web-ext.xmi), and the classes and lib directories.

– WebContent\WEB-INF\classes

  Contains the project's Java-compiled code for regular classes, JavaBeans, and servlets. These are the Java classes that will be published to the application server and loaded at runtime. The class files are automatically placed in this directory when the source files from the JavaSource directory are compiled. Any files placed manually in this directory will be deleted by the Java compiler when it runs.

– WebContent\WEB-INF\lib

  Contains utility JAR files that your Web module references. Any classes contained in these JAR files will be available to your Web module.

11.3.4 Import the ITSO Bank model

The ITSO Bank Web application requires the classes created in Chapter 7, "Develop Java applications" on page 221. This section describes how to import the BankJava.zip Project Interchange file.

1. Select **File** → **Import**.
2. When the Import dialog appears, select **Project Interchange** and click **Next**.
3. When the Import Projects dialog appears, enter `c:\6449code\java\BankJava.zip` in the From zip file field, check the **BankJava** project, and then click **Finish**.

11.4 Define the site navigation and appearance

In this section, we demonstrate how to define the site pages and navigation using the Web Site Designer. In addition, we will create a page template and style sheet to provide a common appearance on the site pages. The page template is used to define a standard page layout (header, navigation menu,

footer, etc.), whereas a stylesheet is used by page templates to define fonts, colors, table formatting, etc.

This section includes the following tasks:

- Launch the Web Site Designer.
- Create a new page template.
- Create the Web site navigation and pages.
- Customize a page template.
- Customize a style sheet.

11.4.1 Launch the Web Site Designer

The Web Site Designer can be launched by clicking the Web Site Navigation file found in the Web Project, as follows:

1. From the Web perspective Project Explorer view, expand **Dynamic Web Projects** → **BankBasicWeb**.
2. Double-click **Web Site Navigation**, as seen in Figure 11-15.

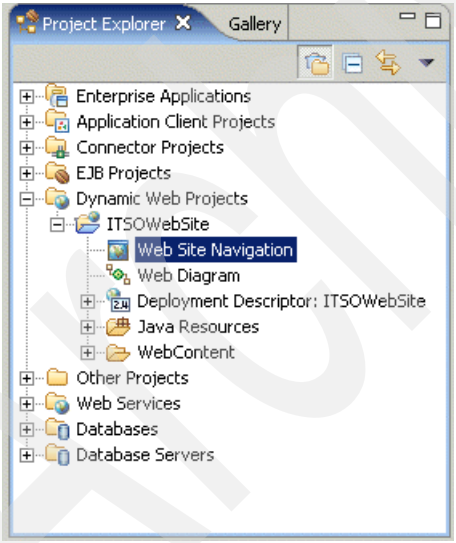


Figure 11-15 Launch Web Site Designer

3. When the Web Site Designer opens (as seen in Figure 11-16 on page 526), take note of the following features.
   - Navigation and Detail view: There is a Navigation view (default) and a Detail view used to visually design the layout of the site; and a Detail view used to define the structure such as ID, navigational label, file path and name /URL, serlet URL, page title, etc.

Archived

– Palette: The Palette parts can be dragged to the Navigation page. For example, the New Page part can be dragged from the Palette to create a new page.

  There is also the capability to create a *New Group*, which can be used to logically organize pages in a hiearchy grouping.

– Site template: In the Properties view, you can select a Site template to define the appearance of the site from a sample page template included with Rational Application Developer, or a user-defined page template.

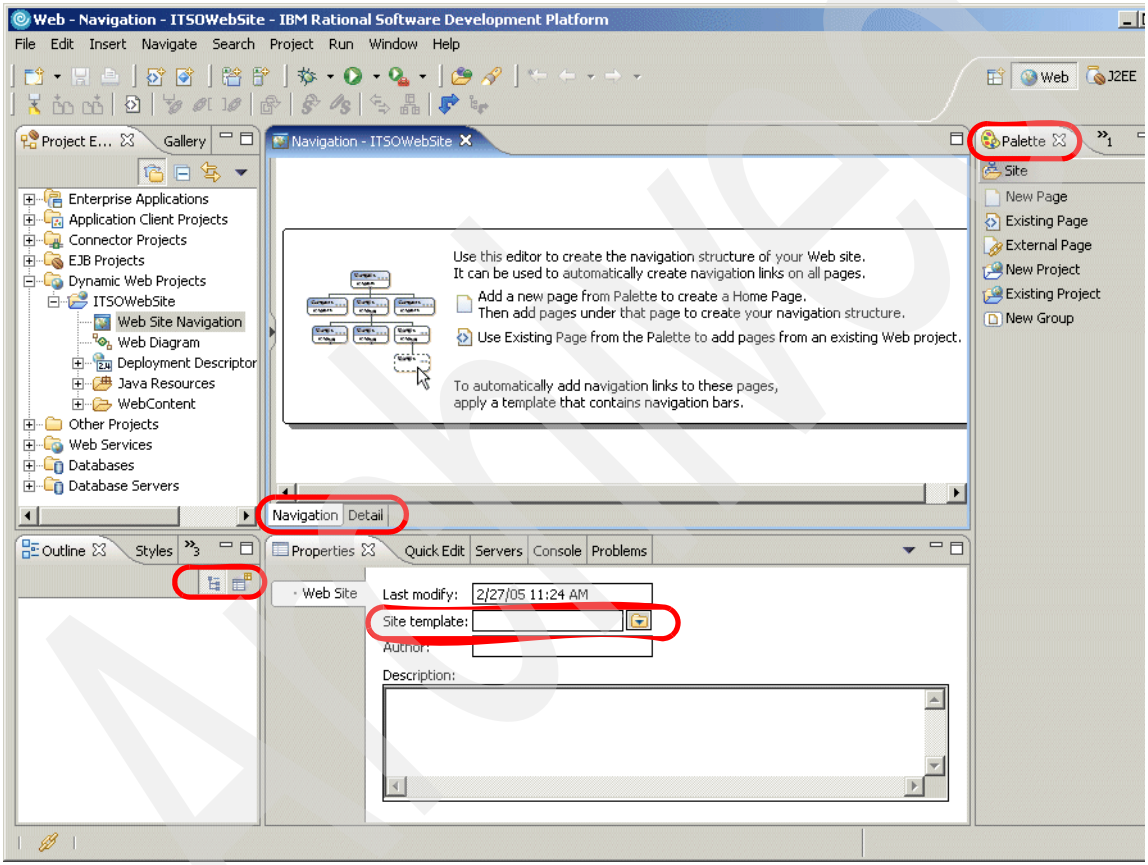


Figure 11-16 Web Site Designer - Navigation view

11.4.2 Create a new page template

Page templates provide an efficient methods of creating a common layout for Web pages. Page templates can be created from an existing sample page template or be user-defined.

The ITSO Web site user interface (view) will be made up of a combination of static HTML pages and dynamic JSPs. In this section we describe how to create a static page template (itso_html_template.htpl) and a dyanamic page template (itso_jsp_template.jtpl) from existing sample page templates, and then customize the page templates to be new user-defined templates.

We recommend that you create a page template prior to creating the pages in the Web Site Designer so that you can specify the page template at the time you create the page.

Tip: The Rational Application Developer Tutorials Gallery includes a Watch and Learn module on Enriching Web pages with page templates. In the Tutorial example, the page template is created from scratch, whereas in our sample we created our page template from an existing sample page template.

Page templates versus style sheets

Though style sheets have not been discussed as of yet, there may be some confusion about page templates and style sheets. Page templates are used if you wish to create a standard layout for the JSP and HTML pages in your application. For example, you may want each page in your application to have a header, a navigation menu on the left side of the page, and a footer. All of these elements can be created in a page template and then used by the pages in your application; this obviously will aid in the maintenance of your application, as changes only need to be made in one area.

Style sheets, on the other hand, are used by page templates (or JSP and HTML pages) to set fonts, colors, table formatting, etc. Again, like page templates, maintenance is done in one area, which will save time.

Create a static page template

To create a new user-defined static page template (itso_html_template. htpl) to be used by the ITSO Web site HTML pages, do the following:

1. Open the Web perspective Project Explorer view.
2. Expand **Dynamic Web Projects** → **BankBasicWeb** → **WebContent** → **theme**. Selecting the folder in which you want to create the template before launching the wizard will save time entering the Folder path.
3. Select **File** → **New** → **Page Template File**.
4. When the New Page Template File dialog appears, we entered the following (as seen in Figure 11-17 on page 528), and then clicked **Next**:
   - Folder: `/BankBasicWeb/WebContent/theme`
   - File name: `itso_html_template`

– Markup language: Select **HTML**.
– Check **Create from page template**.
– Model: Select **Template containing only HTML**.

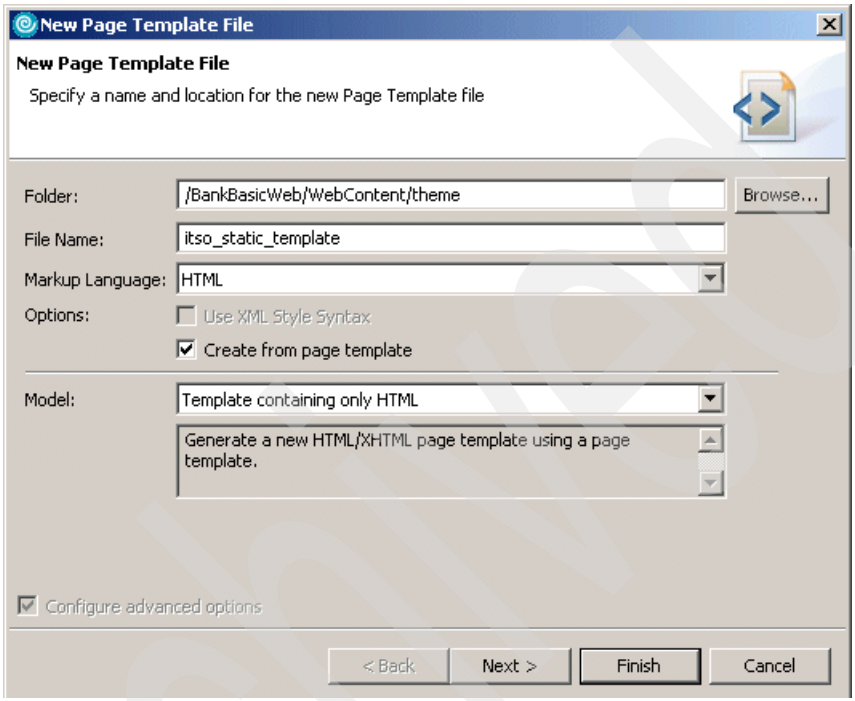


Figure 11-17 New Page Template File - itso_static_template

5. When the Select a page to use when creating this file dialog appears, select **Sample page template** under Page template type, select the **A-01_gray.htpl** thumbnail representing the template (as seen in Figure 11-18 on page 529), and then click **Finish**.

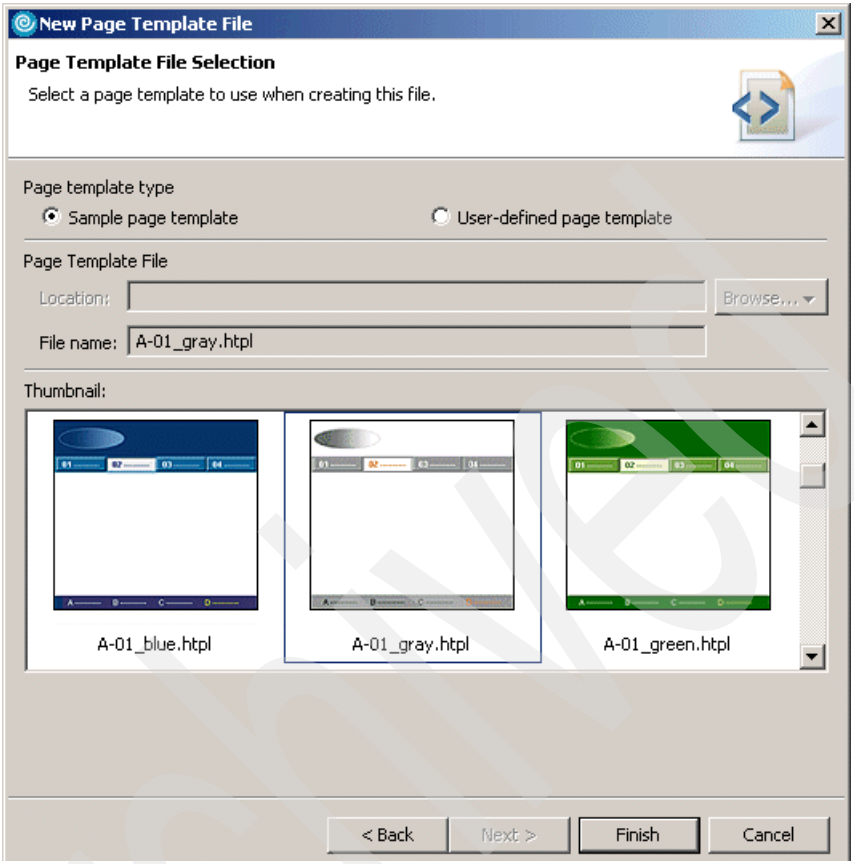


Figure 11-18 Select the sample page template

6. You will see a dialog stating `A page template must contain at least one Content Area which is later filled in by the pages that use the template.` Click **OK**.

   Take note of the files that have been created in the theme directory as a result of creating the page template from an existing sample template.

   – itso_html_template.htpl: This is the new template created with references to the existing template it was created from (A-01_gray.htpl).
   – A-01_gray.htpl: This is copy of the sample template.
   – logo_gray.gif: This is the logo for the A-01_gray.htpl template.
   – gray.css: This is a css file from the existing template.
   – nav_head.html: This is the navigation header file.
   – footer.html: This is the footer file.

7. Create your own page template from the files generated.

> **Note:** The Page Template wizard is designed in such a way that the new page template created references the sample, and thus all page elements of the new template are read-only, including the content area of the new page template. This is not the desired behavior for what we wanted to do, so we created a hybrid procedure of creating the page template and then customizing it to be a user-defined page template.

   a. Delete the itso_html_template.htpl created by the Page Template wizard.
   b. Rename the A-01_gray.htpl page template copied to the theme directory to itso_html_template.htpl. Select the file, right-click, and select **Refactor** → **Rename**.
   c. When prompted with `Do you want to have the links to or from those files fixed accordingly?`, click **Yes**.

      We will later explain how to customize the itso_html_template.htpl in "Customize a page template" on page 531.

Create a dynamic page template

To create a new user-defined dynamic page template (itso_jsp_template.jtpl) to be used by the ITSO Web site JSPs, do the following:

1. Open the Web perspective Project Explorer view.
2. Expand **Dynamic Web Projects** → **BankBasicWeb** → **WebContent** → **theme**. Selecting the folder in which you want to create the template before launching the wizard will save time entering the Folder path.
3. Select **File** → **New** → **Page Template File**.
4. When the New Page Template File dialog appears, we entered the following and then clicked **Next**:
   - Folder: `/BankBasicWeb/WebContent/theme`
   - File name: `itso_jsp_template`
   - Markup language: Select **HTML**.
   - Check **Create from page template**.
   - Model: Select **Template containing JSP**.
5. When the Select a page to use when creating this file dialog appears, select **Sample page template** under Page template type, select the **JSP-A-01_gray.jtpl** thumbnail representing the template, and then click **Finish**.

Archive

6. You will see a dialog stating `A page template must contain at least one Content Area which is later filled in by the pages that use the template`. Click **OK**.

   Take note of the files that have been created in the theme directory as a result of creating the page template from an existing template.

   – itso_jsp_template.jtpl: This is the new template created with references to the existing template it was created from (JSP-A-01_gray.jtpl).
   – JSP-A-01_gray.jtpl: This is copy of the sample template.
   – logo_gray.gif: This is the logo for the JSP-A-01_gray.jtpl template.
   – gray.css: This is a css file from the existing template.
   – nav_head.jsp: This is the navigation header JSP include file.
   – footer.jsp: This is the footer JSP include file.

7. Create your own page template from the files generated.

   Note: The Page Template wizard is designed in such a way that the new page template created references the sample, and thus all page elements of the new template are read-only, including the content area of the new page template. This is not the desired behavior for what we wanted to do, so we created a hybrid procedure of creating the page template and then customizing it to be a user-defined page template.

   a. Delete the itso_jsp_template.jtpl.
   b. Rename the JSP-A-01_gray.jtpl page template copied to the theme directory to itso_jsp_template.jtpl. Select the file, right-click, and select **Refactor** → **Rename**.
   c. When prompted with `Do you want to have the links to or from those files fixed accordingly?`, click **Yes**.

      We will later explain how to customize the itso_jsp_template.jtpl in 11.4.3, “Customize a page template” on page 531.

11.4.3 Customize a page template

This section describes how to make some common page template customizations, such as modifying the layout of the header, navigation, and footer, as well as adding a company logo image.

Customize the static page template

In this section, we describe how to customize the following elements of the static page template (itso_html_template.htpl) created in “Create a static page template” on page 527:

- Customize the logo and title.
- Insert links on the navigation bar.
- Insert the content area.

Customize the logo and title

Modify the page template to include the ITSO logo image and ITSO RedBank title. In order to modify the logo displayed in itso_static_template.htpl, we must really update the image in the A_gray.htpl that is referenced.

1. Import the itso_logo.gif image.
   a. Expand **Dynamic Web Projects** → **BankBasicWeb** → **WebContent**.
   b. Right-click the **theme** folder, and select **File** → **Import**.
   c. Select **File System** and click **Next**.
   d. Enter `c:\6449code\web` in the From directory, check **itso_logo.gif**, and then click **Finish**.
2. Double-click **itso_static_template.htpl** to open the file.
3. You may see the message `A page template must contain at least one Content Area which is later filled in by the pages that use the template.` Click **OK**.
4. Click the **Design** tab.
5. Double-click the oval logo image at the top of the page template in the Design view.
6. Click the **Source** tab. The source with the image should be highlighted in the editor.
7. Modify the source for the following, as seen in Example 11-1:
   – Change the image file name from logo_gray.gif to itso_logo.gif.
   – Change the <td width to 70 (original 150), change the itso_logo.gif <img width to 60 (original 150), and change the height to 50 (original 55).
   – Add `ITSO RedBank` as an <H1.

Example 11-1 Modified itso_html_template.htpl for itso_logo.gif and title

```
<tr>
   <td width="70"><img border="0" width="60" height="50" alt="Company's LOGO"
src="/BankBasicWeb/theme/itso_logo.gif"></td>
   <td><H1>ITSO <Font color="red">RedBank</Font></H1></td>
```

8. Save the itso_html_template.htpl file.
9. Verify that the page template looks as desired by clicking the **Preview** view.

Insert links on the navigation bar

The navigation bar is a set of links to other Web pages, displayed as a horizontal or vertical bar, in a Web site. The navigation bar typically includes links to the home page and other top-level pages. It is beneficial to insert navigation bars (links) in a page template, rather than adding to each individual page.

In our example, we created our page template from a sample page template that included a navigation bar that displays links for the children of the top page (index.html).

Rational Application Developer makes adding a navigation bar very easy. Simply drag and drop the Horizontal or Vertical bar, found under the Web Site Navigation parts Palette, to the page. You can then select the type from sample or user-defined thumbnails representing the bar type.

It is also easily add files or URLs using the **Insert** → **Link** toolbar option, pressing Ctrl+Q, or dragging and droping the Link icon from the Palette. The types of links you can choose from include file, http, FTP, e-mail, or other.

Insert the content area

If you have created a page template from a sample page template, you need to ensure that a content area exists in the page template, which is used to display the unique content on a page. In our sample we do have a content area that was defined as part of creating the page template.

Note: If you need to add a new content area to a page template, do the following:

1. Select the **Page Template** drawer on the Palette and drag **Content Area** from the Palette to the desired location on the page template.
2. When the Insert Content Area for Page Template dialog appears, enter `itso_html_content` and then click **OK**.

Insert a free table layout into the content area

Free layout tables create tables and cells for you automatically so that you can freely place objects on the page. In our example, we chose to add the free table layout to the page template to provide a starting point for adding unique content to the pages.

1. You will notice that the only part of the page that is not read-only is the content area marked `Place your page content here`. Select the text in the content area and delete it.
2. Select the content area, right-click, and select **Layout Mode** → **Free Layout**.
3. Expand **HTML Tags** from the Palette.
4. Drag the free layout table to the content area.
5. Drag the corner of the free layout table to increase the size. For example, we increased the width to match the width of the navigation bar, and increased the height to 300 in the Properties view, as seen in Figure 11-19.

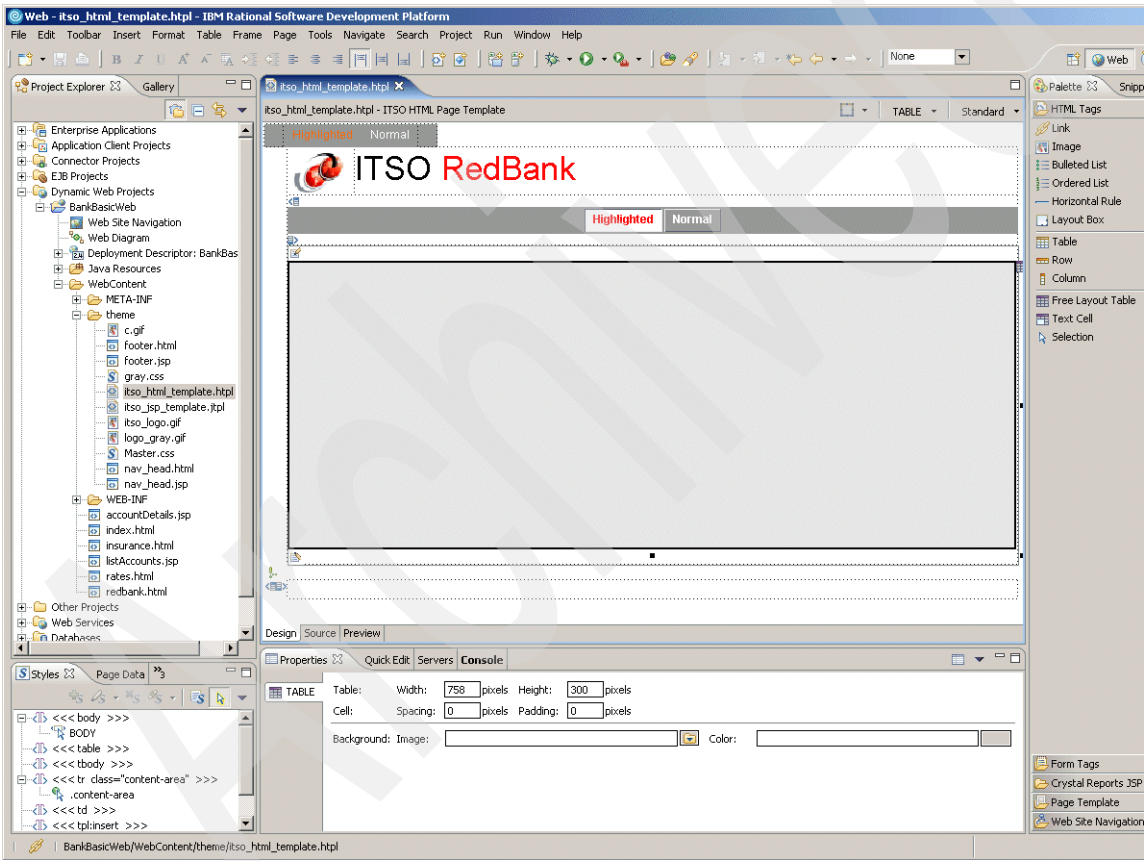


Figure 11-19 Customizing the itso_html_template.htpl

Customize the dynamic page template

The customization of the dynamic page template is virtually identical to that of the static page template for our example.

1. Double-click the **itso_jsp_template.jhtpl** file. Modify the page as described in “Customize the static page template” on page 532.
2. Disable the display of sibling pages on the footer.

   In the case of the dyanamic page template, we want to turn off the display of sibling pages on the footer.

   a. Double-click the **itso_jsp_template.jtpl** file to open in Page Designer.
   b. Click the **Design** tab.
   c. Right-click in the page, and select **Layout Mode** → **Standard**.
   d. Select the icon at the bottom of the page in the footer.
   e. Click **Link Destination** in the Properties view.
   f. Uncheck **Sibling pages**, and check **Top page**.
   g. Save the page template file.

Methods of applying a page template to a page

Page templates can be applied to a Web page in one of the following ways:

- By specifying the template when you use a wizard to create the page. This is true regardless of whether you are using the wizard from a navigation view, a tool bar, or the Web Site Designer.
- By using the Web Site Designer. You can apply a template as you create pages or to existing pages as you add them to the Web Site Configuration. You can also apply or replace templates to pages already in the configuration.
- By using the Page Designer. With the page open, select **Page** → **Page Template**. You can apply, replace, or detach the template. You also have the option of opening the template for modification.

A default page template can be specified when you create a Web Project. If you specify a default template, it will be automatically selected when given the option in wizards to select a template for a page.

11.4.4 Customize a style sheet

Style sheets can be created when a Web Project is created, when a page template is created from a sample, or at any time by launching CSS File creation wizard. In our example, we created a style sheet named gray.css as part of the process of creating a page template from a sample. Both the static (itso_html_template.htpl) and dynamic (itso_jsp_template.jhtpl) reference the gray.css as desired to have a common appearance of fonts and colors for the pages.

Archived

In the following example we customize the colors used for the navigation bar links, and supporting variations such as hover links. By default the link text in the navigation bar is orange (cc6600). We will customize this to be red (FF0000). To find the hexadecimal HTML color code, we suggest you do an Internet search.

To customize the gray.css style sheet, do the following:

1. Expand **Dynamic Web Projects** → **BasicBankWeb** → **WebContent** → **theme**.
2. Double-click the **gray.css** file to open in the CSS Designer.

   The CSS Designer will appear as seen in Figure 11-20. By selecting the source in the right-hand window, the text in the left-hand window will become highlighted and display, for example, the font and color. This makes it easy to change the settings and see the change immediately.

Archived

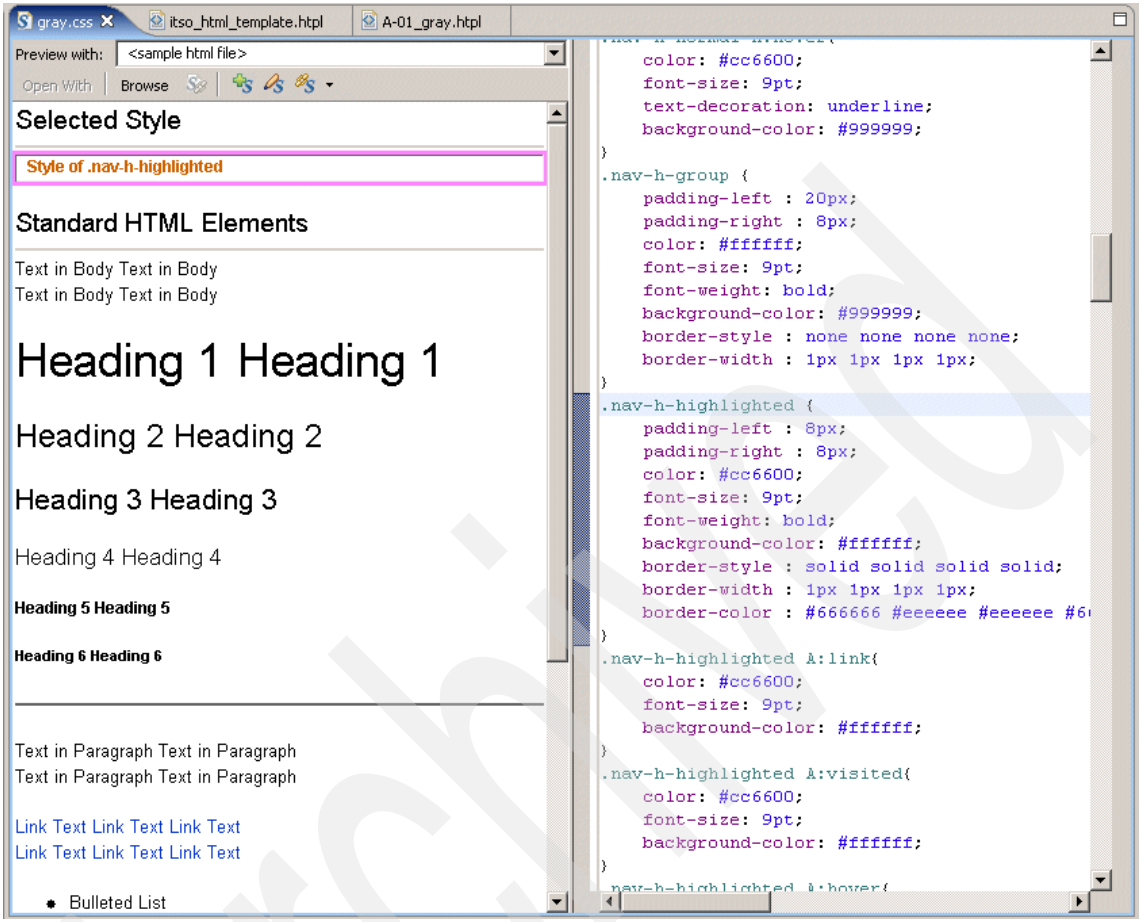


Figure 11-20 CSS Designer - Example gray.css

3. Customize the header highlighted link text.
   a. Move to the source view (right-hand window), and scroll down until you find `.nav-h-highlighted`. We found this by first double-clicking the area we wanted to change in the Design view for the page template, and then looking at the source.
   b. Customize the Hex HTML color code for all nav-h-highlighted entries from `color: #cc6600;` (orange) to `color: #FF0000;` (red).
4. Customize the footer highlighted link text.
   a. Move to the source view (right-hand window), and scroll down until you find `.nav-f-highlighted`. We found this by first double-clicking the area

Archived

we wanted to change in the Design view for the page template, and then looking at the source.

b. Customize the Hex HTML color code for all nav-h-highlighted entries from `color: #cc6600;` (orange) to `color: #FF0000;` (red).

5. Save the gray.css file (press Ctrl+S).

11.4.5 Create the Web site navigation and pages

In this section, we use the Web Site Designer to visually construct the page navigation and page skeletons for the ITSO Bank Web site. In later sections, we add the content for these pages.

We will create the navigation pages and corresponding HTML or JSP page for each of the following:

- itsohome (index.html)
- Rates (rates.html)
- Insurance (insurance.html)
- RedBank (bank.html)
- listaccounts (listAccounts.jsp)
- accountdetails (accountDetails.jsp)
- listtransactions (listTransactions.jsp)
- showexception (showException.jsp)

To define the site navigation, as well as create the HTML and JSP page skeletons, do the following in the Web Site Designer:

1. If you have not already done so, launch the Web Site Desinger.

   For details refer to 11.4.1, “Launch the Web Site Designer” on page 525.

2. Create the root static navigation page.

   We will use the itsohome navigation and corresponding index.html as an example by which the other static pages listed above should be created.

   a. Select **New Page** from the Palette and drag it to the Navigation page.

   b. After the New Page was added, notice you can type the navigation label in the Navigation page or in the Properties view under Navigation label. We entered `itsohome`.

   c. Save the Navigation page (press Ctrl+S).

   d. Double-click the **itsohome** navigation page to create the corresponding HTML file associated with the navigation page.

   e. When the Create a page dialog appears, select **HTML** and click **OK**.

Archived

f. When the HTML File location dialog appears, we entered the following (as seen in Figure 11-21), and then clicked **Next**:

- Folder: `/BankBasicWeb/WebContent`
- File name: `index.html`

  By default a Web Project will look for index.html (or index.htm) when the project is run on server. Although this behavior can be changed, we recommend that you use index.html as the top level page name.
- Markup language: Select **HTML**.
- Options: Check **Create from page template**.

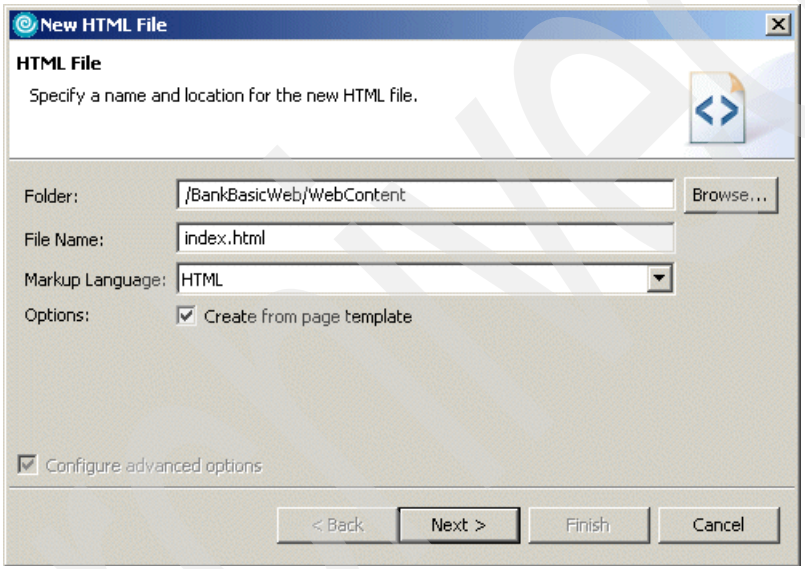


Figure 11-21 Create the itsohome - index.html page

g. When the Specify a Page Template dialog appears, select **User-defined page template**, select the **itso_html_template.htpl (**as seen in Figure 11-22 on page 540), and then click **Finish**.

Note: When the page is created, the page is set as a navigation candidate by default. The top most page (for example, itsohome index.html) should be set as the Navigation root by selecting the page, right-clicking, and selecting **Navigation** → **Set Navigation Root**.

Also, pages are set by default as map cadidates. This feature is a toggle. If you want to change the page to not be visiable in the site map, select the page, right-click, and select **Site Map** → **Site Map**.

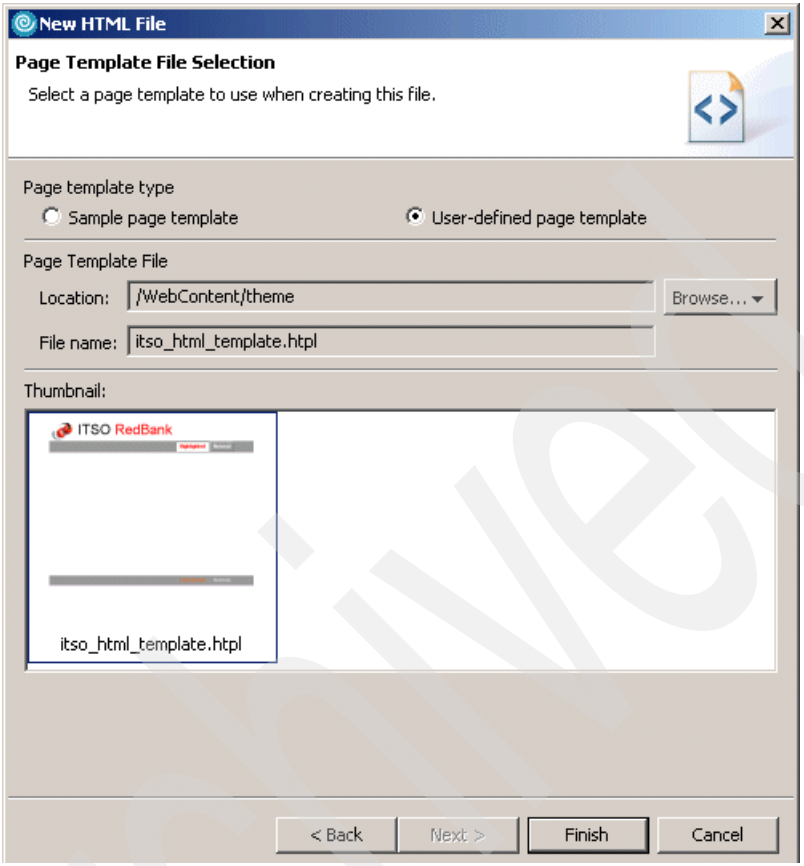


Figure 11-22 Specify a Page Template

3. Define the navigation root.

   In our example, itsohome (index.html) is the *Navigation root*. By default, when a page is created it is set as a *Navigation candidate*, which is the desired format for all other pages.

   To enable itsohome (index.html) as the Navigation root, select **itsohome**, right-click, and select **Navigation** → **Set Navigation Root**.

4. Add the following static pages as children of itsohome.

   – rates (rates.html)
   – redbank (redbank.html)
   – insurance (insurance.html)

   a. Select **itsohome**, right-click, and select **Add New Page** → **As Child**.

Notice the link and child relationship of the new page (rates) to the parent (itsohome) in the Navigation window.

b. The remaining steps are the same as used to create the itsohome static page.

c. Repeat the process to add `insurance` and `redbank` as children pages of itsohome.

d. Save the Navigation page (Ctrl+S).

5. Create a new group named RedBank.

   Groups are used to logically build or organize pages into a reusable group. For example, consider a footer that includes an About, Privacy, Terms of Use, and Contact. These can be logically grouped into a group named Footer.

   In our example, we are using the group to hold JSPs temporarily until we create the servlets that will control the navigation and interaction of the JSPs.

   a. Select the **redbank** page, right-click, and select **Add New Group** → **As Child**.

   b. Save the Navigation page (Ctrl+S).

6. Create the dynamic pages.

   Note: The interaction between the dynamic pages for the ITSO Bank Web is controlled by servlets. At this stage we will use the Web Site Designer to create the following JSPs and assign the itso_jsp_template.htpl page template and place them in the RedBank group. The navigation for these pages is not representative of the final version of the sample.

   We will create the following JSPs and put them in the RedBank group:

   – listaccount (listAccounts.jsp)
   – accountdetails (accountDetails.jsp)
   – listtransactions (listTransactions.jsp)
   – showexception (showException.jsp)

   a. Select **New Page** from the Palette and drag it to RedBank Group.

   b. After the New Page was added, notice you can type the navigation label in the Navigation page or in the Properties view under Navigation label. We entered `listaccount`.

   c. Save the Navigation page (Ctrl+S).

   d. Double-click **listaccount** to create the JSP file associated with the navigation label.

   e. When the Create a page dialog appears, select **JSP** and click **OK**.

Archived

f. When the HTML File location dialog appears, we entered the following and then clicked **Next**:

- Folder: `/BankBasicWeb/WebContent`
- File name: `listAccount.jsp`
- Markup language: Select **HTML**.
- Options: Check **Create from page template**.

g. When the Specify a Page Template dialog appears, select **User-defined page template**, select the **itso_jsp_template.jtpl**, and then click **Finish**.

h. Repeat the process to add the accountdetails (accountDetails.jsp) and showexception (showException.jsp) dynamic pages.

i. Select the **showexception** page, right-click, and select **Navigation** → **Show in Navigation** (toggle, enabled by default).

j. Save the Navigation page (Ctrl+S).

When done adding the navigation, HTML, and JSP pages, the Navigation page should look like Figure 11-23 on page 542 at this stage.

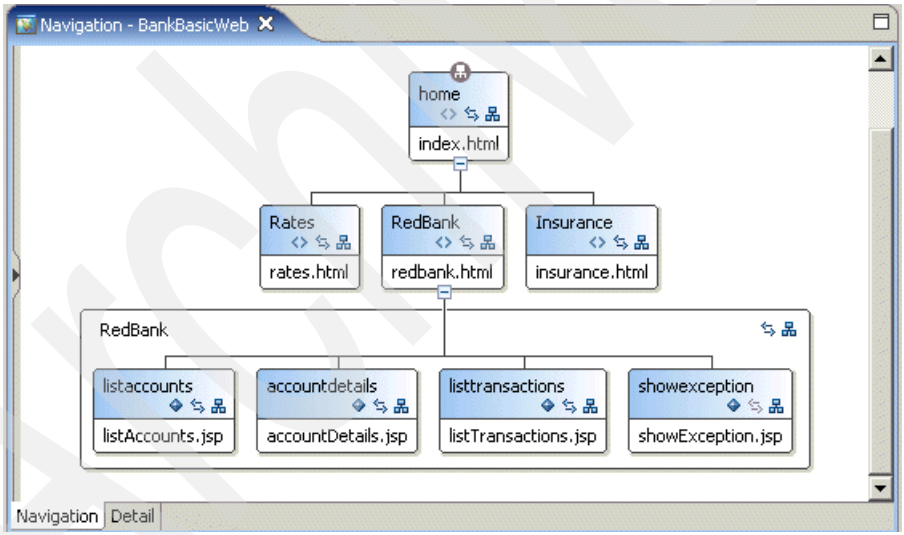


Figure 11-23 Navigation page after adding pages

11.4.6 Verify the site navigation and page templates

At this stage the site navigation is not complete, nor have we completed adding the static or dynamic content. We can, however, verify that the top level site navigation, page templates, and style sheet appearance is as desired. To verify

the appearance, you can use the Preview view for the page, or run the application on a test server.

To run the ITSO RedBank Web application in the WebSphere Application Server V6.0 Test Environment to verify the site navigation and page appearance, do the following:

1. Expand **Dynamic Web Applications**.
2. Right-click **BankBasicWeb**, and select **Run** → **Run on Server**.
3. When the Define a New Server dialog appears, select **Choose and existing server**, select **WebSphere Application Server v6.0**, and then click **Finish**.

> **Note:** If you do not have a test environment configured, refer to Chapter 19, “Servers and server configuration” on page 1043.

4. At this stage there is not a great deal to test, but this same concept can be used as you add static and dynamic content. Notice the URL in the Web browser to distiguish between pages (Rates, RedBank, Insurance), since there is no content at this point (see Figure 11-24 on page 543).

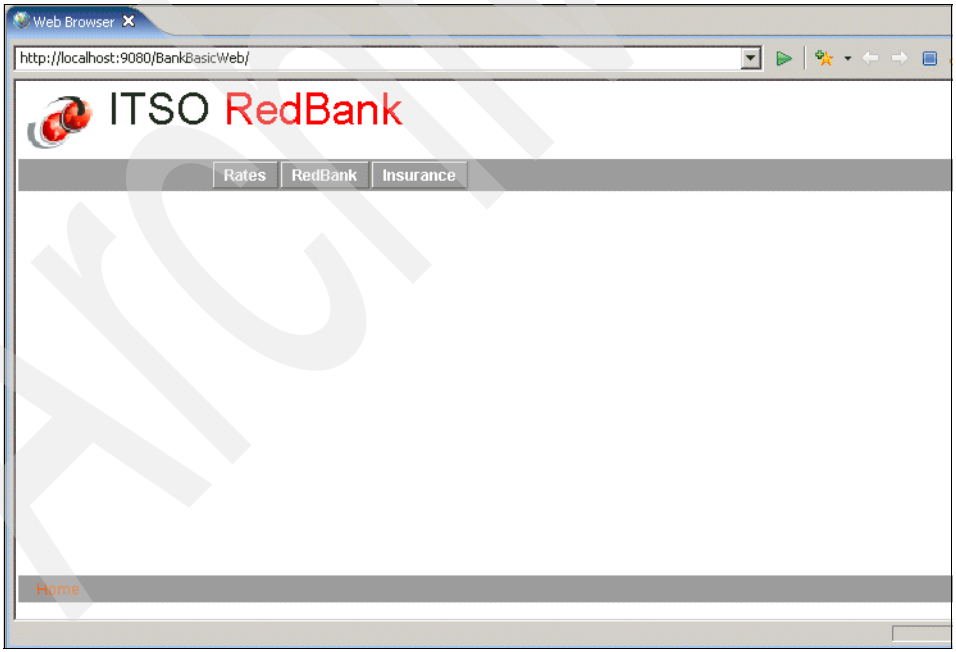


Figure 11-24 ITSO RedBank Web site

Archived

11.5 Develop the static Web resources

In this section we create the content for the four static pages of our sample with the objective of highlighting some of the features of Page Designer. We use each of the following pages to demonstrate features of Page Designer such as adding tables, links, text, images, customizing fonts on the HTML pages, and working with forms.

- Create the index.html page content (text, links).
- Create the rates.html page content (tables).
- Create the insurance.html page content (list).
- Create the redbank.html page content (forms).

In each example the page content will be created in the Design view, reviewed in the Source view, and verified in the Preview view of Page Designer.

11.5.1 Create the index.html page content (text, links)

The ITSO RedBank home page is index.html. The links to the child pages are included as part of the header and footer of the our page template. In the following example, we describe how to add static text to the page, and add a link to the page to the IBM Redbooks site.

1. Expand **Dynamic Web Projects** → **BankBasicWeb** → **WebContent**.
2. Double-click the **index.html** file to open in Page Designer.
3. Click the **Design** tab.
4. Insert the Welcome message text.
   a. Click the content area in which you wish to insert the text.
   b. Right-click and select **Layout Mode** → **Free Layout**.
   c. From the menu bar, select **Insert** → **Paragraph** → **Heading 1**.
   d. Click the cursor on the area of the table in which you wish to insert the heading.
   e. You will see a cell marked on the page. Resize the cell as desired.
   f. Enter `Welcome to the ITSO RedBank!` in the H1 field.
5. Insert a Link to the IBM Redbooks Web site.
   a. From the menu bar, select **Insert** → **Paragraph** → **Normal**.
   b. Click the cursor on the area of the table in which you wish to insert the heading.
   c. You will see a cell marked on the page. Resize the cell as desired.

Archived

d. Enter `For more information on the ITSO and IBM Redbooks, please visit our Internet site.`

e. Select **Internet site**, right-click ,and select **Insert Link**.

f. When the Insert Link dialog appears, select **HTTP**, enter `http://www.ibm.com/redbooks` in the URL field, and then click **OK**.

6. Customize the text font face, size, and color.

   a. Select the text.

   b. Select **Format** → **Font** from the menu bar.

   c. When the Insert Font dialog appears, select the desired font and size, click the Color button, select the desired color, and then click **OK**.

7. Save the index.html page.

8. Click the **Preview** tab.

   Figure 11-25 on page 545 displays a preview of the index.html page.

Figure 11-25 Preview of index.html

Archived

11.5.2 Create the rates.html page content (tables)

In this example we demonstrate how to add a visable table containing interest rates using the Page Designer.

1. Expand **Dynamic Web Projects** → **BankBasicWeb** → **WebContent**.
2. Double-click the **rates.html** file to open in Page Designer.
3. Click the **Design** tab.
4. Expand **HTML Tags** in the Palette.
5. Right-click and select **Layout Mode** → **Free Layout**.
6. Drag **Table** from the Palette to the content area.
7. When the Insert Table dialog appears, enter 5 for Rows and 5 for Columns, then click **OK**.
8. Resize the table as desired.
9. Enter the descriptions and rates (as seen in Figure 11-26 on page 547) into the table.

   Note: Additional table rows and columns can be added and deleted with the Table menu option.

10. Save the rates.html page.
11. Click the **Preview** tab.

    Figure 11-26 displays a preview of the rates.html page.

Archived

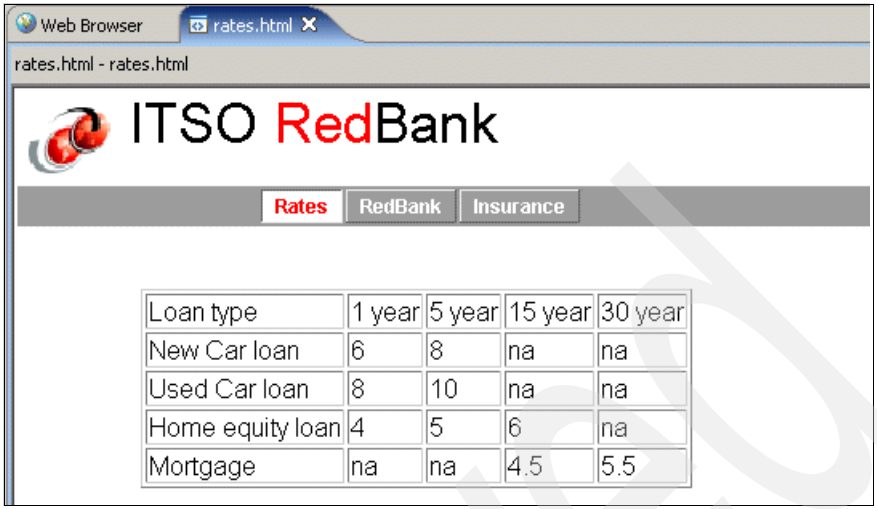


Figure 11-26 Preview the rates.html page

11.5.3 Create the insurance.html page content (list)

In this example, we demonstrate how to add a bulleted list of text and an image to the insurance.html page using the functionality provided by Page Designer.

1. Expand **Dynamic Web Projects** → **BankBasicWeb** → **WebContent**.
2. Double-click the **insurance.html** file to open in Page Designer.
3. Click the **Design** tab.
4. Insert the insurance text.
   a. Click the content area in which you wish to insert the text.
   b. Right-click and select **Layout Mode** → **Free Layout**.
   c. From the menu bar, select **Insert** → **List** → **Bulleted List**.
   d. Click the cursor on the area in which you wish to insert the list.
   e. You will see a cell marked on the page. Resize the cell as desired.
   f. Enter the text for the bulleted list. For example, we added the following:
      - Auto
      - Life
      - Home owners
      - Renters
5. Save the insurance.html page.
6. Click the **Preview** tab to view the page.

Archived

11.5.4 Create the redbank.html page content (forms)

In this example, we demonstrate how to work with *forms* by adding input fields and a button to the redbank.html page. The form will not be fully functional until adding the servlet code in later steps. Form options such as form input fields and submit buttons are available by selecting **Insert** → **Form and Input Fields**.

1. Expand **Dynamic Web Projects** → **BankBasicWeb** → **WebContent**.
2. Double-click the **redbank.html** file to open in Page Designer.
3. Click the **Source** tab.
4. Modify the redbank.html source to add the form, input field, and submit button, as seen in Example 11-2.

Example 11-2 Modified redbank.html snipped for forms

```
<TD valign="top">
   <FORM action="ListAccounts" method ="post">Please enter your customer ID (SSN):<BR>
      <INPUT type="text" name="customerNumber" size="20">
      <BR>
      <BR>
      <INPUT type="submit" name="ListAccounts" value="Submit">
   </FORM>
</TD>
```

Note: At this point, you will see a broken link warning since the ListAccounts servlet has not yet been created.

5. Save the redbank.html file.
6. Click the **Preview** tab to view the redbank.html page.

   Figure 11-27 on page 549 displays the preview of the page.

Archived

Figure 11-27 Preview of the redbank.html page

11.6 Develop the dynamic Web resources

There are many ways to create dynamic Web applications. The most time-consuming method is to build the pages manually, by writing the code line-by-line in a text editor. An easier and more productive way is to use the Application Developer wizards in conjunction with content-specific editors, such as the HTML and CSS editors, which we have already used.

The Web development wizards help you quickly create forms, HTML pages, JavaServer Pages (JSPs), and Java servlets, even if you are not an expert programmer. These files can be used as is, or modified to fit your specific needs.

Application Developer wizards not only support you in creating servlets, JSPs, and JavaBeans, but they also compile the Java code and store the class files in the correct folders for publishing to your application servers. In addition, as the wizards generate project resources, the deployment descriptor file, web.xml, is updated with the appropriate configuration information for the servlets that are

Archived

created. You can test the resulting project resources within the Application Developer using the WebSphere Test Environment, or any other configured server that supports the chosen J2EE specification level.

In the previous section we described how to create each of the static Web pages from scratch. In this section we demonstrate the process of creating and working with servlets and JSPs; however, we rely on the completed samples to be imported as a prerequiste.

11.6.1 Creating model classes

Throughout the remainder of this section, we develop the dynamic behavior of the BankBasic Web application. Before we can create the servlets that manipulate the data model, we must first create the underlying Java classes that represent this model.

Note: In this section we create classes using the visual modelling features of Rational Application Developer. Refer to 7.2, “Develop the Java Bank application” on page 231, for more detailed information about using these features for creating Java applications.

We create the classes listed in Table 11-1.

Table 11-1 Classes to add

Class name	Superclass	Package	Class modifiers
Customer	java.lang.Object	itso.bank.model	public
Account	java.lang.Object	itso.bank.model	public
Transaction	java.lang.Object	itso.bank.model	public, abstract
Credit	Transaction	itso.bank.model	public
Debit	Transaction	itso.bank.model	public
Bank	java.lang.Object	itso.bank.facade	public, abstract
MemoryBank	Bank	itso.bank.facade	public
BankException	java.lang.Exception	itso.bank.exception	public
UnknownCustomerException	BankException	itso.bank.exception	public
UnknownAccountException	BankException	itso.bank.exception	public
InvalidAmountException	BankException	itso.bank.exception	public

Archived

Class name	Superclass	Package	Class modifiers
InsufficientFundsException	BankException	itso.bank.exception	public
ApplicationException	java.lang.Exception	itso.bank.exception	public

Create a class diagram for the model

To create the class diagram that will be used to create the model classes, do the following:

1. Expand and select **Dynamic Web Projects** → **BankBasicWeb**.
2. Select **File** → **New** → **Other**.
3. In the Select a wizard window, expand and select **Modeling** → **Class Diagram** and click **Next**.
4. The Create a Class Diagram wizard appears. Enter the following and click **Finish**:
   - Enter or select the parent folder: `BankBasicWeb/diagrams`
   - File name: `model`

   Rational Application Developer will create and open the editor for the new class diagram.

Create packages

To create the packages listed in Table 11-1 on page 550, do the following:

1. Open the class diagram that was created in the previous section.
2. In the Palette view, select **Java** → **Package** and click anywhere on the free form surface.
3. The New Java Package wizard appears. Enter `itso.bank.model` in the Name field and click **Finish**.
4. Repeat the previous steps to add packages named itso.bank.facade and itso.bank.exception. After this, your class diagram should similar to Figure 11-28.

Archived

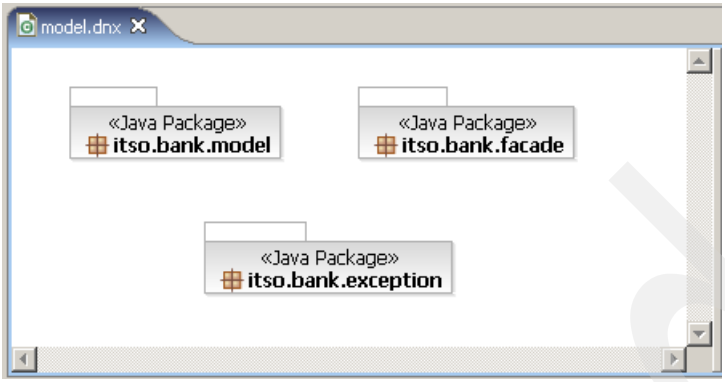


Figure 11-28 Class diagram with packages

Create classes

Do the following to create the classes listed in Table 11-1 on page 550:

1. In the Palette view, select **Java** → **Class** and click anywhere on the free form surface.
2. The New Java Class wizard appears. Enter `itso.bank.model` in the Package field, `Customer` in the Name field, and `java.lang.Object` in the Superclass field. Click **Add**.
3. When the Implemented Interfaces Selection dialog appears, enter `Serializable` in the Choose interfaces field. Select **Serializable** in the Matching types listbox and **java.io** in the Qualifier listbox and click **OK**.
4. When you return to New Java Class wizard it should look similar to in Figure 11-29. Click **Finish**.

Archived

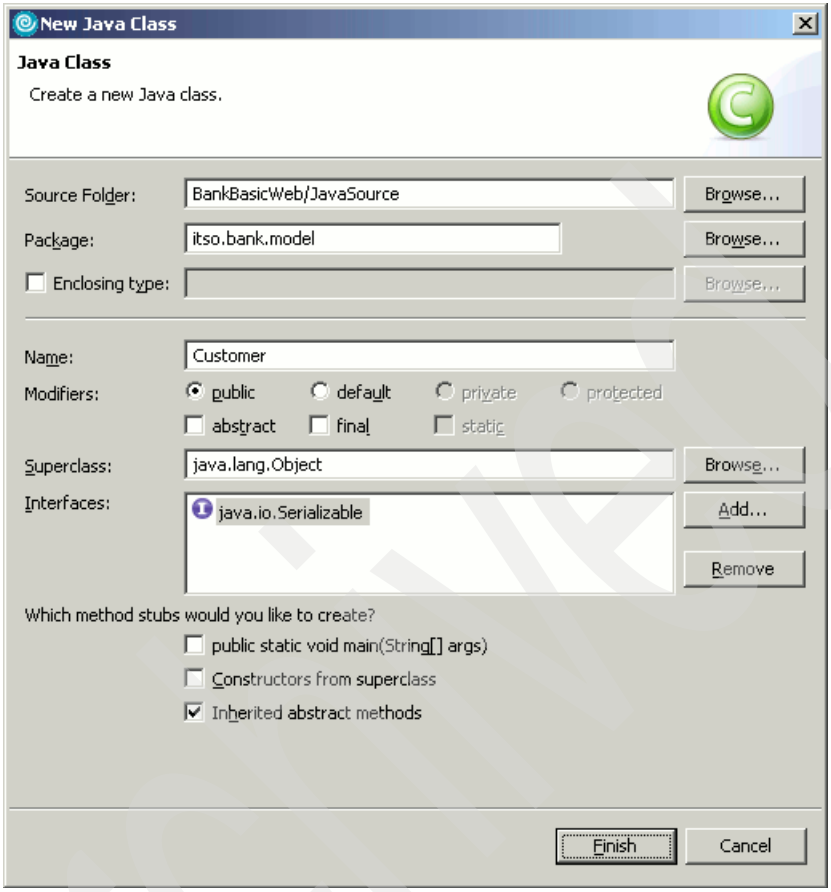


Figure 11-29 Add class Customer

5. Repeat the previous steps to add the classes shown in Table 11-1 on page 550. Remember to add the Serializable interface for all classes.

   When done, your class diagram should similar to Figure 11-30 on page 554.

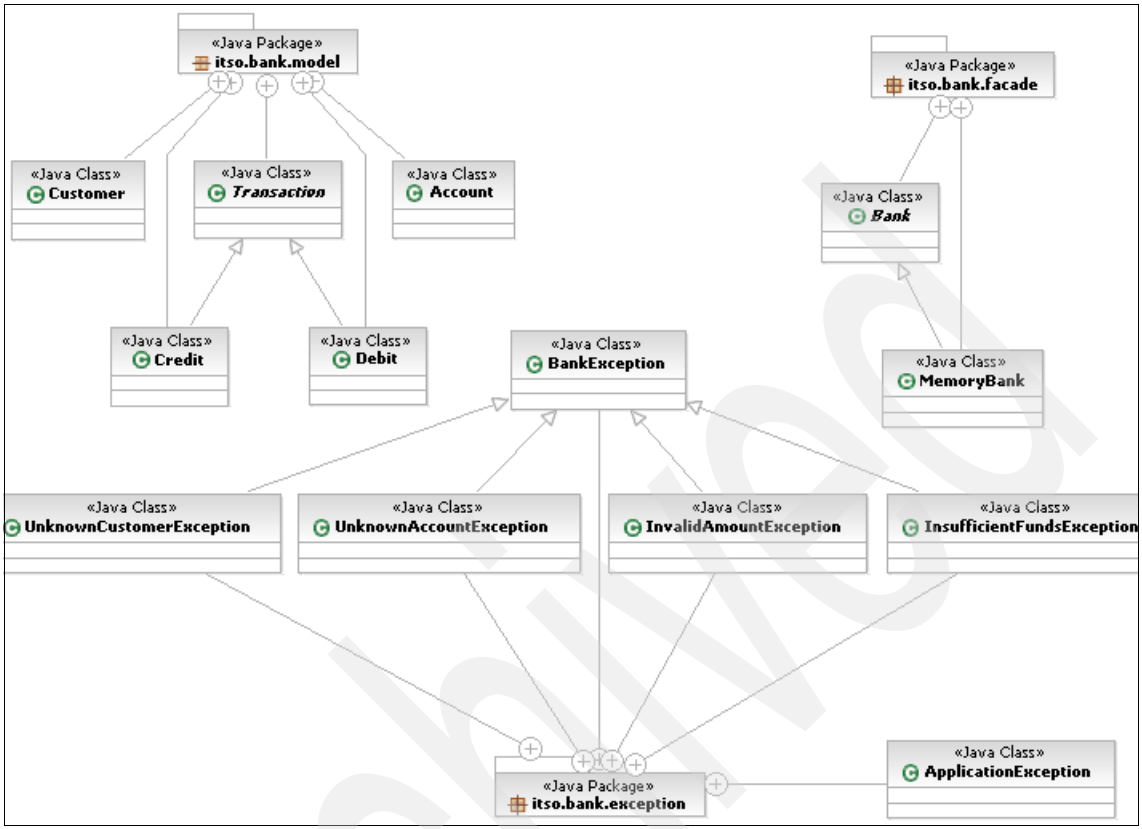


Figure 11-30 Class diagram with packages before adding attributes and methods

Create fields

We will create the fields listed in Table 11-2.

Table 11-2 Fields to add

Class	Field name	Field type	Visibility and modifiers	Initial value
Customer	ssn	String	private	null
	title	String	private	null
	firstName	String	private	null
	lastName	String	private	null

Archive

Class	Field name	Field type	Visibility and modifiers	Initial value
Account	accountNumber	String	private	null
	balance	int	private	0
Transaction	accountNumber	String	private	null
	amount	int	private	0
	timestamp	java.util.Date	private	null
Bank	singleton	Bank	private static	null
MemoryBank	customers	java.util.Map	private	null
	accounts	java.util.Map	private	null
	customerAccounts	java.util.Map	private	null
	transactions	java.util.Map	private	null
InvalidAmountException	amount	String	private	null
UnknownCustomerException	customerNumber	String	private	null
UnknownAccountException	accountNumber	String	private	null
InsufficientFundsException	debitAccountNumber	String	private	null
	amount	int	private	0
	message	String	private	null

To create a field, do the following:

1. Move the cursor anywhere over the class Customer.
2. When the Action bar appears (as shown in Figure 11-31), click the Add new Java field icon (▫) to add attributes to the class.

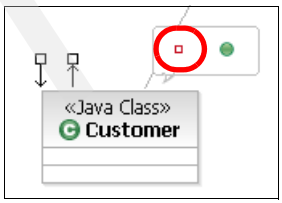


Figure 11-31 The Action bar is used to add attributes and methods to a Java class

3. Then the Create Java Field wizard appears. Do the folowing (as shown in Figure 11-32 on page 556), and click **Finish**:
   - Enter `firstName` in the Name field.
   - Enter `String` in the Type field.
   - Enter `null` in the Initial value field.
   - Select **private**.
   - Ensure that none of the modifiers are checked.

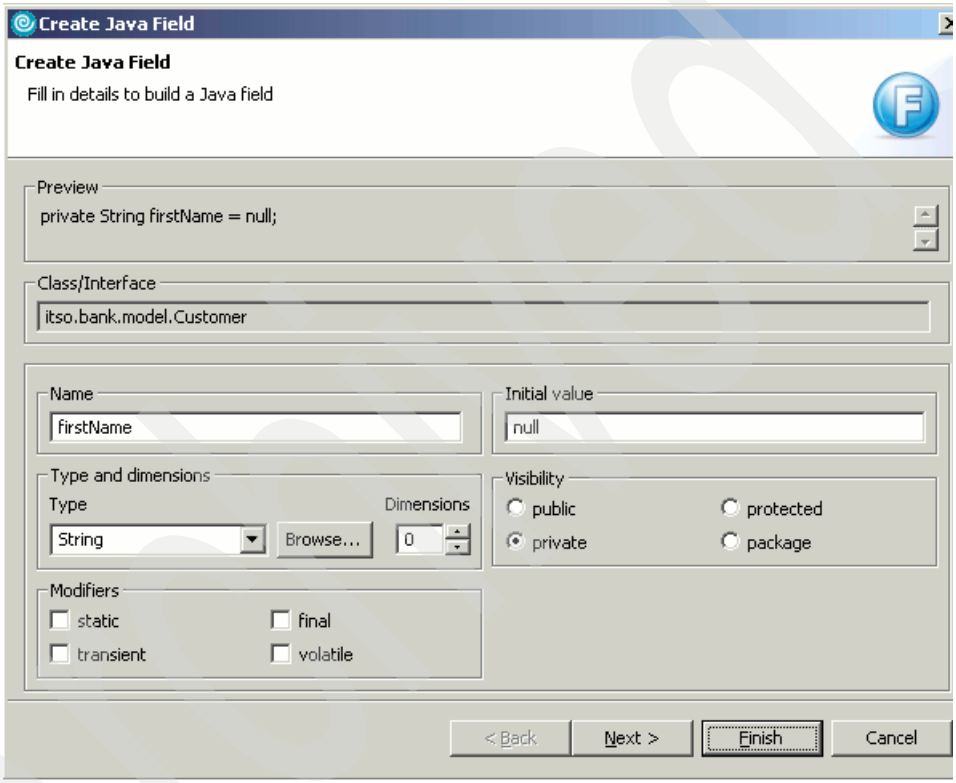


Figure 11-32 Adding the field firstName to the Customer class

4. Repeat the process to add the fields in Table 11-2 on page 554. After this, your class diagram should appear similar to Figure 11-33 on page 557.

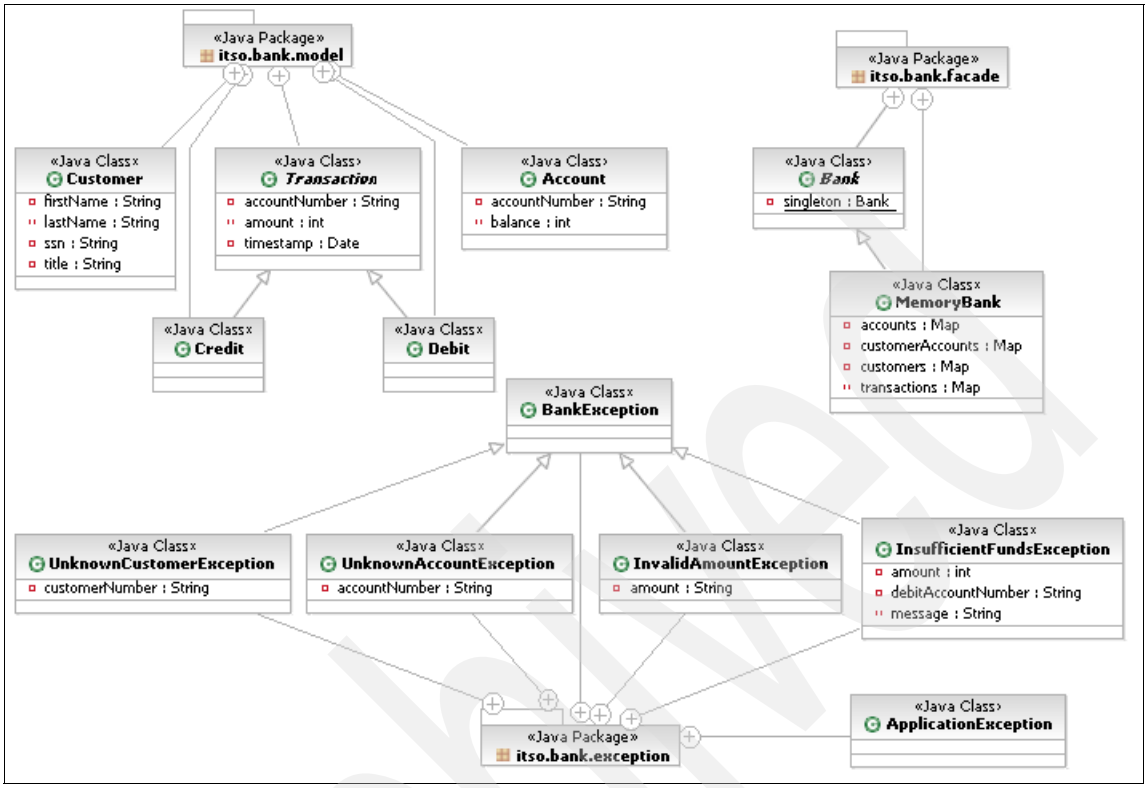


Figure 11-33 Class diagram with fields

Adding accessors

In this section, we add accessors to the model classes. The accessors we create are summarized in Table 11-3 on page 558.

Archive

Table 11-3 Accessors for model classes

Class	Accessor
Customer	getSsn
	setSsn
	getTitle
	setTitle
	getFirstName
	setFirstName
	getLastName
	setLastName
Account	getAccountNumber
	setAccountNumber
	getBalance
	setBalance
Transaction	getAmount
	setAmount
	getAccountNumber
	setAccountNumber
	getTimestamp
	setTimestamp
InvalidAmountException	getAmount
UnknownCustomerException	getCustomerNumber
UnknownAccountException	getAccountNumber
InsufficientFundsException	getDebitAccountNumber
	getAmount
	getMessage

Archived

To add the accessors, do the following:

1. In the Project Explorer, expand **Dynamic Web Projects** → **BankBasicWeb** → **Java Resources** → **JavaSource** → **itso.bank.exception**.
2. Right-click **InvalidAmountException.java** and select **Source** → **Generate Getters and Setters**.
3. When the Generate Getters and Setters window appears, check **getAmount** and click **OK**.
4. Repeat the above steps to generate the remaining accessors, shown in Table 11-3 on page 558.

After adding all accessors, the class diagram will look similar to Figure 11-34 on page 560.

Archived

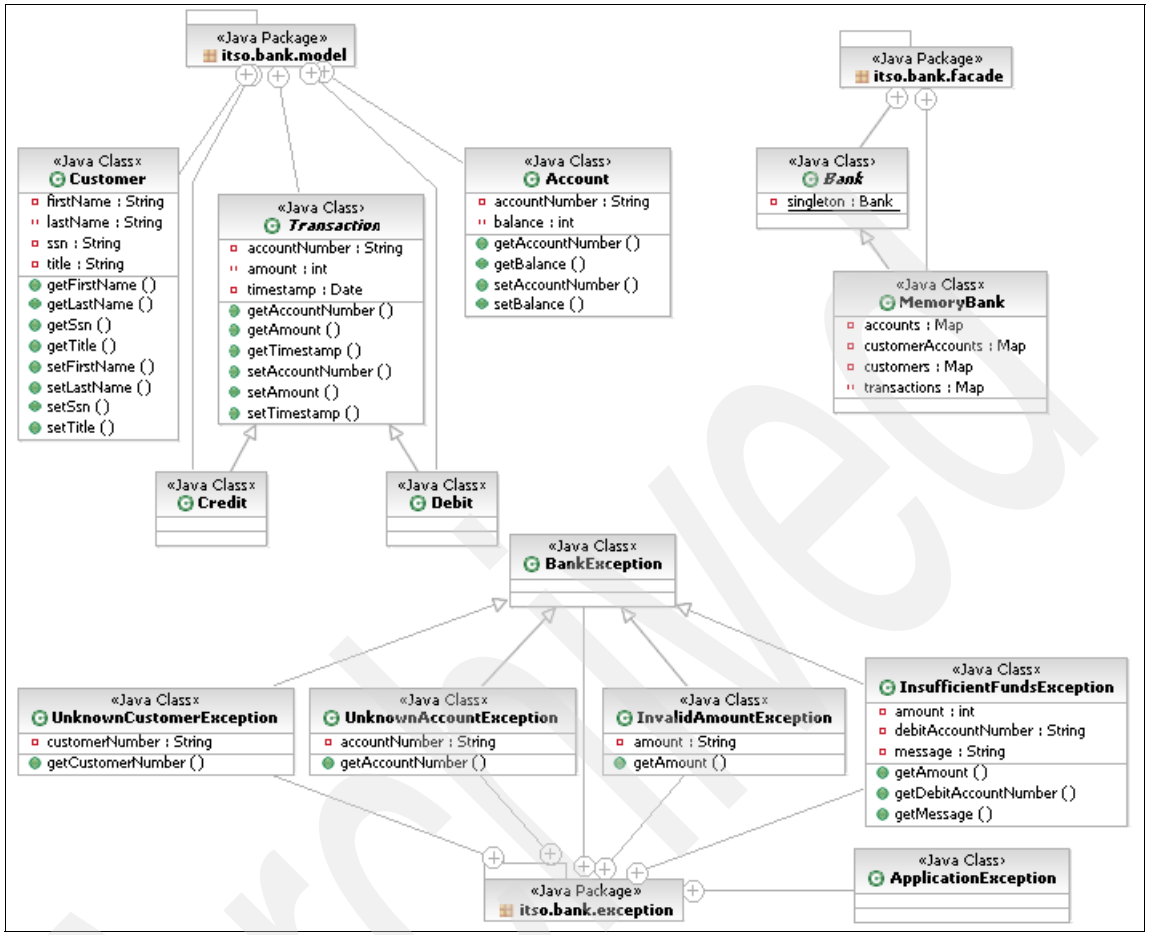


Figure 11-34 Class diagram with fields and accessors

Implement the abstract Bank facade

Now that the framework for the model classes is in place, we can implement the logic and interface of the abstract Bank, defining how the ITSO bank is used.

Note: This code is shipped in the finished BankBasicWeb.zip Project Interchange file, as well as a standalone file. The standalone file can be found as c:\6449code\web\source\Bank.java.

Refer to Appendix B, “Additional material” on page 1395, for more information about the additional material.

Archive

We will add the following methods to the Bank facade class, as shown in Example 11-4 on page 566:

- public static Bank getBank()

  This method is used by a client to get a reference to the actual Bank implementation class.

  For the in-memory bank used in the sample in this chapter, the method creates and returns an instance of the MemoryBank class.

- public abstract Customer getCustomer(String customerNumber)

  This method is used to retrieve a Customer object from a customer number (SSN). If not found, the method will throw an UnknownCustomerException exception.

- public abstract Account[] getAccounts(String customerNumber)

  This method is used to retrieve an array of Account objects, representing the accounts for a customer, as specified by the customer number (SSN). If the customer cannot be found, the method will throw an UnknownCustomerException exception. If that customer has no account, an empty array is returned.

- public abstract Account getAccount(String accountNumber)

  This method is used to retrieve an Account object from an account number. If not found, the method will throw an UnknownAccountException exception.

- public abstract Transaction[] getTransactions(String accountId)

  This method is used to retrieve a list of the transactions performed on the given account. If the account cannot be found, the method will throw an UnknownAccountException exception. If no transactions have been performed on the account, an empty array is returned.

- public abstract void updateCustomer(String ssn, String title, String firstName, String lastName)

  This method will update the basic information about a customer. If the customer cannot be found, the method will throw an UnknownCustomerException exception.

- public abstract void deposit(String accountId, int amount)

  This method is used to deposit the given number of cents to the specified account. If the account cannot be found, the method will throw an UnknownAccountException exception.

- public abstract void withdraw(String accountId, int amount)

  This method is used to withdraw the given number of cents from the specified account. If the account cannot be found, the method will throw an

UnknownAccountException exception. If the withdrawal would result in an overdraft, an InvalidTransactionException is thrown.

- public abstract void transfer(String debitAccountNumber, String creditAccountNumber, int amount)

  This method is used to transfer the given number of cents from the specified debitor account to the specified creditor account. If any of the accounts cannot be found, the method will throw an UnknownAccountException exception. If the transfer would result in an overdraft of the debitor account, an InvalidTransactionException is thrown.

Example 11-3 The method definitions for the abstract Bank class

```
/**
 * Create a singleton. The method will ensure that only
 * one Bank object is created.
 *
 * @throws ApplicationException If an application-level non-business related
 *         exception occurred.
 *
 * @see Bank#Bank()
 */
public static Bank getBank() throws ApplicationException {
   if (singleton == null) {
      // no singleton has been created yet - create one
      singleton = new MemoryBank();
   }

   return singleton;
}

/**
 * Retrieve a Customer object for a given customer number (SSN).
 *
 * @param customerNumber The customer number (SSN) to look up.
 * @return A Customer instance, representing that customer.
 * @throws UnknownCustomerException If the specified customer
 *         does not exist.
 * @throws ApplicationException If an application-level non-business related
 *         exception occurred.
 *
 * @see Customer
 */
public abstract Customer getCustomer(String customerNumber)
       throws UnknownCustomerException, ApplicationException;

/**
 * Retrieve an array of the accounts for a given customer
 * number (SSN).
```


Archived

```
     *
     * @param customerNumber The customer number (SSN) to look up.
     * @return An array of Account instances, each representing an
     *         account that the customer owns within the bank.
     * @throws UnknownCustomerException If the specified customer
     *         does not exist.
     * @throws ApplicationException If an application-level non-business related
     *         exception occurred.
     *
     * @see Account
     */
    public abstract Account[] getAccounts(String customerNumber)
          throws UnknownCustomerException, ApplicationException;

    /**
     * Retrieve an Account object for a given account number.
     *
     * @param accountNumber The account number to look up.
     * @return An Account instance, representing the given
     *         account number.
     * @throws UnknownAccountException If the account does not
     *         exist.
     * @throws ApplicationException If an application-level non-business related
     *         exception occurred.
     *
     * @see Account
     */
    public abstract Account getAccount(String accountNumber)
          throws UnknownAccountException, ApplicationException;

    /**
     * Retrieve an array of transaction records for the given account.
     *
     * @param accountId The account number to retrieve the transaction
     *        log for.
     * @return An array of instances of the Transaction class, each
     *         representing a movement on the account. If the account has
     *         had no transaction, an empty array (an array of length zero)
     *         is returned.
     * @throws UnknownAccountException If the specified account does not exist.
     * @throws ApplicationException If an application-level non-business related
     *         exception occurred.
     *
     * @see Transaction
     */
    public abstract Transaction[] getTransactions(String accountId)
          throws UnknownAccountException, ApplicationException;
```

Archived

```
/**
 * Update the customer data for the specified customer. The specified SSN
 * must match an existing customer.
 *
 * @param ssn
 * @param title
 * @param firstName
 * @param lastName
 * @throws UnknownCustomerException If either of the accounts do not exist.
 * @throws ApplicationException If an application-level non-business related
 *         exception occurred.
 */
public abstract void updateCustomer(String ssn, String title,
      String firstName, String lastName)
      throws UnknownCustomerException, ApplicationException;

/**
 * Deposit (credit) the specified amount of cents to
 * the specified account.
 *
 * After a successful transaction, the transaction log for
 * the account will be updated.
 *
 * @param accountId The account number to deposit into.
 * @param amount The amount to deposit, in cents.
 * @throws UnknownAccountException If the account does not exist.
 * @throws ApplicationException If an application-level non-business related
 *         exception occurred.
 */
public abstract void deposit(String accountId, int amount)
      throws UnknownAccountException, ApplicationException;

/**
 * Withdraw (debit) the specified amount of cents from
 * the specified account.
 *
 * After a successful transaction, the transaction log for
 * the account will be updated.
 *
 * @param accountId The account number to withdraw from.
 * @param amount The amount to withdraw, in cents.
 * @throws UnknownAccountException If the account does not exist.
 * @throws InsufficientFundsException If the amount exceeds the
 *         current balance of the account.
 * @throws ApplicationException If an application-level non-business related
 *         exception occurred.
 */
public abstract void withdraw(String accountId, int amount)
      throws UnknownAccountException, InsufficientFundsException, ApplicationException;
```


Archived

```
/**
 * Transfer the specified amount of cents from one account to
 * another. The accounts do not need to be owned by the same customer.
 *
 * After a successful transaction, the transaction log for
 * both accounts will be updated.
 *
 * @param debitAccountNumber The account number to withdraw (debit) from.
 * @param creditAccountNumber The account number to deposit (credit) to.
 * @param amount The amount to transfer, in cents.
 * @throws UnknownAccountException If either of the accounts do not exist.
 * @throws InsufficientFundsException If the amount exceeds the
 *         current balance of the debit account.
 * @throws ApplicationException If an application-level non-business related
 *         exception occurred.
 */
public abstract void transfer(String debitAccountNumber, String creditAccountNumber,
      int amount)
      throws UnknownAccountException, InsufficientFundsException, ApplicationException;
```

Implement the MemoryBank facade

While the abstract `Bank` class defines the interface to access an ITSO Bank, the concrete MemoryBank class implements the bank functionality as an in-memory database. In Chapter 15, “Develop Web applications using EJBs” on page 827, this will be augmented with a bank facade that uses EJBs to store the information in a relational database.

Note: This code is shipped in the finished BankBasicWeb.zip Project Interchange file, as well as a standalone file. The standalone file can be found as c:\6449code\web\source\MemoryBank.java.

We will add implementation of the abstract methods from the parent `Bank` class, as well as the following methods to the `MemoryBank` facade class. The finished methods is shown in Example 11-4 on page 566.

- protected MemoryBank()

  This constructor, which is protected to ensure that only classes in the itso.bank.facade package can create new instances, initializes the memory structures for the ITSO Bank example.

- private Transaction addDebitTransaction(String accountId, int amount)

  This method is used by the transaction methods to add information to the transaction history about a debit transaction. The method will instantiate an

Archived

object of the Debit class, populate it and call addTransactionToLog to add the transaction to the account's transaction log.

– private Transaction addCreditTransaction(String accountId, int amount)

Like addDebitTransaction, but creates an instance of the Credit Transaction class.

- private void addTransactionToLog(Transaction transaction)

  This method adds the specified transaction to the account specified in the accountNumber field of the transaction object.

- private void addCustomer(String ssn, String title, String firstName, String lastName)

  This method is used by the constructor to create the test data used in the application.

- private void addAccount(String ssn, String accountNumber, int balance)

  This method is used by the constructor to create the test data used in the application.

Note: After implementing the methods shown in Example 11-4 you will have a number of errors. These will be resolved when the transaction and exception classes have been completed in the following sections.

Example 11-4 Code for the MemoryBank facade class

```
/**
 * Create the in-memory bank, initialized with the default data.
 */
protected MemoryBank() {
   // create the maps
   customers = new HashMap();
   accounts = new HashMap();
   customerAccounts = new HashMap();
   transactions = new HashMap();

   // seed with data
   addCustomer("111-11-1111", "MR", "John", "Ganci");
   addCustomer("222-22-2222", "MR", "Richard", "Raszka");
   addCustomer("333-33-3333", "MR", "Fabio", "Ferraz");
   addCustomer("444-44-4444", "MR", "Neil", "Weightman");
   addCustomer("555-55-5555", "MR", "Kiriya", "Keat");
   addCustomer("666-66-6666", "MR", "Hari", "Kanangi");
   addCustomer("777-77-7777", "MR", "Juha", "Nevalainen");
   addCustomer("999-99-9999", "Sir", "Nicolai", "Nielsen");

   addAccount("111-11-1111", "001-999000777", 123456789);
```

```
    addAccount("111-11-1111", "001-999000888", 654321);
    addAccount("111-11-1111", "001-999000999", 9876);
    addAccount("222-22-2222", "002-999000777", 6548423);
    addAccount("222-22-2222", "002-999000888", 8796);
    addAccount("222-22-2222", "002-999000999", 65465);
    addAccount("333-33-3333", "003-999000777", 987652);
    addAccount("333-33-3333", "003-999000888", 56879);
    addAccount("333-33-3333", "003-999000999", 2156);
    addAccount("444-44-4444", "004-999000777", 98765);
    addAccount("444-44-4444", "004-999000888", 145645646);
    addAccount("444-44-4444", "004-999000999", 2315646);
    addAccount("555-55-5555", "005-999000777", 6589);
    addAccount("555-55-5555", "005-999000888", 7221341);
    addAccount("555-55-5555", "005-999000999", 89755);
    addAccount("666-66-6666", "006-999000777", 50000);
    addAccount("666-66-6666", "006-999000888", 10000);
    addAccount("666-66-6666", "006-999000999", 1000000);
    addAccount("777-77-7777", "007-999000777", 250000000);
    addAccount("777-77-7777", "007-999000888", 100000000);
    addAccount("777-77-7777", "007-999000999", 123);
    addAccount("999-99-9999", "009-999000999", 65860042);
}

/**
 * @see Bank#getCustomer(String)
 */
public Customer getCustomer(String customerNumber)
       throws UnknownCustomerException {

    Customer customer = (Customer)customers.get(customerNumber);

    if (customer == null) {
      // not found
      throw new UnknownCustomerException(customerNumber);
    }

    return customer;
}

/**
 * @see itso.bank.facade.Bank#updateCustomer(java.lang.String, java.lang.String,
java.lang.String, java.lang.String)
 */
public void updateCustomer(String ssn, String title, String firstName, String lastName) throws
UnknownCustomerException, ApplicationException {

    Customer customer = getCustomer(ssn);

    customer.setTitle(title);
```

Archived

```
   customer.setFirstName(firstName);
   customer.setLastName(lastName);
}

/**
 * @see Bank#getAccounts(String)
 */
public Account[] getAccounts(String customerNumber)
       throws UnknownCustomerException {

   if (!customers.containsKey(customerNumber)) {
      // no such customer
      throw new UnknownCustomerException(customerNumber);
   }

   Collection accounts = (Collection)customerAccounts.get(customerNumber);

   if (accounts == null) {
      // no accounts - return empty array
      return new Account[0];
   }
   else {
      // copy to array of Account objects and cast
      return (Account[])accounts.toArray(new Account[0]);
   }
}

/**
 * @see Bank#getAccount(String)
 */
public Account getAccount(String accountNumber)
       throws UnknownAccountException {

   Account account = (Account)accounts.get(accountNumber);

   if (account == null) {
      // not found
      throw new UnknownAccountException(accountNumber);
   }

   return account;
}

/**
 * @see Bank#getTransactions(String)
 */
public Transaction[] getTransactions(String accountId)
       throws UnknownAccountException {
```

Archived

```
   if (accounts.containsKey(accountId)) {

      Collection accountTransactions = (Collection)transactions.get(accountId);

      if (accountTransactions == null) {
         // no transactions - return empty array
         return new Transaction[0];
      }
      else {
         // copy to array of Transaction objects and cast
         return (Transaction[])accountTransactions.toArray(new Transaction[0]);
      }
   }
   else {
      throw new UnknownAccountException(accountId);
   }
}

/**
 * @see Bank#deposit(String, int)
 */
public void deposit(String accountId, int amount)
      throws UnknownAccountException, ApplicationException {

   Account account = getAccount(accountId);

   Transaction transaction = addCreditTransaction(accountId, amount);

   account.setBalance(account.getBalance()+transaction.getSignedAmount());
}

/**
 * @see Bank#withdraw(String, int)
 */
public void withdraw(String accountId, int amount)
      throws UnknownAccountException, InsufficientFundsException, ApplicationException {
   Account account = getAccount(accountId);

   if (account.getBalance() > amount) {

      Transaction transaction = addDebitTransaction(accountId, amount);

      account.setBalance(account.getBalance()+transaction.getSignedAmount());
   }
   else {
      // would result in overdraft
      throw new InsufficientFundsException(accountId, amount);
   }
}
```

Archived

```
/**
 * @see Bank#transfer(String, String, int)
 */
public void transfer(String debitAccountNumber, String creditAccountNumber, int amount)
       throws UnknownAccountException, InsufficientFundsException, ApplicationException {
   Account debitAccount = getAccount(debitAccountNumber);
   Account creditAccount = getAccount(creditAccountNumber);

   if (debitAccount.getBalance() > amount) {

      Transaction credit = addCreditTransaction(creditAccountNumber, amount);
      Transaction debit = addDebitTransaction(debitAccountNumber, amount);

      debitAccount.setBalance(debitAccount.getBalance()+credit.getSignedAmount());
      creditAccount.setBalance(creditAccount.getBalance()+debit.getSignedAmount());
   }
   else {
      // would result in overdraft
      throw new InsufficientFundsException(debitAccountNumber, amount);
   }
}

/**
 * Add a Debit transaction for the specified account's transaction log.
 *
 * @param accountId The account to add a record for.
 * @param amount The amount debited from the account.
 * @return The added transaction.
 * @throws UnknownAccountException If the specified account could not be found.
 */
protected Transaction addDebitTransaction(String accountId, int amount) throws
UnknownAccountException {
   Transaction transaction = new Debit();
   transaction.setAccountNumber(accountId);
   transaction.setAmount(amount);
   transaction.setTimestamp(new Date());

   addTransactionToLog(transaction);

   return transaction;
}

/**
 * Adds a Credit transaction for the specified account's transaction log.
 *
 * @param accountId The account to add a record for.
 * @param amount The amount credited to the account.
 * @return The added transaction.
```

Archived

```
 * @throws UnknownAccountException If the specified account could not be found.
 */
protected Transaction addCreditTransaction(String accountId, int amount) throws
UnknownAccountException {
   Transaction transaction = new Credit();
   transaction.setAccountNumber(accountId);
   transaction.setAmount(amount);
   transaction.setTimestamp(new Date());

   addTransactionToLog(transaction);

   return transaction;
}

/**
 * Add the specified transaction to the account's transaction log.
 *
 * @param transaction The transaction to add (the account number is
 *        stored in the Transaction obejct).
 * @throws UnknownAccountException If the specified account could not be found.
 */
private void addTransactionToLog(Transaction transaction) throws UnknownAccountException {
   if (accounts.containsKey(transaction.getAccountNumber())) {

      Collection accountTransactions =
(Collection)transactions.get(transaction.getAccountNumber());

      if (accountTransactions == null) {
         // no transactions create new list
         accountTransactions = new LinkedList();
         transactions.put(transaction.getAccountNumber(), accountTransactions);
      }
      accountTransactions.add(transaction);
   }
   else {
      throw new UnknownAccountException(transaction.getAccountNumber());
   }
}

/**
 * Add a customer record with the given information to the customer database.
 * This method is used by the constructor to seed the in-memory database.
 *
 * @param ssn The customer number (SSN) of the new customer.
 * @param title The salutation for the new customer
 * @param firstName The new customer's first name.
 * @param lastName The new customer's last name.
 *
 * @see Customer
```

Archived

```
 * @see #Bank(Bank)
 */
private void addCustomer(String ssn, String title, String firstName, String lastName) {
   Customer customer = new Customer();
   customer.setSsn(ssn);
   customer.setTitle(title);
   customer.setFirstName(firstName);
   customer.setLastName(lastName);

   customers.put(ssn, customer);
}

/**
 * Add a new account with the given information to the account database.
 * This method is used by the constructor to seed the in-memory database.
 *
 * @param ssn The customer number (SSN) of the customer owning the account.
 * @param accountNumber The number of the new account.
 * @param balance The initial balance, in cents.
 *
 * @see Account
 * @see #Bank(Bank)
 */
private void addAccount(String ssn, String accountNumber, int balance) {
   Account account = new Account();
   account.setAccountNumber(accountNumber);
   account.setBalance(balance);

   Collection customerAccountsColl = (Collection)customerAccounts.get(ssn);

   if (customerAccountsColl == null) {
      customerAccountsColl = new LinkedList();
      customerAccounts.put(ssn, customerAccountsColl);
   }

   customerAccountsColl.add(account);
   accounts.put(accountNumber, account);
}
```

Implement the exception classes

In order to be able to use the exception classes, we need to create constructors that initialize the internal variables in a way that allows for useful exception messages.

Note: The code for the exception classes can be copied from the file c:\6449code\web\source\exception.jpage.

Archived

Example 11-5 shows the implementation of the constructors for the base exception class BankException. This class functions as a base class for the business-specific exceptions in the Bank application.

Example 11-5 Constructors for the class BankException

```
/**
 * Create a new BankException instance without any details.
 */
public BankException() {
   super();
}
/**
 * Create a new BankException instance.
 *
 * @param message A message, describing the exception.
 */
public BankException(String message) {
   super(message);
}
```

Example 11-6 shows the implementation of the constructors for the base exception class ApplicationException. This exception signals that a non-business related application exception, such as a database problem, has occurred.

Example 11-6 Constructors for the class ApplicationException

```
/**
 * Create a new ApplicationException instance without any details.
 */
public ApplicationException() {
   super();
}
/**
 * Create a new ApplicationException instance.
 *
 * @param message A message, describing the exception.
 */
public ApplicationException(String message) {
   super(message);
}
/**
 * Create a new ApplicationException instance.
 *
 * @param message A message, describing the exception.
 * @param cause The root cause of the exception.
 */
public ApplicationException(String message, Throwable cause) {
   super(message, cause);
```

Archived

```
}
```

Example 11-7 shows the implementation of the constructor for the exception class UnknownCustomerException. The constructor initializes the exception message by passing the message to the Exception superclass.

Example 11-7 Constructor for the class UnknownCustomerException

```
/**
 * Create a new UnknownCustomerException.
 *
 * @param customerNumber The customer number (SSN) that was specified.
 */
public UnknownCustomerException(String customerNumber) {
   super("Unknown customer: "+customerNumber);
   this.customerNumber = customerNumber;
}
```

Example 11-8 shows the implementation of the constructor for the exception class UnknownAccountException. The constructor initializes the exception message by passing the message to the Exception superclass.

Example 11-8 Constructor for the class UnknownAccountException

```
/**
 * Create a new UnknownAccountException.
 * @param accountNumber The account number attempted to be used.
 */
public UnknownAccountException(String accountNumber) {
   super("Unknown account: "+accountNumber);
   this.accountNumber = accountNumber;
}
```

Example 11-9 shows the implementation of the constructor for the exception class InvalidAmountException. The constructor initializes the exception message by passing the message to the Exception superclass.

Example 11-9 Constructor for the InvalidAmountException class

```
/**
 * Create an InvalidAmountException.
 * @param strAmount The string that was attempted to be
 *        used as a monetary amount.
 */
public InvalidAmountException(String strAmount) {
   super("Invalid amount: "+strAmount);
   this.amount = strAmount;
```

Archived

```
}
```

Example 11-10 shows the implementation of the constructor for the exception class InsufficientFundsException. Since the exception message is rather large, the exception message is implemented as a field and a getMessage method that overrides the method from the Exception superclass.

Example 11-10 Constructor for the InsufficientFundsException class

```
/**
 * Create a new InsufficientFundsException.
 * @param accountNumber The account number for the account being debited.
 * @param amount The amount attempted to transfer.
 *
 */
public InsufficientFundsException(
       String debitAccountNumber,
       int amount) {
   this.debitAccountNumber = debitAccountNumber;
   this.amount = amount;

   // initialize the exception message string
   StringBuffer buf = new StringBuffer(256);
   buf.append("Insufficient funds for transfer. ");
   buf.append(" Debitor: ");
   buf.append(debitAccountNumber);
   buf.append(" Amount: ");
   buf.append(amount);
   message = buf.toString();
}
```

Implement the Transaction classes

Two abstract method must be added to the Transaction class and implemented in the two concrete classes, Credit and Debit:

- public abstract String getTransactionType()

  This method returns a descriptive name for the concrete transaction.

- public abstract int getSignedAmount()

  This method returns the transaction amount with a sign that signals the direction of the transaction. This value can be used to add to an account balance to simulate the transaction.

Note: The code for the Transaction classes can be copied from the file c:\6449code\web\source\transactions.jpage.

Archived

The abstract method declarations for the Transaction class are shown in Example 11-11.

Example 11-11 Method declarations for the Transaction class

```
/**
 * @return A textual description of the transaction type.
 */
public abstract String getTransactionType();

/**
 * @return An amount that reflect the "direction" of the transaction.
 */
public abstract int getSignedAmount();
```

The concrete method declarations for the `Credit` class are shown in Example 11-12.

Example 11-12 Method implementation for the Credit class

```
public int getSignedAmount() {
   return getAmount();
}
public String getTransactionType() {
   return "Credit";
}
```

The concrete method declarations for the `Debit` class are shown in Example 11-13.

Example 11-13 Method implementation for the Debit class

```
public int getSignedAmount() {
   return -getAmount();
}
public String getTransactionType() {
   return "Debit";
   }
```

11.6.2 Working with servlets

Servlets are flexible and scalable server-side Java components based on the Sun Microsystems Java Servlet API, as defined in the Sun Microsystems Java Servlet Specification. For J2EE 1.4, the supported API is Servlet 2.4.

Archived

Servlets generate dynamic content by responding to Web client requests. When an HTTP request is received by the application server, the server determines, based on the request URI, which servlet is responsible for answering that request and forwards the request to that servlet. The servlet then performs its logic and builds the response HTML that is returned back to the Web client, or forwards the control to a JSP.

Application Developer provides the necessary features to make servlets easy to develop and integrate into your Web application. Without leaving your Workbench, you can develop, debug, and deploy them. You can set breakpoints within servlets, and step through the code. You can make changes that are dynamically folded into the running servlet on a running server, without having to restart the server each time.

In this section, we show you how to create and implement the servlets ListAccounts, UpdateCustomer, AccountDetails, and Logout servlets. Then we show you how to use the command, or action, pattern to implement the `PerformTransaction` servlet.

Adding the ListAccounts servlet to the Web Project

Application Developer provides a servlet wizard to assist you in adding servlets to your Web applications.

1. Select **File** → **New** → **Other**.
2. The New wizard will open. Select **Web** → **Servlet** and click **Next**.

   Tip: The Create Servlet wizard can also be accessed by right-clicking the deployment descriptor and selecting **New** → **Servlet**.

3. The first page of the Create Servlet wizard appears. Enter the name of the new servlet and click **Next**. In our case, we entered `ListAccounts`, as shown in Figure 11-35 on page 578.

   The wizard will automatically generate the URL mapping /ListAccounts for us. If we were intereseted in a different, or additional URL mappings for this servlet, we could add these here.

   The wizard will also allow us to add servlet initialization parameters. These are used to parameterize a servlet. Servlet initialization parameters can be changed at runtime from within the WebSphere Application Server Administration Console.

   In our sample, we do not require additional URL mappings or initialization parameters.

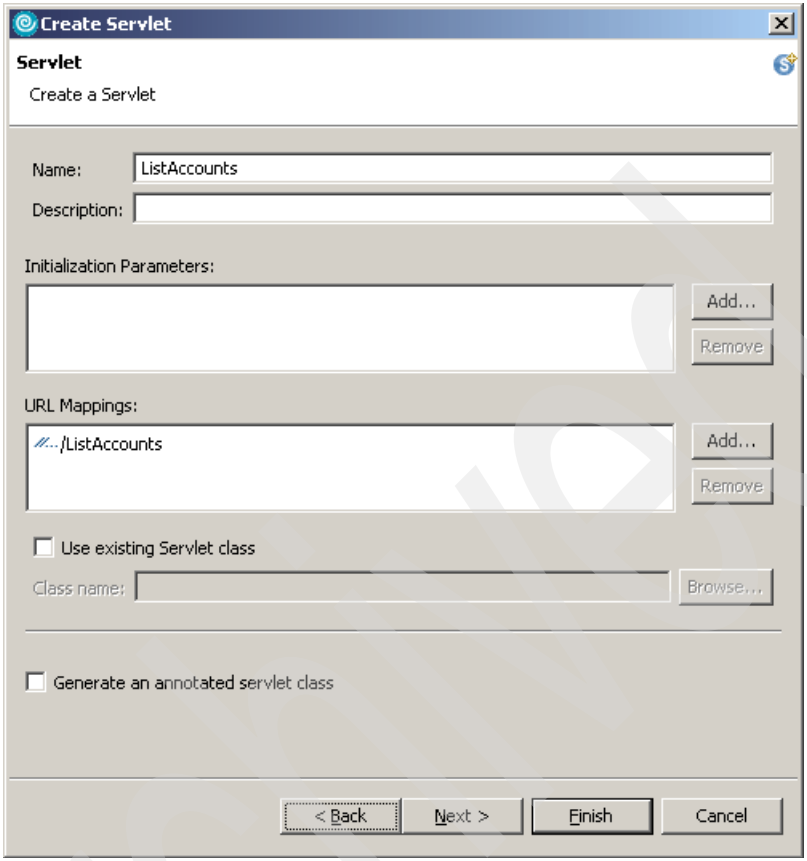


Figure 11-35 New servlet wizard (page 1)

4. The second page of the Create Servlet wizard appears. On this page we can specify specifics about the Java class that will be created to implement the new servlet. We accepted the defaults, except for the package name. We entered the name `itso.bank.servlet` (as shown in Figure 11-36 on page 579), and clicked **Next**.

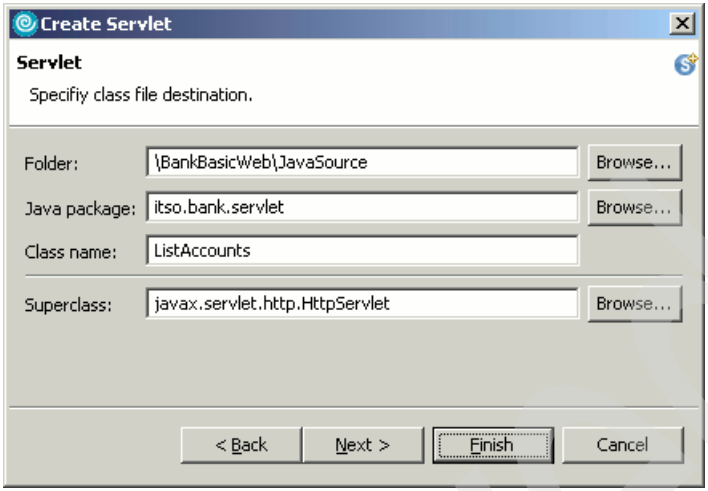


Figure 11-36 New servlet wizard (page 2)

5. The third and last page of the Create Servlet wizard appears.

   This page lets you select the appropriate method stubs to be created in the servlet code. These are the servlet's life-cycle methods, along with its service methods specific to the HTTP protocol (the methods that start with `do`).

   For our example, we need both doGet and doPost selected. Both are read methods. Usually, HTTP gets are used with direct links, when no information needs to be sent to the server. HTTP posts are typically used when information in a form has to be sent to the server.

   Only one instance of a servlet is created in the application server. If you want to perform any initialization when the servlet instance is created, select the `init` method to be created. This method is invoked after the servlet instance has been created and you can perform the initialization tasks.

   Uncheck Constructors from superclass and ensure that doPost and doGet are both checked (as shown in Figure 11-37), and click **Finish**.

Archive

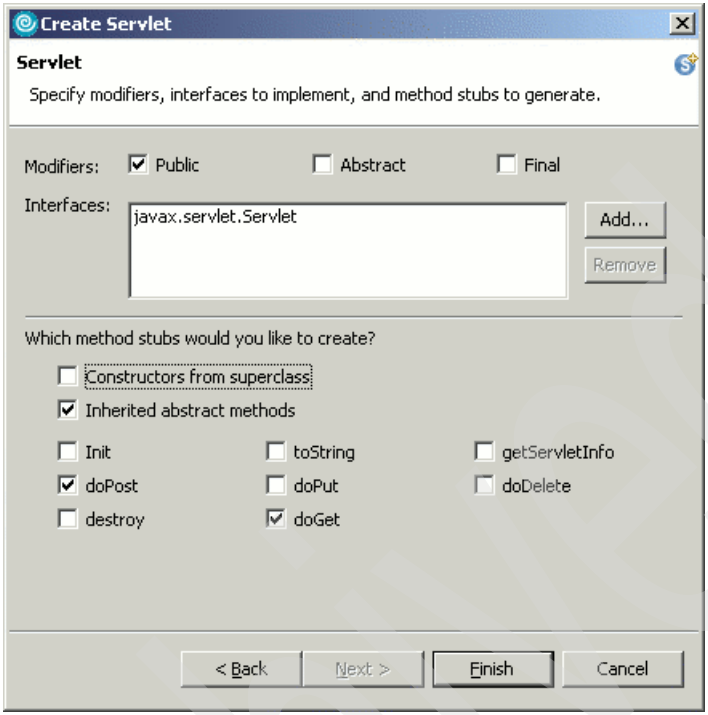


Figure 11-37 New servlet wizard (page 3)

The servlet is then generated and added to the project. The source code can be found in the JavaSource folder of the project, while the configuration for the servlet is found in Servlets tab of the Web Deployment Descriptor.

Implementing the ListAccounts servlet

We now have a skeleton servlet that does not perform any actions when it is invoked. We now need to add code to the servlet in order to implement the required behavior.

Note: The code for the following servlets can be copied from the file c:\6449code\web\source\servlets.jpage.

1. Expand **Dynamic Web Projects** → **BankBasicWeb** → **Java Resources** → **JavaSource** → **itso.bank.servlet**.
2. Double-click **ListAccounts.java**.
3. Add the import statements from Example 11-14 to the existing import statements.

Example 11-14 Additional import statements for ListAccounts.java

```
import javax.servlet.ServletContext;
import javax.servlet.http.HttpSession;
import javax.servlet.RequestDispatcher;
import itso.bank.facade.Bank;
import itso.bank.model.Account;
import itso.bank.model.Customer;
```

4. Change the doGet and doPost methods to look like Example 11-15.

   As Example 11-15 shows, both service methods call a third method, performTask. This means that HTTP GET and POST methods will be handled identically.

Example 11-15 Modified doGet and doPost service methods for ListAccounts.java

```
/**
 * HTTP GET service method. Calls performTask to service requests.
 *
 * @see performTask(HttpServletRequest req, HttpServletResponse resp)
 * @see javax.servlet.http.HttpServlet#doGet(HttpServletRequest req,
 *      HttpServletResponse resp)
 */
protected void doGet(HttpServletRequest req, HttpServletResponse resp)
      throws ServletException, IOException
{
   performTask(req, resp);
}

/**
 * HTTP POST service method. Calls performTask to service requests.
 *
 * @see performTask(HttpServletRequest req, HttpServletResponse resp)
 * @see javax.servlet.http.HttpServlet#doPost(HttpServletRequest req,
 *      HttpServletResponse resp)
 */
protected void doPost(HttpServletRequest req, HttpServletResponse resp)
      throws ServletException, IOException
{
   performTask(req, resp);
}
```

5. Implement the performTask method, as shown in Example 11-16.

Example 11-16 The performTask method for the ListAcocunts servlet

```
private void performTask(HttpServletRequest req, HttpServletResponse resp)
      throws ServletException, IOException
{
```

```
        try
        {
            // Get input parameter and keep it on the HTTP session
            String customerNumber = req.getParameter("customerNumber");
            HttpSession session = req.getSession();

            if (customerNumber == null)
                customerNumber = (String) session.getAttribute("customerNumber");
            else
                session.setAttribute("customerNumber", customerNumber);

            // Control logic - Create the new banking facade
            Bank bank = Bank.getBank();

            // Retrieve customer and related accounts
            Customer customer = bank.getCustomer(customerNumber);
            Account[] accounts = bank.getAccounts(customerNumber);

            // Response - Set the request attributes for future rendering
            req.setAttribute("customer", customer);
            req.setAttribute("accounts", accounts);

            // Call the presentation renderer
            ServletContext ctx = getServletContext();
            RequestDispatcher disp = ctx.getRequestDispatcher("listAccounts.jsp");
            disp.forward(req, resp);
        }
        catch (Exception e)
        {
            // set up error information and forward to the error page
            req.setAttribute("message", e.getMessage());
            req.setAttribute("forward", "index.html");
            ServletContext ctx = getServletContext();
            RequestDispatcher disp = ctx.getRequestDispatcher("showException.jsp");
            disp.forward(req, resp);
        }
    }
```

The performTask method is divided into three main sections:

- The first section deals with the HTTP parameters. This servlet expects to either receive a parameter called customerNumber or none at all. If the parameter is passed, we store it in the HTTP session for future use. If it is not passed, we look for it in the HTTP session, because it might have been stored there earlier.
- The second section deals with the control logic. We create a new Bank facade and use it to get the customer object and the array of accounts for that customer.

Archived

- The third section sees that the presentation renderer (listAccounts.jsp) gets the parameters it requires to perform its job (customer and accounts). The parameters are passed in the request context, where they can be picked up my the JSP.
- The final part is the error handler. If an exception is thrown in the previous code, the catch block will ensure that control is passed to the showException.jsp page.

6. Save your changes and close the source editor.

Implementing the UpdateCustomer servlet

The UpdateCustomer servlet is used for updating the customer information. The servlet accepts the following parameters:

- title
- firstName
- lastName

The servlet requires that the SSN of the customer that should be updated is already placed on the session, as done by the ListAccounts servlet.

Follow the procedures described in “Implementing the ListAccounts servlet” on page 580 for preparing the servlet, including the doGet and doPost methods. The performTask implementation for the UpdateCustomer servlet is shown in Example 11-17.

Example 11-17 The performTask method for the UpdateCustomer servlet

```
private void performTask(HttpServletRequest req, HttpServletResponse resp)
      throws ServletException, IOException
{
   try
   {
      // Get input parameters
      String title = req.getParameter("title");
      String firstName = req.getParameter("firstName");
      String lastName = req.getParameter("lastName");

      // retrieve the SSN from the session
      HttpSession session = req.getSession();
      String ssn = (String) session.getAttribute("customerNumber");

      // Control logic - Create the new banking facade
      Bank bank = Bank.getBank();

      // Update customer information
      bank.updateCustomer(ssn, title, firstName, lastName);
```

Archived

```
        // Retrieve customer information
        Customer customer = bank.getCustomer(ssn);

        // Call the presentation renderer
        ServletContext ctx = getServletContext();
        RequestDispatcher disp = ctx.getRequestDispatcher("listAccounts.jsp");
        disp.forward(req, resp);
    }
    catch (Exception e)
    {
        // set up error information and forward to the error page
        req.setAttribute("message", e.getMessage());
        req.setAttribute("forward", "index.html");
        ServletContext ctx = getServletContext();
        RequestDispatcher disp = ctx.getRequestDispatcher("showException.jsp");
        disp.forward(req, resp);
    }
}
```

Implementing the AccountDetails servlet

The AccountDetails servlet is used to retrieve account details and forward to the accountDetails.jsp to show these details. The servlet accepts the parameter accountId, specifying the account for which data should be shown.

Note: A real-life implementation would perform authorization, that is verify that the current user has the required access rights to the requested account.

Follow the procedures described in “Implementing the ListAccounts servlet” on page 580 for preparing the servlet. The performTask implementation for the AccountDetails servlet is shown in Example 11-18.

Example 11-18 The performTask method for the AccountDetails servlet

```
private void performTask(HttpServletRequest req, HttpServletResponse resp)
        throws ServletException, IOException
{
    try
    {
        // parameters
        String accountNumber = req.getParameter("accountId");

        // Control logic - Create the new banking facade
        Bank bank = Bank.getBank();

        // Retrieve customer and related accounts
        Account account = bank.getAccount(accountNumber);
```

```
      // Response - Set the request attributes for future rendering
      req.setAttribute("account", account);

      // Call the presentation renderer
      ServletContext ctx = getServletContext();
      RequestDispatcher disp = ctx.getRequestDispatcher("accountDetails.jsp");
      disp.forward(req, resp);
   }
   catch (Exception e)
   {
      // set up error information and forward to the error page
      req.setAttribute("message", e.getMessage());
      req.setAttribute("forward", "index.html");
      ServletContext ctx = getServletContext();
      RequestDispatcher disp = ctx.getRequestDispatcher("showException.jsp");
      disp.forward(req, resp);
   }
}
```

Implementing the Logout servlet

The Logout servlet is used for logging off the redbank. The servlet requires no parameters, and the only logic performed in the servlet is to remove the SSN from the session, simulating a log off action.

Follow the procedures described in “Implementing the ListAccounts servlet” on page 580 to prepare the servlet. The performTask implementation for the Logout servlet is shown in Example 11-19.

Example 11-19 The performTask method for the Logout servlet

```
private void performTask(HttpServletRequest req, HttpServletResponse resp)
throws ServletException, IOException
{
   try
   {
      // remove the customer number from the session
      HttpSession session = req.getSession();
      session.removeAttribute("customerNumber");
      session.invalidate();

      // Call the presentation renderer
      ServletContext ctx = getServletContext();
      RequestDispatcher disp = ctx.getRequestDispatcher("redbank.html");
      disp.forward(req, resp);
   }
   catch (Exception e)
   {
```

Archived

```
        // set up error information and forward to the error page
        req.setAttribute("message", e.getMessage());
        req.setAttribute("forward", "index.html");
        ServletContext ctx = getServletContext();
        RequestDispatcher disp = ctx.getRequestDispatcher("showException.jsp");
        disp.forward(req, resp);
    }
}
```

Implementing the PerformTransaction servlet framework

In this section, we show you how to use the command design pattern to implement a fron controller, PerformTransaction, that forwards control to one of the four command objects, Deposit, Withdraw, Transfer, and ListTransactions.

Note: The code for the following classes and interfaces can be copied from files in the c:\6449code\web\source directory. The code will be in separate .java files, named for the class or interface.

Implementing the Command interface

The Command interface is an interface that describes the contract to a command. It is implemented by each of the four command classes Deposit, Withdraw, Transfer, and ListTransactions.

Use the New Java Interface wizard to create an interface named Command in the package itso.bank.command. The entire source code for the new interface is shown in Example 11-20.

Example 11-20 The command interface

```
package itso.bank.command;

import java.io.IOException;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

/**
 * The Command interface. This is based on the Task, or Command, design
 * pattern.
 * Used for servlet processing, the class provides only two methods:
 * <tt>execute</tt> and <tt>getForwardView</tt>.
 */
public interface Command
{
    /**
     * Execute the command. The passed-in serlvet request and response
```

```
     * can be used just like any servlet. If an exception is thrown, control
     * is forwarded to the showException.jsp page.
     *
     * @param req The HTTP request
     * @param resp The HTTP response
     * @throws ServletException
     * @throws IOException
     */
    public void execute(HttpServletRequest req, HttpServletResponse resp)
          throws Exception;

    /**
     * @return The requested view to forward to after executing the command,
     *         or <tt>null</tt> if no forwarding should take place.
     */
    public String getForwardView();
}
```

Implementing the DepositCommand command

The DepositCommand class implements the Command interface, created previously, to realize a deposit of a specified amount to a given account.

Use the New Java Class wizard to create a class named DepositCommand in the package itso.bank.command. The class should implement the interface itso.bank.command.Command. The entire source code for the new interface is shown in Example 11-21.

Example 11-21 The completed DepositCommand class

```
package itso.bank.command;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import itso.bank.facade.Bank;
import itso.bank.model.Account;

/**
 * Deposit command. Will perform a deposit of the specified amount to
 * the specified account.
 *
 * Parameters:
 * <dl>
 * <dt>amount</dt><dd>The amount cents to deposit to the account</dd>
 * <dt>accountId</dt><dd>The account number to deposit to</dd>
 * </dl>
 */
public class DepositCommand
```

```
      implements Command
{

   /**
    * Do the actual deposit.
    *
    * @see
itso.bank.command.Command#execute(javax.servlet.http.HttpServletRequest,
javax.servlet.http.HttpServletResponse)
    */
   public void execute(HttpServletRequest req, HttpServletResponse resp)
         throws Exception
   {
      // Parameters
      String accountId = req.getParameter("accountId");
      String strAmount = req.getParameter("amount");
      int iAmount = Integer.parseInt(strAmount);

      // Control logic
      Bank bank = Bank.getBank();
      bank.deposit(accountId, iAmount);

      // Response
      Account account = bank.getAccount(accountId);
      req.setAttribute("account", account);
   }

   /**
    * @see itso.bank.command.Command#getForwardView()
    */
   public String getForwardView() {
      return "accountDetails.jsp";
   }
}
```

Implementing the WithdrawCommand command

The WithdrawCommand class implements the Command interface, created previously, to realize a withdrawal of a specified amount from a given account.

Use the New Java Class wizard to create a class named WithdrawCommand in the package itso.bank.command. The class should implement the interface itso.bank.command.Command. The entire source code for the new interface is shown in Example 11-22.

Example 11-22 The completed WithdrawCommand class

```
package itso.bank.command;
```


Archived

```
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import itso.bank.facade.Bank;
import itso.bank.model.Account;

/**
 * Withdrawal command. Will perform a withdrawal of the specified amount
 * from the specified account.
 *
 * Parameters:
 * <dl>
 * <dt>amount</dt><dd>The amount cents to withdraw</dd>
 * <dt>accountId</dt><dd>The account number to withdraw from</dd>
 * </dl>
 */
public class WithdrawCommand
       implements Command
{

   /**
    * Do the actual withdrawal.
    *
    * @see
itso.bank.command.Command#execute(javax.servlet.http.HttpServletRequest,
javax.servlet.http.HttpServletResponse)
    */
   public void execute(HttpServletRequest req, HttpServletResponse resp)
         throws Exception
   {
      // Parameters
      String accountId = req.getParameter("accountId");
      String strAmount = req.getParameter("amount");
      int iAmount = Integer.parseInt(strAmount);

      // Control logic
      Bank bank = Bank.getBank();
      bank.withdraw(accountId, iAmount);

      // Response
      Account account = bank.getAccount(accountId);
      req.setAttribute("account", account);
   }

   /**
    * @see itso.bank.command.Command#getForwardView()
    */
   public String getForwardView() {
      return "accountDetails.jsp";
```

Archived

```
    }
}
```

Implementing the TransferCommand command

The `TransferCommand` class implements the Command interface, created previously, to realize a transfer from one account to another.

Use the New Java Class wizard to create a class named TransferCommand in the package itso.bank.command. The class should implement the interface itso.bank.command.Command. The entire source code for the new interface is shown in Example 11-23.

Example 11-23 The completed TransferCommand class

```
package itso.bank.command;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import itso.bank.exception.InvalidAmountException;
import itso.bank.facade.Bank;
import itso.bank.model.Account;

/**
 * Transfer command. Will perform a transfer of the specified amount
 * from one account to another.
 *
 * Parameters:
 * <dl>
 * <dt>amount</dt><dd>The amount of cents to transfer</dd>
 * <dt>accountId</dt><dd>The debit account</dd>
 * <dt>targetAccountId</dt><dd>The credit account</dd>
 * </dl>
 */
public class TransferCommand
      implements Command
{
   /**
    * Do the actual transfer.
    *
    * @see
itso.bank.command.Command#execute(javax.servlet.http.HttpServletRequest,
javax.servlet.http.HttpServletResponse)
    */
   public void execute(HttpServletRequest req, HttpServletResponse resp)
         throws Exception
   {
      // Parameters
```


Archived

```
        String debitAccountNumber = req.getParameter("accountId");
        String creditAccountNumber = req.getParameter("targetAccountId");
        String strAmount = req.getParameter("amount");
        int iAmount = 0;

        try
        {
           iAmount = Integer.parseInt(strAmount);
        }
        catch (NumberFormatException x)
        {
           throw new InvalidAmountException(strAmount);
        }

        // Control logic
        Bank bank = Bank.getBank();
        bank.transfer(debitAccountNumber, creditAccountNumber, iAmount);

        // Response
        Account account = bank.getAccount(debitAccountNumber);
        req.setAttribute("account", account);
    }

    /**
     * @see itso.bank.command.Command#getForwardView()
     */
    public String getForwardView() {
        return "accountDetails.jsp";
    }
}
```

Implementing the ListTransactionsCommand command

The ListTransactionsCommand class implements the Command interface, created previously, to retrieve the transaction history for an account and forward to the view listTransactions.jsp to display this information to the user.

Use the New Java Class wizard to create a class named ListTransactionsCommand in the package itso.bank.command. The class should implement the interface itso.bank.command.Command. The entire source code for the new interface is shown in Example 11-24.

Example 11-24 The completed ListTransactionsCommand class

```
package itso.bank.command;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
```


Archived

```
import itso.bank.facade.Bank;
import itso.bank.model.Account;
import itso.bank.model.Transaction;

/**
 * List transactions command. Will retrieve the account history and forward
 * to a JSP that can show this information to the user.
 *
 * Parameters:
 * <dl>
 * <dt>accountId</dt><dd>The account to show the transaction history for</dd>
 * </dl>
 */
public class ListTransactionsCommand
       implements Command
{
   /**
    * Retrieve the transactions for the account.
    *
    * @see
itso.bank.command.Command#execute(javax.servlet.http.HttpServletRequest,
javax.servlet.http.HttpServletResponse)
    */
   public void execute(HttpServletRequest req, HttpServletResponse resp)
         throws Exception
   {
      // Parameters
      String accountId = req.getParameter("accountId");

      // Control logic
      Bank bank = Bank.getBank();

      // Response
      Account account = bank.getAccount(accountId);
      Transaction[] transactions = bank.getTransactions(accountId);

      req.setAttribute("account", account);
      req.setAttribute("transactions", transactions);
   }

   /**
    * @see itso.bank.command.Command#getForwardView()
    */
   public String getForwardView() {
      return "listTransactions.jsp";
   }
}
```

Archived

Implementing the PerformTransaction servlet

Now that all the commands for the PerformTransaction framework have been realized, we can create the PerformTransaction servlet. The servlet uses the value of the transaction request parameter to determine what command to execute.

Use the Create Servlet to create a servlet named PerformTransaction. The servlet class should be placed in the package itso.bank.servlet. The source code for the completed servlet is shown in Example 11-25.

Example 11-25 The PerformTransaction servlet

```
package itso.bank.servlet;

import java.io.IOException;
import java.util.HashMap;
import java.util.Map;

import javax.servlet.RequestDispatcher;
import javax.servlet.Servlet;
import javax.servlet.ServletContext;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import itso.bank.command.*;

public class PerformTransaction
       extends HttpServlet
       implements Servlet
{
   private Map commands;

   public PerformTransaction()
   {
      commands = new HashMap();

      commands.put("deposit", new DepositCommand());
      commands.put("withdraw", new WithdrawCommand());
      commands.put("transfer", new TransferCommand());
      commands.put("list", new ListTransactionsCommand());
   }

   /**
    * HTTP GET service method. Calls performTask to service requests.
    *
    * @see performTask(HttpServletRequest req, HttpServletResponse resp)
```

Archived

```
 * @see javax.servlet.http.HttpServlet#doGet(HttpServletRequest req,
 *      HttpServletResponse resp)
 */
protected void doGet(HttpServletRequest req, HttpServletResponse resp)
      throws ServletException, IOException
{
   performTask(req, resp);
}

/**
 * HTTP POST service method. Calls performTask to service requests.
 *
 * @see performTask(HttpServletRequest req, HttpServletResponse resp)
 * @see javax.servlet.http.HttpServlet#doPost(HttpServletRequest req,
 *      HttpServletResponse resp)
 */
protected void doPost(HttpServletRequest req, HttpServletResponse resp)
      throws ServletException, IOException
{
   performTask(req, resp);
}

private void performTask(HttpServletRequest req, HttpServletResponse resp)
      throws ServletException, IOException
{
   String transaction = null;

   try
   {
      // Get input parameter and keep it on the HTTP session
      transaction = req.getParameter("transaction");

      Command command = (Command)commands.get(transaction);
      if (command != null)
      {
         command.execute(req, resp);

         String forwardView = command.getForwardView();

         if (forwardView != null)
         {
            // Call the presentation renderer
            ServletContext ctx = getServletContext();
            RequestDispatcher disp = ctx.getRequestDispatcher(forwardView);
            disp.forward(req, resp);
         }
      }
      else
      {
```

```
            // set up error information and forward to the error page
            req.setAttribute("message", "Unknown transaction: "+transaction);
            req.setAttribute("forward", "index.html");
            ServletContext ctx = getServletContext();
            RequestDispatcher disp = ctx.getRequestDispatcher("showException.jsp");
            disp.forward(req, resp);
         }
      }
      catch (Exception e)
      {
         // set up error information and forward to the error page
         req.setAttribute("message", e.getMessage());
         req.setAttribute("forward", "index.html");
         ServletContext ctx = getServletContext();
         RequestDispatcher disp = ctx.getRequestDispatcher("showException.jsp");
         disp.forward(req, resp);
      }
   }
}
```

11.6.3 Working with JSPs

Now that we have created a servlet that prepares the required data for, and forwards controls to, the listAccounts.jsp page, we can complete the listAccounts.jsp to display the customer details and account overview.

JSP files are edited in Page Designer, the very same editor you used to edit the HTML page. When working with a JSP page, Page Designer has additional elements (JSP tags) that can be used, such as JavaBean references, Java Standard Template Language (JSTL) tags, and scriptlets containing Java code.

In this section we do the following:

- Implement listAccounts.jsp
- Implement accountDetails.jsp
- Implementing listTransactions.jsp
- Implementing showException.jsp

Implement listAccounts.jsp

You can customize the recently created JSP file by adding your own static content, just like you would to a regular HTML file. Along with that, you can use the standard JSP declarations, scriptlets, expressions, and tags, or any other custom tag that you might have retrieved from the Internet or developed yourself.

To finish the listAccounts.jsp file, do the following:

1. Expand **Dynamic Web Projects** → **BankBasicWeb** → **WebContent**.

Archived

2. Double-click the **listAccounts.jsp** file to open in Page Designer.
3. Click the **Design** tab.
4. Click the content area and select **Center** in the Horizontal alignment drop-down box of the Properties view.
5. Add the customer and accounts variables to the page data meta information. These variables are sent to the JSP from the ListAccounts servlet, as can be seen in Example 11-16 on page 581. We need to make the Page Designer aware of these variables:
   a. Right-click anywhere on the Page Data view and select **New** → **Scripting Variable** → **Request Scope Variable**.
   b. The Add Request Scope Variable window opens. Enter the following information in the window and click **OK**:
      - Variable name: `customer`
      - Type: `itso.bank.model.Customer`

      Tip: You can use the browse-button (marked with an ellipsis) to find the class using the class browser.

   c. Repeat this procedure to add the following request scope variable:
      - Variable name: `accounts`
      - Type: `itso.bank.model.Account[]`

      Important: Note the square brackets—the variable *accounts* is an array of accounts.

6. In the Palette view, select **Form Tags** → **Form** and click anywhere on the JSP page. A dashed box will appear on the JSP page, representing the new form.
7. In the Properties view, enter the following:
   – Action: `UpdateCustomer`
   – Method: Select **Post**.
8. Add a table with customer information:
   a. In the Page Data view, expand and select **Scripting Variables** → **requestScope** → **customer (itso.bank.model.Customer)**.
   b. Click and drag the customer request object to area inside the form that was previously created.
   c. When the Insert JavaBean window appears, do the following:
      - Select **Displaying data (read-only)**.

- Use the arrow up and down buttons to arrange the fields in the order shown in Figure 11-38.
- Edit the labels as shown in Figure 11-38.
- Click **Finish**.

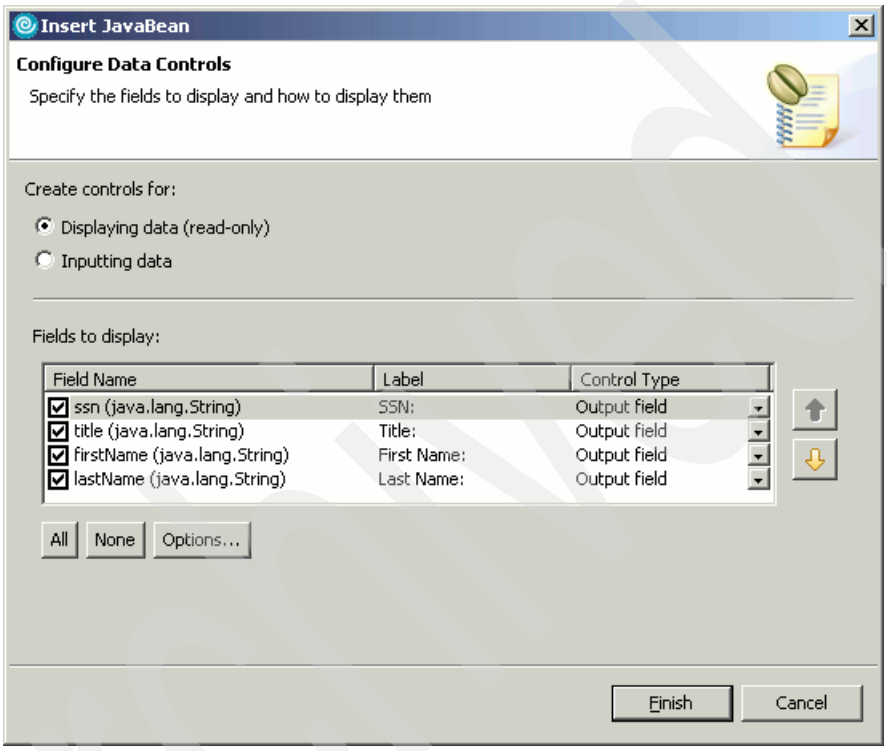


Figure 11-38 Inserting the customer JavaBean

> **Note:** The newly created table with customer data will be changed in a later stage to use input fields for the title, first name and last name fields. At the time of writing, this was not possible to achieve through the use of the available wizards.

9. Right-click the last row of the newly created table and select **Table** → **Add Row Below**.
10. In the Palette view, select **Form Tags** → **Submit Button** and click in the right-hand cell of the new row.
11. When the Insert Submit Button window appears, enter `Update` in the Label field and click **OK**.

12. In the Palette view, select **HTMLTags** → **Horizontal Rule** and click in the area below the form.
13. In the Page Data view, expand and select **Scripting Variables** → **requestScope** → **accounts (itso.bank.model.Account[])**.
14. Click and drag the customer request object to area below the horizontal rule.
15. When the Insert JavaBean wizard appears, do the following, as seen in Figure 11-39 on page 598, and click **Finish**.
    - For both fields, select **Output link** in the Control Type column.
    - In the Label field for the accountNumber field, enter `Account Number`.
    - Ensure that the order of the fields is:
      - accountNumber
      - balance

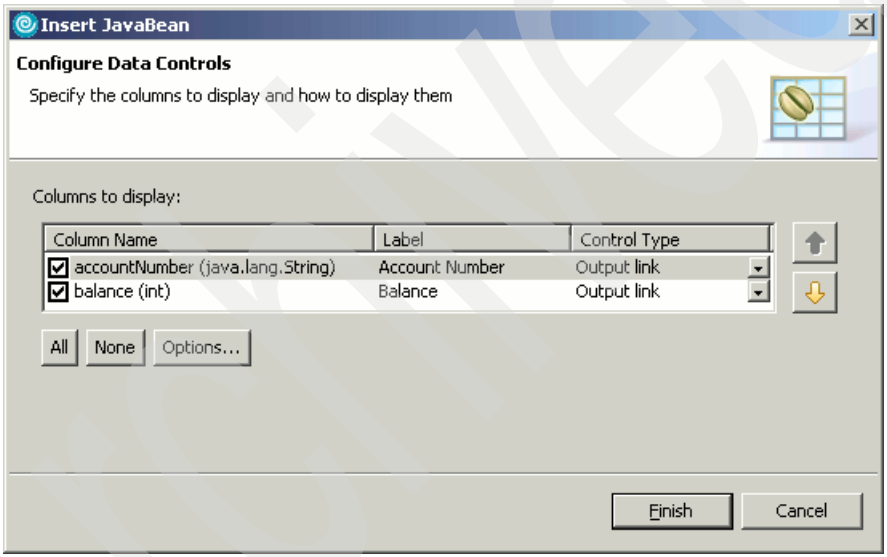


Figure 11-39 Inserting the accounts JavaBean

16. The wizard will insert a JSTL `c:forEach` tag and an HTML table with headings, as selected in the Insert JavaBean window. Since we selected **Output link** as the Control Type for each column, corresponding `c:url` tags have been inserted. We now need to edit the URL for these links to be identical:
    a. Select the first c:url tag: `${varAccounts.accountNumber}{}`. In the Properties view, enter the following in the Value field:

       ```
       AccountDetails
       ```

    b. Select the second c:url tag: `${varAccounts.balance}{}`. In the Properties view, enter the following n the Value field:

AccountDetails

c. In the Palette view, select JSP Tags → Parameter and click the first `c:url` tag (`${varAccounts.accountNumber}{}`).

d. In the Properties view, enter `accountId` in the Name field and `${varAccounts.accountNumber}` in the Value field.

e. Repeat the two previous steps to add a parameter to the second `c:url` tag (`${varAccounts.balance}{}`).

17. Select the `c:out` tag for outputting the balance (the tag with the text ${varAccounts.balance}). In the Properties view, enter `${varAccounts.balance/100}` in the Value or EL expression field.

18. In the Properties view, select the lower-most **td** tab and select **Right** in the Horizontal alignment list box.

19. In the Palette view, select **HTMLTags** → **Horizontal Rule** and click in the area below the account details table.

20. Add a logout form:

a. In the Palette view, select **Form Tags** → **Form** and click below the new horizontal rule. A dashed box will appear on the JSP page, representing the new form.

b. In the Properties view, enter the following:

- Action: `Logout`
- Method: Select **Post**.

c. In the Palette view, select **Form Tags** → **Submit Button** and click in the right-hand cell of the new row.

d. When the Insert Submit Button window appears, enter `Logout` in the Label field and click **OK**.

The JSP should now look similar to Figure 11-40 on page 600. The only part missing is to change the title, first name and last name to be entry fields, such that the user can update the customer details.

Archived

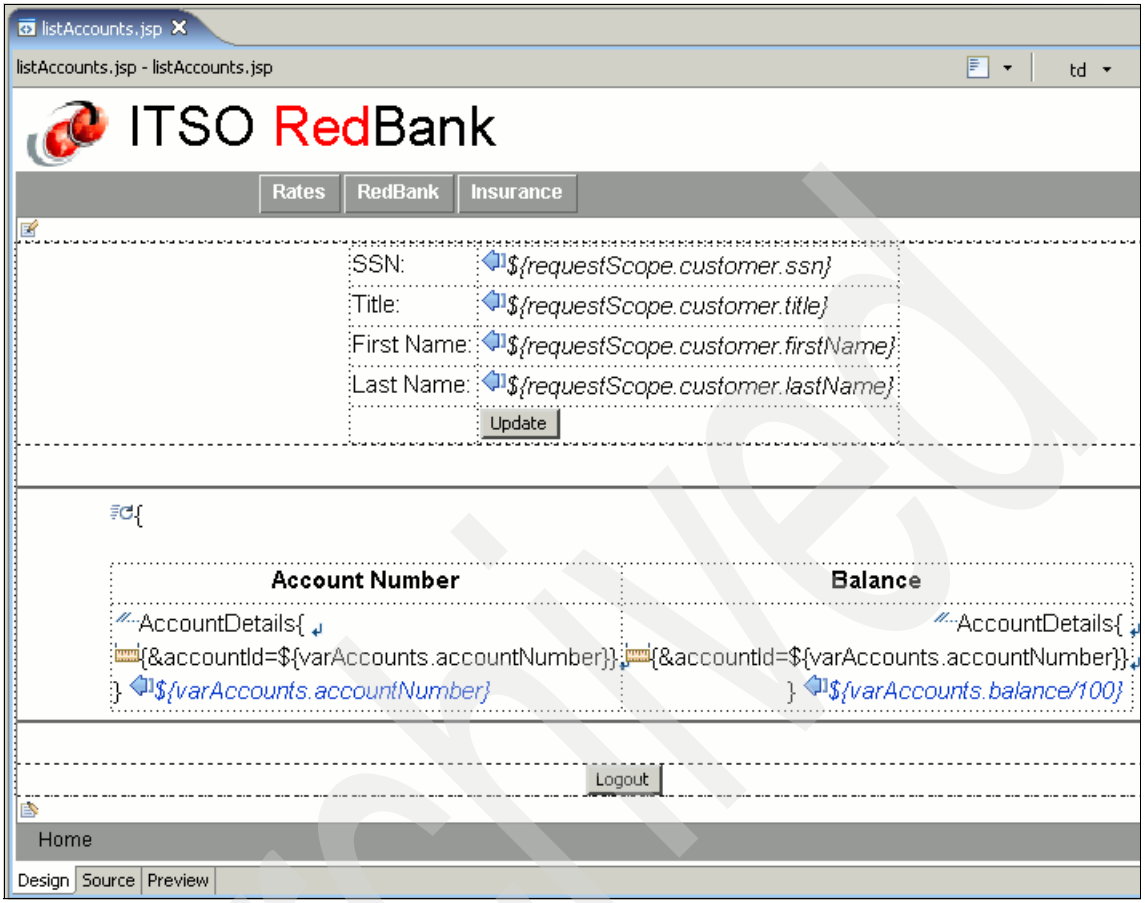


Figure 11-40 listAccounts.jsp before adding entry fields

21. Do the following to convert the Title, First Name, and Last Name text fields to allow text entry:

    a. Click the **Source** tab.

    b. Locate the lines shown in Example 11-26 in the source code.

Example 11-26 Source code lines that must be changed

```
<tr>
   <td align="left">SSN:</td>
   <td><c:out value="${requestScope.customer.ssn}" /></td>
</tr>

<tr>
   <td align="left">Title:</td>
```

```
   <td><c:out value="${requestScope.customer.title}" /></td>
</tr>

<tr>
   <td align="left">First Name:</td>
   <td><c:out value="${requestScope.customer.firstName}" /></td>
</tr>

<tr>
   <td align="left">Last Name:</td>
   <td><c:out value="${requestScope.customer.lastName}" /></td>
</tr>
```

c. Replace the lines with the lines shown in Example 11-27 in the source code. The differences have been highlighted in **bold**.

Example 11-27 Changed lines to support entry fields

```
<tr>
   <td align="left">SSN:</td>
   <td><c:out value="${requestScope.customer.ssn}" /></td>
</tr>

<tr>
   <td align="left">Title:</td>
   <td><input type="text" name="title"
         value="<c:out value='${requestScope.customer.title}' />" /></td>
</tr>

<tr>
   <td align="left">First Name:</td>
   <td><input type="text" name="firstName"
         value="<c:out value='${requestScope.customer.firstName}' />" /></td>
</tr>

<tr>
   <td align="left">Last Name:</td>
   <td><input type="text" name="lastName"
         value="<c:out value='${requestScope.customer.lastName}' />" /></td>
</tr>
```

22. Save the file by pressing Ctrl+S.

The listAccounts.jsp page is now complete. The page should look like Figure 11-41 on page 602 when shown in the Design view.

Archived

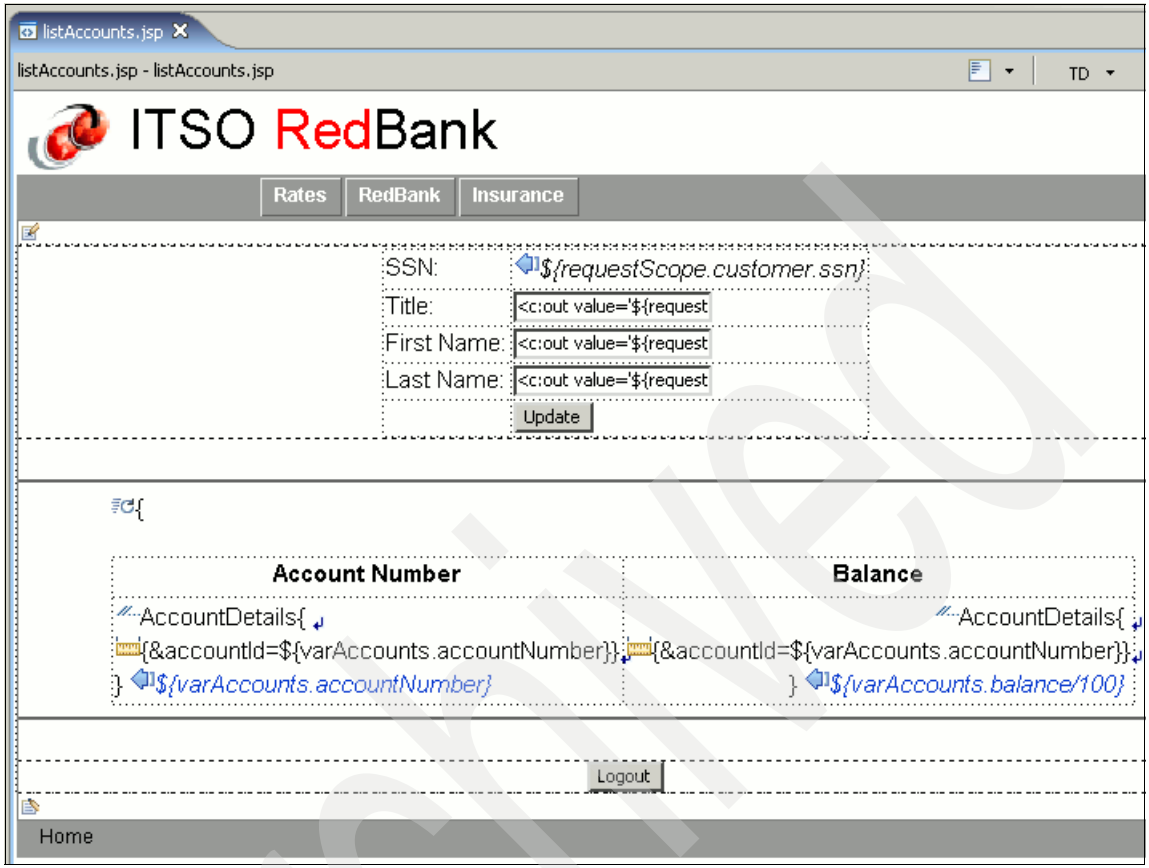


Figure 11-41 Finished listAccounts.jsp in the Preview view

The JSP tags are shown as icon boxes. Double-click a JSP tag icon and view the definition in the Attributes view, or view the content in the Source view.

Implement accountDetails.jsp

The listAccounts.jsp page will forward the user to the AccountDetails servlet when one of the lines in the account list is clicked. As shown in Example 11-18 on page 584, the servlet will add the account attribute to the request and forward control to the accountDetails.jsp page.

The following steps describe how to implement the accountDetails.jsp page:

1. Expand **Dynamic Web Projects** → **BankBasicWeb** → **WebContent**.
2. Double-click the **accountDetails.jsp** file to open in Page Designer.
3. Click the **Design** tab.

Archived

4. Click the content area and select **Center** in the Horizontal alignment drop-down box of the Properties view.
5. Using the method described in “Implement listAccounts.jsp” on page 595 to add the request scope variable `account` to the page. The variable has the type itso.bank.model.Account.
6. Drag the `account` request scope variable to the content area. When the Insert JavaBean window appears:
   - Select **Displaying data (read-only)**.
   - Ensure that **accountNumber** and **balance** appear in that order.
   - Enter `Account Number:` in the Label field for the accountNumber variable.
   - Click **Finish**.
7. Select the balance tag (the c:out tag with the display text ${requestScope.account.balance}). In the Properties view, enter `${requestScope.account.balance/100}` in the Value or EL expression field.
8. Add a horizontal rule below the new table.
9. Add a form under the horizontal rule. The form should have the following properties:
   - Action: `PerformTransaction`
   - Method: Select **Post**.
10. In the Properties view, select the **Hidden fields** tab below the **FORM** tab.

> **Tip:** This tab may be hidden. If so, click the small triangular down-arrow below the **FORM** tab to scroll down in the tab list.

11. Click **Add** and enter the following in the first row of the Hidden fields table:
    - Name: `accountId`
    - Value: `<c:out value="${requestScope.account.accountNumber}" />`
12. Add a table to the form. The table should have 5 rows and 4 columns, have no borders and have a padding of 5 pixels inside the cells.
13. In the Palette view, select **Form Tags** → **Radio Button** and click the top-left cell of the table.
14. When the Insert Radio Button window appears, enter `transaction` in the Group name field, `list` in the Value field, check **Selected** and click **OK**.
15. Enter the description `List transactions` in the cell to the right of the new radio button.
16. Repeat the previous steps to add the remaining radio buttons with descriptions from Table 11-4 to the cells in the first column of the table.

Archived

Table 11-4 Radio buttons on the accountDetails.jsp page

Value	Selected	Description
list	Checked	List transactions
withdraw	Unchecked	Withdraw
deposit	Unchecked	Deposit
transfer	Unchecked	Transfer

17. Enter the text `Amount:` to the right of the `Withdraw` description (third cell in the second row).
18. Enter the text `To account:` to the right of the `Transfer` description (third cell in the fourth row).
19. In the Palette view, select **Form Tags** → **Text Field** and click the cell to the right of the text `Amount:` (fourth cell in the second row).
20. When the Insert Text Field window opens, enter `amount` in the Name field and click **OK**.
21. In the Palette view, select **Form Tags** → **Text Field** and click the cell to the right of the text `To account:` (fourth cell in the fourth row).
22. When the Insert Text Field window opens, enter `targetAccountId` in the Name field and click **OK**.
23. In the Palette view, select **Form Tags** → **Submit Button** and click the first cell in the last row of the table.
24. When the Insert Submit Button window opens, enter `Submit` in the Label field and click **OK**.
25. Perform the following to merge cells in the table. The finished table should look similar to Figure 11-43 on page 606.
    a. Select cells two through four on the first row of the table (the cell with the description `List transactions` and the two adoining cells to the right), right click and select **Table** → **Join Selected Cells** to merge the cells.
    b. Select the cells in the second column of rows two and three (the cell with the description `Amount:` and the cell below it), right click and select **Table** → **Join Selected Cells** to merge the cells.
    c. Select the cells in the last column of rows two and three (the cell with the `amount` Text the cell below it), right click and select **Table** → **Join Selected Cells** to merge the cells.
    d. Select all the cells in the last row, right click and select **Table** → **Join Selected Cells** to merge the cells.

Archived

e. With the last row still selected, select **Center** in the Horizontal Alignment list box in the Properties view.

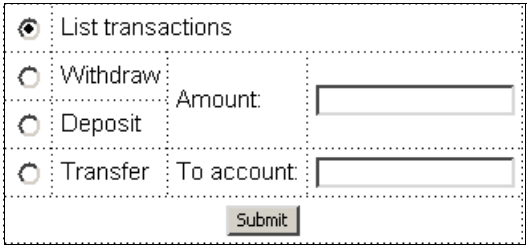


Figure 11-42 accountDetails.jsp table after merging cells

26. Add a new form under the table. The form should have the following properties:
    - Action: `ListAccounts`
    - Method: Select **Post**.
27. Add a Submit Button with the label `Customer Details` to the new form.
28. Save the page. The page should look similar to Figure 11-43 on page 606.

Archived

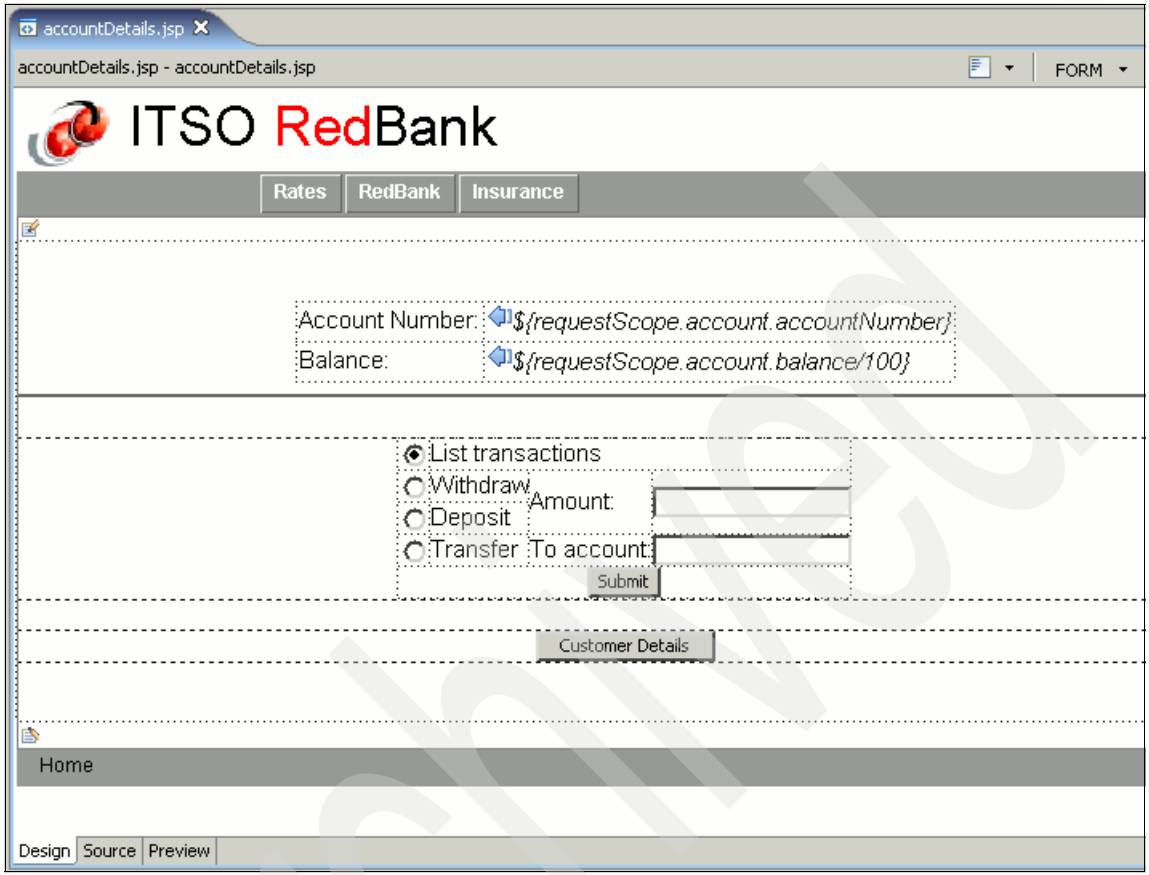


Figure 11-43 Completed accountDetails.jsp in the Design view

Implementing listTransactions.jsp

As shown in Example 11-24 on page 591, the `ListTransactionsCommand` will forward the user to the listTransactions.jsp page after adding ther `account` and `transactions` request to the request.

The following steps describe how to implement the `listTransactions.jsp` page:

1. Expand **Dynamic Web Projects** → **BankBasicWeb** → **WebContent**.
2. Double-click the **listTransactions.jsp** file to open in Page Designer.
3. Click the **Design** tab.
4. Click the content area and select **Center** in the Horizontal alignment drop-down box of the Properties view.

Archived

5. Using the method described in “Implement listAccounts.jsp” on page 595 to add the two request scope variables shown in Table 11-5 to the page.

Table 11-5 New request scope variables for listTransactions.jsp

Variable Name	Variable Type
account	itso.bank.model.Account
transactions	itso.bank.model.Transaction[]

6. Drag the `account` request scope variable to the content area. When the Insert JavaBean window appears:
   - Select **Displaying data (read-only)**.
   - Ensure that **accountNumber** and **balance** appear in that order.
   - Enter `Account Number:` in the Label field for the `accountNumber` variable.
   - Click **OK**.
7. Add a horizontal rule below the new table.
8. Drag the **transactions** request scope variable to the area below the horizontal rule. When the Insert JavaBean window appears, ensure that only the columns shown in Table 11-6 are checked and that the occur in the order shown in the table. Change the values in the Label column to match information shown in Table 11-6 and click **Finish**.

Table 11-6 Columns to show in the transaction list

Column name	Label
timestamp	Time
transactionType	Type
amount	Amount

9. Select the JSTL tag under the `Time` heading (the tag in the first cell in the second row of the new table).
10. In the Properties view, select **Date and time** and enter `yyyy-MM-dd HH:mm:ss` in the format field to the right of the **Date and time** radio button.
11. Select the JSTL tag under the `Amount` heading (the tag in the right-most cell in the second row of the new table).
12. In the Properties view, enter `${varTransactions.amount/100}` in the Value or EL expression field.
13. In the Properties view, select the **td** tab just above the currently highlighted **c:out** tab.

14. Select **Right** in the Horizontal alignment drop-down box.
15. In the Properties view, select the bottom-most **table** tab.
16. Enter `1` in the Border field and `2` in the Padding field.
17. Add a horizontal rule below the new table.
18. Add a form under the horizontal rule. The form should have the following properties:
    - Action: `AccountDetails`
    - Method: Select **Post**.
19. In the Properties view, select the **Hidden fields** tab below the **FORM** tab.
20. Click **Add** and enter the following in the first row of the Hidden fields table:
    - Name: `accountId`
    - Value: `<c:out value="${requestScope.account.accountNumber}" />`
21. Add a Submit Button with the label `Account Details` to the new form.
22. Save the page. The page should look similar to Figure 11-44.

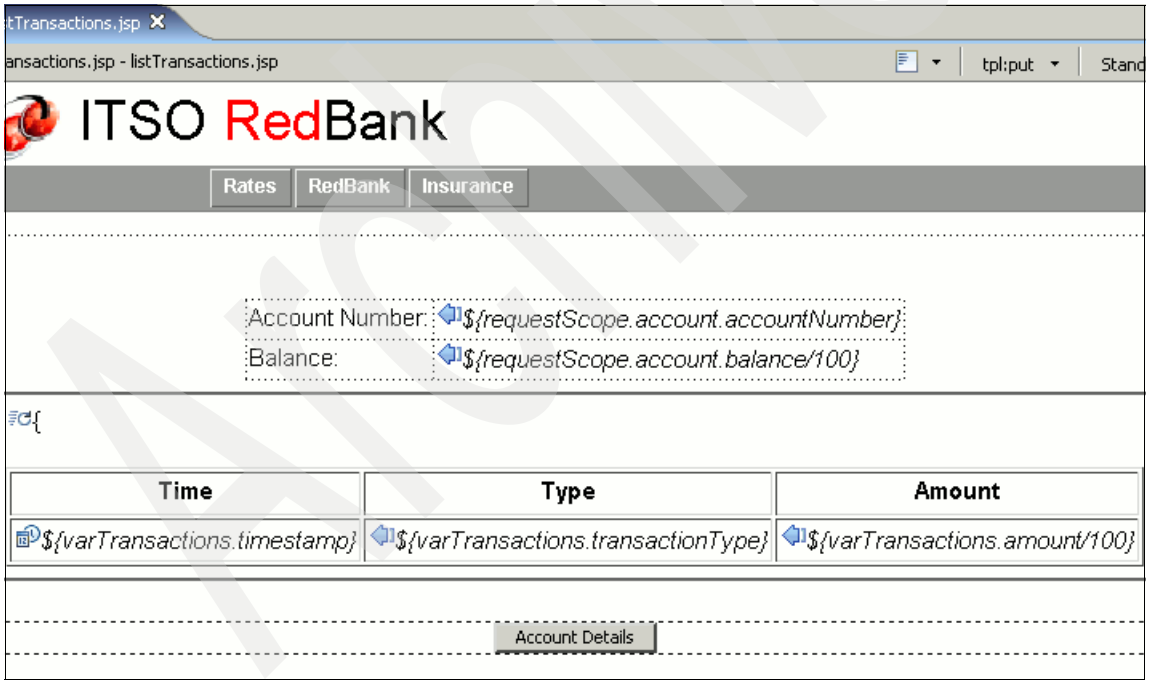


Figure 11-44 Completed listTransactions.jsp in the Design view

Archived

Implementing showException.jsp

As shown in Example 11-16 on page 581, the user will be forwarded to the showException.jsp page in case an error occurs in the servlet processing. The servlet will add the following two String attributes to the request:

- message

  A message, descibing the error condition. This will be of a technical nature, often just the exception text.

- forward

  The page that the user should be sent to after reading the error message.

The following steps describe how to implement a simple error page:

1. Expand **Dynamic Web Projects** → **BankBasicWeb** → **WebContent**.
2. Double-click the **showException.jsp** file to open in Page Designer.
3. Click the **Design** tab.
4. Click the content area and select **Center** in the Horizontal alignment drop-down box of the Properties view.
5. Using the method described in “Implement listAccounts.jsp” on page 595 to add the two request scope variables shown in Table 11-7 to the page.

Table 11-7 New request scope variables for showException.jsp

Variable Name	Variable Type
message	java.lang.String
forward	java.lang.String

6. Drag the `message` request scope variable to the content area. When the Insert JavaBean window appears:
   - Select **Displaying data (read-only)**.
   - Enter `An error has occurred:` in the Label field.
   - Click **OK**.
7. Drag the `forward` request scope variable to the content area. When the Insert JavaBean window appears:
   - Select **Displaying data (read-only)**.
   - Clear the contents of the Label field.
   - Select **Output link** in the Control Type drop-down box.
   - Click **OK**.

Archived

8. Select the **c:out** tag (the second tag with the text *${requestScope.forward}* in blue) and enter the text `Click here to continue` in the Value or EL expression field in the Properties view.
9. Save the page. The page should look similar to Figure 11-45.

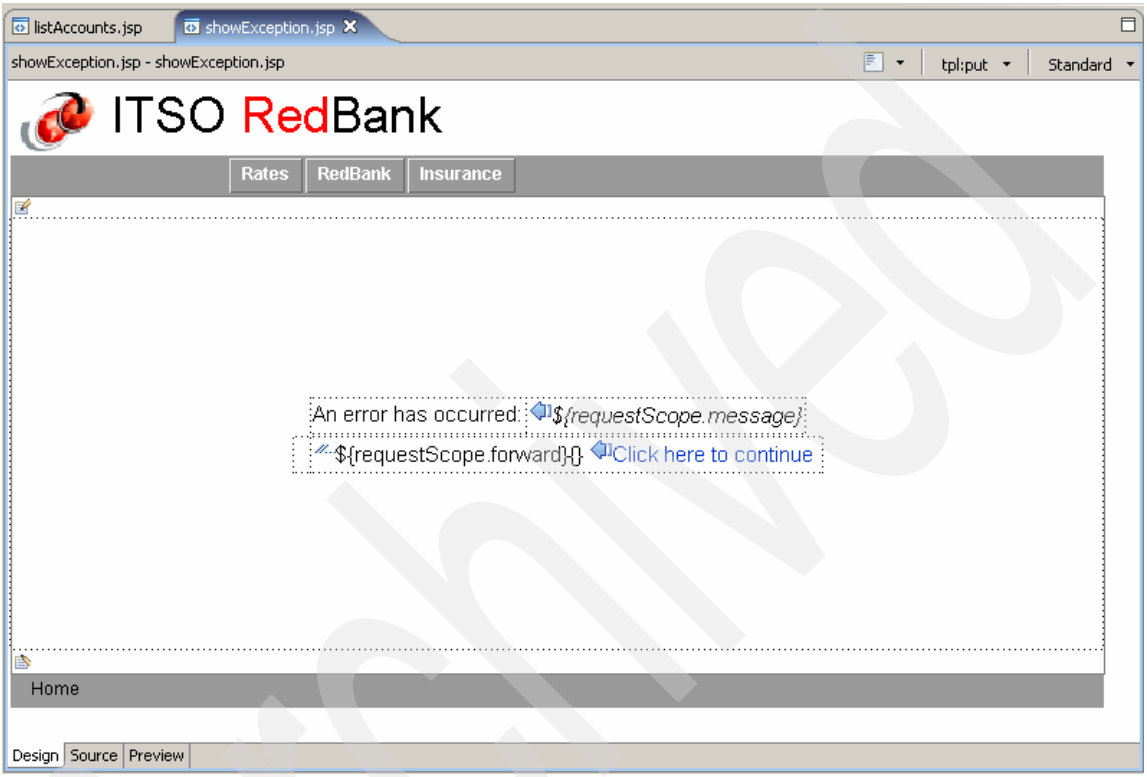


Figure 11-45 Completed showException.jsp in the Design view

11.7 Test the application

This section demonstrates how to run the sample Web application built using servlets and JSPs.

Archived

11.7.1 Prerequisites to run sample Web application

In order to run the Web application you will need to have completed one of the following:

- Complete the sample following the procedures described in the following sections:
  - 11.3, “Prepare for the sample” on page 513.
  - 11.4, “Define the site navigation and appearance” on page 524.
  - 11.5, “Develop the static Web resources” on page 544.
  - 11.6, “Develop the dynamic Web resources” on page 549.
- Import the completed sample c:\6449code\web\BankBasicWeb.zip Project Interchange file. Refer to “Import sample code from a Project Interchange file” on page 1398 for details.

11.7.2 Run the sample Web application

To run the sample Web application in the test environment, do the following:

1. Open the Web perspective Project Explorer view.
2. Expand and select **Dynamic Web Projects** → **BankBasicWeb**.
3. Right-click **BankBasicWeb**, and select **Run** → **Run on Server**.
4. When the Server Selection dialog appears, select **Choose an existing server**, select **WebSphere Application Server v6.0**, and click **Finish**.

The main page of the Web application should be displayed in a Web browser inside Rational Application Developer.

11.7.3 Verify the sample Web application

Once you have launched the application by running it on the test server, there are some basic steps that can be taken to verify the Web application is working properly.

1. From the main page, select the RedBank menu option.
2. When the ITSO RedBank login page appears, enter 111-11-1111 in the customer SSN field as seen in Figure 11-46 on page 612, and then click **Enter**.

Archived

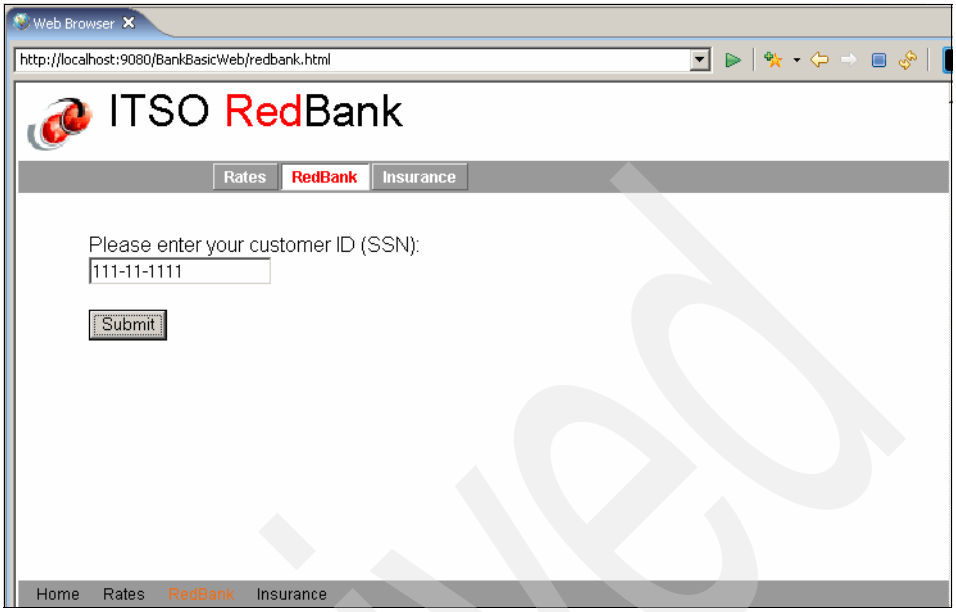


Figure 11-46 ITSO RedBank Login page

The resulting page should look like Figure 11-47.

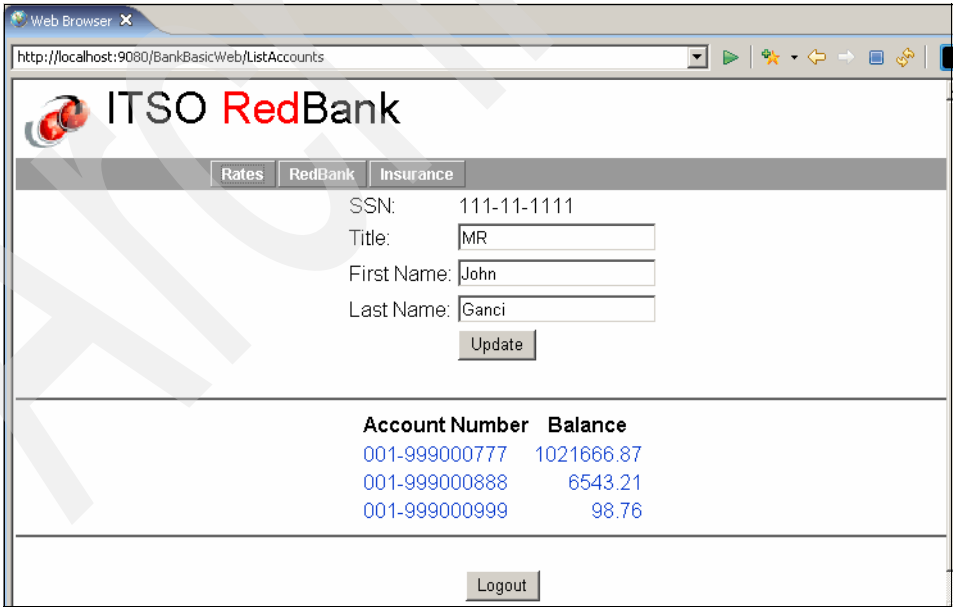


Figure 11-47 Display of customer accounts

Archived

3. From the page displayed in Figure 11-47, you can do one of the following:
   - Change the fields and then click Update. This verifies the UpdateCustomer servlet. For example, change Title to Sir and then click Update.
   - You can click Logout, which will perform a logout and return to the Login page.
   - You can click on the accounts to display the accounts information, resulting in the page displayed in Figure 11-48 on page 613.

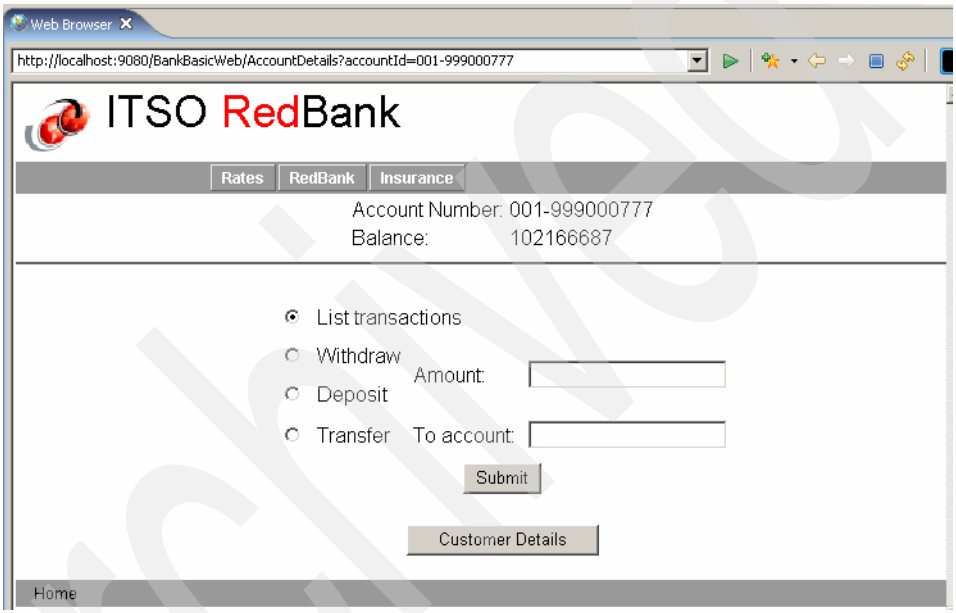


Figure 11-48 Details for a selected account

4. From the page displayed in Figure 11-48, you can do one of the following:
   - Select one of the radio buttons and click Submit. This tests the framework, created in "Implementing the PerformTransaction servlet framework" on page 586. The radio buttons works as follows:
     - List transactions: This will result in the page displayed in Figure 11-49 on page 614.

> **Note:** The page in Figure 11-49 on page 614 is the result of first testing a few withdrawals and deposits. Before testing these transactions, the List transactions page does not contain any details.

Archive

- Withdraw: This will result in the number of cents entered in the Amount field to be withdrawn from the account and the current page to be redisplayed with the updated balance.
- Deposit: This will result in the number of cents entered in the Amount field to be deposited to the account and the current page to be redisplayed with the updated balance.
- Transfer: This will result in the number of cents entered in the Amount field to be transferred to the account number entered in the To account field and the current page to be redisplayed with the updated balance.

– You can click Customer Details, which will return you to the page displayed in Figure 11-47 on page 612.

– You can click Logout, which will perform a logout and return to the home page.

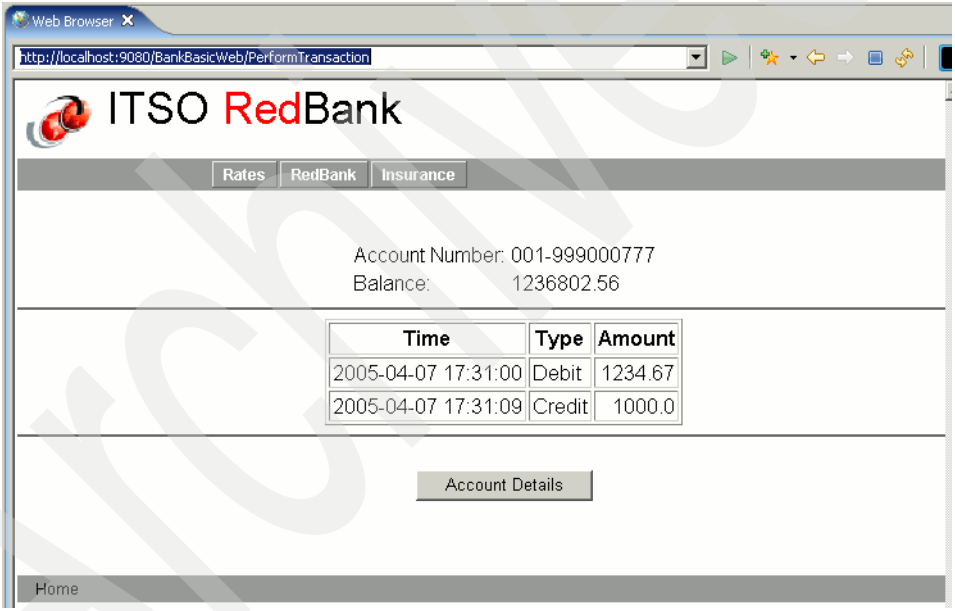


Figure 11-49 List of transactions for an account

Archived

12

Develop Web applications using Struts

Jakarta Struts is an open source framework maintained by the Apache Software Foundation, that simplifies building and maintaining Web applications.

In this chapter we introduce the concepts of the Struts framework and demonstrate the Rational Application Developer tooling used for Struts Web application development. Lastly, we have provided a completed Web application using Struts that can be imported and run within the test environment.

The chapter is organized into the following sections:

- Introduction to Struts
- Prepare for the sample application
- Develop a Web application using Struts
- Import and run the Struts sample application

© Copyright IBM Corp. 2005. All rights reserved.

12.1 Introduction to Struts

The Struts framework control layer uses technologies such as servlets, JavaBeans, and XML. The view layer is implemented using JSPs. The Struts architecture encourages the implementation of the concepts of the model-view-controller (MVC) architecture pattern. By using Struts you can get a clean separation between the presentation and business logic layers of your application.

Struts also speeds up Web application development by providing an extensive JSP tag library, parsing and validation of user input, error handling, and internationalization support.

The focus of this chapter is on the Rational Application Developer tooling used to develop Struts-based Web applications. Although we do introduce some basic concepts of the Struts framework, we recommend that you refer to the following sites:

- Apache Struts home page:

  http://struts.apache.org/

- Apache Struts User Guide:

  http://struts.apache.org/userGuide/introduction.html

Note: IBM Rational Application Developer V6.0 includes support for Struts Version 1.1. At the time of writing this book, the latest version of the Struts framework was V1.2.4.

12.1.1 Model-view-controller (MVC) pattern with Struts

In 11.1.2, “Model-view-controller (MVC) pattern” on page 503, we described the general concepts and architecture of the MVC pattern. Figure 12-1 on page 617 depicts the Struts components in relation to the MVC pattern.

- Model: Struts does not provide model classes. The business logic must be provided by the Web application developer as JavaBeans or EJBs.
- View: Struts provides action forms to create form beans that are used to pass data between the controller and view. In addition, Struts provides custom JSP tag libraries that assist developers in creating interactive form-based applications using JSPs. Application resource files hold text constants and error message, translated for each language, that are used in JSPs.
- Controller: Struts provides an action servlet (controller servlet) that populates action forms from JSP input fields and then calls an action class where the developer provides the logic to interface with the model.

Archived

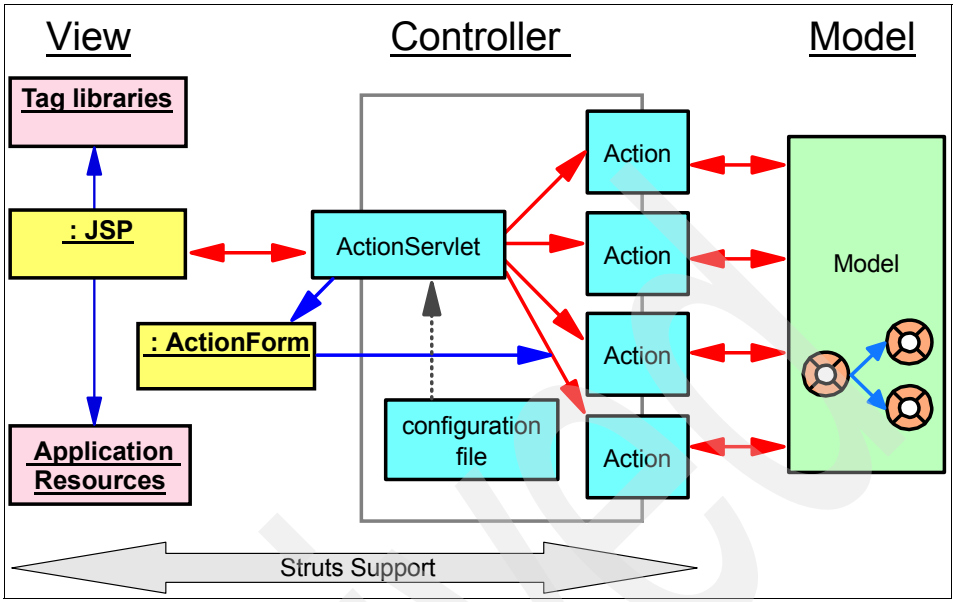


Figure 12-1 Struts components in the MVC architecture

A typical Struts Web application is composed of the following components:

- A single servlet (extending org.apache.struts.action.ActionServlet) implements the primary function of mapping a request URI to an action class. Before calling the action class, it populates the form bean associated to the action with the fields from the input JSP. If specified, the action servlet also requests the form bean to validate the data. It then calls the action class to carry out the requested function. If form bean validation fails, control is returned to the input JSP so the user can correct the data. The action servlet is configured by an XML configuration file that specifies the environment and the relationship between the participating components.
- Multiple JSPs that provide the end-user view. Struts includes an extensive tag library to make JSP coding easier. The JSPs display the information prepared by the actions and requests new information from the user.
- Multiple action classes (extending any one of the Struts action classes like org.apache.struts.action.Action) that interface with the model. When an action has performed its processing, it returns an action forward object, which determines the view that should be called to display the response. The action class prepares the information required to display the response, usually as a form bean, and makes it available to the JSP. Usually the same form bean that was used to pass information to the action is used also for the response, but it is also common to have special view beans tailored for displaying the data. An action forward has properties for its name, address (URL), and a flag

specifying if a forward or redirect call should be made. The address to an action forward is usually hard coded in the action servlet configuration file, but can also be generated dynamically by the action itself.

- Multiple action forms (extending any one of the Struts Action Form classes like org.apache.struts.action.ActionForm) to help facilitate transfer form data from JSPs. The action forms are generic Javabeans with getters and setters for the input fields available on the JSPs. Usually there is one form bean per Web page, but you can also use more coarse-grained form beans holding the properties available on multiple Web pages (this fits very well for wizard-style Web pages). If data validation is requested (a configurable option) the form bean is not passed to the action until it has successfully validated the data. Therefore the form beans can act as a sort of firewall between the JSPs and the actions, only letting valid data into the system.
- One application resource file per language supported by the application holds text constants and error messages and makes internationalization easy.

Figure 12-2 shows the basic flow of information for an interaction in a Struts Web application.

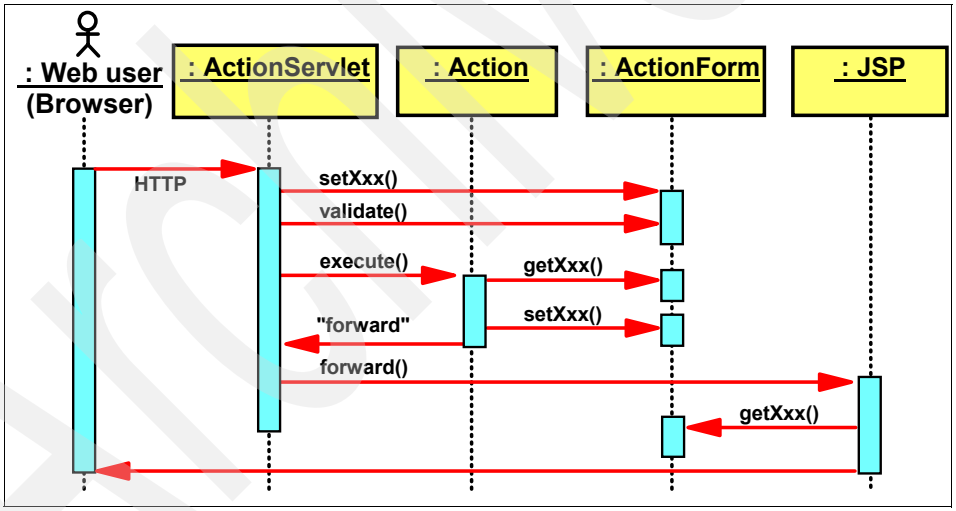


Figure 12-2 Struts request sequence

A request from a Web browser reaches the Struts ActionServlet. If the action that will handle the request has a form bean associated with it, Struts creates the form bean and populates it with the data from the input form. It then calls the validate method of the form bean. If validation fails, the user is returned to the input page to correct the input. If validation succeeds, Struts calls the action's execute method. The action retrieves the data from the form bean and performs the appropriate logic. Actions often call session EJBs to perform the business

logic. When done, the action either creates a new form bean (or other appropriate view bean) or reuses the existing one, populates it with new data, and stores it in the request (or session) scope. It then returns a forward object to the Struts action servlet, which forwards to the appropriate output JSP. The JSP uses the data in the form bean to render the result.

12.1.2 Rational Application Developer support for Struts

Rational Application Developer provides the following support for Struts-based Web applications:

- A Web Project can be configured for Struts. This adds the Struts runtime (and dependent JARs), tag libraries, and action servlet to the project, and creates skeleton Struts configuration and application resources files. Rational Application Developer provides support for Struts 1.1, selectable when setting up the project. This field is selectable, as at the time of this writing, support for Struts 1.2.x is being added to Rational Application Developer.
- A set of *Struts Component Wizards* to define action form classes, action classes with action forwarding information, and JSP skeletons with the tag libraries included.
- The *Struts Configuration Editor* to maintain the control information for the action servlet.
- A graphical design tool to edit a graphical view of the Web application from which components (forms, actions, JSPs) can be created using the wizards. This graphical view is called a *Web diagram*. The *Web diagram editor* provides top-down development (developing a Struts application from scratch), bottom-up development (that is, you can easily diagram an existing Struts application that you may have imported), and meet-in-the-middle development (that is, enhancing or modifying an existing diagrammed Struts application).
- The *Project Explorer view* provides a hierarchical (tree-like) view of the application. This view shows the Struts artifacts (such as Actions, Formbeans, Global Forwards, Global Exceptions, and Web pages). You can expand the artifacts to see their attributes. For example, an Action can be expanded to see the formbeans, and forwards and local exceptions associated with the selected Action. This is useful for understanding specific execution paths of your application. The Project Explorer view is available in the Web perspective.
- The *JSP Page Designer* support for rendering the Struts tags, making it possible to properly view Web pages that use the Struts JSP tags. This support is customizable using Rational Application Developer's Preferences settings.

Archive

- Validators to validate the Struts XML configuration file and the JSP tags used in the JSP pages.

12.2 Prepare for the sample application

This section describes the tasks that need to be completed prior to developing the Web application using Struts.

Note: A completed version of the ITSO RedBank Web application built using Struts can be found in the c:\6449code\struts\BankStrutsWeb.zip Project Interchange file.

If you do not wish to develop the sample yourself, but want to see it run, follow the procedures described in 12.4, “Import and run the Struts sample application” on page 665.

12.2.1 ITSO Bank Struts Web application overview

We will use the ITSO Bank as the theme of our sample application. Similar samples were developed in the following chapters using other Web application technologies:

- Chapter 11, “Develop Web applications using JSPs and servlets” on page 499
- Chapter 13, “Develop Web applications using JSF and SDO” on page 673

The ITSO Bank sample application allows a customer to enter a customer ID (social security number), select an account to view detailed transaction information, or perform a deposit or withdrawal on the account. The model layer of the application is implemented within the Action classes using Java beans for the sake of simplicity. We will implement the business tier using EJB technology developed in Chapter 15, “Develop Web applications using EJBs” on page 827.

In the banking sample, we will use the Struts framework for the controller and view components of the Web application, implement the model using JavaBeans and Cloudscape bundled with Rational Application Developer.

Figure 12-3 on page 621 displays the Struts Web diagram for the sample banking application. The basic description and flow of the banking sample application are as follows

1. The logon.jsp page is displayed as the initial page of the banking sample application. The customer is allowed to enter her or his social security number. In our case we will use simple validation to check for an empty value

entered using the Struts framework. If the customer does not enter a valid value, the Struts framework will return to the logon.jsp page and display the appropriate message to the user.

2. The logon action logs in the user, and on successful logon retrieves the customer's account information and lists all the accounts associated with the customer using the customerlisting.jsp Web page.
3. In cutsomerlisting.jsp, the customer can select to see details of a selected account or select to perform a transaction on an account using the accountDetails and performTransaction actions, respectively.
4. The customer can log off using the logoff link, which will invoke the logoff action.

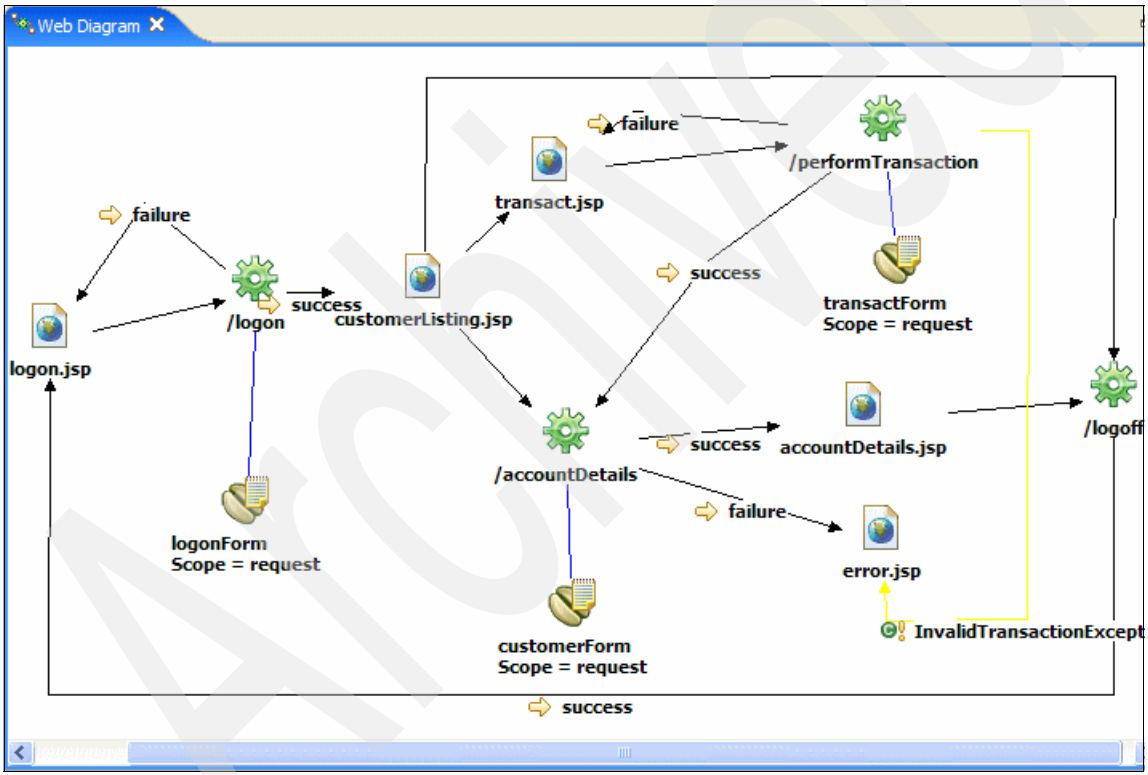


Figure 12-3 Struts Web diagram - Bank sample

In this section we focus on creating the various Struts components, including the Struts Controller, Action, Form bean, and Web pages, and relate these

components together. We will implement the following steps to demonstrate the capabilities of Rational Application Developer:

- Create a Dynamic Web Application with Struts support: In this section the process of creating a dynamic Web application with Struts support and the wizard generated support for Struts will be described.
- Create Struts Components: In this section we will focus on creating Web pages, Actions, Form Beans, Exceptions (Local and Global), and Forwards using *Web diagrams*, and modify the properties of Struts components using the Struts Configuration Editor.
- Use the Struts Configuration Editor: In this section we will focus on creating Struts components using the Struts Configuration Editor that provides a visual editor to modify the Struts Configuration file struts-config.xml.
- Import the complete Sample Banking Web-application: In the previous sections we created various Struts components. Here we will import the complete banking sample implemented as shown in Figure 12-3 on page 621.
- Run the sample Banking Application: In this section we will verify the datasource configurations in the Extended Application Descriptor of the imported sample and then run and test the application in the WebSphere V6.0 Test Environment.

Note: Because this chapter focuses on Application Developer's Struts tools and wizards (more than the architecture and best practices of a Struts application) we try to use the Struts tools and wizards as much as possible when creating our application.

After having used the wizards to create some components (JSPs, form beans, actions) you may find it faster to create new components by copying and pasting from your existing components than by using all the wizards.

12.2.2 Create a Dynamic Web Project with Struts support

To create a dynamic Web Project with Struts support, do the following:

1. Open the Web perspective.
2. Select **File** → **New** → **Project** from the menu.
3. When the New Project Wizard appears, select **Dynamic Web Project** and click **Next**.
4. Enter `BankStrutsWeb` in the Name field and click **Next**.
5. When the Features dialog appears, check **Struts** as seen in Figure 12-4 on page 623, accept defaults for other features, and then click **Next**.

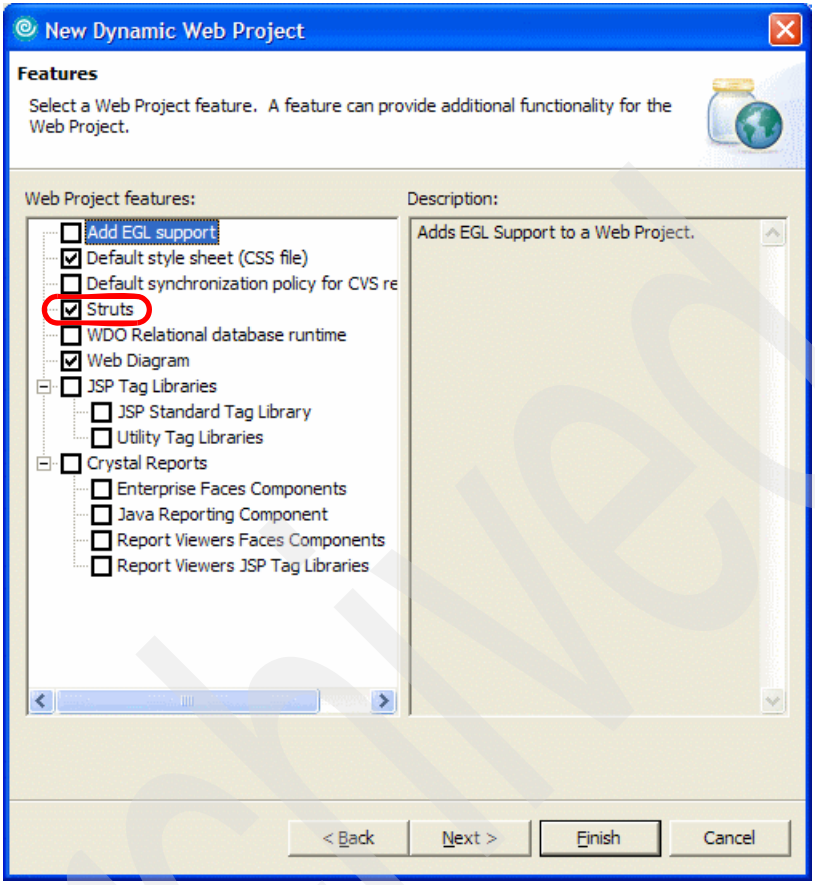


Figure 12-4 Create a Struts-based Dynamic Web Project - Select Features

6. We accepted the default on the Select a Page Template for the Web site page. Click **Next**.
7. When the Struts Settings page appears, since we enabled Struts, do the following (as seen in Figure 12-5 on page 624), and then click **Finish**:
   - Check **Override default settings**.
   - Struts version: Select **1.1** (default).
   - Default Java package prefix: `itso.bank.model.struts`

Note: At the time of writing, Struts V1.1 is the only version supported and available from the list box.

Archived

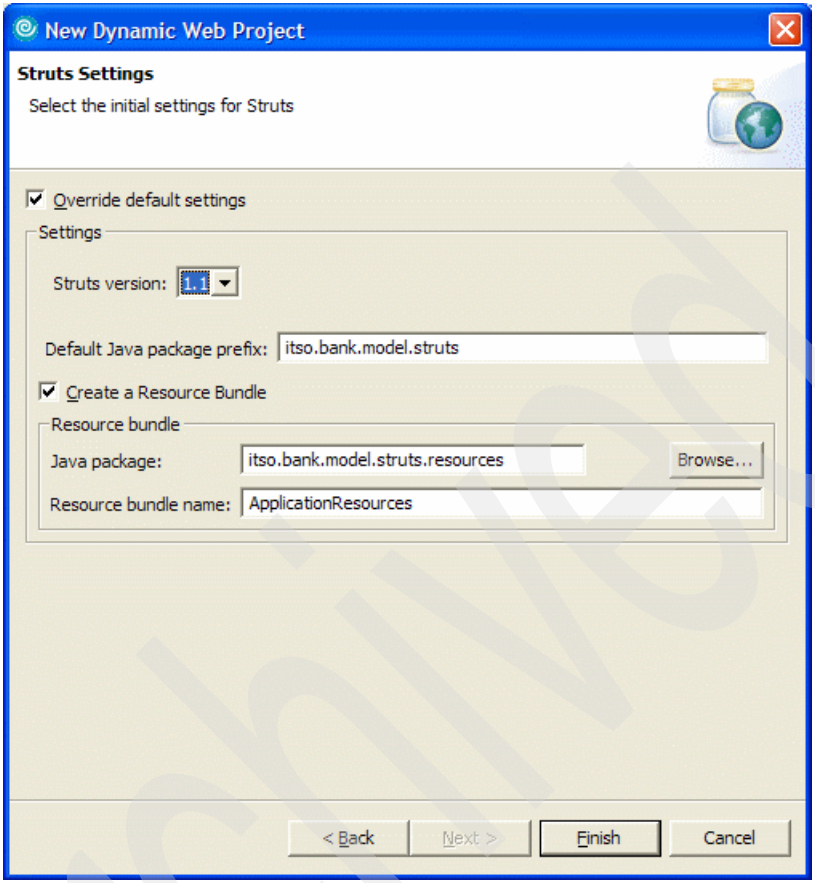


Figure 12-5 Create a Struts-based Dynamic Web Project - Struts Settings

At this point a new dynamic Web Project with Struts support called *BankStrutsWeb* has been created.

Note: If you create a Web Project as part of creating an Enterprise Application Project you will not be given the option to add Struts support at that time. You will have to add the Struts support afterwards by selecting **Properties** from the Web Project's context menu, selecting **Web Project Features** in the left pane, and checking the **Struts** option there.

Archived

The following Struts-specific artifacts are created, and Web application configurations are modified by the wizard related to Struts when a new dynamic Web application is created with Struts support:

- The wizards add the Struts configuration file struts-config.xml, the Struts jar files under the WEB-INF/lib directory, and the tag library(tild) files under the WEB-INF directory, as shown in Figure 12-6 on page 626.
- Modifies the Web deployment descriptor web.xml and adds the following elements to the deployment descriptor:
  - The Struts Action Servlet and a servlet mapping for the Action servlet to handle all client requests matching the regular expression pattern *.do, as shown in Figure 12-7 on page 627.
  - The tld files References for the Struts tag libraries, as shown in Figure 12-8 on page 628.

  Note: The Struts action servlet is configured (in web.xml) to intercept all requests with a URL ending in .do (the servlet mapping is *.do). This is common for Struts applications, but equally common is using a servlet mapping of /action/* to intercept all URLs beginning with /action.

- Adds the ApplicationResources.properties in the package specified in step 4 in the creation of a dynamic project. This is a property file used in the Struts Web framework to display messages and form field names in a language and locale independent fashion.
- A default Struts Module called <default Module> is created, under which all the Struts components are created.
- A default Web diagram diagram.gph is created. The Struts Web Diagram Navigator is used to create Struts components. The Web Diagram Navigator is used in a top-down design approach for creating Web Applications.

Archived

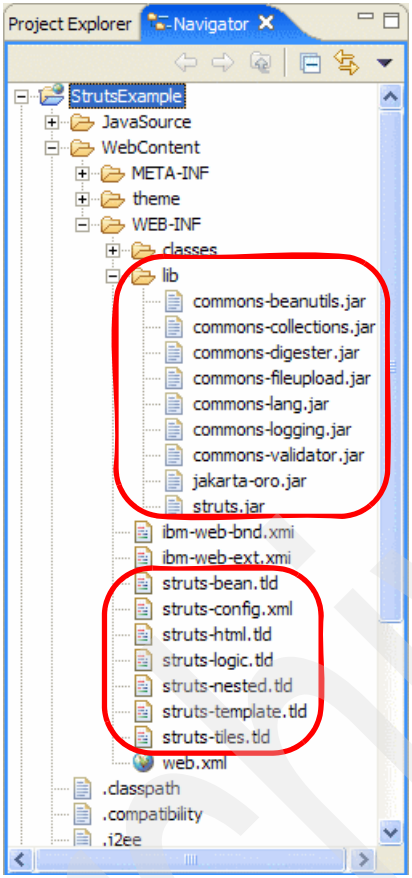


Figure 12-6 Generated struts-config.xml, Struts jars, and tag library definitions

Archived

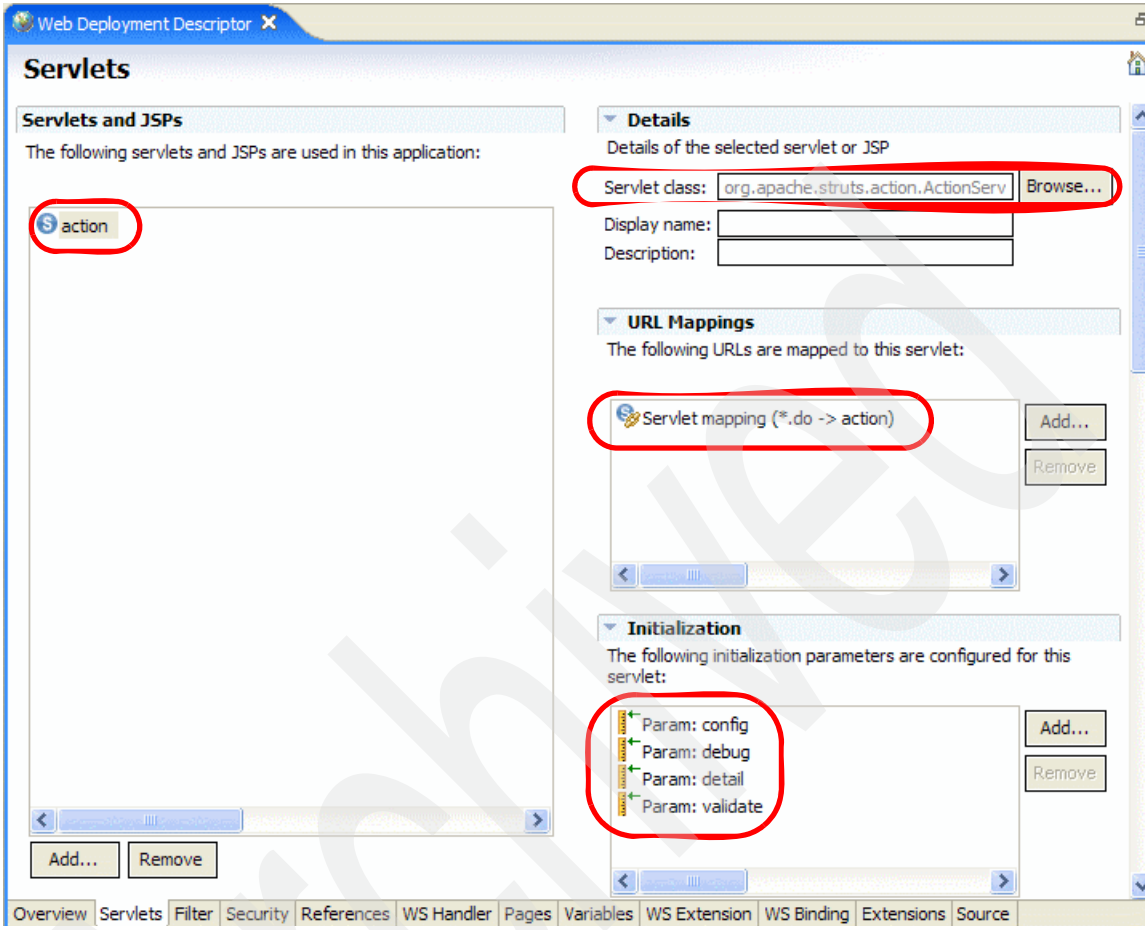


Figure 12-7 Generated Web Deployment Descriptor config for action servlet

Archive

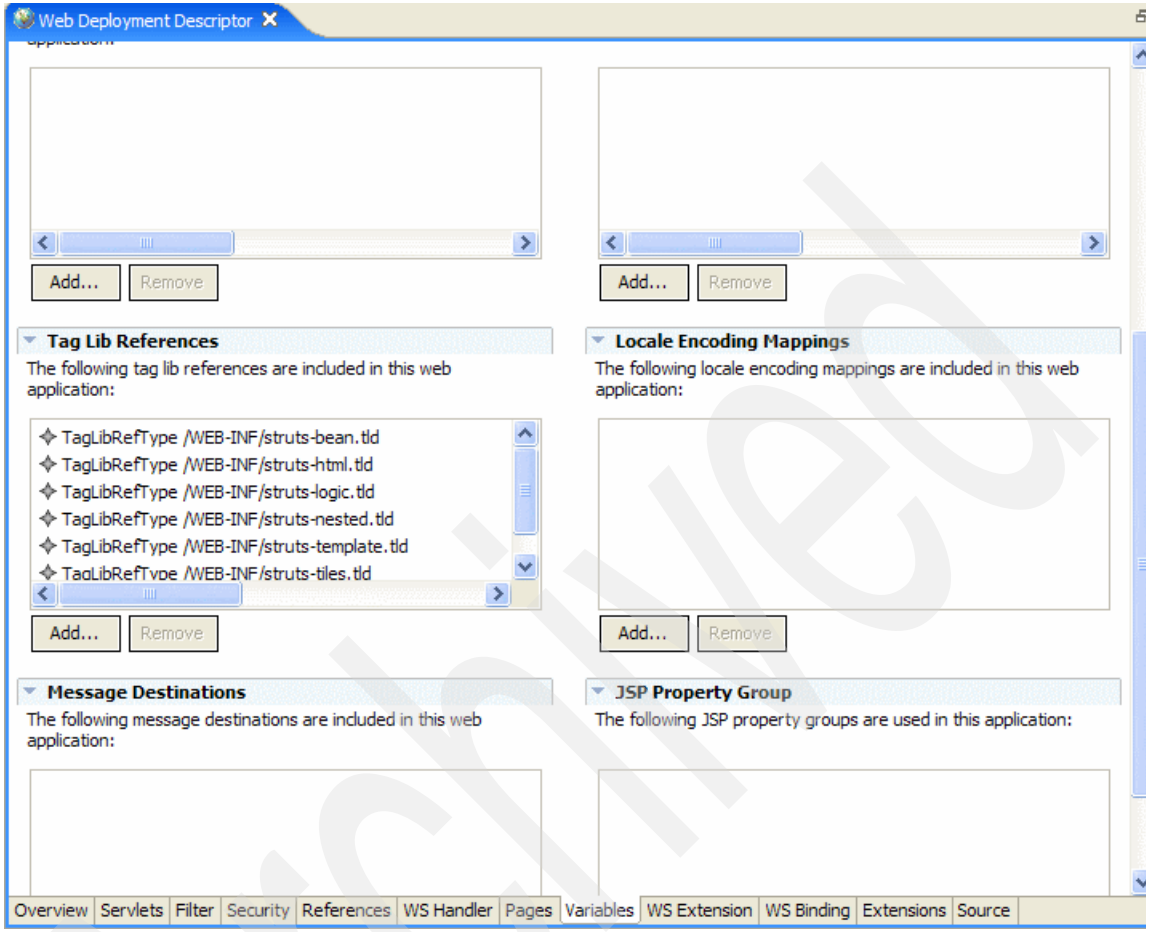


Figure 12-8 Generated Web Deployment Descriptor config for Struts tag library references

12.2.3 Add JDBC driver for Cloudscape to project

To add the Cloudscape JDBC driver to the project, do the following:

1. Open the Web perspective Project Explorer view.
2. Expand **Dynamic Web Projects**.
3. Right-click **BankStrutsWeb** and select **Properties**.
4. Select **Java Build Path**.
5. Add the JAR file that contains the JDBC drivers for the Cloudscape database we will be using in the sample.
   a. Select the **Libraries** tab at the top of the dialog and click **Add Variable...**.

Archived

A further dialog appears, allowing you to select from a list of predefined variables. By default, there is no variable defined for the JAR file we need, so we will have to create one.

b. Click **Configure Variables...** and in the resulting dialog click **New...**.

c. Enter `CLOUDSCAPE_DRIVER_JAR` in the Name field and click **File...**.

d. Find the appropriate JAR file, which is in <rad_home>\runtimes\base_v6\cloudscape\lib and is called db2j.jar.

e. Click **Open**, **OK**, and **OK** and you will be back at the New Variable Classpath Entry dialog.

f. Click **OK** when prompted for a Full Rebuild of the project workspace with the new variable added to classpath.

g. Select the **CLOUDSCAPE_DRIVER_JAR** variable you just created and click **OK**.

6. If you were not prompted for a full rebuild of the project workspace in step 3e, Rational Application Developer has been configured to not perform automatic builds. Perform a project build by selecting **Project** → **Build All**. (This option is disabled if automatic builds are selected; **Project** → **Build Automatically**).

7. Unless you have previously turned this feature off, Rational Application Developer will display a Confirm Perspective Switch dialog asking whether you want to switch to the Java perspective. Click **Yes**. If you have turned this feature off, you will need to open the Java perspective now.

12.2.4 Set up the sample database

This section provides instructions for deploying the BANK sample database and populating the database with sample data. For simplicity we will use the built-in Cloudscape database.

To create the database, create the connection, and create and populate the tables for the BANK sample from within Rational Application Developer, do the following:

1. Create the database and connection to the Cloudscape BANK database from within Rational Application Developer.

   For details refer to “Create a database connection” on page 347.

2. Create the BANK database tables from within Rational Application Developer.

   For details refer to “Create database tables via Rational Application Developer” on page 350.

3. Populate the BANK database tables with sample data from within Rational Application Developer.

Archived

For details refer to “Populate the tables within Rational Application Developer” on page 352.

12.2.5 Configure the data source

There are a couple of methods that can be used to configure the datasource, including using the WebSphere Administrative Console or using the WebSphere Enhanced EAR, which stores the configuration in the deployment descriptor and is deployed with the application.

This section describes how to configure the datasource using the WebSphere Enhanced EAR capabilities. The enhanced EAR is configured in the Deployment tab of the EAR deployment descriptor.

The procedure found in this section considers two scenarios for using the enhanced EAR:

- If you choose to import the complete sample code, you will only need to verify that the value of the databaseName property in the deployment descriptor matches the location of your database.
- If you are going to complete the working example Web application found in this chapter, you will need to create the JDBC provider, the datasource, and update the databaseName property.

Note: For more information on configuring data sources and general deployment issues, refer to Chapter 23, “Deploy enterprise applications” on page 1189.

Access the deployment descriptor

To access the deployment descriptor where the enhanced EAR settings are defined, do the following:

1. Open the Web perspective Project Explorer view.
2. Expand **Enterprise Applications** → **BankStrutsWebEAR**.
3. Double-click **Deployment Descriptor : BankStrutsWebEAR** to open the file in the Deployment Descriptor Editor.
4. Click the **Deployment** tab.

Note: For JAAS Authentication, when using Cloudscape, the configuration of the user ID and password for the JAAS Authentication is not needed.

When using DB2 Universal Database or other database types that require a user ID and password, you will need to configure the JAAS authentication.

Archived

Configure a new JDBC provider

By default, when using Cloudscape, the Cloudscape JDBC Provider (XA), supporting two-phase commit, is predefined in the JDBC provider list. In our example, we do not take advantage of the two-phase commit feature, but it does not cause any harm using this provider.

Note: This step is optional for our example, since we will use the predefined Cloudscape JDBC Provider (XA).

The following procedure is only needed if you wish to add a new JDBC provider using the enhanced EAR capability in the deployment descriptor:

1. From the Deployment tab of the Application Deployment Descriptor, click **Add** under the JDBC provider list.
2. When the Create a JDBC Provider dialog appears, select **Cloudscape** as the Database type, select **Cloudscape JDBC Provider** as the JDBC provider type, and then click **Next**.
3. Enter `Cloudscape JDBC Provider` in the Name field and then click **Finish**.

Configure the data source

To configure a new data source using the enhanced EAR capability in the deployment descriptor, do the following:

1. From the Deployment tab of the Application Deployment Descriptor, select the JDBC provider.
2. Click **Add** next to data source.
3. When the Create a Data Source dialog appears, select **Cloudscape JDBC Provider (XA)** under the JDBC provider, select **Version 5.0 data source**, and then click **Next**.
4. When the Create a Data Source dialog appears, enter the following and then click **Finish**:
   - Name: `BankDS`
   - JNDI name: `jdbc/BankDS`

Configure the databaseName property

To configure the databaseName in the new data source using the enhanced EAR capability in the deployment descriptor to define the location of the database for your environment, do the following:

1. Select the data source created in the previous section.
2. Select the **databaseName** property under the Resource properties.

Archive

3. Click **Edit** next to Resource properties to change the value for the databaseName.
4. When the Edit a resource property dialog appears, enter `c:\databases\BANK` in the Value field and then click **OK**.

   In our example, c:\databases\BANK is the database created for our sample application.
5. Save the Application Deployment Descriptor.

   The changes will not take effect until the server is restarted.

Note: If your enterprise application is already deployed, it may be necessary to remove the application from the server and then republish it.

12.3 Develop a Web application using Struts

This section describes how to develop a Web application using Struts with the tooling provided by Rational Application Developer.

The section is organized into the following tasks:

- Create the Struts components.
- Realize the Struts components.
- Modify ApplicationResources.properties.
- Struts validation framework.
- Page Designer and the Struts tag library.
- Using the Struts configuration file editor.

Important: This section demonstrates how to develop a Web application using Struts with the tooling included with Rational Application Developer. We do not cover the details for all of the sample code. A procedure to import and run the completed Struts Bank Web application sample can be found in 12.4, “Import and run the Struts sample application” on page 665.

12.3.1 Create the Struts components

There are several ways to create Struts components:

- In the Project Explorer, expand and select **Dynamic Web Projects** → *<project>* → **Struts**. Use New from the context menu of the Struts to create Struts Modules, Actions, FormBeans, Global Forwards, and Global Exceptions.
- Using the Struts Configuration Editor: Rational Application Developer provides a Struts configuration editor, which is used to create Struts

components and to modify the Struts configuration file struts-config.xml. Using the Struts Configuration Editor will be described in detail in 12.3.6, “Using the Struts configuration file editor” on page 659.

- In the Struts Web Diagram Navigator, use the Struts Palette to create Struts components.

In this chapter, we take a top-down approach to design the Web application by laying out all the components in the Web diagram using the Web Diagram Navigator.

This section is organized into the following tasks:

- Start the Web Diagram editor.
- Create a Struts Action.
- Create a Struts Form Bean.
- Create a Web Page.
- Create a Struts Web connection.

Start the Web Diagram editor

To launch the Web Diagram editor, do the following:

1. Open the Web perspective Project Explorer view.
2. Expand **Dynamic Web Projects** → **BankStrutsWeb**.
3. Double-click **Web Diagram** from the Project Explorer view.

   The Web Designer should be open, as seen in Figure 12-9 on page 633.

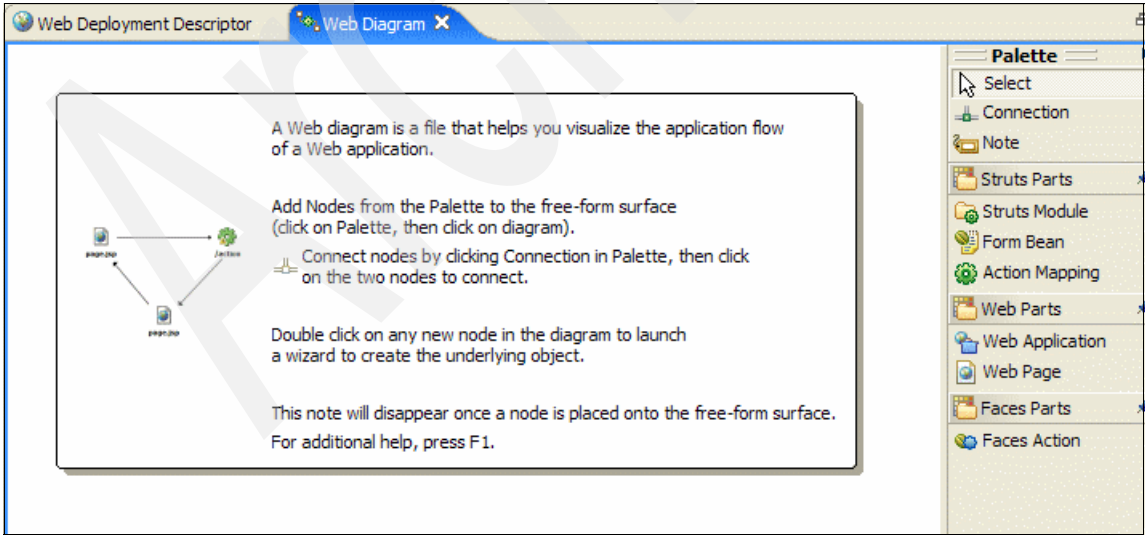


Figure 12-9 Web Diagram with the Struts Drawer open in the Palette

Archived

Create a Struts Action

To create the Struts Action for logon, do the following:

1. From the Palette, expand **Struts Parts**.
2. Drag and drop the Action Mapping icon (Action Mapping) from the palette.
3. Change the name of the action to `/logon`, as seen in Figure 12-10 on page 634.

   The logon component appears in black and white. This is because the component has not yet been realized. Once the action is realized it will be displayed in color.

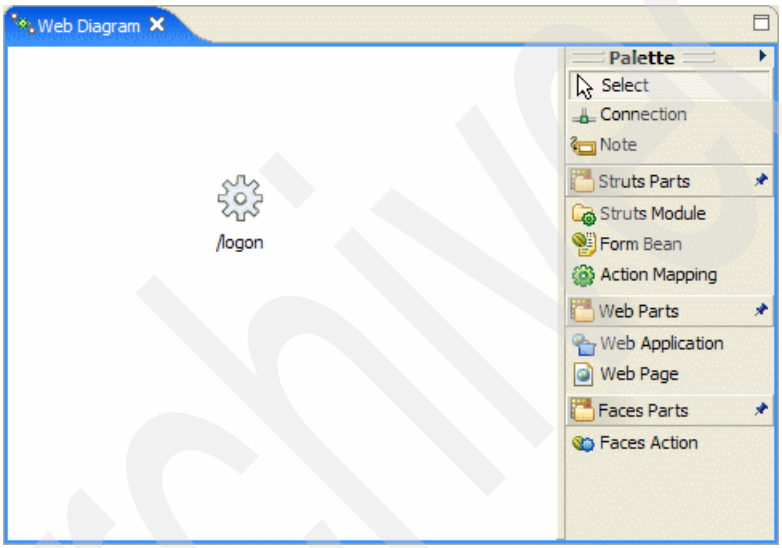


Figure 12-10 Struts Components - Create Struts action

Create a Struts Form Bean

In the previous section we created the logon action. The logon action is invoked when the customer enters a customer ID (social security number). This information is passed to the action as a Struts Form bean.

To create and associate the logonForm form bean to the logon action, do the following:

1. From the Struts Parts drawer in the Palette, drag and drop the Form Bean icon (Form Bean) from the Palette to the editor.
2. When the Form Bean Attributes dialog appears, do the following and then click **OK**:
   - Form Bean Name: `logonForm`

– Form Bean Scope: select **request**

You can choose to store the form bean in either the request or session scope. We choose to store logonForm in the request scope in our example.

3. Associate the Logon action to the LogonForm form bean.
   a. To create a connection, select the **Connection** icon (Connection) from the Palette.
   b. Single click the **logon** action, and drag and drop the connection on logonForm.

   The Web diagram should now look like Figure 12-11 on page 635.

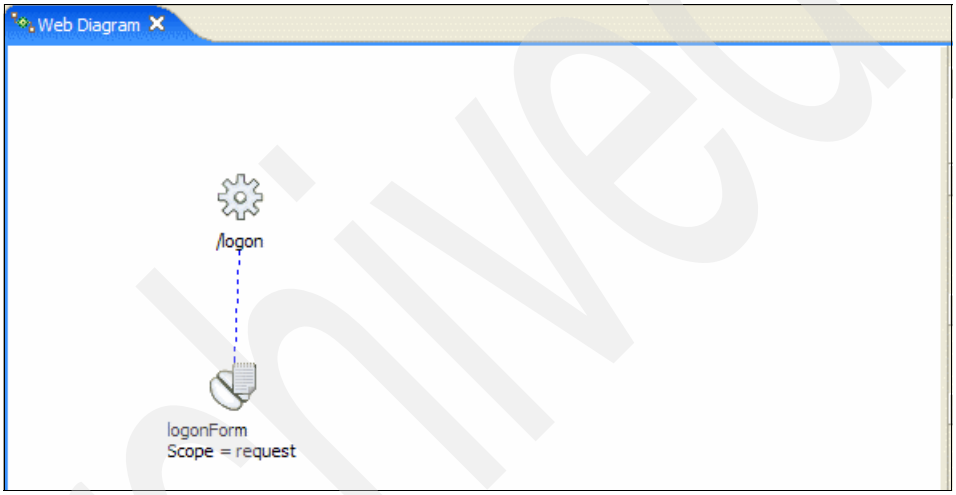


Figure 12-11 Struts Components - Creating Struts Form Bean

Again notice that the diagram is in black and white. This is because none of the components have been realized. We discuss realization and more about realizing the Web components in 12.3.2, “Realize the Struts components” on page 640.

Create a Web Page

Thus far, we have created the logon action and the logonForm form bean. We will create the input and output pages to the logon action. The input page (logon.jsp) lets the customer enter a customer ID (SSN) through the input form. The form action is mapped to the logon action. The form data is passed to the action through the logonForm logon form bean.

Archived

To create the logon.jsp and customerListing.jsp Web pages, do the following:

1. From the Web Parts drawer of the Palette, drag and drop the **Web Page** icon (Web Page) into the Web diagram.
2. Enter `logon.jsp` as the page name.
3. Repeat the steps to create the customerListing.jsp.

   The Web diagram should now look like Figure 12-12 on page 636.

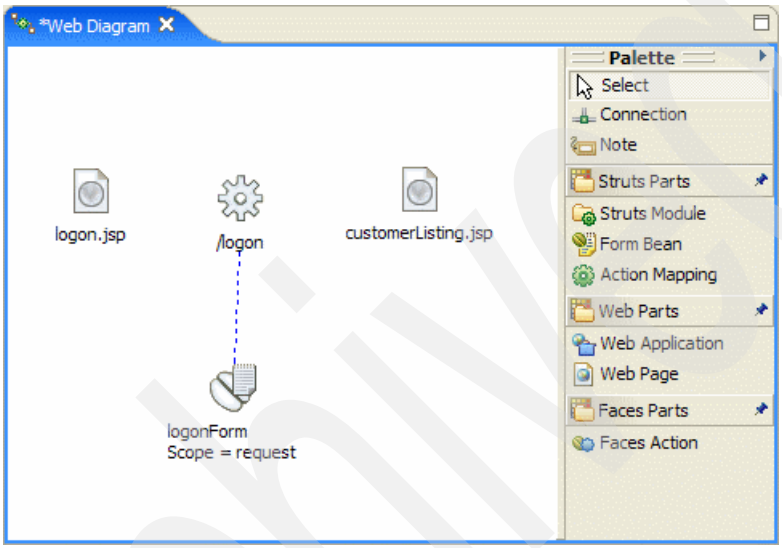


Figure 12-12 Struts components - Create Web pages

Create a Struts Web connection

A connection is typically used to connect two nodes in a Web diagram. In a Struts context, when a connection is dragged from a Struts action, a pop-up connection wizard is displayed, enabling the user to create the connection. When any of these connections are realized, the corresponding Struts action mapping entry in the Struts configuration file struts-config.xml is modified appropriately. For now we will create the connection, and realize all the components we have created thus far in 12.3.2, “Realize the Struts components” on page 640.

When you select **Connection** from the palette and drag it from a Struts Action to any other node, you will be able to create the following:

- Local Exception: When **Local Exception** is selected, the Action Mapping entry is modified in the struts-config.xml file. The handler class created during the creation is invoked when the Struts action throws the local exception.

- Action Input: When the Struts validation framework is used, the action needs to forward the control back to the page that invoked the action in case of validation failures. This is specified by specifying an action input. The action mapping entry is modified in the struts-config.xml file.
- Include Action Mapping: The action in the Action Mapping entry in struts-config.xml is configured as a include.
- Forward Action Mapping: The action in the Action Mapping entry in struts-config.xml is configured as a forward.
- Global Forward: When Global Forward is selected, a global forward entry is added in the struts-config.xml configuration file.
- Local Forward: When Local Forward is selected, a local forward entry is added within the corresponding *action mapping* entry instead of globally in the struts-config.xml configuration file.
- Global Exception: When Global Exception is selected, a global exception entry is added in the struts-config.xml configuration file.

In our sample, when a user enters an invalid customer ID (SSN), the *logon* action fails and then forwards the user back to the *logon* page to enable the user to re-enter this information. Likewise, if the *logon* action succeeds, the customer has to be forwarded to the customerListing page that displays the customer's account information.

To create the local forwards for success and failures for the logon action, do the following:

1. Create a connection from logon.jsp to the logon action.

   This is done to indicate that the action that will be invoked when the form is submitted will be logon.

   Select **Connection** from the Palette, click **logon.jsp**, and drag the connection to the logon action.

2. Create an Action Input.

   The Action Input will be used by the Struts validation framework for validating the form data in logon.jsp to redirect the user back to the logon.jsp page.

   Select **Connection** from the Palette, click the **logon** action, and drag it back to logon.jsp.

3. When the Choose a connection dialog appears, expand **Action Input** and select **<new>** (as seen in Figure 12-13), and then click **OK**.

Archived

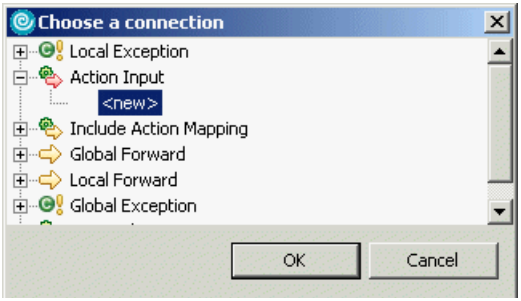


Figure 12-13 Struts Components - Creating a connection

You will now see a dotted red arrow from the logon action to the logon.jsp page, as shown in Figure 12-14 on page 638.

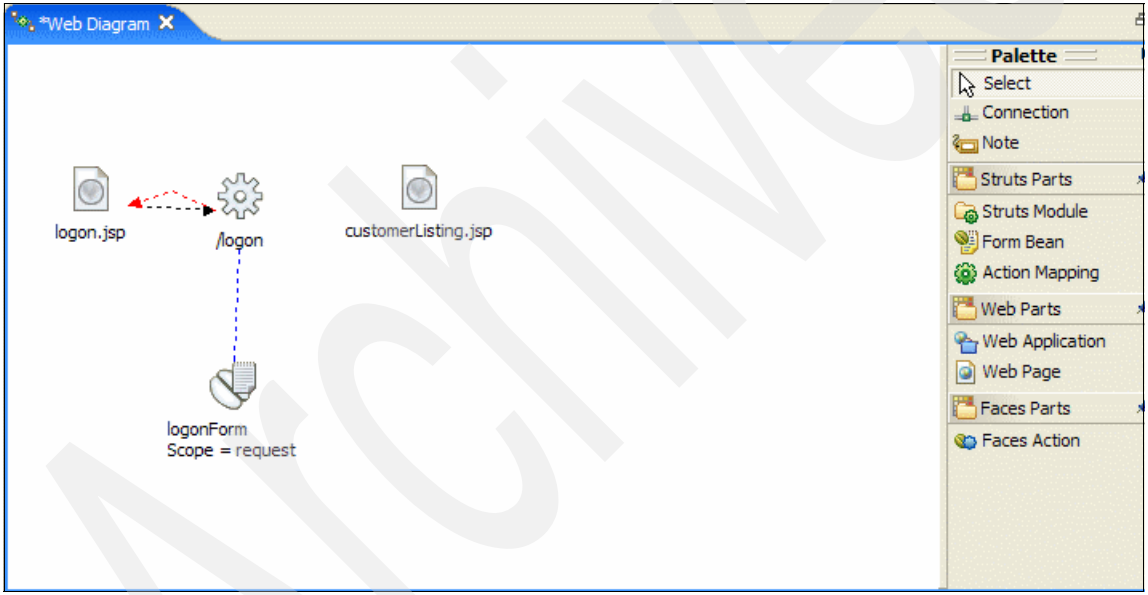


Figure 12-14 Struts Components - Creating Struts Action Input

4. Create a local forward back to logon.jsp.

   This will be used to forward the user back to the logon page when business exceptions occur in the logon action.

   a. Select **Connection**, click the **logon** action, and drag it back to logon.jsp.

   b. When the Choose a connection dialog appears, expand **Local Forward**, select **<new>**, and then click **OK**.

   c. Rename the forward to `failure`.

Archived

You will now see a dotted arrow from the logon action to the logon.jsp page.

5. Create a local forward to the customerListing.jsp.

   a. Select **Connection**, click the **logon** action, and drag it back to customerListing.jsp.

   b. When the Choose a connection dialog appears, expand **Local Forward**, select **<new>**, and then click **OK**.

   c. Rename the forward to `success`.

   You will now see a dotted arrow line from the Struts action to the Web page, as shown in Figure 12-15 on page 639.

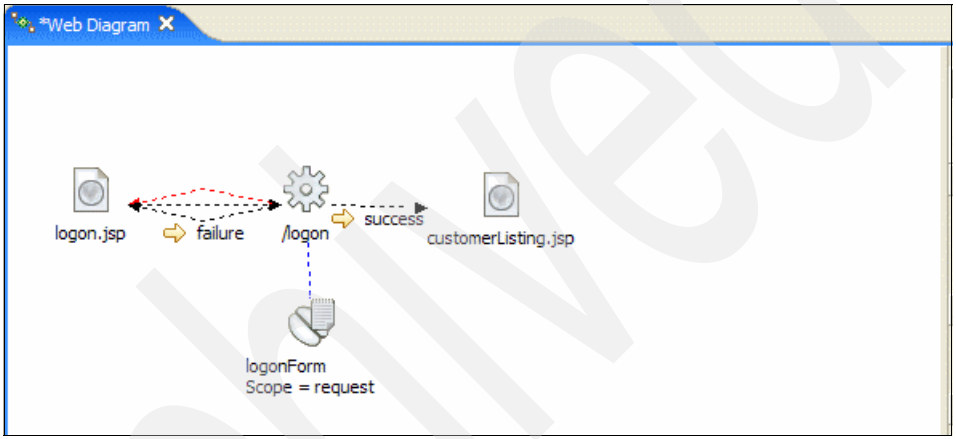


Figure 12-15 Struts components - Creating local action forwards

6. Save the Web diagram (Ctrl+S).

Tip: You can select a component in the Web diagram, right-click, and from the context menu select:

- **Change Path** to change the name of a JSP or action
- **Edit the forward name** to change the name of a connection
- **Change Description** to add descriptive text to any components

To improve the layout of the application flow, you can drag components to another spot and you can rearrange connections by dragging their middle point.

Archived

12.3.2 Realize the Struts components

Up to this point, the components (action, form bean, Web pages, local forwards) displayed on the Web diagram (see Figure 12-15) are black and white. This shows that the components have not been *realized*.

To realize a *resource* in a Web diagram means to tie the node/connection in the Web diagram to its underlying resource and bring the underlying resource into existence.

To realize the *component*, you can either double-click the component to invoke the appropriate wizard or you can change the path of the component to an existing resource. Table 12-1 on page 640 shows the action that occurs when double-clicking the component.

Table 12-1 Struts components realization result

Object	Resulting action when realized
Struts Form Beans	If the form bean does not exist, by default the New ActionForm Class wizard opens. If the form bean exists but is not defined in a configuration file, a configuration dialog box opens, and the dialog box lists all the Struts configuration files that are defined in the web.xml file for the Struts application. The Struts configuration file editor is opened on the chosen configuration file, and a new form bean entry is generated and selected. This behavior can be changed by setting the Web diagram preferences.
Struts Actions	If the action mapping does not exist, the Action Mapping wizard opens. If the action mapping exists but is not configured, a configuration dialog box opens. The former case is the default. In the latter case the dialog box lists all the Struts configuration files that are defined for the Struts application (via the web.xml file). The Struts configuration file editor opens, and a new action mapping entry is generated and selected. This behavior can be changed by setting the Web diagram preferences.

Archive

Object	Resulting action when realized
Local/Global Forwards, Exceptions, Action Input Connections	When a Struts connection is realized, the Struts configuration file struts-config.xml is updated appropriately based on the type of connection being realized.
JSP	The JSP Wizard opens.

Note: Do not select a template from sample page templates, because this produces a JSP file with the Struts tags removed. Instead of samples, select a user-defined page template that you created. To create a page template, create a Struts JSP file and save it as page template from the menu (**Page Designer's File** → **Save As Page Template**).

Realize a Struts form bean

To realize a Struts form bean named logonForm, do the following:

1. Double-click the **logonForm** form bean in the Web diagram.
2. The New Form Bean dialog should be displayed with all the fields populated by default. Ensure that the **Create New ActionForm class or Struts dynamform using DynaActionForm** radio button is selected, and **Generic Form-Bean Mapping** is selected in the Model field. Accept the default, as shown in Figure 12-16, and click **Next**.

Archived

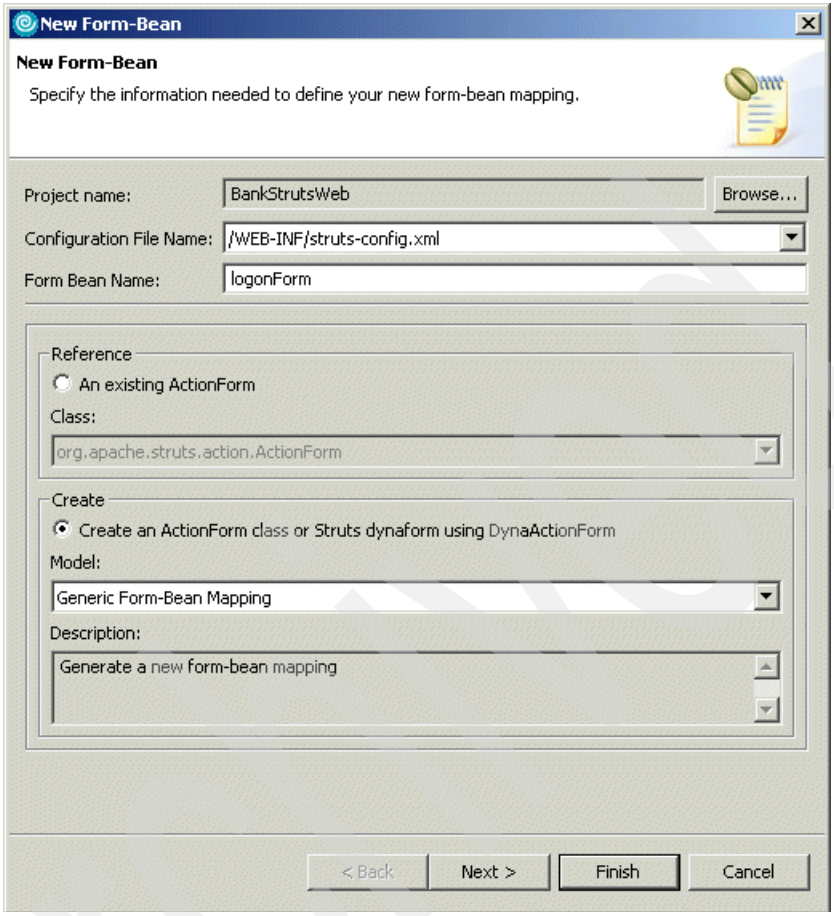


Figure 12-16 Realize Struts components - Logon Form Bean - Bean information

3. When the Choose New Fields dialog appears, we could choose an existing Form in a HTML/JSP file and add it to the form bean directly, but since we have not yet realized the logon.jsp, we skip this and create the fields directly in the bean. Click **Next**.
4. When the Create New Fields for your ActionForm class dialog appears, do the following:
   a. Click **Add**.
   b. Enter `ssn` in the new field.
   c. When complete, the dialog should look like Figure 12-17. Click **Next** to continue.

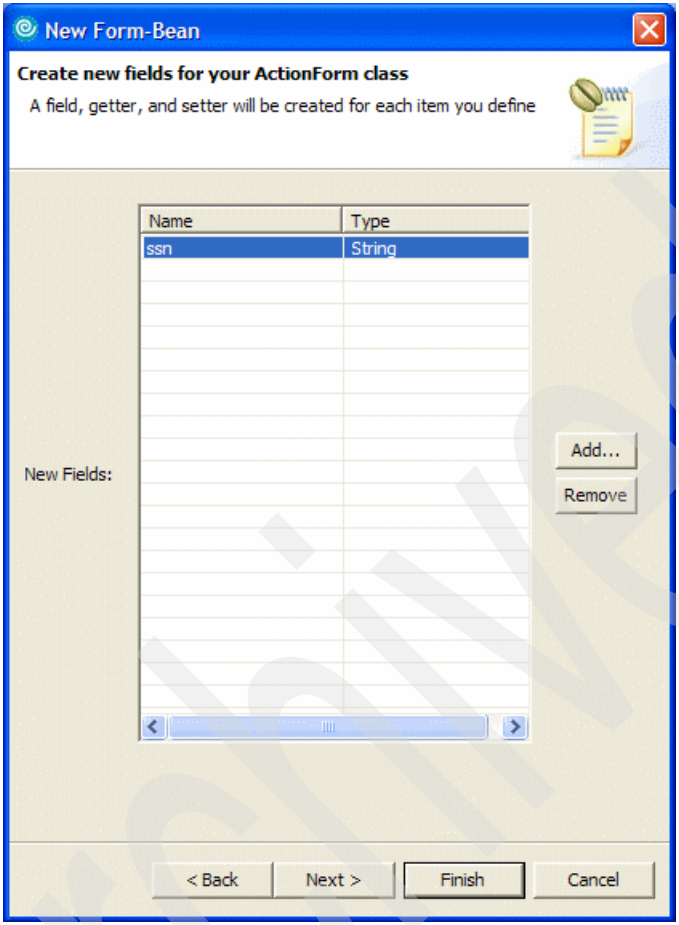


Figure 12-17 Realize Struts components - Logon Form Bean - Create new fields

5. When the Create Mapping for the ActionForm class dialog appears, enter the following (as seen in Figure 12-18 on page 644) and then click **Finish**:
   - Java package: `itso.bank.model.struts.form`
   - ActionForm class name: `org.apache.struts.validator.ValidatorForm`
   - Method stubs: Uncheck **validate** and **reset**.

Archived

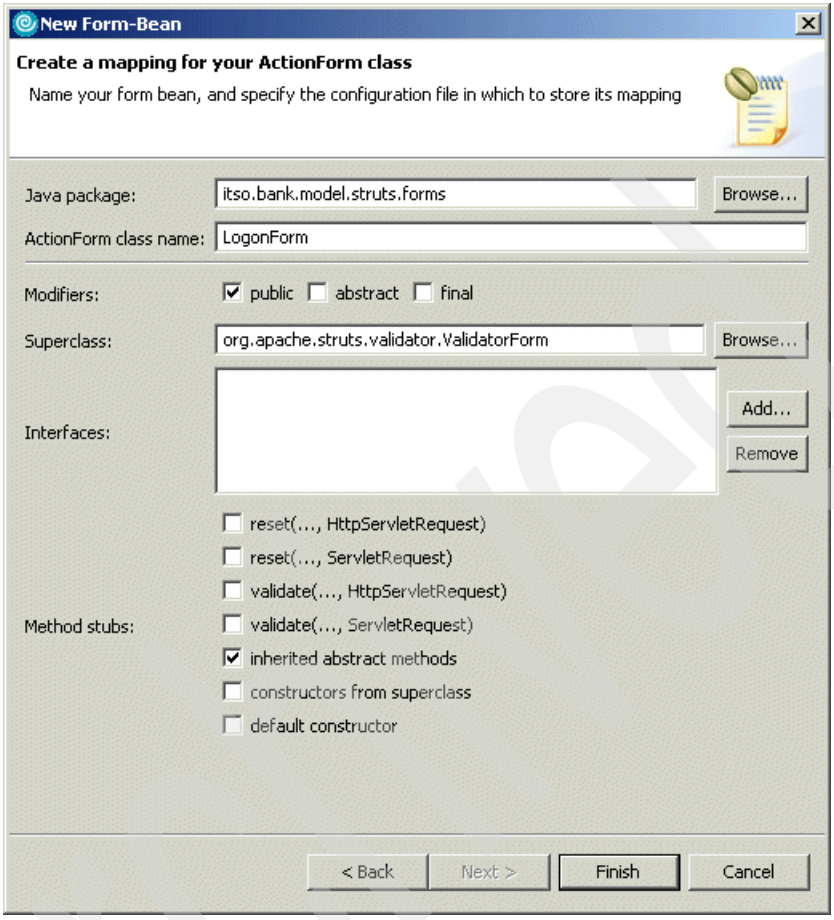


Figure 12-18 Create a mapping for your ActionForm class

6. Save the Web diagram.

The following actions have been completed by the wizard:

- A class LogonForm has been created in the package specified in step 1.
- The Struts configuration file struts-config.xml has been updated with the form bean information, as shown in Example 12-1.

Example 12-1 Struts configuration file struts-config.xml snippet

```
<!-- Form Beans -->
<form-beans>
   <form-bean name="logonForm" type="itso.bank.model.struts.forms.LogonForm">
   </form-bean>
```

```
</form-beans>
```

- Now that the Web diagram has been updated, notice that the form bean appears in the diagram and that the color has changed. The color change in Figure 12-19 for the logonForm denotes the Struts component has been realized.

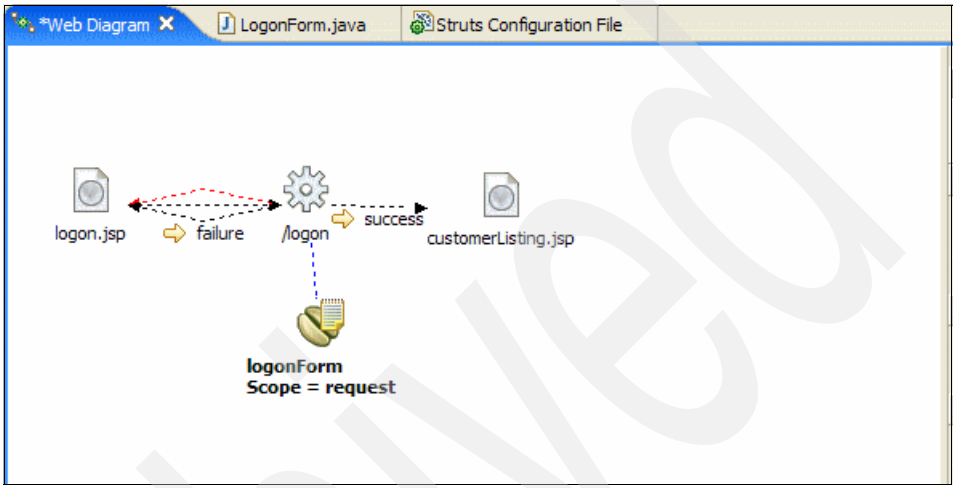


Figure 12-19 Realize Struts components - Logon form bean realized

Realize a Struts action

To realize the Struts action named logon, do the following:

1. Double-click the **logon** action in the Web diagram.
2. When the New Action Mapping wizard appears, the fields should already be filled in by the wizard, as seen in Figure 12-20 on page 646. Click **Next**.

Archived

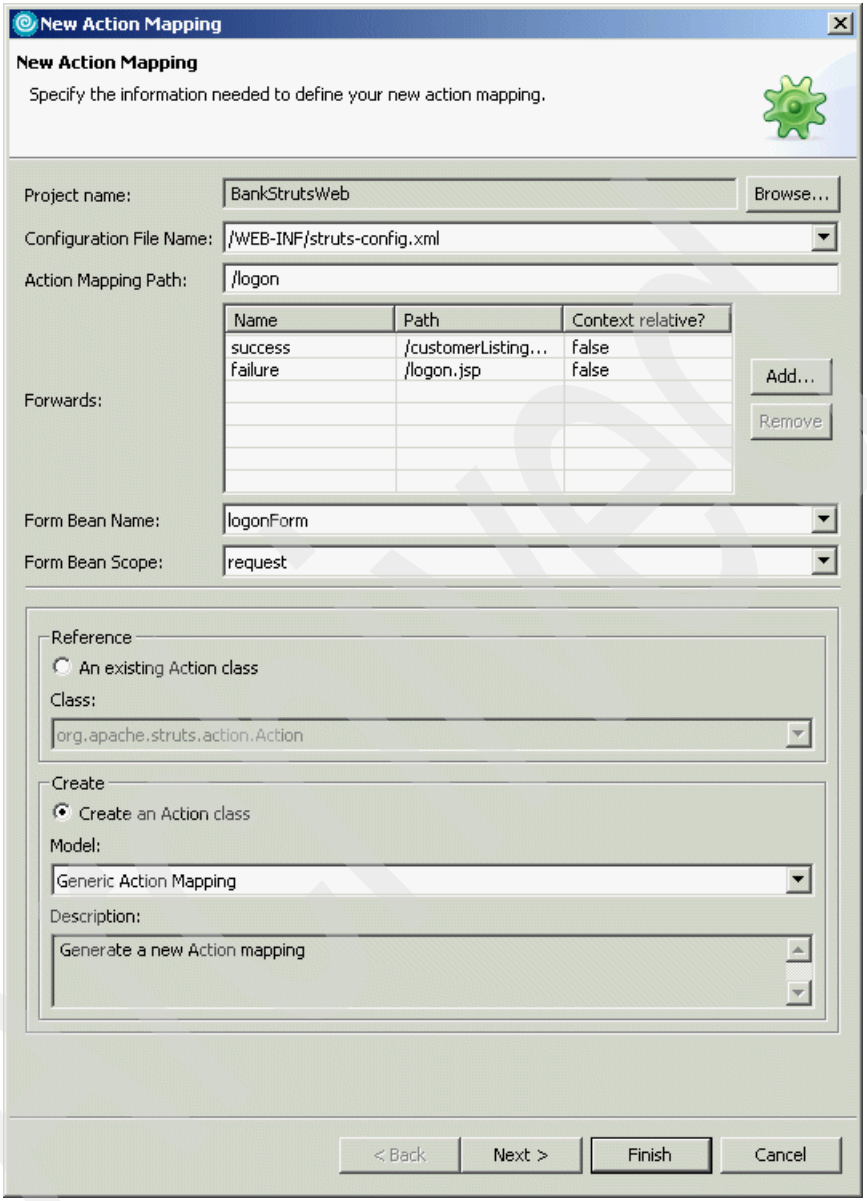


Figure 12-20 New Action Mapping for logon action

3. When the Create an Action class for your mapping dialog appears, we entered the values seen in Figure 12-21 on page 647 and then clicked **Finish**.

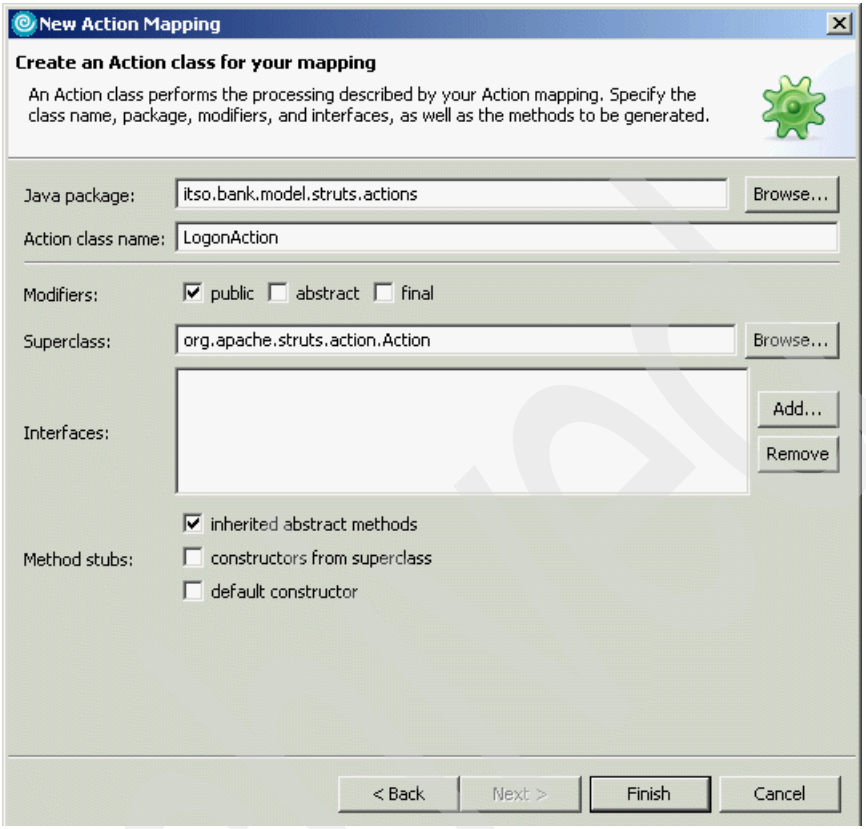


Figure 12-21 Realize Struts components - Logon action - Create action class

The following actions have been completed by the wizard:

- A class LogonAction has been created in the package specified in step 1.
- The Struts configuration file struts-config.xml has been updated with the LogonAction Action Mapping information, as shown in Example 12-2.

Example 12-2 struts-config.xml snippet

```
<!-- Action Mappings -->
<action-mappings>
   <action name="logonForm" path="/logon" scope="request"
type="itso.bank.model.struts.actions.LogonAction">
      <forward name="failure" path="/logon.jsp">
      </forward>
      <forward name="success" path="/customerListing.jsp">
      </forward>
   </action>
```

```
</action-mappings>
```

- The Web diagram has been updated, and now the *Logon Action* and the local forwards *failure* and *success* appear in color to indicate they have been realized, as shown in Figure 12-22.

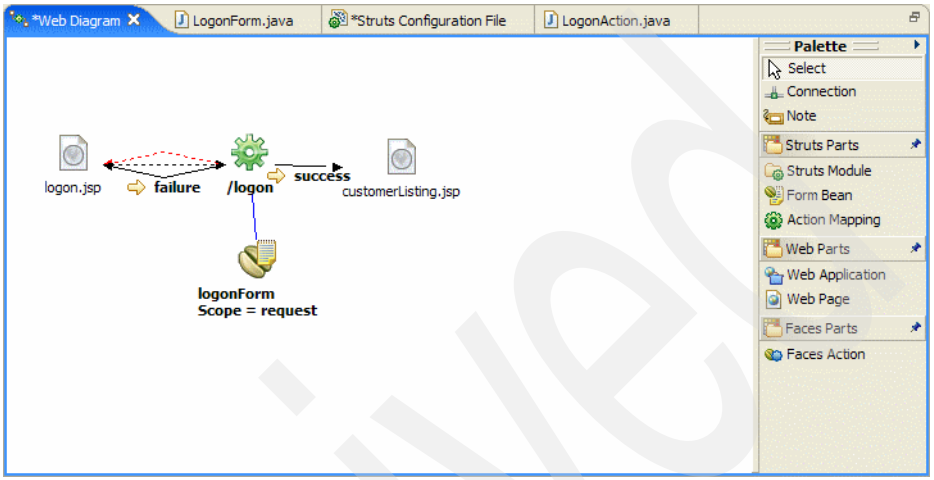


Figure 12-22 Realize Struts components - Logon action realized

Realize a JSP

Realizing a JSP and other Web components is described in detail in Chapter 11, “Develop Web applications using JSPs and servlets” on page 499. Here we just indicate briefly how to realize the logon.jsp and customerListing jsp.

1. Realize the logon.jsp.
   a. Double-click the **logon.jsp** in the Web diagram.
   b. When the New JSP File wizard appears, the wizard will provide the default values, which we chose to accept. Ensure that **Struts JSP** is selected in the Model field, and **Configure advanced options** is checked. Click **Next**.

Archived

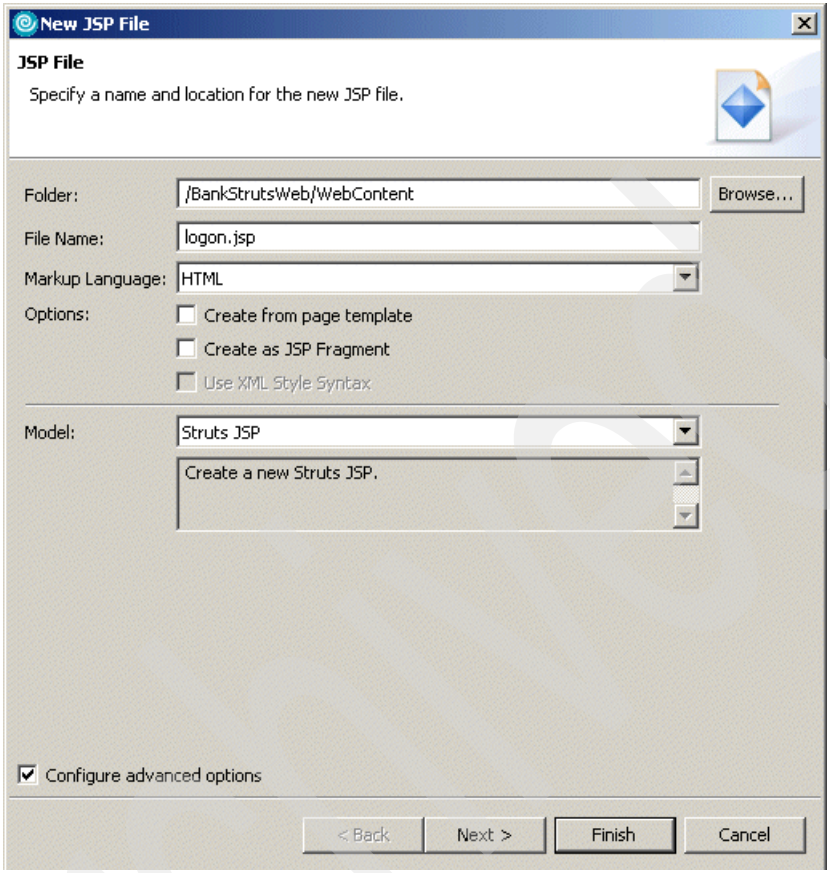


Figure 12-23 Realize Struts components - Logon JSP - JSP File

c. When the Tag Libraries dialog appears, accept the defaults of adding support for the Struts bean and html tag libraries. You can add more tag libraries if needed here. Click **Next**.

d. When the JSP File Options dialog appears, accept the defaults for HTML settings and click **Next**.

e. When the JSP File Choose Method Stubs to generate dialog appears, accept the default settings and click **Next**.

f. When the Form Field Selection dialog appears, the wizard fills in the Struts Form Bean and Struts Action information by default based on the Web diagram. Check the **ssn** field and then click **Finish**.

2. Realize the customerListing.jsp.

   a. Double-click the **customerListing.jsp** in the Web diagram.

b. When the New JSP File wizard appears, the wizard will provide the default values, which we chose to accept. Ensure that **Struts JSP** is selected in the Model field, and **Configure advanced options** is checked. Click **Finish**.

The logon.jsp and customerListing.jsp pages are now created, as is the Web diagram, as shown in Figure 12-24.

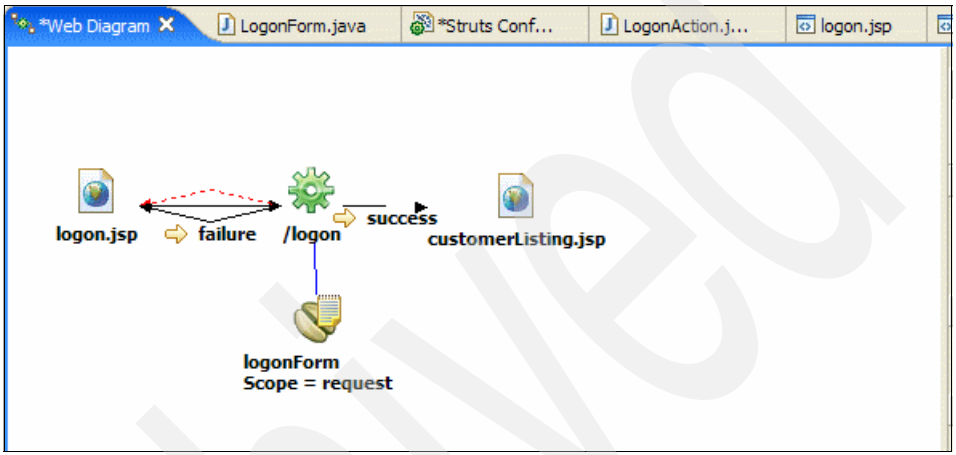


Figure 12-24 Realize Struts components - Realizing JSP pages

Realize the Struts Web connections

When connections are not realized, the Web diagram displays dashed lined arrows. Realized connections are indicated by solid arrows instead of dashed arrows.

To realize connections we follow the same procedure of double-clicking the arrow to bring the wizard up. In our example, notice that most of the arrows in Figure 12-24 have been realized automatically by the wizard when the action, form beans, and JSP pages were realized.

We will demonstrate realization of Web connections using the *Action Input* in our sample that we still have to realize for the *logon* action.

Double-click the red dashed arrow that goes from the *logon Action* to *logon.jsp*.

The wizard automatically adds the necessary *input attribute* to the *logon* action mapping, as shown in bold in Example 12-3.

Example 12-3 struts-config.xml snippet - Logon action mapping

```
<!-- Action Mappings -->
```

```
<action-mappings>
   <action name="logonForm" path="/logon" scope="request"
type="itso.bank.model.struts.actions.LogonAction" input="/logon.jsp">
      <forward name="failure" path="/logon.jsp">
      </forward>
      <forward name="success" path="/customerListing.jsp">
      </forward>
   </action>
</action-mappings>
```

The Web diagram, after all the realizations have been completed, is shown in Figure 12-25.

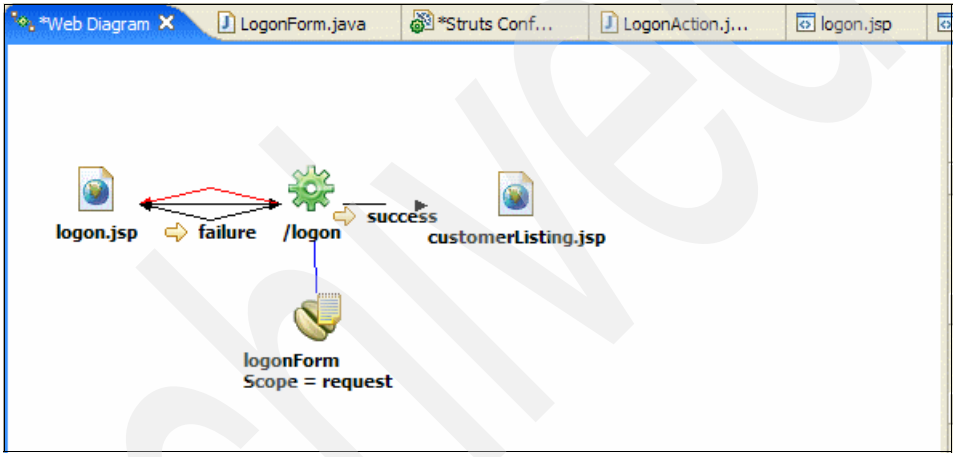


Figure 12-25 Realize Struts components - Realizing connections

Note: If deleting a component from the Web diagram, you are prompted whether you also want to delete the underlying resource. Underlying resource here refers to the mapping in the struts-config.xml file, not the implemented component itself.

This means that if you delete a form bean and also delete the underlying resource, it will remove the form bean from the Web diagram and its mapping from the struts-config.xml file. It will not, however, delete the Java source or class file for the form bean. If you then add a new form bean with the same name to the Web diagram and attempt to implement it, the Finish button that you must click to create the component is deactivated in the wizard. This is due to the fact that this class already exists—it was not deleted.

Archived

12.3.3 Modify ApplicationResources.properties

The wizard created an empty ApplicationResource.properties file for us and we have to update it with the texts and messages for our application.

While developing Struts applications, you will usually find yourself having this file open, because you will typically add messages to it as you go along writing your code. Example 12-4 shows a snippet of the ApplicationResources.properties file.

Example 12-4 ApplicationResources.properties snippet

```
# Optional header and footer for <errors/> tag.
errors.header=<ul>
errors.footer=</ul>
errors.prefix=<li>
errors.suffix=</li>

form.ssn=SSN
form.accountId=Account Id
form.balance=Balance
form.amount=Amount

...

errors.required={0} is a required Field
error.ssn=Verify that the customer ssn entered is correct.
error.amount=Verify that the amount entered is valid.
error.timeout=Your session has timed out. Please login again.
errors.systemError=The system is currently unavailable. Please try again later.
```

To modify the ApplicationResources.properties file, do the following:

1. Open the Web perspective Project Explorer view.
2. Expand **Dynamic Web Project** → **BankStrutsWeb** → **Java Resources** → **JavaSource** → **itso.bank.model.struts.resources**.
3. Double-click the **ApplicationResources.properties** file.
4. Append the contents of Example 12-5 to the end of the ApplicationResources.properties file.

Example 12-5 ApplicationResources.properties - Add lines to the bottom of the file

```
form.ssn=SSN
errors.required={0} is a required field.
```

5. Save the file (Ctrl+S) and close the editor.

Archived

We will use this message to validate, using the Struts Validation Framework, the logon form in logon.jsp to ensure that the user enters a value for the customer (SSN).

12.3.4 Struts validation framework

The Struts validation framework provides automatic validation of forms using configuration files. The validation.xml and validator-rules.xml are the two configuration files used by the Struts Validation Framework to validate forms.

Note: More information on the architecture and further documentation of Struts Validation Framework can be found at:

`http://struts.apache.org`

To validate the logonForm using the Struts validation framework, do the following:

1. We have provided a validation.xml and validation-rules.xml as part of the sample code.

   Import the validation.xml and validator-rules.xml from the c:\6449code\struts directory into the BankStrutsWeb\WebContent\WEB-INF directory of the workspace.

2. Add the Struts validator plug-in and required property to the plug-in indicating the location of the validation configuration files.

   a. Expand **Dynamic Web Projects** → **BankStrutsWeb** → **WebContent** → **WEB-INF**.

   b. Double-click the **struts-config.xml** file to open in the Struts Configuration Editor.

   c. Click the **Plug-ins** tab in the Struts Configuration Editor.

   d. Click **Add...** in the Plug-ins field and select the **ValidatorPlugIn** in the Class Selection Wizard. Click **OK** to close the Class Selection Wizard. The Struts Validator Plug-in has now been added.

   e. Add the required parameter by clicking **Add** in the Plug-in Mapping Extension field.

   f. In the Property and Value fields, enter in `pathnames` and `/WEB-INF/validator-rules.xml,/WEB-INF/validation.xml`, respectively, as seen in Figure 12-26 on page 654.

Archive

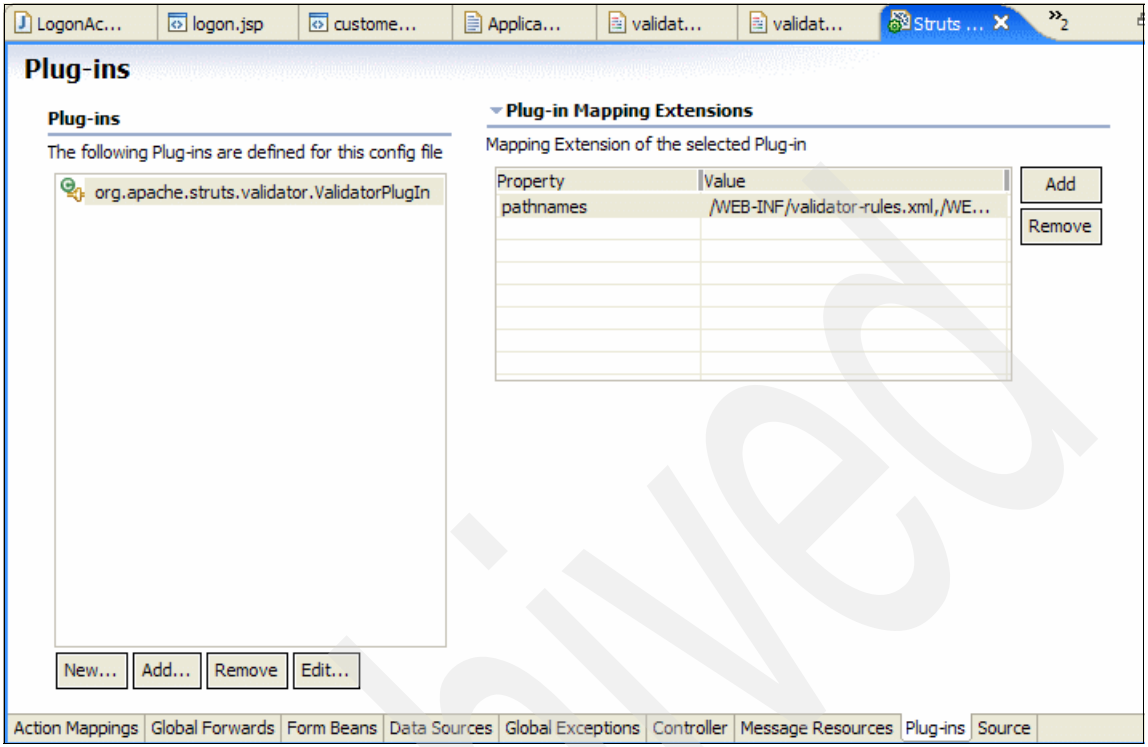


Figure 12-26 Struts Configuration Editor - Adding the validator plug-in

3. Save the configuration file and close the Struts configuration file.

The validation.xml file contains all the Struts form beans and the fields within the form bean that will be validated, and the rule that will be applied to validate the bean. The snippet that validates the logonForm is shown in Example 12-6.

Example 12-6 validation.xml snippet - LogonForm

```
<form-validation>
   <formset>
      <form name="logonForm">
         <field property="ssn" depends="required">
            <arg0 key="form.ssn" />
         </field>
      </form>
   </formset>
   .......
   .......
</form-validation>
```

Archived

The validator-rules.xml file contains the rule configurations for all the rules defined in the validation.xml file. In our example above, the rule that is defined is *required* for the field ssn, as shown in Example 12-6 on page 654. The snippet for the *required* rule is shown in Example 12-7.

Example 12-7 validator-rules.xml snippet - Rule configuration for the required rule

```
<form-validation>
   <global>
      <validator name="required"
         classname="org.apache.struts.validator.FieldChecks"
         method="validateRequired"
         methodParams="java.lang.Object,
org.apache.commons.validator.ValidatorAction,
org.apache.commons.validator.Field, org.apache.struts.action.ActionErrors,
javax.servlet.http.HttpServletRequest"
         msg="errors.required" />
   </global>
</form-validation>
```

12.3.5 Page Designer and the Struts tag library

The Struts framework provides a tag library to help in the development of Struts-based Web applications. The *Page Designer* is used to design and develop HTML and dynamic Web pages within Rational Application Developer. The features of the Page Designer are explained in detail in Chapter 11, “Develop Web applications using JSPs and servlets” on page 499.

Stuts tag library overview

In this section we explore the support for the Struts tag libraries within the Page Designer. The Page Designer supports the Struts tag libraries by allowing dropping of tags from the Page Designer palette and into the design view of the Page Designer.

- Struts-html tags: Struts provides tags to render html content. Examples of the Struts html tags are button, cancel, checkbox, form, errors, etc. These tags can be dragged and dropped into a page in the page designer’s design perspective. We will drag and drop the errors tag to the logon.jsp and display a message to the user if no ssn is entered later on in this section. The Struts-html Palette is displayed in Figure 12-27 on page 656.
- Struts-bean tags: The bean tag library provides tags to access bean properties, request parameters, create page attributes, etc. The Struts-bean Palette is displayed in Figure 12-27 on page 656.

Archived

- Struts-Logic tags: The Logic tag library provides tags to implement conditional, looping, and control related functionality. The Struts-logic Palette is displayed in Figure 12-27.
- Struts-Nested tags: The Nested tag library provides tags to access complex beans with nested structures. The Struts-nested Palette is displayed in Figure 12-27.
- Struts-Template tags: The Template tag library provides tags for creation of dynamic JSP templates. The Struts-template is displayed in Figure 12-27.
- Struts-Tiles Tags: The Tiles tag library facilitates development of dynamic Web applications in a tiled fashion where the tile can be reused throughout the application. The Tiles tag library is displayed in Figure 12-27.

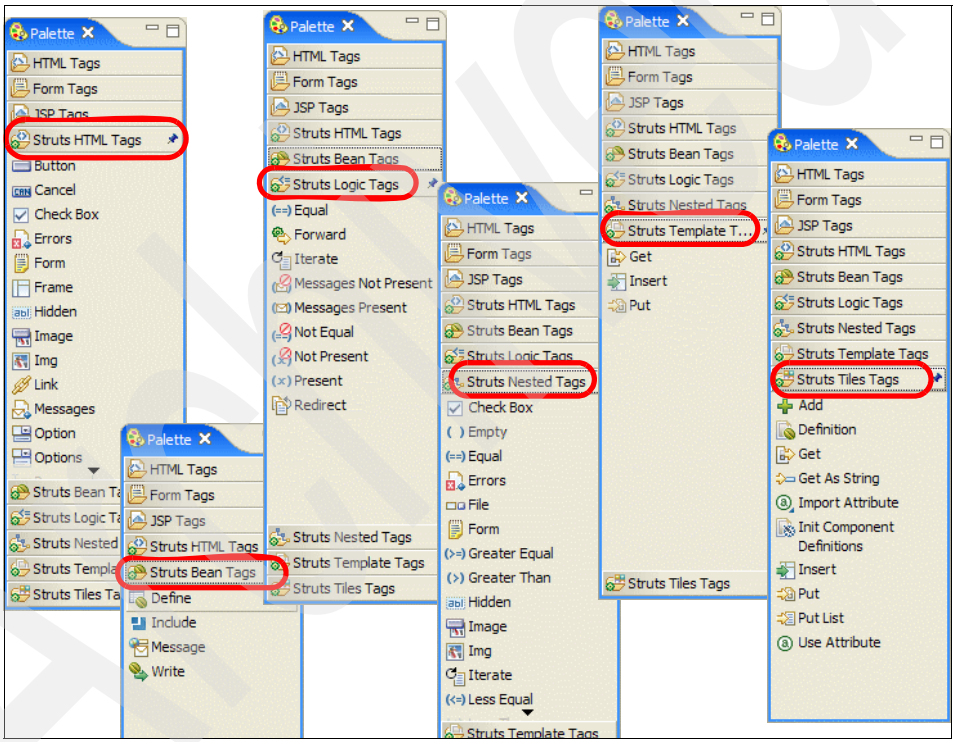


Figure 12-27 Struts tag library support - Web Page Designer Struts tag drawers

Tip: If you do not see all the Struts tag libraries in the palette, right-click the palette and select **Customize**. From the Customize Palette dialog box, select all drawers you want to be available in the palette.

Add html error tag to logon.jsp

We will now add a simple html error tag to logon.jsp, which will display an HTML error message that occurs due to the validation check by the Struts Validation Framework. To add the html error tag to logon.jsp, do the following:

1. Open the Web perspective Project Explorer view.
2. Double-click **logon.jsp** to open in Page Designer.
3. Click the **Design** tab of the Page Designer.
4. Insert a space before the HTML form, where the Struts html error tag will be placed. The logon.jsp is shown in Figure 12-28.

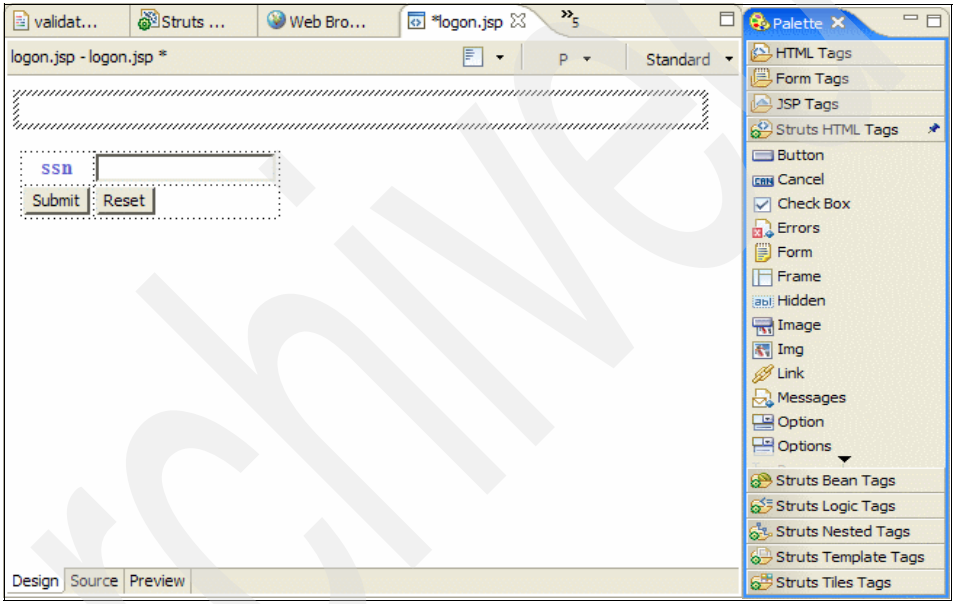


Figure 12-28 Struts tags - Adding a Struts tag

5. Click the **Errors** icon (Errors) from the Struts HTML Tags drawer and drop it in the newly created space before the HTML form.

   The Page Designer renders the html for the tag in the design view as shown in Figure 12-29 on page 658.

Archived

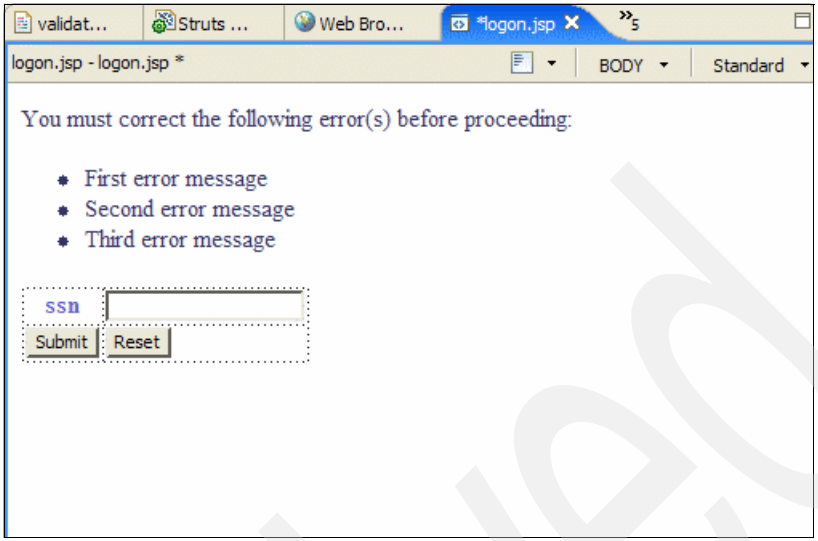


Figure 12-29 Struts tags - Struts html tag rendered by the Page Designer

6. The source for the rendered logon.jsp is shown in the Example 12-8. The html:error tag has been added. Notice also that the wizard has added the Struts html tags for rendering the form itself.

Example 12-8 Struts tags - Logon.jsp snippet of the tag in the source view

```
......
<P><html:errors /></P>
<html:form action="/logon">
   <TABLE border="0">
      <TBODY>
         <TR>
            <TH>ssn</TH>
            <TD><html:text property="ssn" /></TD>
         </TR>
         <TR>
            <TD><html:submit property="submit" value="Submit" /></TD>
            <TD><html:reset /></TD>
         </TR>
      </TBODY>
   </TABLE>
</html:form>
......
```

Archived

12.3.6 Using the Struts configuration file editor

Rational Application Developer provides an editor for the Struts struts-config.xml configuration file. This editor is yet another way you can add new form beans and actions and customize their attributes. You can also directly edit the XML source file should you prefer to do it by hand instead of using the wizards. The Struts Configuration Editor is shown in Figure 12-30.

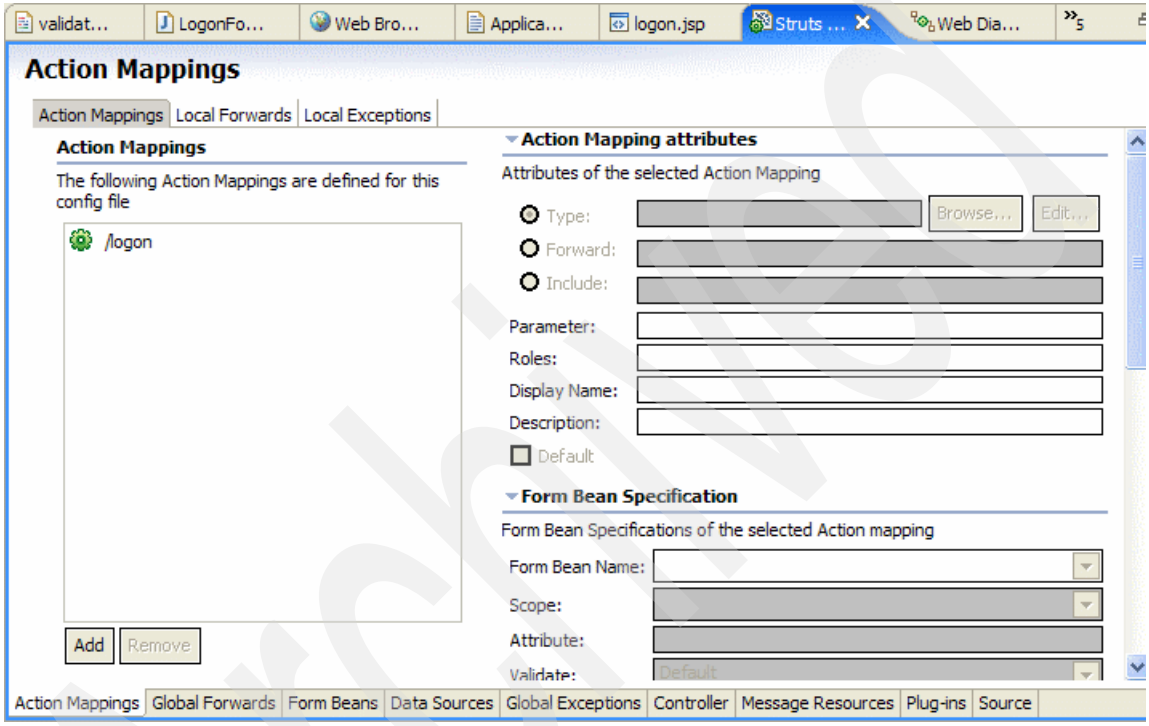


Figure 12-30 Struts Configuration Editor

We will use this editor to add a local forward called *cancel* to the *logon action*. This forward can be used by the logon action's execute method to forward the user to the logon.jsp page, as we will map this forward to the logon.jsp page.

We also specify the input attribute for all our actions. This attribute specifies which page should be displayed if the validate method of a form bean or the Struts validation framework fails to validate. Usually you want to display the input page where the user entered the data so they can correct their entry.

1. Expand **Dynamic Web Projects** → **BankStrutsWeb** → **WebContent** → **WEB-INF**.

2. Double-click the **struts-config.xml** file to open in the Struts Configuration Editor.

   The editor has tabs at the bottom of the screen to navigate between the different Struts artifacts (actions, form beans, global forwards, data sources) it supports.

3. Click the **Action Mappings** tab.

4. Select the **/logon** action. In the Input field enter `/logon.jsp` if it is not already specified, as shown in Figure Figure 12-31.

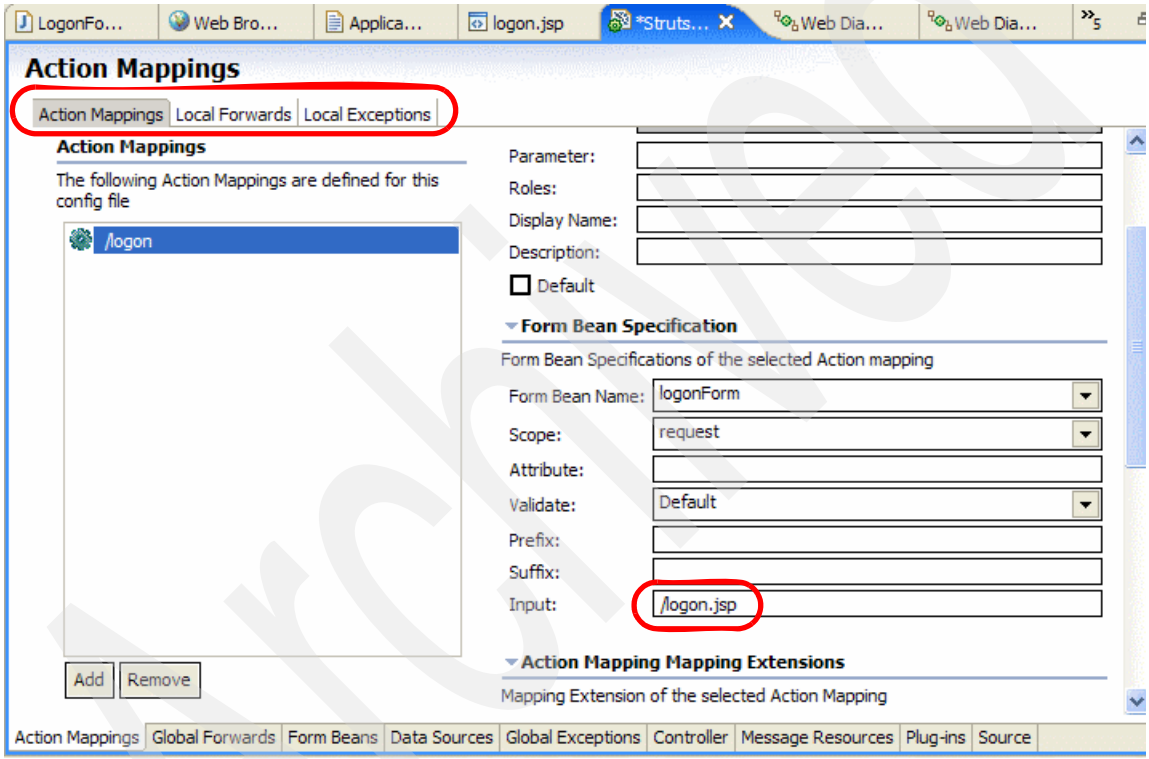


Figure 12-31 Struts Configuration Editor - Action input attribute

5. Create a forward.

   a. Click the **Local Forwards** tab found at the top of the Action Mappings page.

   b. Click **Add** to specify in the *Local Forwards* module. A new forward with the name *no name* is created.

   c. Rename *no name* to `cancel` as the name and enter `/logon.jsp` in the Path field in the Forward Attributes, as shown in Figure 12-32 on page 661.

Note: The Redirect check box allows you to select if a redirect or forward call should be made. A forward call keeps the same request with all attributes it contains and just passes control over to the path specified. A redirect call tells the browser to make a new HTTP request, which creates a new request object (and you lose any attributes set in the original request).

A forward call does not change the URL in the browser's address field, as it is unaware that the server has passed control to another component. With a redirect call, however, the browser updates the URL in its address field to reflect the requested address.

You can also redirect or forward to other actions. It does not necessarily have to be a JSP.

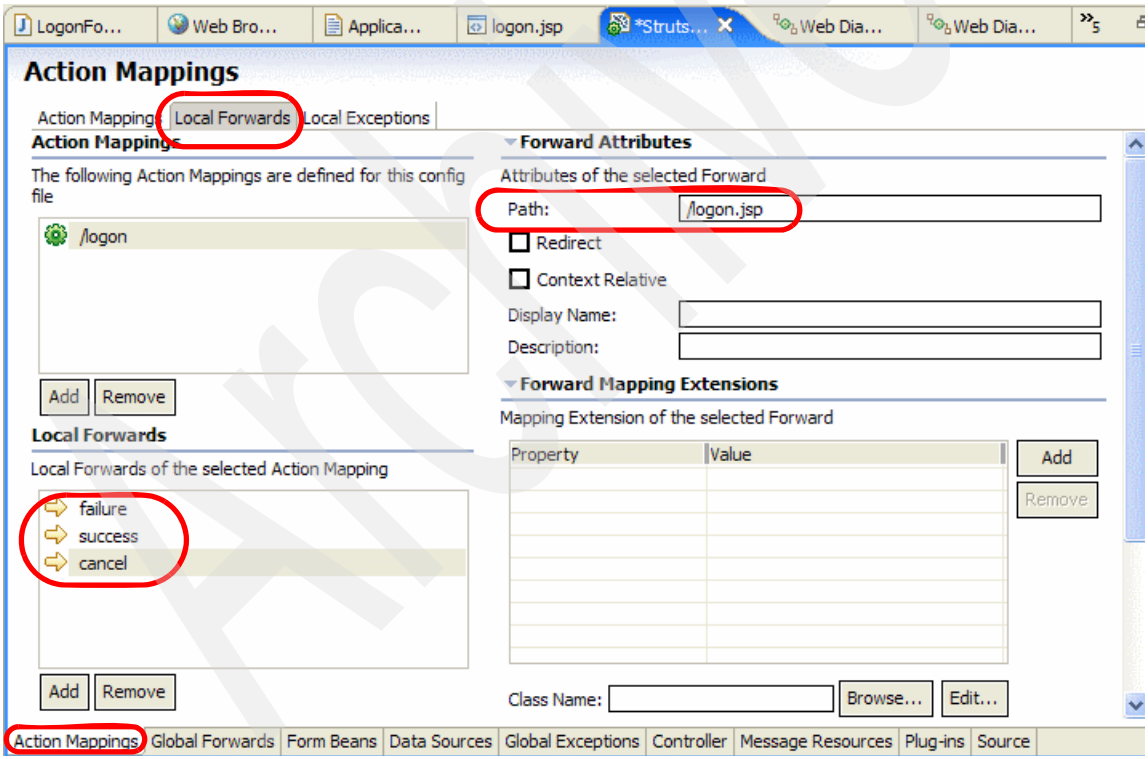


Figure 12-32 Struts Configuration Editor - Creating new forward

6. Save the file (Ctrl+S).

7. Click the **Source** tab to look at the Struts configuration file XML. Example 12-9 displays the struts-config.xml XML source.

Example 12-9 Struts configuration file - struts-config.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE struts-config PUBLIC "-//Apache Software Foundation//DTD Struts
Configuration 1.1//EN"
"http://jakarta.apache.org/struts/dtds/struts-config_1_1.dtd">
<struts-config>

   <!-- Data Sources -->
   <data-sources>
   </data-sources>

   <!-- Form Beans -->
   <form-beans>
      <form-bean name="logonForm"
type="itso.bank.model.struts.forms.LogonForm">
      </form-bean>
   </form-beans>

   <!-- Global Exceptions -->
   <global-exceptions>
   </global-exceptions>

   <!-- Global Forwards -->

   <!-- Action Mappings -->
   <action-mappings>
      <action name="logonForm" path="/logon" scope="request"
type="itso.bank.model.struts.actions.LogonAction" input="/logon.jsp">
         <forward name="failure" path="/logon.jsp">
         </forward>
         <forward name="success" path="/customerListing.jsp">
         </forward>
         <forward name="cancel" path="/logon.jsp">
         </forward>
      </action>
   </action-mappings>

   <!-- Message Resources -->
   <message-resources
parameter="itso.bank.model.struts.resources.ApplicationResources"/>
   <plug-in className="org.apache.struts.validator.ValidatorPlugIn">
      <set-property property="pathnames"
value="/WEB-INF/validator-rules.xml,/WEB-INF/validation.xml"/>
   </plug-in>
```

Archived

```
</struts-config>
```

As you can see, the Rational Application Developer Struts tools have defined the logonform bean in the <form-beans> section and the logon action in the <action-mappings> section. The JSPs, however, are not specified in the Struts configuration file. They are completely separate form the Struts framework (only the forwarding information is kept in the configuration file). At the end of the file is the name of the application resources file where our texts and error messages are stored.

The Struts configuration editor does round-trip editing, so if you edit something in the XML view it is reflected in the other views.

The forwards we use are local to each action, meaning that only the action associated with the forward can look it up. In the <global-forwards> section you can also specify global forwards that are available for all actions in the application. Normally you have a common error page to display any severe error messages that may have occurred in your application and that prevents it from continuing. Pages like these are good targets for global forwards and so are any other commonly used forward. Local forwards override global forwards.

8. You can now close the configuration file editor.
9. Update the Web diagram.

   As we have now added a forward through the Struts Configuration Editor, we should update the Web diagram to show them as well.

   a. If the Web diagram is not open, open it by double-clicking **\diagram.gph**.
   b. In the Web diagram select the **logon** action and choose **Draw** → **Draw Selected** from its context menu.
   c. Select **On Input Error** → **logon.jsp** and **cancel** → **logon.jsp** (as shown in Figure 12-33 on page 664), and click **OK**.

      This draws the red line from the logon action back to logon.jsp indicating the input page, and a new black line named *cancel* from *logon* action to logon.jsp to indicate the forward.

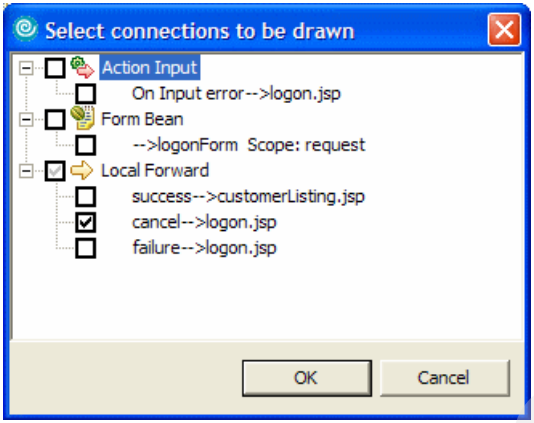


Figure 12-33 Struts Configuration Editor - Selecting Struts components to draw

10. Select the lines and drag the small dot in the middle of the line to avoid the lines from crossing over other components in the Web Diagram, if needed.

    The result is shown in Figure 12-34.

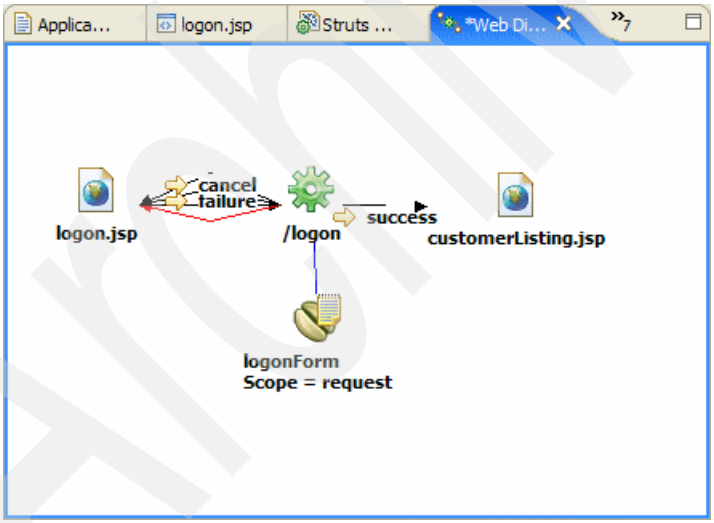


Figure 12-34 Struts Configuration Editor - Updated Web diagram

11. Save the Web diagram (Ctrl+S).

Archived

Notes: We recommend that all requests for JSPs go through an action class so that you have control over the flow and can prepare the view beans (form beans) necessary for the JSP to display properly. Struts provides simple forwarding actions that you can use to accomplish this.

In our example we do not perform any customization on the Struts action servlet (org.apache.struts.action.ActionServlet). If you have to do any initialization (for example, initialize data sources), you would want to create your own action servlet, extending the ActionServlet class, and overriding the init and destroy methods. You would then also modify the \WEB-INF\web.xml file and replace the name of the Struts action servlet with the name of your action servlet class.

12.4 Import and run the Struts sample application

In the previous sections of this chapter, we demonstrated how create the various Struts components including Struts actions, form beans, local forwards, inputs, and exceptions. We implemented a simple sequence that uses most Struts components to demonstrate the capabilities of Rational Application Developer support for Struts-based development.

This section describes how to import the completed Struts Bank Web application and run it within the test environment. You will see that the logon.jsp form validates, and if a number is entered you are forwarded to a customerListing.jsp, which just displays a message `Place Content here`, as the logon action's execute method does not do anything but forward the page using the local forward *success* created earlier. If nothing is submitted to the logon action you will notice that the Struts validation framework brings you back to the logon.jsp page.

You can continue to implement the rest of diagram based on the instructions provided in the previous sections of this chapter to create the various Struts components and complete the diagram shown in Figure 12-3 on page 621 or import the sample Web application from the source directory provide.

12.4.1 Import the Struts Bank Web application sample

We will now import the sample banking application that implements the diagram shown in Figure 12-3 on page 621.

To import the sample application, do the following:

1. Open the Web perspective Project Explorer view.

2. The imported sample project interchange file will use the project name BankStrutsWeb. If you completed the previous samples of this chapter, this project will already exist. Simply refactor the existing project by selecting the project, right-clicking, and selecting **Refactor** → **Rename**.
3. Select **File** → **Import**.
4. When the Import Wizard wizard appears, select **Project Interchange** and click **Next**.
5. In the Import Projects screen, browse to the c:\6449code\struts folder and select **BankStrutsWeb.zip**. Click **Open**.
6. Select the **BankStrutsWeb** and **BankStrutsWebEAR** projects, and then click **Finish**.

   Now a dynamic Web Project, BankStrutsWeb, that is Struts enabled and an Enterprise Application Project, BankStrutsWebEAR, are imported into the workspace. The solution shown in Figure 12-3 on page 621 is implemented. You can access this by double-clicking **diagram.gph** under the BankStrutsWeb project.

12.4.2 Prepare the application and sample database

Ensure that you have completed the following steps prior to running the application:

1. Add the JDBC driver for Cloudscape to the BankStrutsWeb project.

   Refer to 12.2.3, “Add JDBC driver for Cloudscape to project” on page 628.
2. Set up the sample BANK database using the sample Table.ddl and loadData.sql.

   Refer to 12.2.4, “Set up the sample database” on page 629.
3. Ensure that the data source is configured to use the database you defined in the previous step.

   Refer to 12.2.5, “Configure the data source” on page 630.

12.4.3 Run the Struts Bank Web application sample

In this section we run the sample application and explore the functionality built using Rational Application Developer and its support for rapid development of Struts-based Web applications.

To run the sample Bank application, do the following:

1. Open the Web perspective Project Explorer view.
2. Expand **Dynamic Web Projects** → **BankStrutsWeb** → **WebContent**.

3. Select **logon.jsp**, right-click, and select **Run** → **Run on Server**.
4. When the Server Selection wizard appears, select **Choose and existing server**, select the **WebSphere Application Server v6.0** server, and click **Finish**.

   The home page logon.jsp should be displayed as shown in Figure 12-35.

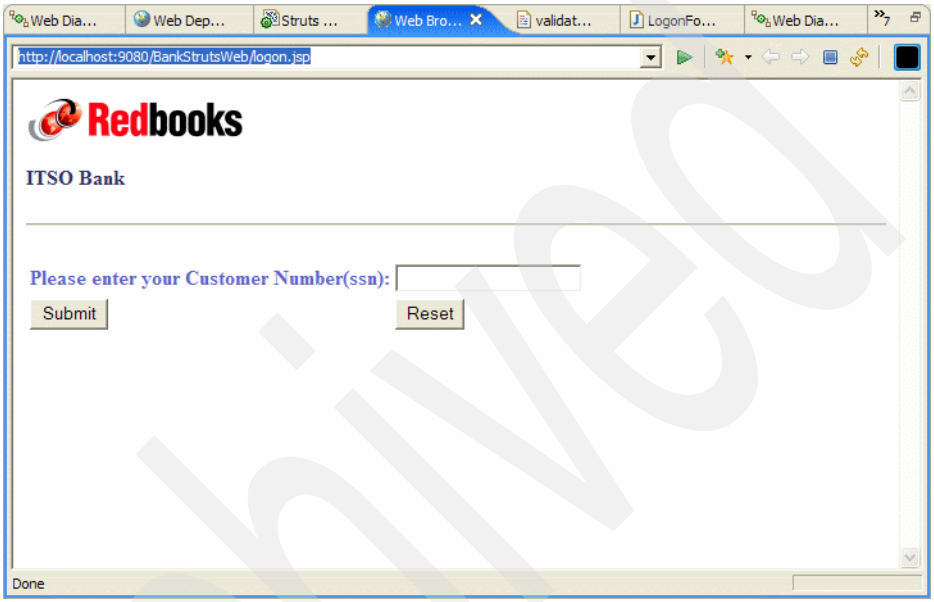


Figure 12-35 Running the sample - Logon.jsp

5. If you click **Submit** without entering an ssn, the Struts Validation Framework kicks in and displays the error message added to ApplicationResources.properties, as shown in Figure 12-36 on page 668.

Archived

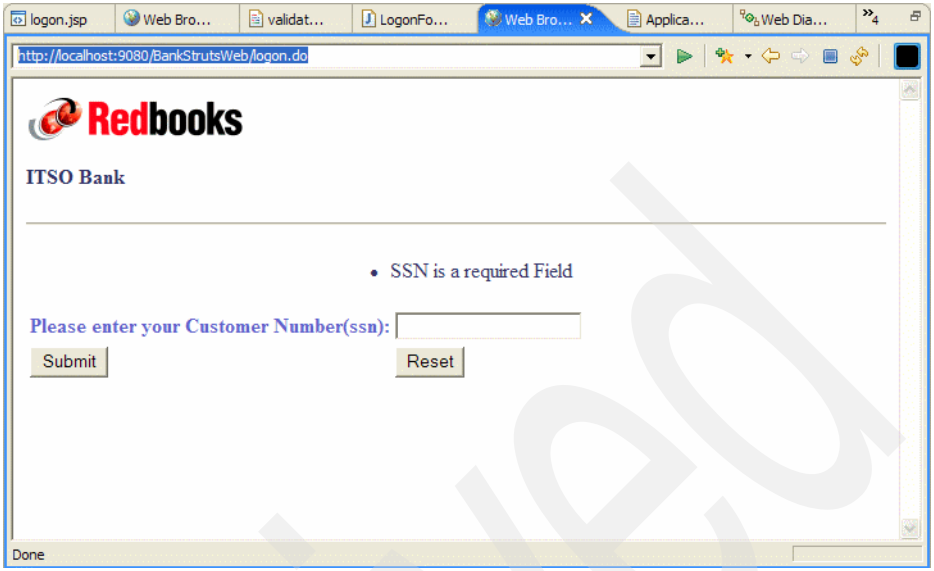


Figure 12-36 Running the sample - Logon.jsp - Invalid input

6. Enter a sample ssn 111-11-1111 and click **Submit**. A list of accounts associated with the account is displayed. Here the logon action is responsible for retrieving this information. In a real world application the logon action will talk to the business tier to retrieve this information. The Business Tier Architecture is described in detail in Chapter 15, “Develop Web applications using EJBs” on page 827.

Archived

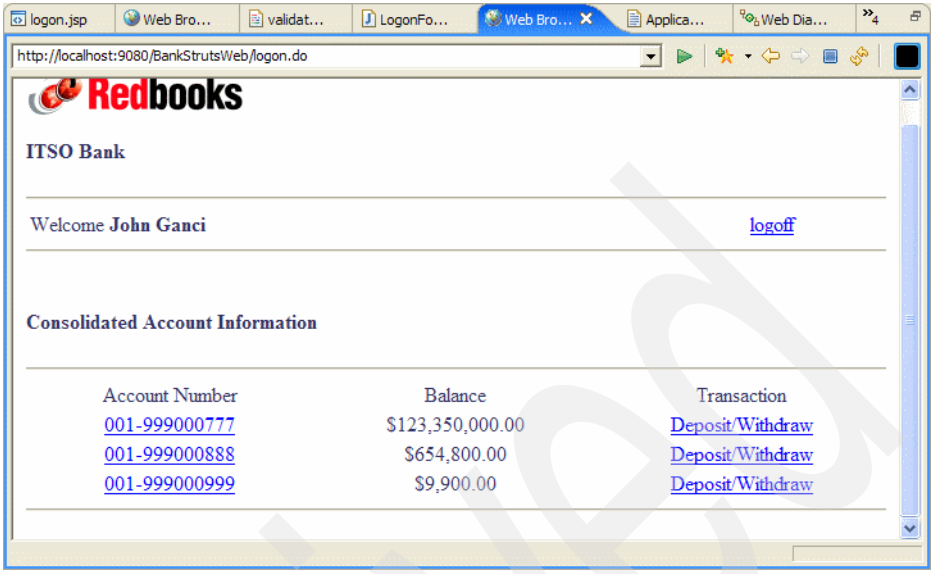


Figure 12-37 Running the sample - Account listing

7. To view details of a particular account, click the **Account Number** and the list of transactions on the account is listed, as shown in Figure 12-38.

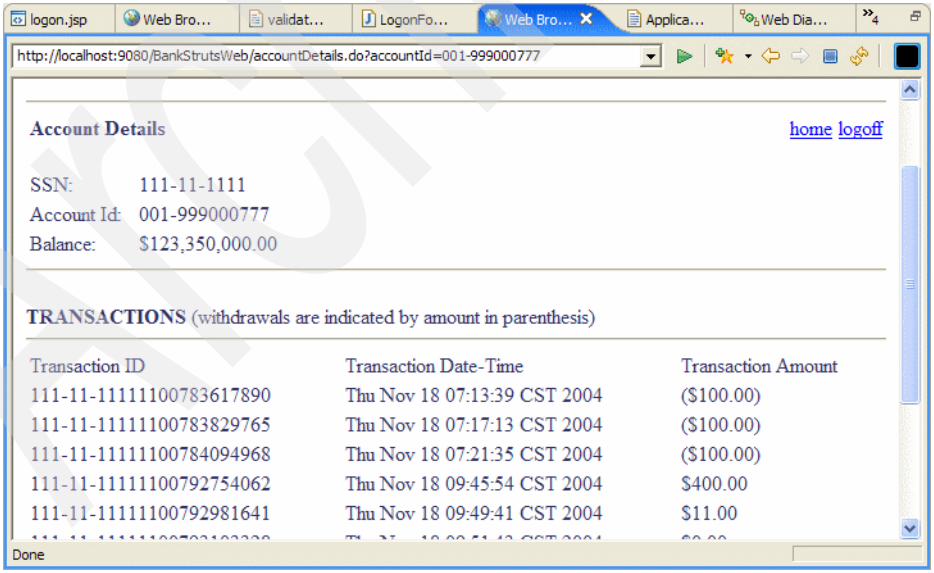


Figure 12-38 Running the sample - Account details

Archived

8. To perform a transaction on the account, click the **Deposit/Withdraw** link in Figure 12-39.
9. The transaction page is displayed. Enter a positive number for deposits and negative number for withdrawal. We will enter in -50000 to indicate a withdrawal of $50,000, and click **Submit**.

Figure 12-39 Running the sample - Performing transactions

10. The Account Details page is displayed with a new entry made for the last transaction, as shown in Figure 12-40 on page 671.

Archive

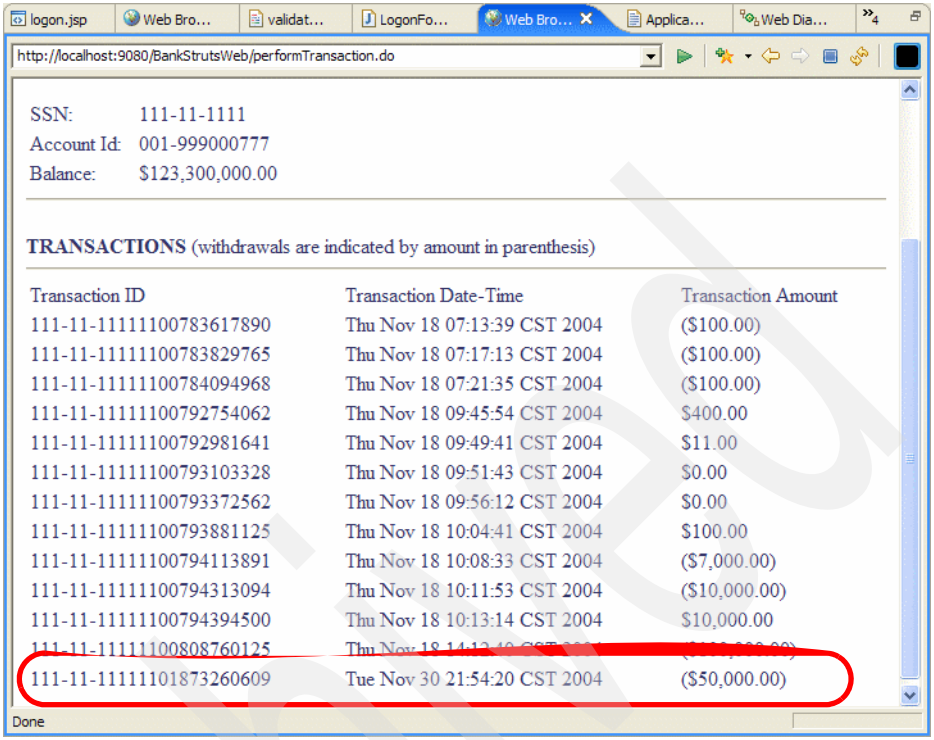


Figure 12-40 Running the sample - Transaction listing

Archive

Archived

13

Develop Web applications using JSF and SDO

JavaServer Faces (JSF) is a framework that simplifies building user interfaces for Web applications. Service Data Objects (SDO) is a data programming architecture and API for the Java platform that unifies data programming across data source types.

This chapter introduces the features, benefits, and architecture of JSF and SDO. The focus of the chapter is to demonstrate the Rational Application Developer support and tooling for JSF. The chapter includes a working example Web application using JSF and SDO.

The chapter is organized into the following sections:

- Introduction to JSF and SDO
- Prepare for the sample
- Develop a Web application using JSF and SDO
- Run the sample Web application

© Copyright IBM Corp. 2005. All rights reserved.

13.1 Introduction to JSF and SDO

This section provides an introduction to JavaServer Faces (JSF) and Service Data Objects (SDO).

13.1.1 JavaServer Faces (JSF) overview

JavaServer Faces is a framework that simplifies building user interfaces for Web applications. The combination of the JSF technology and the tooling provided by IBM Rational Application Developer allows developers of differing skill levels the ability to achieve the promises of rapid Web development.

This section provides an overview of the following aspects of JSF:

- JSF features and benefits
- JSF application architecture
- IBM Rational Application Developer support for JSF

Note: Detailed information on the JSF specification can be found at:

`http://java.sun.com/j2ee/javaserverfaces/download.html`

JSF features and benefits

The following is a list of the key features and benefits of using JSF for Web application design and development:

- JSF is a standards-based Web application framework.

  JavaServer Faces technology is the result of the Java Community process JSR-127 and evolved from Struts. JSF addresses more of the Model-View-Controller pattern than Struts, in that it more strongly addresses the view or presentation layer though UI components, and addresses the model through managed beans. Although JSF is an emerging technology and will likely become a dominant standard, Struts is still widely used.

  JSF is targeted at Web developers with little knowledge of Java and eliminates much of the hand coding involved in integrating Web applications with back-end systems.

- Event driven architecture.

  JSF provides server-side rich UI components that respond to client events.

- User interface development.

  UI components are de-coupled from its rendering. This allows for other technologies such as WML to be used (for example, mobile devices).

  JSF allows direct binding of user interface (UI) components to model data.

Archive

Developers can use extensive libraries of prebuilt UI components that provide both basic and advanced Web functionality.

- Session and object management

  JSF manages designated model data objects by handling their initialization, persistence over the request cycle, and cleanup.

- Validation and error feedback

  JSF allows direct binding of reusable validators to UI components. The framework also provides a queue mechanism to simplify error and message feedback to the application user. These messages can be associated with specific UI components.

- Internationalization

  JSF provides tools for internationalizing Web applications, including supporting number, currency, time, and date formatting, and externalization of UI strings.

JSF application architecture

The JSF application architecture can be easily extended in a variety of ways to suit the requirements of your particular application. You can develop custom components, renderers, validators, and other JSF objects and register them with the JSF runtime.

The focus of this section is to highlight the JSF application architecture depicted in Figure 13-1 on page 676:

- JSF page JSPs are built from JSF components, where each component is represented by a server-side class.
- Faces Servlet: One servlet (FacesServlet) controls the execution flow.
- Configuration file: An XML file (faces-config.xml) that contains the navigation rules between the JSPs, validators, and managed beans.
- Tag libraries: The JSF components are implemented in tag libraries.
- Validators: Java classes to validate the content of JSF components, for example, to validate user input.
- Managed beans: JavaBeans defined in the configuration file to hold the data from JSF components. Managed beans represent the data model and are passed between business logic and user interface. JSF moves the data between managed beans and user interface components.
- Events: Java code executed in the server for events (for example, a push button). Event handling is used to pass managed beans to business logic.

Archived

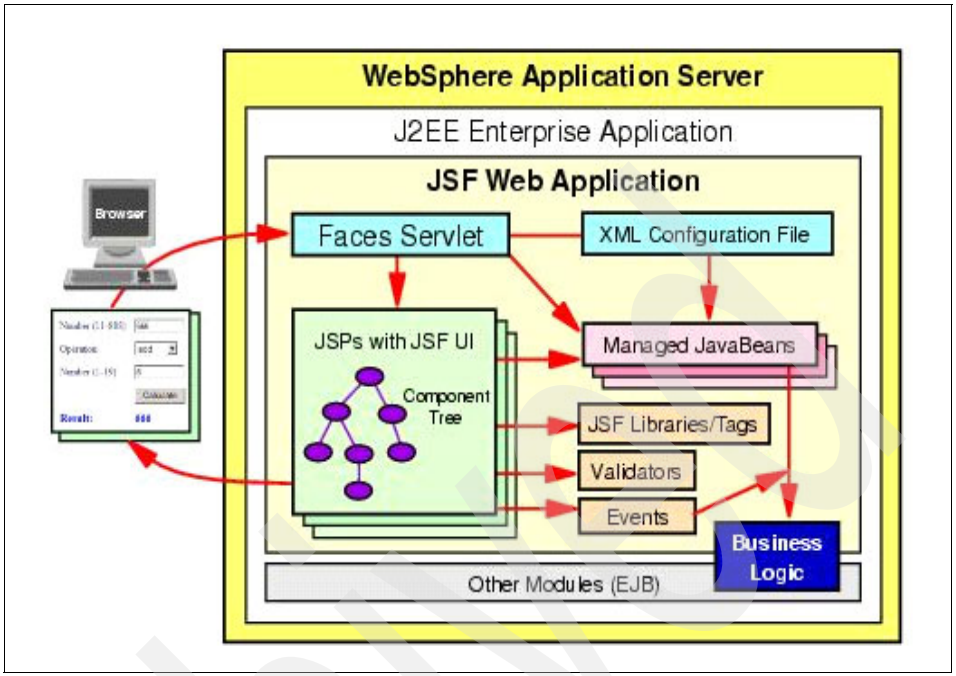


Figure 13-1 JSF application architecture

Figure 13-2 on page 677 represents the structure of a simple JSF application created in Rational Application Developer.

Archived

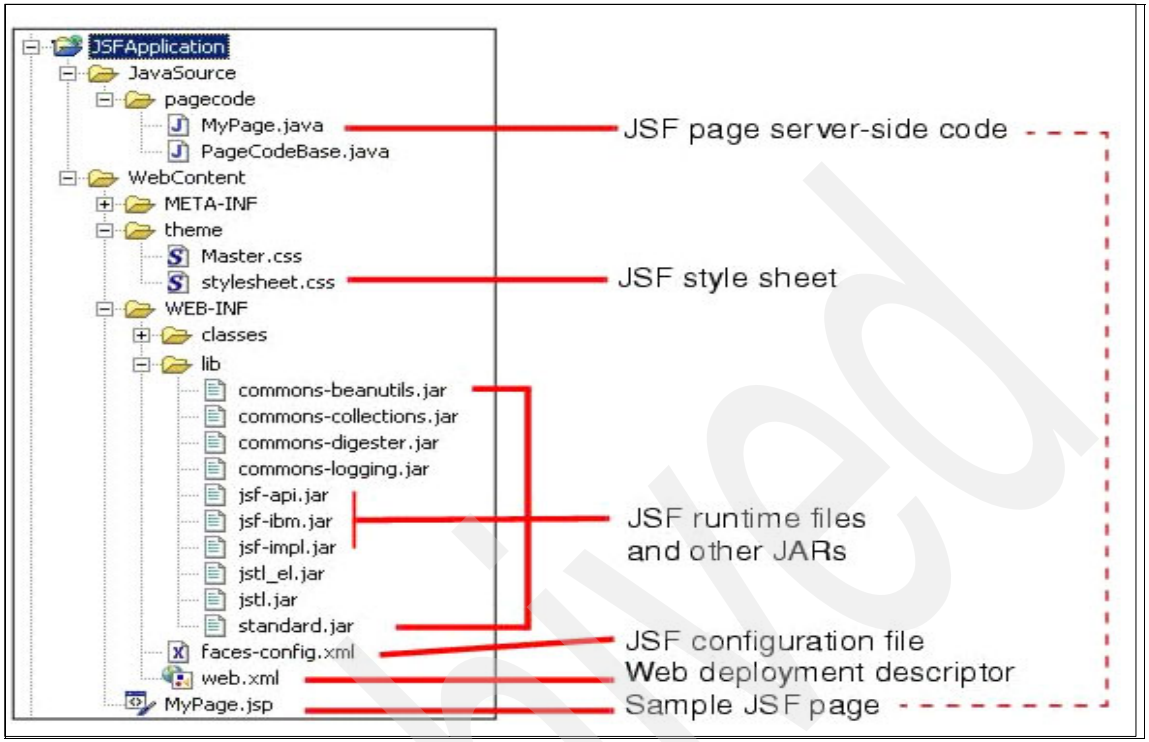


Figure 13-2 JSF application structure within Rational Application Developer

IBM Rational Application Developer support for JSF

IBM Rational Application Developer V6.0 includes a wide range of features for building highly functional Web applications. Application Developer includes full support to make drag-and-drop Web application development a reality.

Rational Application Developer includes the following support and tooling for JSF Web application development:

- Visual page layout of JSF components using Page Designer.
- Built-in Component Property Editor.
- Built-in tools to simplify and automate event handling.
- Built-in tools to simplify page navigation.
- Page navigation defined declaratively.
- Automatic code generation for data validation, formatting, and CRUD functions for data access.
- Relational database support.

Archived

- EJB support.
- Web Services support.
- Data abstraction objects for easy data connectivity (Service Data Objects).
- Data objects can be bound easily to user interface components.

13.1.2 Service Data Objects (SDO)

Service Data Objects is a data programming architecture and API for the Java platform that unifies data programming across data source types. SDO provides robust support for common application patterns, and enables applications, tools, and frameworks to more easily query, view, bind, update, and introspect data.

The Java specification request is JSR-235 for Service Data Objects (SDO) and can be found at:

http://www.jcp.org/en/jsr/detail?id=235

Note: For more detailed information on JavaServer Faces and Service Data Objects, we recommend the following:

- *WebSphere Studio 5.1.2 JavaServer Faces and Service Data Objects*, SG24-6361
- *Developing JSF Applications using WebSphere Studio V5.1.1 -- Part 1*, found at:

  http://www.ibm.com/developerworks/websphere/techjournal/0401_barcia/barcia.html

13.2 Prepare for the sample

This section describes the tasks that need to be completed prior to developing the JSF and SDO sample application.

Note: A completed version of the ITSO RedBank Web application built using JSF and SDO can be found in the c:\6449code\jsf\BankJSF.zip Project Interchange file. If you do not wish to create the sample yourself, but want to see it run, follow the procedures described in 13.4, “Run the sample Web application” on page 746.

13.2.1 Create a Dynamic Web Project

To create a new Dynamic Web Project with support for JSF and SDO, do the following:

1. Open the Web perspective Project Explorer view.
2. Right-click the **Dynamic Web Projects** folder.
3. In the context menu select **New** → **Dynamic Web Project**.
4. When the New Dynamic Web Project wizard appears, enter the following (as seen in Figure 13-3), and then click **Next**:
   - Name: `BankJSF`
   - Click **Show Advanced**.
   - Check **Add support for annotated Java classes**.

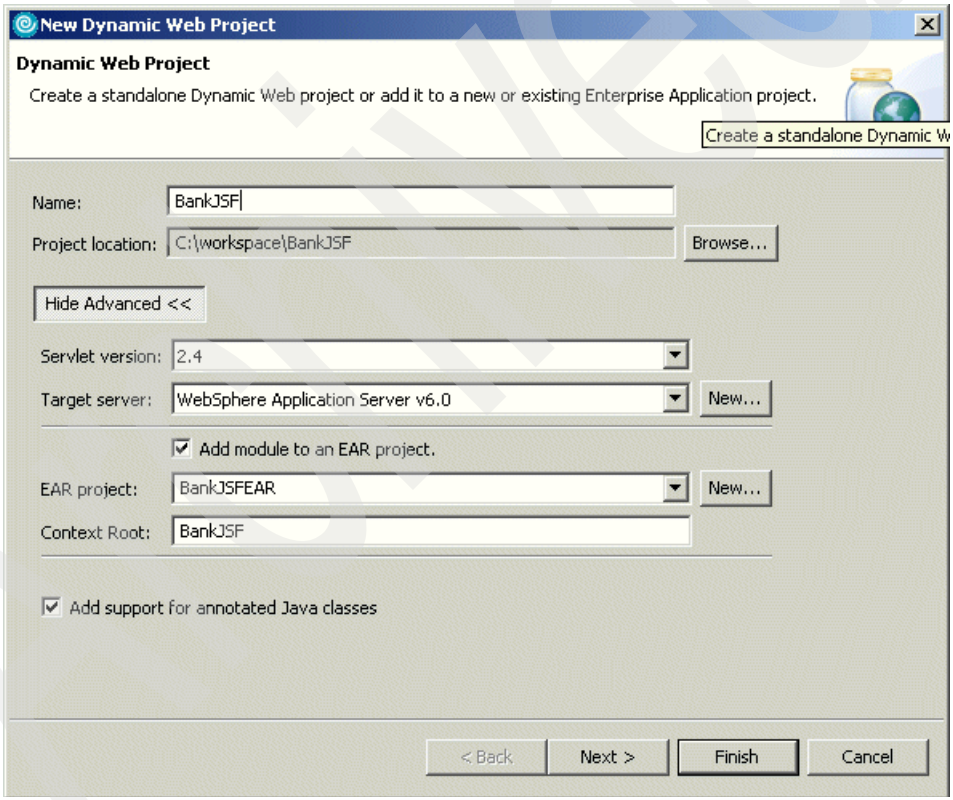


Figure 13-3 New Dynamic Web Project Wizard

5. When the Features dialog appears, select the features you wish to include in your project.

For our example, ensure that you select the following features (as seen in Figure 13-4) and then click **Next**:

- Check **WDO Relational Database runtimes**.
- Check **Web Diagram**.
- Check **JSP Tag Libraries** (includes both the JSP Standard Tag Library and Utility Tag Library).

Note: At the time of writing, the SDO feature is named WDO in various parts of Application Developer.

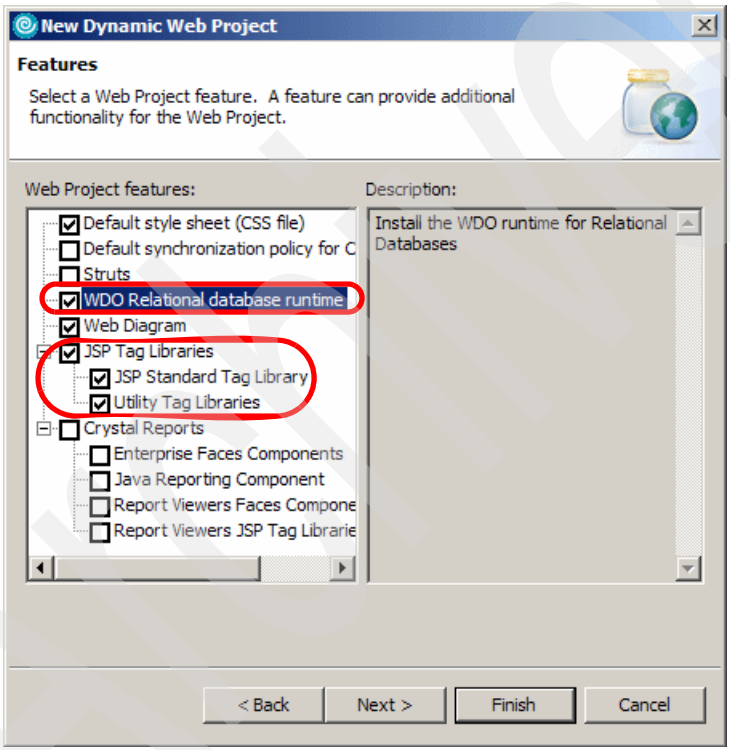


Figure 13-4 The features dialog for New Dynamic Web Projects

Tip: Dynamic Web Project features can also be added to a project later using the Web Project Features panel of the Project Properties window.

6. When the Select a Page Templates for the Web site dialog appears, do not check this option. We will create a user-defined page template as part of our sample. Click **Finish**.

13.2.2 Set up the sample database

In order for us to use SDO components we will need to have a relational database to connect to. This section provides instructions for deploying the BANK sample database and populating the database with sample data. For simplicity we will use the built-in Cloudscape database.

To create the database, create the connection, and create and populate the tables for the BANK sample from within Rational Application Developer, do the following:

1. Create the database and connection to the Cloudscape BANK database from within Rational Application Developer.

   Note: For this chapter we created a connection named BankJSF_Con1.

   For details refer to “Create a database connection” on page 347.

2. Create the BANK database tables from within Rational Application Developer.

   For details refer to “Create database tables via Rational Application Developer” on page 350.

3. Populate the BANK database tables with sample data from within Rational Application Developer.

   For details refer to “Populate the tables within Rational Application Developer” on page 352.

13.2.3 Configure the data source via the enhanced EAR

There are a couple of methods that can be used to configure the datasource, including using the WebSphere Administrative Console or using the WebSphere Enhanced EAR, which stores the configuration in the deployment descriptor and is deployed with the application.

This section describes how to configure the datasource using the WebSphere Enhanced EAR capabilities. The enhanced EAR is configured in the Deployment tab of the EAR deployment descriptor.

Archived

The procedure found in this section considers two scenarios for using the enhanced EAR:

- If you choose to import the complete sample code, you will only need to verify that the value of the databaseName property in the deployment descriptor matches the location of your database.
- If you are going to complete the working example Web application found in this chapter, you will need to create the JDBC provider and the datasource, and update the databaseName property.

Note: For more information on configuring data sources and general deployment issues, refer to Chapter 23, “Deploy enterprise applications” on page 1189.

Access the deployment descriptor

To access the deployment descriptor where the enhanced EAR settings are defined, do the following:

1. Open the Web perspective Project Explorer view.
2. Expand **Enterprise Applications** → **BankJSFEAR**.
3. Double-click **Deployment Descriptor : BankJSFEAR** to open the file in the Deployment Descriptor Editor.
4. Click the **Deployment** tab.

Note: When using Cloudscape, the configuration of the user ID and password for the JAAS Authentication is not needed.

When using DB2 Universal Database or other database types that require a user ID and password, you will need to configure the JAAS authentication.

Configure a new JDBC provider

To configure a new JDBC provider using the enhanced EAR capability in the deployment descriptor, do the following:

1. From the Deployment tab of the Application Deployment Descriptor, click **Add** under the JDBC provider list.
2. When the Create a JDBC Provider dialog appears, select **Cloudscape** as the Database type, select **Cloudscape JDBC Provider** as the JDBC provider type, and then click **Next**.

Note: The JDBC provider type list for Cloudscape will contain two entries:

- Cloudscape JDBC Provider
- Cloudscape JDBC Provider (XA)

Since we will not need support for two-phase commits, we choose to use the non-XA JDBC provider for Cloudscape.

3. Enter `Cloudscape JDBC Provider - SDO` in the Name field and then click **Finish**.

Configure the data source

To configure a new data source using the enhanced EAR capability in the deployment descriptor, do the following:

1. From the Deployment tab of the Application Deployment Descriptor, select the JDBC provider created in the previous step.
2. Click **Add** next to data source.
3. When the Create a Data Source dialog appears, select **Cloudscape JDBC Provider** under the JDBC provider, select **Version 5.0 data source**, and then click **Next**.
4. When the Create a Data Source dialog appears, enter the following and then click **Finish**:
   - Name: `BankJSF_Con1`

     Note: This name should match the connection name used to select a database when adding the SDO relational record to the page (see “Add SDO to a JSF page” on page 720).

   - JNDI name: `jdbc/BankJSF_Con1`
   - Description: `BankJSF_Con1`

Configure the databaseName property

To configure the databaseName in a new data source using the enhanced EAR capability in the deployment descriptor to define the location of the database for your environment, do the following:

1. Select the data source created in the previous section.
2. Select the **databaseName** property under the Resource properties.
3. Click **Edit** next to Resource properties to change the value for the databaseName.

Archived

4. When the Edit a resource property dialog appears, enter `c:\databases\BANK` in the Value field and then click **OK**.

   In our example, c:\databases\BANK is the database created for our sample application in 13.2.2, “Set up the sample database” on page 681.

5. Save the Application Deployment Descriptor.
6. Restart the test server for the changes to the deployment descriptor to take effect.

Tip: You can verify that the data source configuration has been picked up by the server by inspecting the console output after the server has been restarted. The following lines should appear in the server console output:

```
WSVR0200I: Starting application: BankJSFEAR
WSVR0049I: Binding BankJSF_Con1_CF as eis/jdbc/BankJSF_Con1_CMP
WSVR0049I: Binding BankJSF_Con1 as jdbc/BankJSF_Con1
SRVE0169I: Loading Web Module: BankJSF.
```

If these lines do not appear, you may have to remove the BankJSF application from the server and then republish it.

13.3 Develop a Web application using JSF and SDO

This section describes how to develop the ITSO RedBank sample Web application using JSF and SDO.

This section includes the following tasks:

- Create a page template.
- Useful views for editing page template files.
- Customize the page template.
- Create JSF resources using the Web Diagram tool.

13.3.1 Create a page template

A page template contains common areas that you want to appear on all pages, and content areas that will be unique on the page. They are used to provide a common look and feel for a Web Project.

Create a page template

To create a page template containing JSF components, do the following:

1. Open the Web perspective Project Explorer view.
2. Expand **Dynamic Web Projects** → **BankJSF**.

3. Right-click the **WebContent** folder, and select **New** → **Page Template File** from the context menu.
4. When the New Page Template wizard appears, enter the following (as seen in Figure 13-5 on page 685), and then click **Finish**:
   - Folder: `/BankJSF/WebContent`
   - File name: `BankJSFTemplate`
   - Model: Select **Template containing Faces Components**.

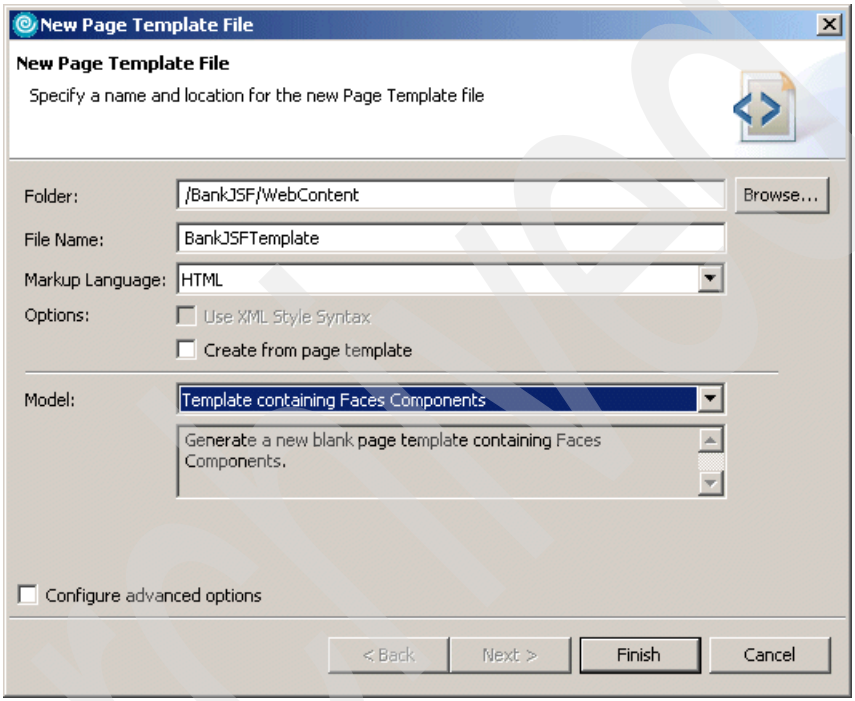


Figure 13-5 The new page template wizard design

5. When prompted with the message displayed in Figure 13-6, click **OK**. We will add a content area as part of our page template customization.

Accepted

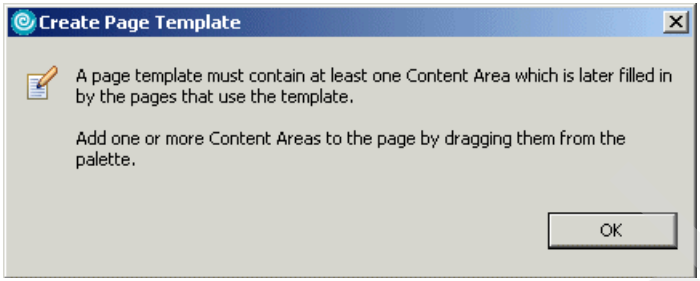


Figure 13-6 Content area warning

Review of files generated for page template

Now that we have created the page template, we would like to highlight some of the files that have been generated.

In the Project Explorer view (shown in Figure 13-7) you will see that there are some files added to the Java Resources folder. The JavaSource folder contains the generated managed bean classes. The PageCodeBase.java class file is a super class for all the generated managed bean classes. A BankJSFTemplate.java class was created as the managed bean for the template page. Each JSF page will have its own managed bean class generated.

Archived

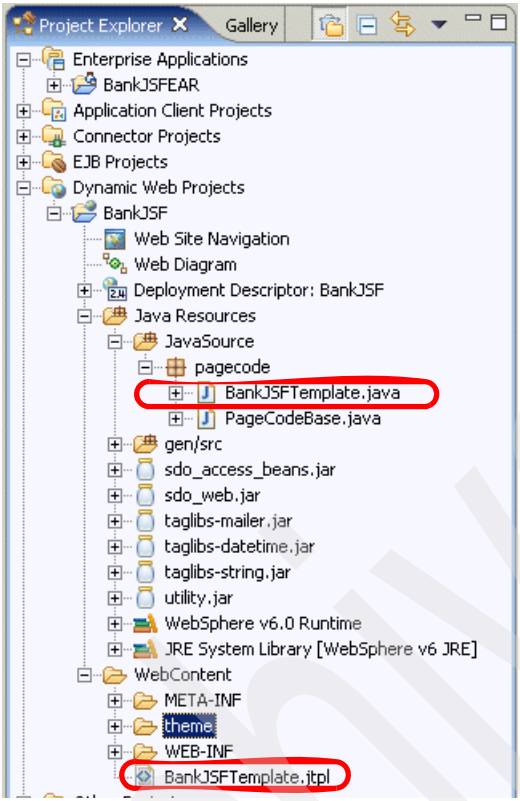


Figure 13-7 JSF page template files

13.3.2 Useful views for editing page template files

There are many views and tools that can be used when editing page templates. This section highlights some views that we found helpful for editing a page template.

> **Note:** This section is not required for the working example. To continue with the working example, skip to 13.3.3, “Customize the page template” on page 695.

Page Template Editor view

This is the main editor window and it comprised of three tabs: Design, Source, and Preview.

- Design tab: The Design tab is the main GUI editor that allows you to select the components on the page in a graphical manner. It also gives you a good

representation of the resulting page. Also, you will notice in the top right of Figure 13-8 that there are some drop-down options; these options allow you to select the layout mode, select an existing component on the page, and select font options for presentation, respectively. If you right-click the Design editor, the context menu will allow you to be able to insert a variety of components, as well as edit embedded documents.

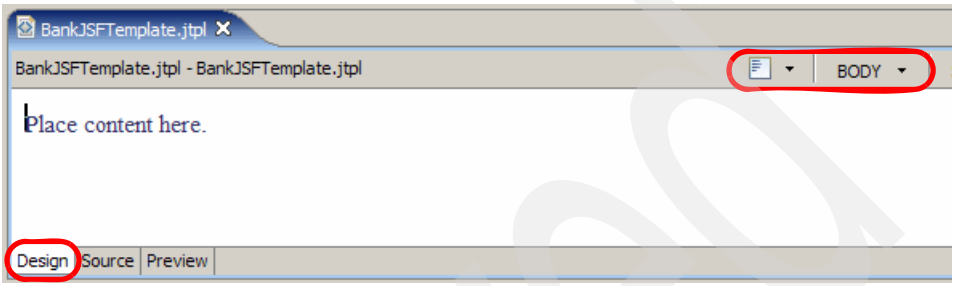


Figure 13-8 Design tab of Page Template Editor

- Source tab: The Source tab allows you to view the page template source. This tab allows you to tweak the template by changing the source directly. If you right-click in the source editor you have the option to clean up the document, format the source, and refactor components.
  - The cleanup option will present the Cleanup dialog, as shown in Figure 13-10 on page 689. This dialog allows you to change the casing of HTML tags and attributes. It also gives you the ability to clean up tags with missing attributes, place quotes around attributes, insert missing tags, and format the source.
  - The format option allows you to format the whole document based on your formatting preferences, or just format the active element. To set your preferences you can right-click the source editor and select **Preferences...**, and the preferences dialog will be displayed, as shown in Figure 13-11 on page 690. The preferences dialog is similar to the one that is displayed when you select the preferences through the Windows menu; however, only the relevant preferences are include in the dialog.

Archive

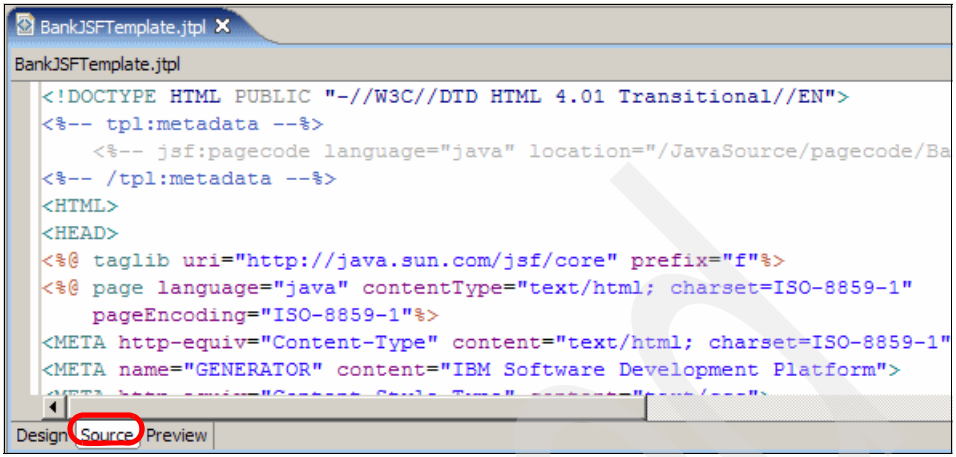


Figure 13-9 Source tab of the Page Template Editor

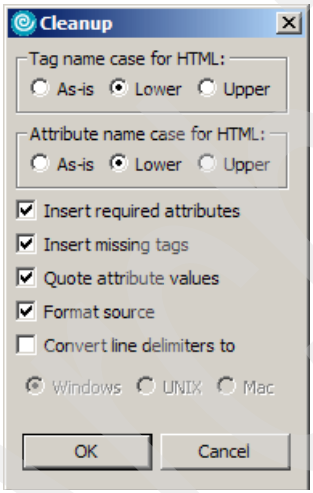


Figure 13-10 Cleanup dialog

Archived

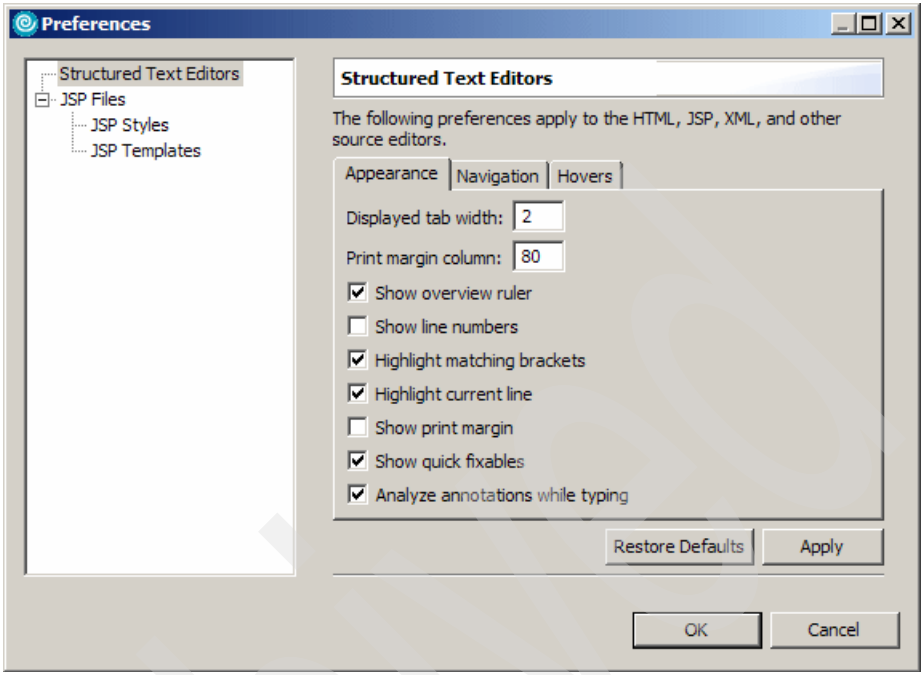


Figure 13-11 Preferences dialog

- Preview tab: The Preview tab allows you to preview the resulting JSF page template. It presents the page template as it would appear in a browser. You will notice in the top right corner there are some buttons; these button are similar to the navigation buttons on a browser. They allow you to go to the previous page, next page, or reload the page, respectively.

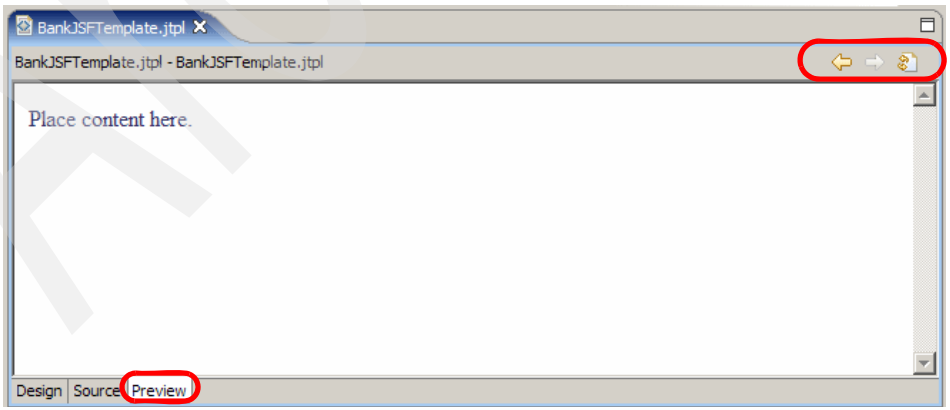


Figure 13-12 Preview tab for Page Template Editor

Note: The Preview tab within the Page Designer for templates does not always give you a graphical preview when there are extended components on the page. The Preview tab on the Page Designer for JSF does give you a graphical preview of the page.

Palette view

This view allows you to select palettes to work with. Examples of palettes in the Palette view can be seen in Figure 13-13 on page 692. Palettes contain the reusable components that can be dragged on to the page to be added to a file. The palettes that are available for page template files include:

- HTML Tags: This palette contains common HTML tags such as images, lists, tables, links, etc.
- JSP Tags: This palette contains common JSP tags such as beans, expressions, scriptlets, declarations, include directives, as well as some logical tags such as if, when, otherwise, etc.
- Crystal Report Faces Components: This palette contains Crystal Report Faces Components. You can drag these reusable Crystal Report Components to be used on the page template.
- Faces Components: This palette contains the common Faces tags, as well as the IBM extended Faces tags.
- Page Template: This palette contains page Template tags such as content area and page fragment.
- Web Site Navigation: This palette contains IBM's site edit tags, which can be used to add site navigation to your page template.
- Data: This palette contains data components that can be added to the page template such as Web services, EJB session bean, SDO relational records, SDO relational record list, and Java beans.

The Palette view can be customized to hide and display palettes that you wish to use. The palettes listed above are the default palettes presented when editing a Page Template with JSF and SDO components. Other palettes that can be included are EGL, Struts, Server Side Include directives, etc.

To customize the Palette view, right-click the Palette view and select **Customize**. The Customize Palette dialog will be presented, as shown in Figure 13-14 on page 693. This dialog allows you to hide and show palettes, as well as reorganize the order in which they appear. You will notice that in the dialog the hidden palettes are a lighter shade of gray.

Some palettes contain more components that can be presented on the screen at any one time, so there are some arrows to allow you to scroll up and down, as

Archive

seen in Figure 13-14 on page 693. Also, the Palette view allows you to pin open frequently used palettes to stop them from being closed when you open other palettes.

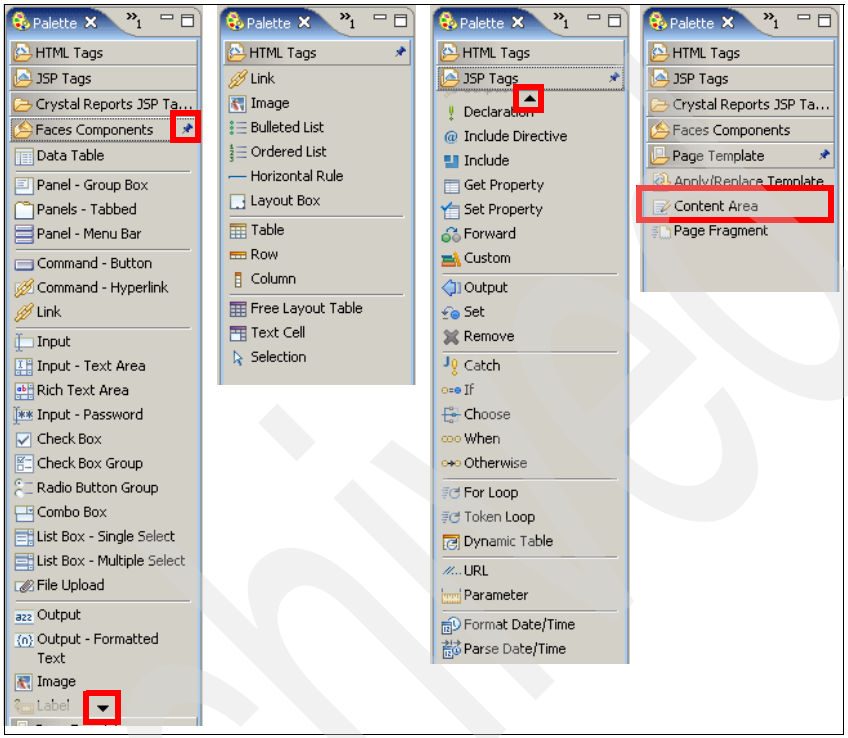


Figure 13-13 Palettes views

Archive

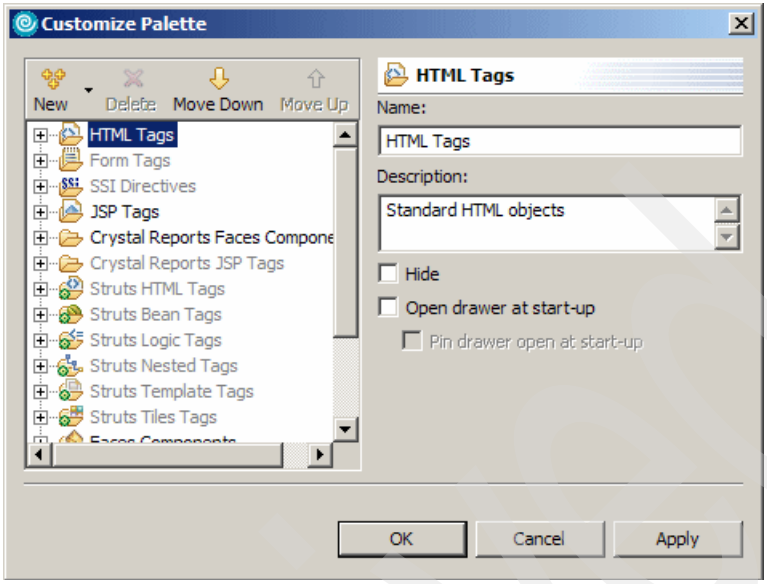


Figure 13-14 Customize Palette dialog

Properties view

The Properties view allows you to define the properties of the selected component. It is context sensitive and will change its content based on the type of component selected. The properties view (as seen in Figure 13-15) shows the properties of the body tag. It also has a Text tab, which allows you to define the text style for the page template. You will notice the highlighted button on the top right of Figure 13-15. This button allows you to view all the attributes of the tag in tabular format, as shown in Figure 13-16 on page 694.

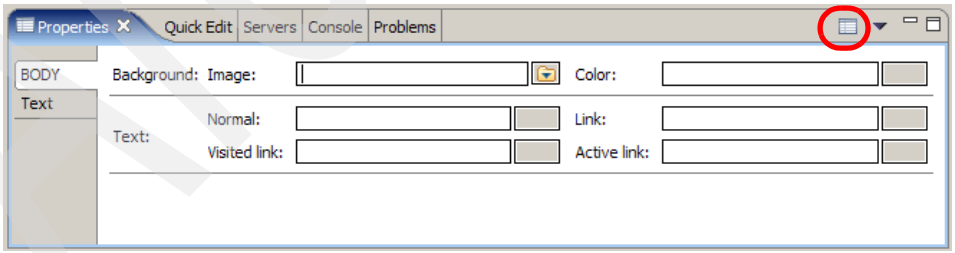


Figure 13-15 Properties view

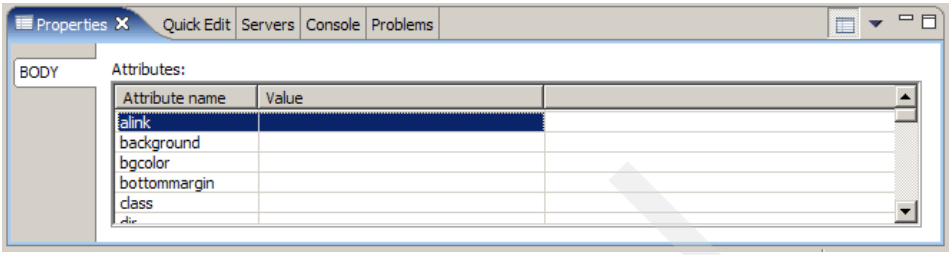


Figure 13-16 Properties view - All attributes

Page Data view

The Page Data view allows you to create data elements, as well as select methods from you managed beans to be added to your page or bound to your page components. The button in Figure 13-17 allows you to add new data components such as:

- EJB Session Beans
- Web Services
- Param scope variable
- Application scope variables
- Request scope variables
- Session scope variables
- SDO Relational record list
- SDO Relation records
- Java Beans

You can also add these components through the context menu by right-clicking the Page Data view.

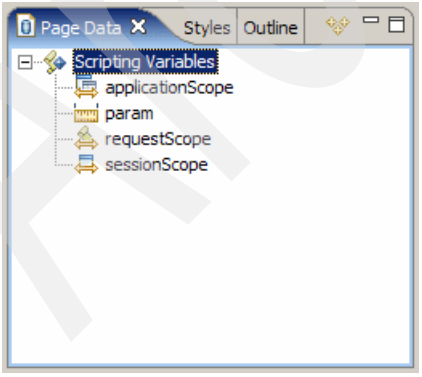


Figure 13-17 Page Data view

Archived

13.3.3 Customize the page template

Now that you have created a page template, it is likely you will want to customize the page. This section demonstrates how to make the following common customizations to a page template:

- Customize the logo image and title of the page template.
- Customize the style of the page template.
- Add the content area to the page template.

Customize the logo image and title of the page template

To customize the page template to include the ITSO logo image and ITSO RedBank title, do the following:

1. Import the itso_logo.gif image.
   a. Expand **Dynamic Web Projects** → **BankJSF** → **WebContent** → **theme**.
   b. Right-click the **theme** folder, and select **Import**.
   c. Select **File System** and click **Next**.
   d. Enter `c:\6449code\web` in the From directory, check **itso_logo.gif** (as seen in Figure 13-18 on page 696), and then click **Finish**.

Archived

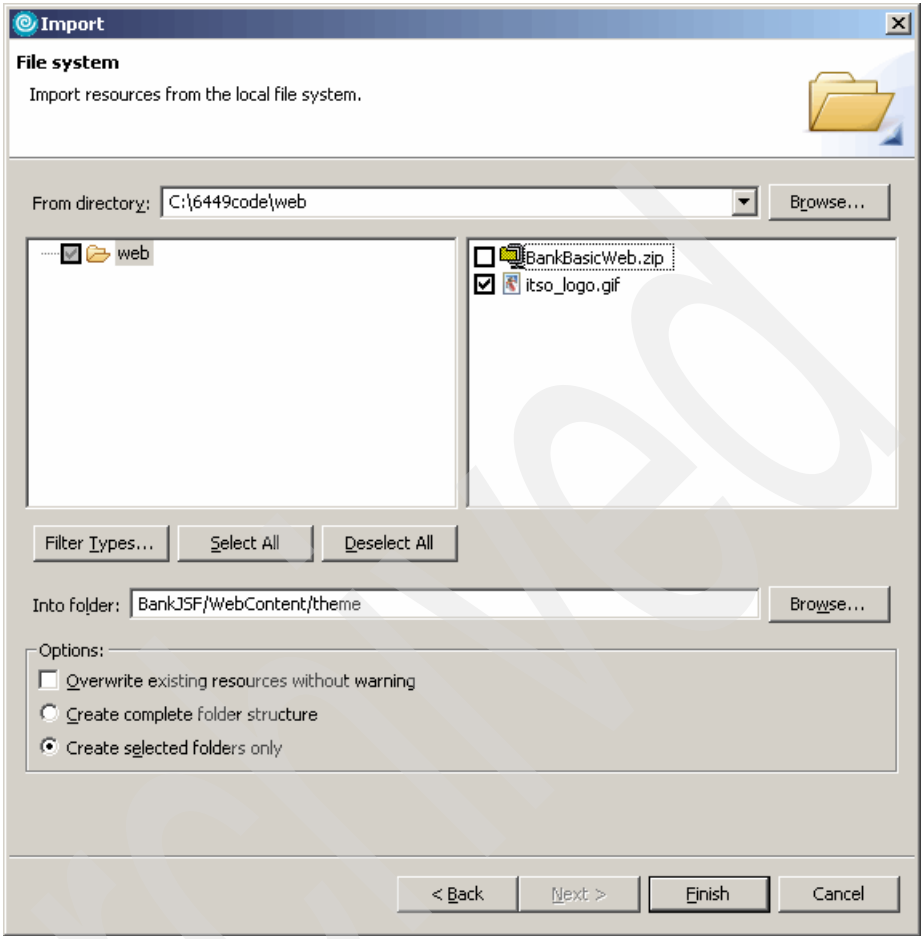


Figure 13-18 Adding the ITSO logo to the Dynamic Web Project

2. Expand **Dynamic Web Projects** → **BankJSF** → **WebContent**.
3. Double-click **BankJSFTemplate.jtpl** to open the file created in 13.3.1, “Create a page template” on page 684.

> **Note:** You may see the message shown in Figure 13-6 on page 686. If this happens, just click **OK**.

4. Click the **Design** tab.
5. Select the text **Place content here**, right-click, and select **Delete**.
6. From the Palettes view, expand the **Faces Components**.

7. Select the **Panel - Group Box** and drag it onto the page.
8. When the Select Type dialog appears, select **List** as seen in Figure 13-19, and then click **OK**.

> **Note:** You maybe asked to confirm resource copy. This is because the project does not contain resources that are required for the component that you are adding to the page template. Click the **OK** button to proceed. Notice the faces-config.xml file has been added to the WEB-INF folder.

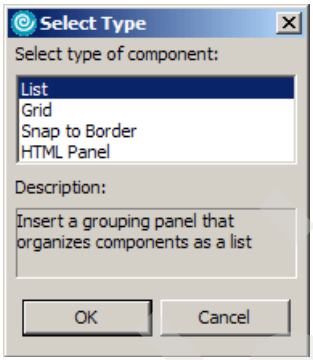


Figure 13-19 Select Type - List

9. You should now see a box on your page with the text `box1: Drag and Drop Items from the palette to this area to populate this region.`

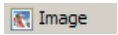
   From the Faces Components, select **Image**  and drag it to the panel. After adding the image, the page should look like Figure 13-20.

Figure 13-20 View of page template after adding image to panel - Group box

Archived

10. Update the image values in the Properties view.

    a. Select the image on the page (see Figure 13-20 on page 697) to highlight the image.

    b. In the Properties view, enter `headerImage` in the Id field, as seen in Figure 13-21.

    c. Click the folder icon next to File and select **Import**. Enter the path to the image and click **Open**. In our example, we entered `/WebContent/themes/itso_logo.gif` to import the image.

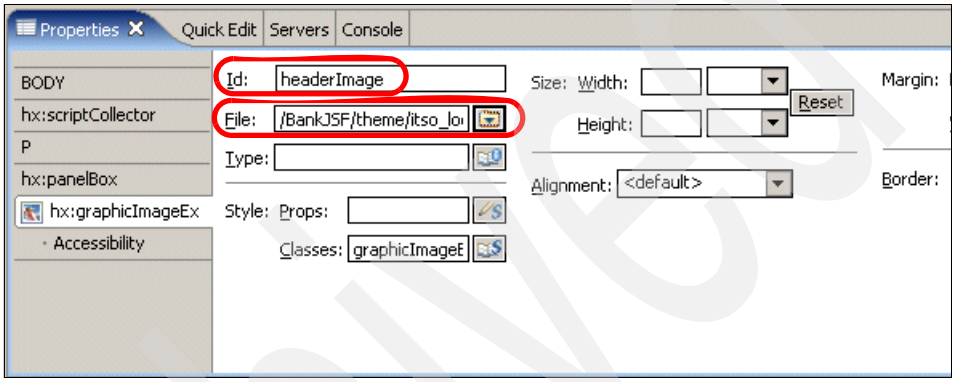


Figure 13-21 Properties for an image component

> **Note:** We found that the File field in the Properties view showed /BankJSF/theme/itso_logo.gif after the import of the image.

    d. You will notice in Figure 13-21 that the absolute reference has been entered. We need to make it a relative reference by removing the /BankJSF from the File field. After making the change, it will be theme/itso_logo.gif without any leading slash.

11. From the Faces Components palette, select **Output** Output and drag it under the image.

12. In the Properties view enter `ITSO RedBank` into the Value field.

13. Select the Output box (ITSO RedBank) and move it to the right of the image.

Customize the style of the page template

To customize the style of the page template, do the following:

1. Select the Output text box on the page.

2. In the Properties view, click the button next to the Style: Props: field. The Add style dialog will be displayed, as shown in Figure 13-22 on page 699.

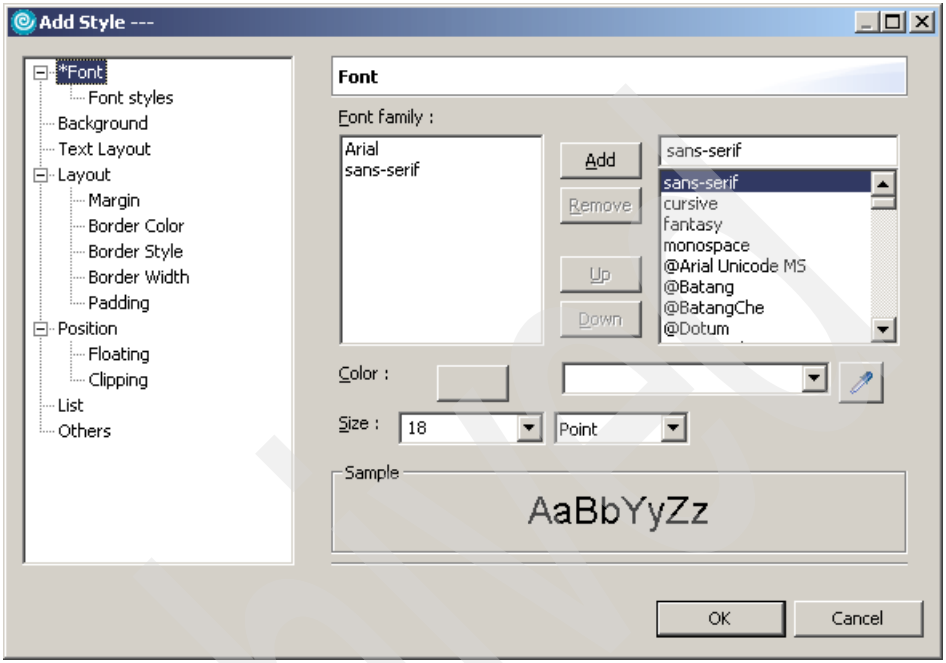


Figure 13-22 Style dialog used to configure the style of the selected components

3. Change the *Size* field value to 18.
4. Select **Arial** for the Font and click **Add**.
5. Select **sans-serif** for the Font and click **Add**.
6. Click **OK**.

Add the content area to the page template

To add the required content area to the page template, do the following:

1. Right-click under the output field and from the context menu select **Insert** → **Horizontal Rule**.
2. Expand Page Template in the Palette view.
3. From the Page Template, select the **Content Area** Content Area and drag it under the horizontal rule.
4. When the Insert Content Area for Page Template dialog appears, accept the default name (bodyarea) and click **OK**.

5. Right-click under the content area and from the context menu select **Insert** → **Horizontal Rule**.
6. Save the page template file.

The customized page template file should look similar to Figure 13-23.

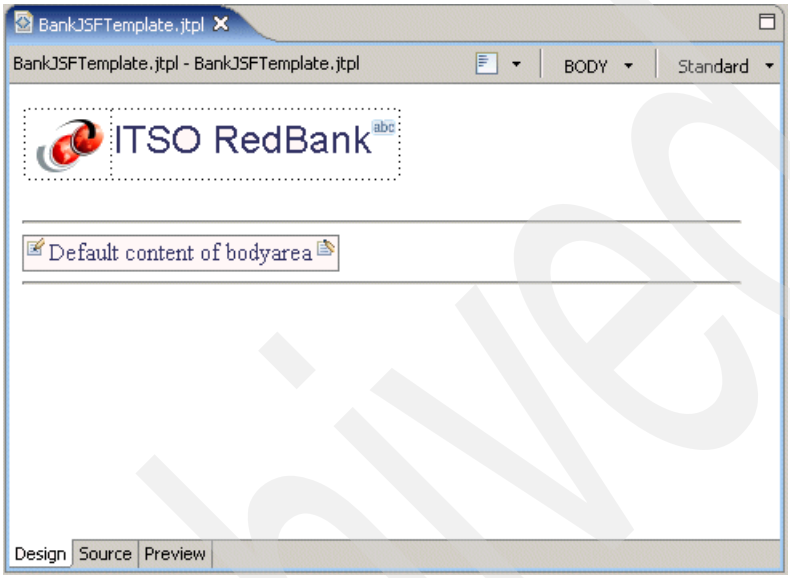


Figure 13-23 Customized page template - BankJSFTemplate.jtpl

13.3.4 Create JSF resources using the Web Diagram tool

This section demonstrates using the Web Diagram tool to create JSF pages, JSPs, Faces actions, and connections between pages and actions.

The sample application will consist of the following three pages:

- Login page (BankJSFLogin): Validate the entered CustomerSSN. If it is valid it will then display the customer details for the entered customer.
- Customer details page (BankJSFCustomerDetails): Display all the customer's account details and allow you to select an account to view the transactions.
- Account details page (BankJSPAccountDetails): Display the selected account details.

Create a JSF page using the Web Diagram tool

To create a JSF page using the Web Diagram tool, do the following:

1. Open the Web perspective Project Explorer view.

Archived

2. Expand Dynamic **Web Projects** → **BankJSF**.
3. Double-click **Web Diagram** (as shown in Figure 13-24) to open.

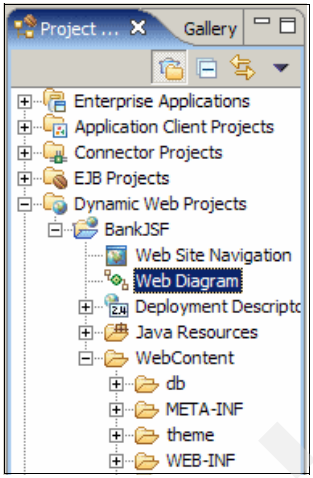


Figure 13-24 Select Web Diagram in the Project Explorer

4. When the Web Diagram appears in the Web Diagram Editor (as seen in Figure 13-25), select **Web Page** Web Page from the Web Parts palette and drag it onto the page.

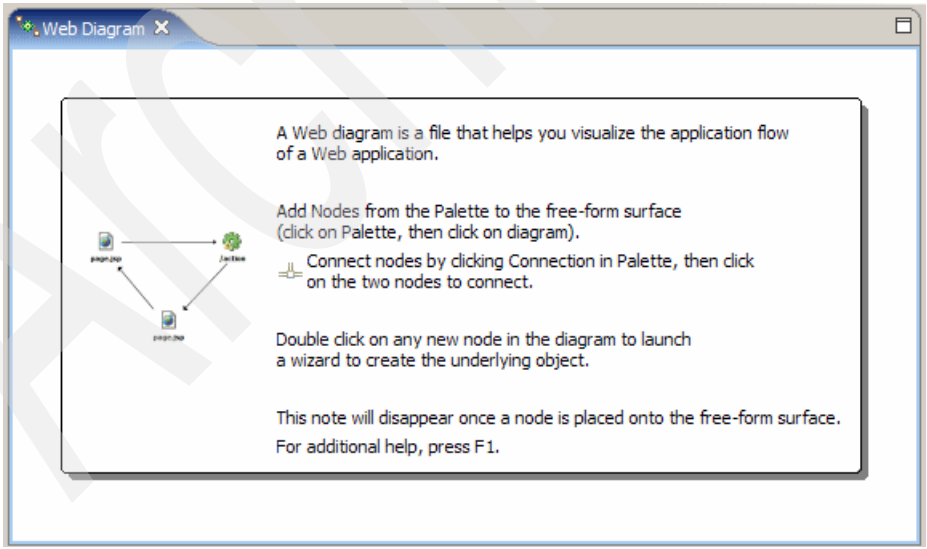


Figure 13-25 Initial Web Diagram page

Archived

5. In the Properties view change the Web Page Path value to `/BankJSFLogin.jsp`, and change the description to `The login page`.

> **Note:** Now that we have created a Web page in the Web Diagram tool, you may notice that the BankJSFLogin.jsp icon has a gray-blue tint. The reason for this is that it is not linked to an actual JSP file. We will use this diagram to create the actual JSP file that this icon will link to in a later step.

6. Repeat the process to create Web pages for the JSF pages shown in Table 13-1.

Table 13-1 Additional Web pages for the BankJSF application

Web page path	Description
/BankJSFCustomerDetails.jsp	Customer details and account overview
/BankJSFAccountDetails.jsp	Account details

Create a JSP file

To create the JSP file from a page template using the Web diagram, do the following:

1. Double-click the **BankJSFLogin.jsp** in the Web diagram.
2. When the New Faces JSP File wizard appears, enter the following (as seen in Figure 13-26 on page 703), and then click **Next**:
   - Folder: `/BankJSF/WebContent`
   - File name: `BankJSFLogin.jsp`
   - Options: Check **Create from page template**.

> **Note:** If you have not already created the page template refer to 13.3.1, “Create a page template” on page 684, to create a page template.

Archived

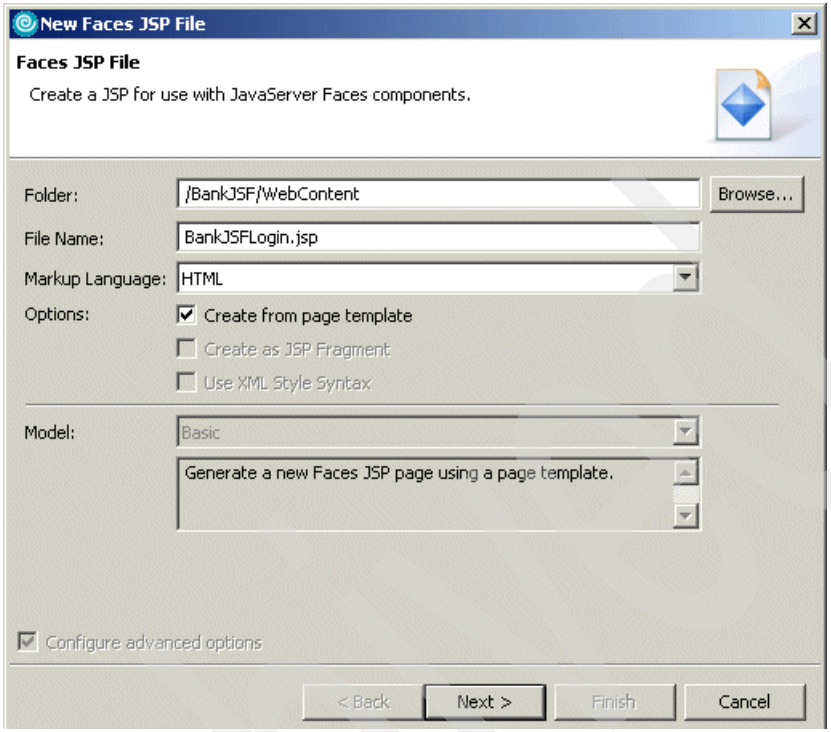


Figure 13-26 New Faces JSP File wizard - Create from page template

3. When the Page Template File Selection page appears, select the **User-defined page template**, select **BankJSFTemplate.jtpl** (as seen in Figure 13-27 on page 704), and then click **Finish**.

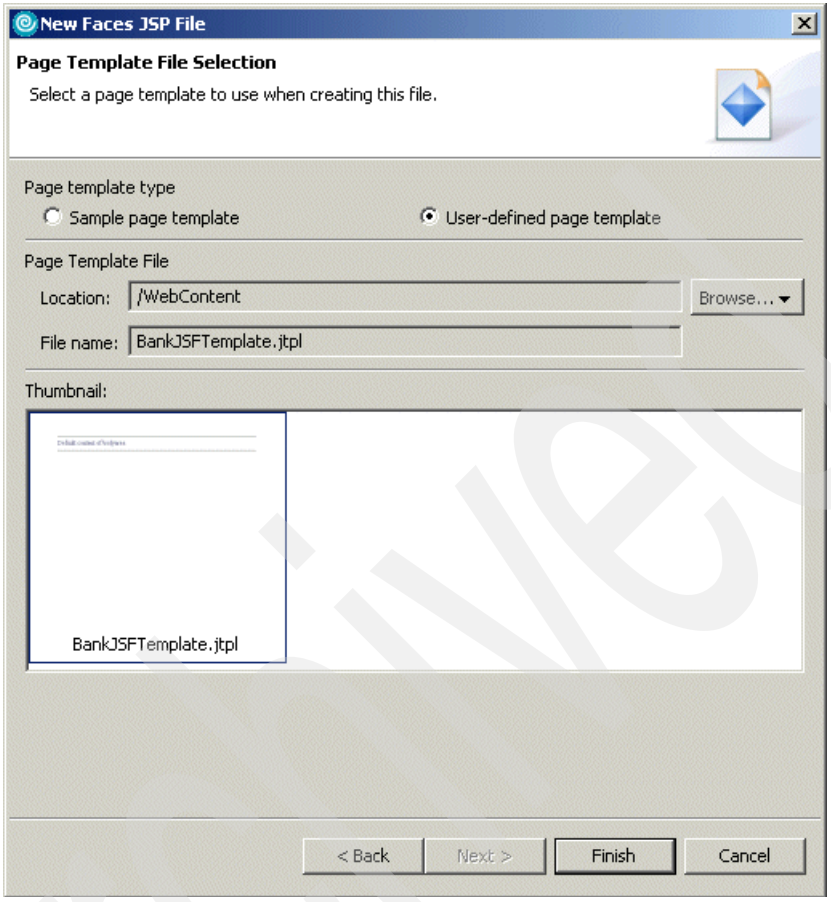


Figure 13-27 New Faces JSP file select template

4. The new Faces JSP file will be loaded into the Page Designer. At this point, the newly create JSF page should look like the page template.
5. Click the **Web Diagram** to repeat the process to create the following Faces JSPs using the BankJSFTemplate.jtpl:
   - BankJSFCustomerDetails.jsp
   - BankJSFAccountDetails.jsp
6. Save the Web diagram.

Once you have created all the Faces JSPs, the Web diagram should look similar to Figure 13-28 on page 705. Notice the Web page icons now have color, and the icon titles are shown in bold. This is because they have been realized.

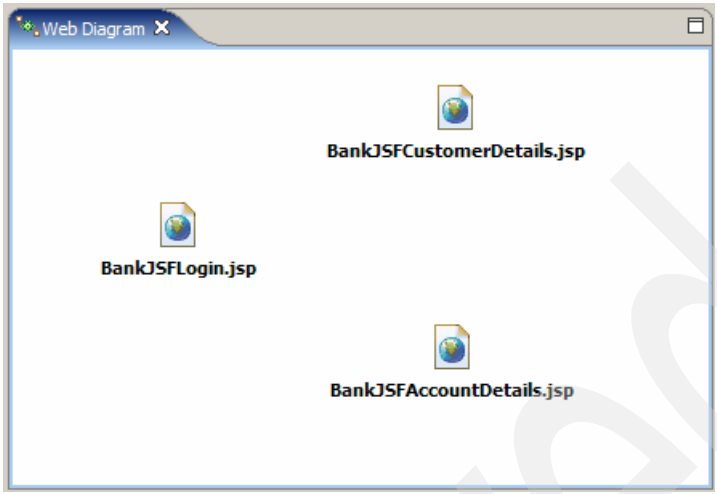


Figure 13-28 Realized Faces JSPs for the sample application

Create connections between JSF pages

Now that the pages have been created, we can create connections between the pages.

To add connections between pages, do the following:

1. Click **Connection** Connection from the Palette, click **BankJSFLogin.jsp**, and then click **BankJSFCustomerDetails.jsp**.

> **Note:** The Connection palette item is not dragged to the Web diagram like the remaining palette items.

2. When the Choose a Connection dialog appears, select **Faces Outcome** → **<new>** (as seen in Figure 13-29), and then click **OK**.

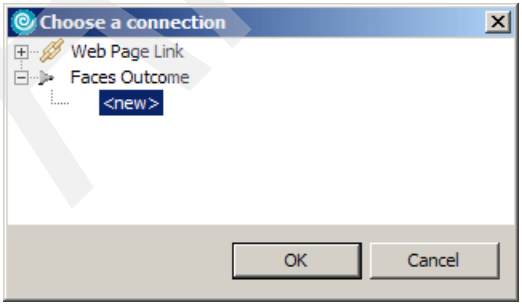


Figure 13-29 Choose a connection dialog

Archived

3. An arrow with a dotted line with the label of <new> will be drawn from the BankJSFLogin icon to the BankJSFCustomerDetails icon. The line is dotted because the connection has not been realized (added to the faces-config.xml). Once realized, the line will become solid.

   Click the **<new>** label to change the value of the outcome, and enter `login`. Alternatively, you can select the line and change the value in the properties window.

4. Realize the connection.

   a. Double-click the line.

      Alternatively, right-click the line and select **Create outcome**.

   b. When the Edit Navigation Rule appears, do the following (as seen in Figure 13-30), and then click **OK**:

      - From page: `/BankJSFLogin.jsp`
      - To page: `/BankJSFCustomerDetails.jsp`
      - When action returns outcome: Select **The outcome named**, enter `login`.
      - This rule is used by: Select **Any action**.
      - When following this rule: Select **Use request forwarding (parameters work automatically)**.

Archived

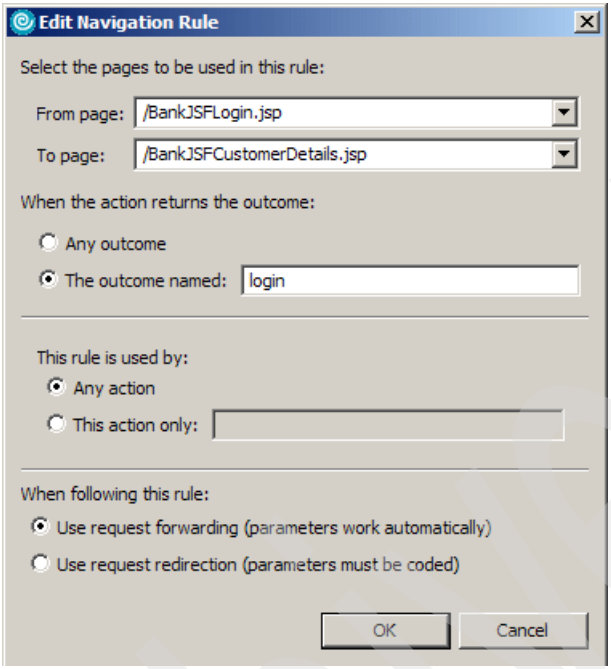


Figure 13-30 Edit Navigation Rule dialog

5. Review the modified faces-config.xml.

   Now we have a link from the BankJSFLogin.jsp to the BankJSFCustomerDetails.jsp. This link will be activated when the outcome of an action on the BankJSFLogin page is login. To review how this link is realized in the JSF configuration, do the following:

   a. Expand **Dynamic Web Project** → **BankJSF** → **WebContent** → **WEB-INF**.

   b. Double-click **face-config.xml** to open the file.

      Verify the navigation rule was added as displayed in Example 13-1.

      The connection in the Web Diagram view should now be a solid line and the Web diagram should look similar to Figure 13-31.

Example 13-1 New navigation rule added to faces-config.xml (snippet)

```
<navigation-rule>
   <from-view-id>/BankJSFLogin.jsp</from-view-id>
   <navigation-case>
      <from-outcome>login</from-outcome>
      <to-view-id>/BankJSFCustomerDetails.jsp</to-view-id>
   </navigation-case>
```

```
</navigation-rule>
```

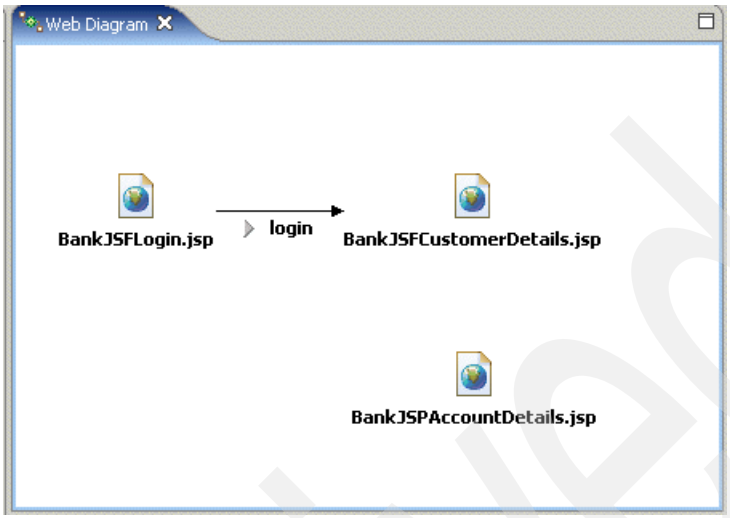


Figure 13-31 Web diagram with a realized connection

Create Faces action

The Web Diagram tool provides the means to create faces actions and connect Web pages to these actions.

To create a new Faces action, do the following:

1. From the Faces Parts palette select **Faces Action** Faces Action and drag it onto the page.
2. When the Faces Action Attributes dialog appears, enter `Logout` in the Managed Bean field, enter `logout` in the Method field (as seen in Figure 13-32), and then click **OK**.

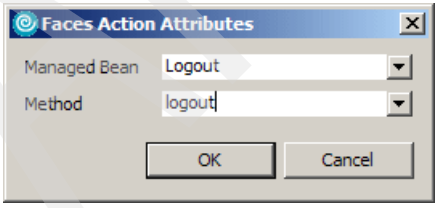


Figure 13-32 Faces Action Attributes dialog

3. Create the bean class.

Archived

Again you will notice that the new icon is lacking in color. This is because the Faces Action bean is linked to an actual class or is unrealized. Double-click the **Logout.logout** icon to create the bean class.

4. When the New Faces Managed Bean dialog appears, the *manage bean name* will be populated for you, since this is the first we are creating.

   Select the **New managed-bean class** option to create a new managed bean, and ensure that the selected Model is `managed-bean-class extends Action` (as seen in Figure 13-33 on page 709), and then click **Next**.

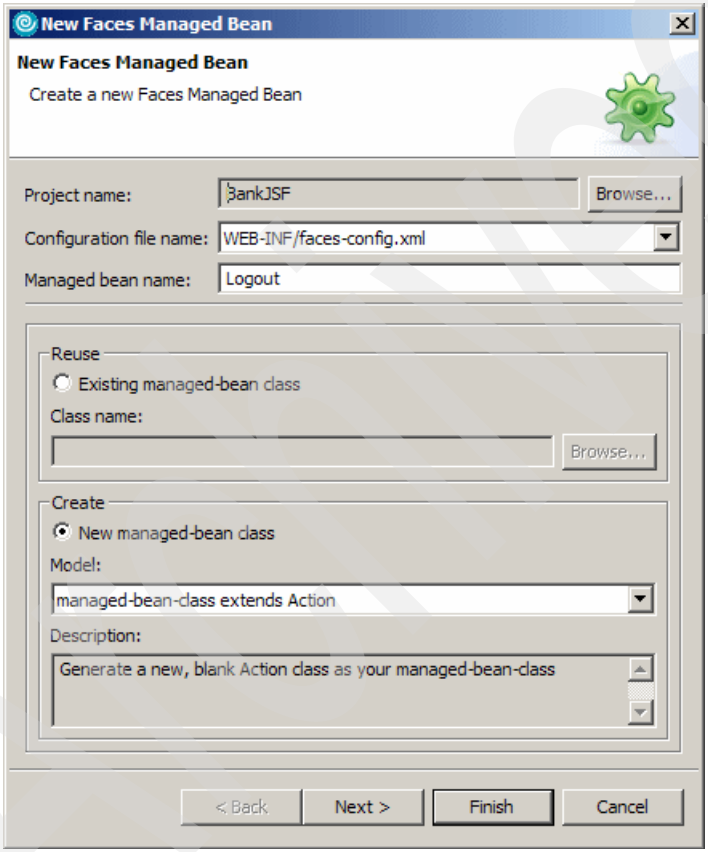


Figure 13-33 New Managed Bean wizard

5. When the Specify the managed-bean class you wish to generate dialog appears, enter `com.ibm.bankjsf.actions` in the Java package field (as displayed in Figure 13-34 on page 710), and click **Finish**.

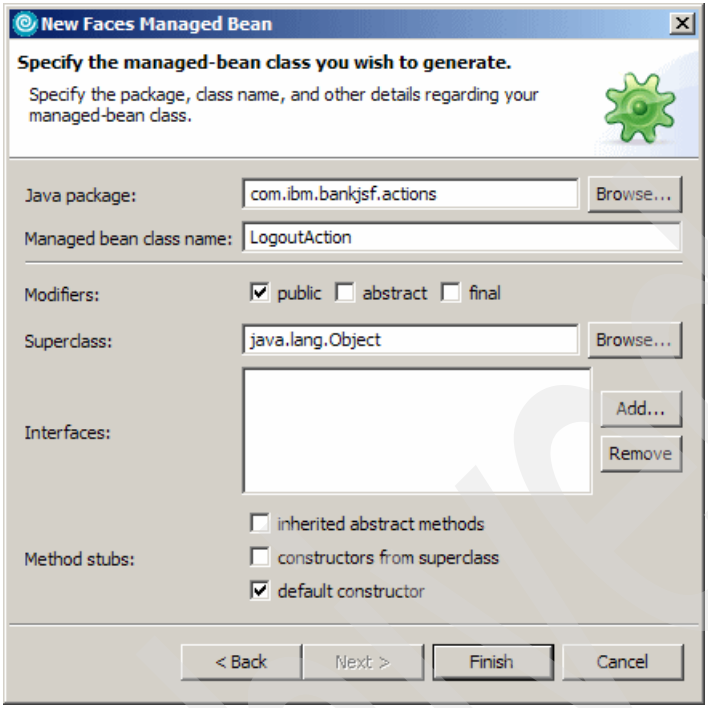


Figure 13-34 New Faces Managed Bean class details

6. The new action class will be displayed in the editor. Enter the source code found in Example 13-2 to the relevant sections of the LogoutAction class.

> **Note:** To save time entering, the source code can be copied from c:\6449code\jsf\BankJSFSource.jpage.

Example 13-2 ITSO sample LogoutAction.java

```
package com.ibm.bankjsf.actions;

/** Import statements **/
import java.util.Map;
import javax.faces.context.*;

public class LogoutAction {

    /** Constructor **/
    public LogoutAction() {

    }
```

```
    /** Member variables **/
    protected FacesContext facesContext;
    protected Map sessionScope;

    private static final String CUSTOMERSSN_KEY = "customerSSN";
    private static final String SESSION_SCOPE = "#{sessionScope}";
    private static final String OUTCOME_LOGOUT = "logout";


    /** logout method **/
    public String logout(){
        facesContext = FacesContext.getCurrentInstance();
        sessionScope =
            (Map) facesContext
                .getApplication()
                .createValueBinding(SESSION_SCOPE)
                .getValue(facesContext);

        if (sessionScope.containsKey(CUSTOMERSSN_KEY)){
            sessionScope.remove(CUSTOMERSSN_KEY);
        }

        return OUTCOME_LOGOUT;
    }
}
```

The Web Diagram should now look similar to Figure 13-35.

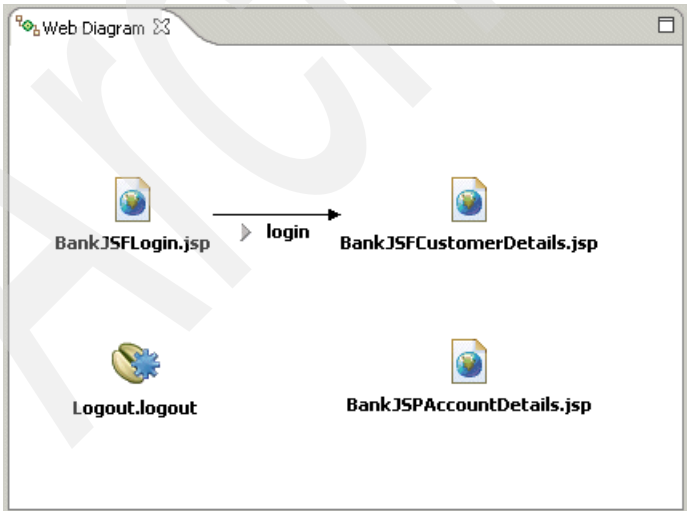


Figure 13-35 Logout bean in the Web diagram

Archived

Add a connection for the action

To add a connection from the Action to a page, do the following:

1. Select Connection

from the palette, click the **Logout.logout** icon, and then click on the **BankJSFLogin.jsp** icon.
2. Select **<new>** on the dotted line between Logout.logout and BankJSFLogin.jsp, and rename the outcome to `logout`.
3. Double-click **logout** on the dotted line between Logout.logout and BankJSFLogin.jsp to realize the connection.
4. When the Edit Navigation Rule dialog appears, do the following, as seen in Figure 13-36, and then click **OK**:
   a. Enter /* in the From Page field (the * makes the navigation rule applicable for all pages).
   b. Ensure that **Any action** is selected.

> **Note:** At the time of writing, the settings for this navigation rule would change if the logout line was subsequently double-clicked. The settings would then reflect those of Figure 13-37 on page 713. Note how the selection has changed from `Any action` to `This action only`.

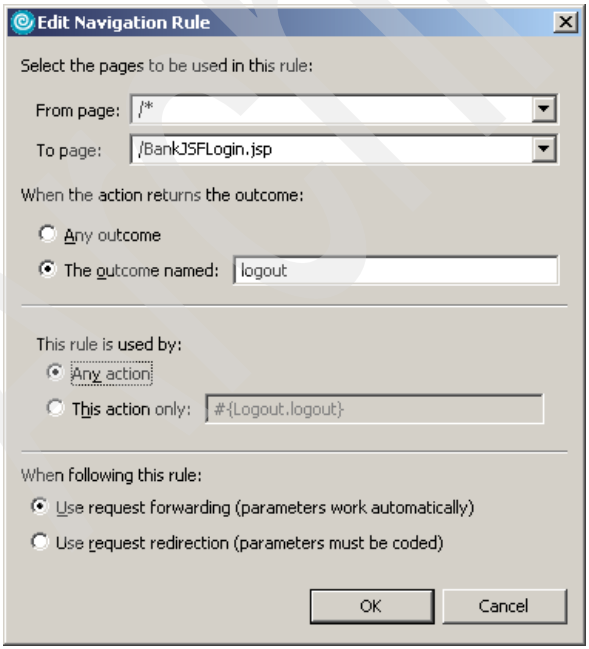


Figure 13-36 Realize global logout navigation rule

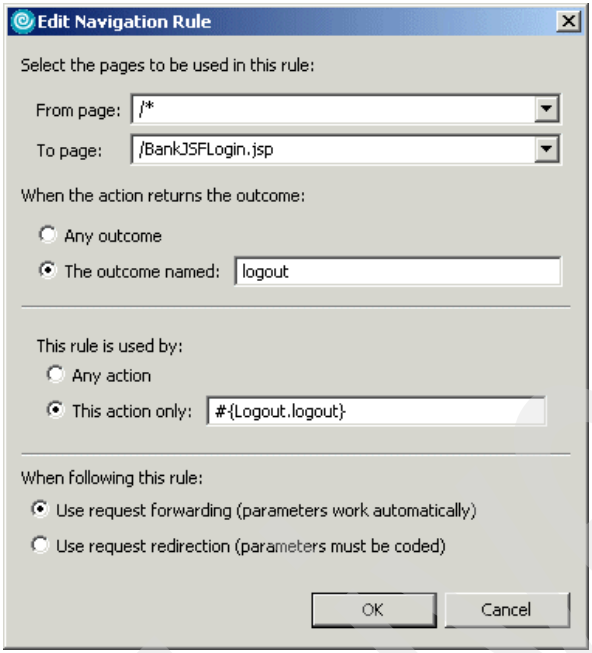


Figure 13-37 Wrong settings for the logout navigation rule

This will result in the new navigation rule being added to the faces-config.xml file, as shown in Example 13-3.

Example 13-3 New navigation rule added to faces-config.xml for the logout action

```
<navigation-rule>
   <from-view-id>/*</from-view-id>
   <navigation-case>
      <from-outcome>logout</from-outcome>
      <to-view-id>/BankJSFLogin.jsp</to-view-id>
   </navigation-case>
</navigation-rule>
```

Add remaining navigation rules

We now have an action bean that will perform some action and return the user to the BankKJSFLogin screen.

1. Add the navigation rules listed in Table 13-2 on page 714, following the previously described procedures. Note that the navigation rules that point to the logout action do not require any outcome information.

Table 13-2 Navigation rules and connections

From	To	Outcome
BankJSFCustomerDetails.jsp	BankJSFAccountDetails.jsp	accountDetails
BankJSFAccountDetails.jsp	BankJSFCustomerDetails.jsp	customerDetails
BankJSFCustomerDetails.jsp	Logout.logout	
BankJSFAccountDetails.jsp	Logout.logout	

2. Once you have added the connections the Web Diagram should look similar to Figure 13-38.

Tip: To change the shape of a connection, select the connection, and in the center of the straight line you will see an anchor point (a small square). You can drag the anchor to change the shape of the connection for better readability.

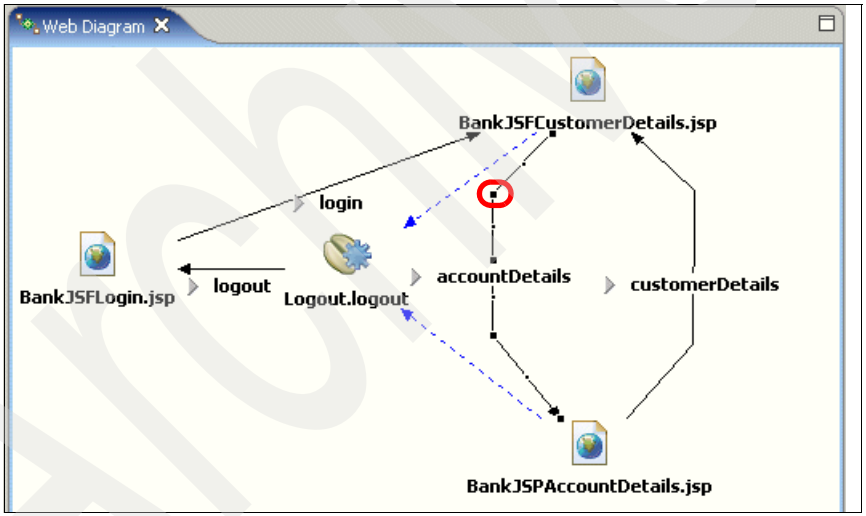


Figure 13-38 Completed Web diagram

Other useful buttons for editing the Web diagram are shown in Figure 13-39 on page 715.

- Align selected objects horizontally.
- Align selected object vertically.
- Zoom in and out.
- Configure the diagram setting such as grid line and snap to grid.
- Filter certain types of objects from view.

Archive

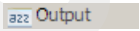
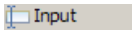
Figure 13-39 Web diagram menu bar

13.3.5 Edit a JSF page

This section demonstrates the JSF editing features in Application Developer by using the JSF pages created in 13.3.4, “Create JSF resources using the Web Diagram tool” on page 700.

Add the UI components

To add the UI components to the BankJSFLogin.jsp page, do the following:

1. Open **BankJSFLogin.jsp** by double-clicking the file in the Web diagram or in the WebContent folder.
2. Click the **Design** tab.
3. Select the text **Default content of bodyarea**, right-click, and select **Delete**.
4. From the Faces Components palette select **Output**  and drag it into the content area.
5. In the Properties view for the new output component, do the following:
   a. Enter `Login` into the Value field.
   b. Click the icon next to Style: Props: to change the font and size.
      i. Select the Font size **14**.
      ii. Select **Arial** font, and click **Add**.
      iii. Select **sans-serif** font, and click **Add**.
      iv. Click **OK**.
6. Place the cursor right after the Output component and press Enter.
7. Enter the text `Enter Customer SSN:`. Select the text you have just entered and use the Properties view to change the font to Arial.
8. From the Faces Components select **Input**  and drag it next to the text entered in the previous step.
9. In the Properties view of the input component, enter `customer` in the Id field.
10. From the Faces Components select the **Command - Button** and drag it to the right of the Input component.
11. In the Properties view for the command button select the **Display options** tab, and in the Button label type `Enter`, as seen in Figure 13-40 on page 716.

Archived

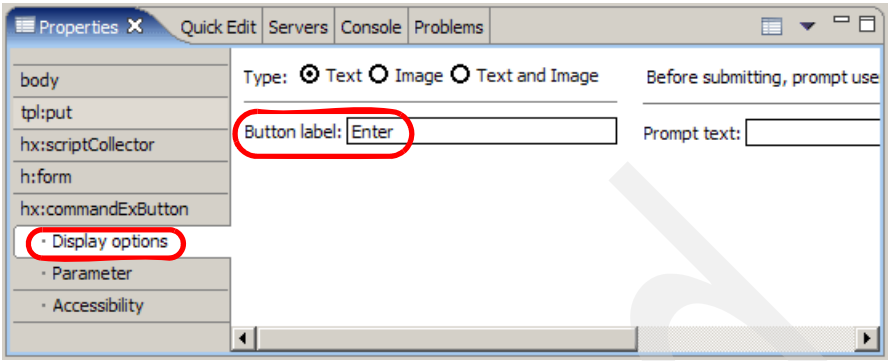


Figure 13-40 Command button Properties view

12. Save the BankJSFLogin.jsp file (Ctrl+S).
13. Click the **Preview** tab.

    The resulting page should be similar to Figure 13-41.

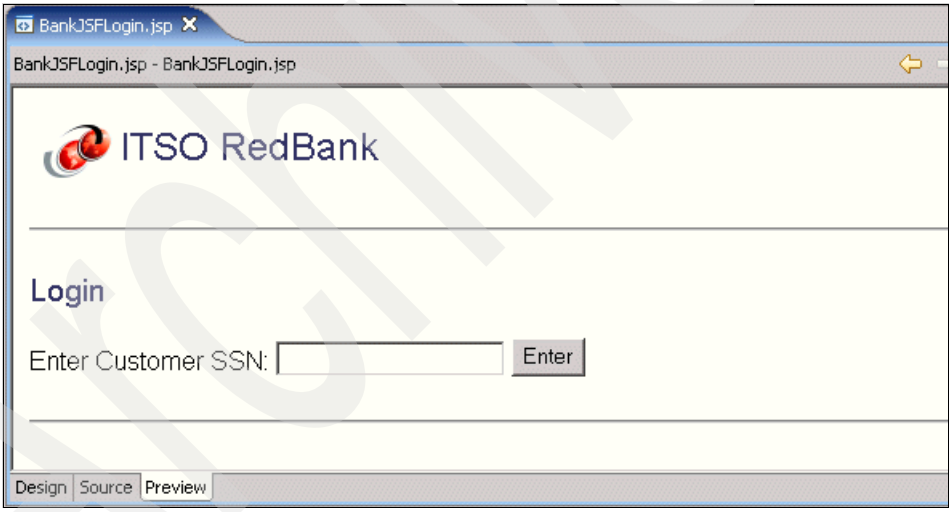


Figure 13-41 Preview of the BankJSFLogin.jsp

Add variables

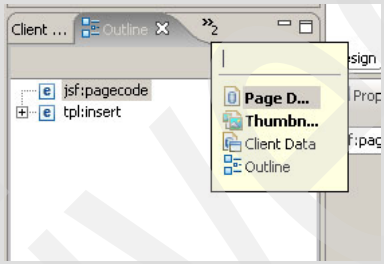
When a page has a field where text is entered, the input needs can be stored. This is accomplished by creating a session scope variable to store the entered value and bind it with the input field created.

Archived

To create a variable and bind it to the input field, do the following:

1. Double-click **BankJSFLogin.jsp** and change to the Design view if necessary.
2. In the Page Data view click the Create new data component button ( ) (upper right of view).

Tip: The Page Data view may be collapsed. If this is the case, click the Show List icon on the right of the other tabs in the panel where the view is normally placed (this is typically in the lower-left part of the Workbench) and select **Page Data** from the list.



3. When the Add Data Component dialog appears, select the **Session Scope Variable**, and then click **OK**.
4. When the Add Session Scope Variable dialog appears, enter `customerSSN` in the Variable name field, enter `java.lang.String` in the Type field (as seen in Figure 13-42), and then click **OK**.

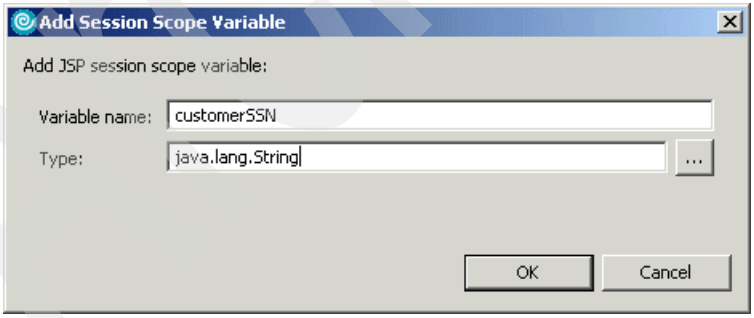


Figure 13-42 Add session scope variable

5. From the Page Data view, expand and select **Scripting Variables** → **sessionScope** → **customerSSN**. Drag customerSSN to the input component. The help-tip will show the text `Drop here to bind “customerSSN” to the control customerId`.

6. Save BankJSFLogin.jsp.

BankJSFLogin.jsp should look like Figure 13-43 from the Design view. Notice that the input field is now bound to the session scope variable customerSSN, as denoted by the text `{customerSSN}`.

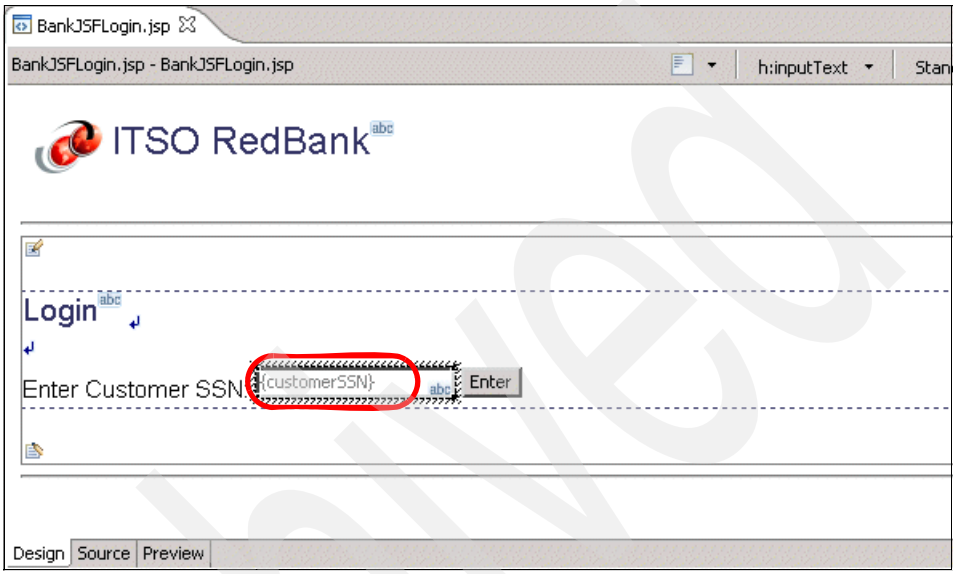


Figure 13-43 Design view - Bind input field and session scope variable

Add simple validation

JSF offers a framework for performing common validation on input fields such as required values, validation on the length of the input, and input check for all alphabetic or digits.

To add simple validation to an input field, do the following:

1. Select the Input component in the Design view.
2. In the Properties view for the input component select the **Validation** tab.
3. Enter the following in the Validation tab, as seen in Figure 13-44 on page 719:
   a. Check **Value is required**.
   b. Check **Display validation error message in an error message control**. When you select the second checkbox an error message component will be added next to the Input box.
   c. Enter 11 in the Minimum length field.

Archived

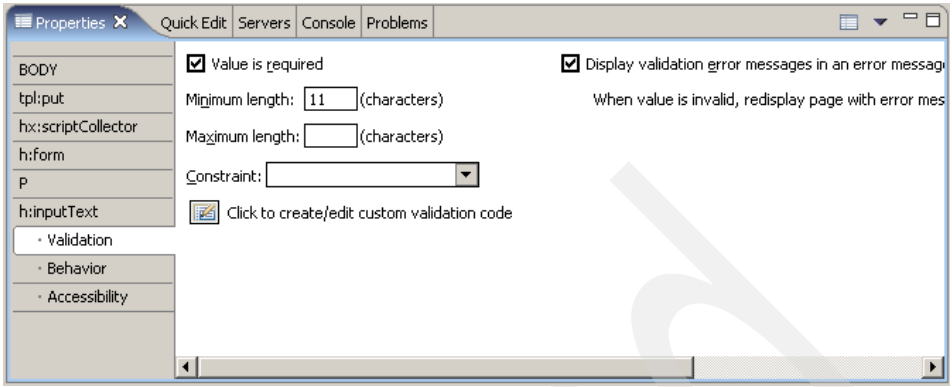


Figure 13-44 Input box validation properties

We have now added some simple validation for the input component.

Add static navigation to a page

To add static navigation to page, do the following:

1. Double-click the **BankJSFLogin.jsp** to open in the editor.
2. Select the **Command Button** component (Enter).
3. In the Properties view, click the All Attributes ( ) icon found in the top right.

   The Properties view will be changed to show all the attributes of the selected component, as seen in Figure 13-45.

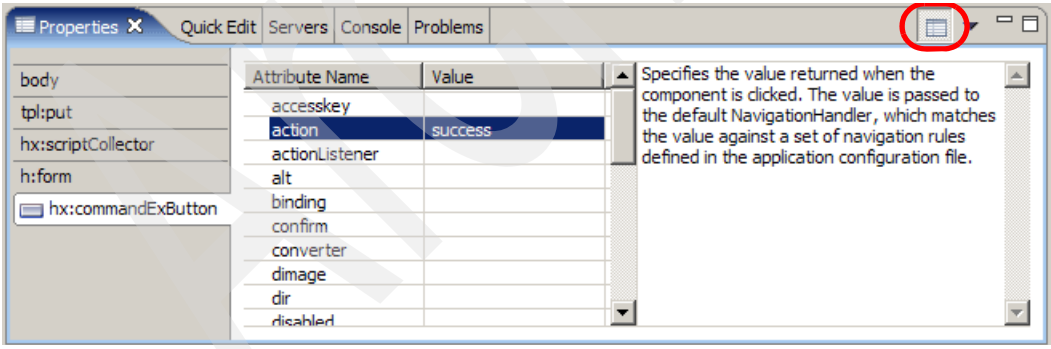


Figure 13-45 Properties view showing all attributes

4. Select the **action** attribute, and enter `success`, as seen in Figure 13-45.

   This will be the outcome of the page when the Command Button (Enter) is pressed, and as a result the navigation rule that we created will be invoked.

Archived

5. Save the BankJSFLogin.jsp.

Now that we have set up static navigation, we need to compare the entered customer SSN to the values in the database. We will do so by creating an SDO object to retrieve the records from the database.

Add SDO to a JSF page

To add an SDO relational record to a JSF page, do the following:

1. Ensure you have set up the sample database as described in 13.2.2, “Set up the sample database” on page 681. We will use the *Bank Connection* as part of the following procedure.
2. Ensure the BankJSF_Con1 connection is connected so that it can access the c:\databases\BANK database.
3. Double-click the **BankJSFLogin.jsp** to open it in the editor.
4. Right-click the **Page Data** view, and select **New** → **Relational Record List**.
5. When the Add Relational Record List dialog appears, enter `customers` in the Name field (as shown in Figure 13-46), and then click **Next**.

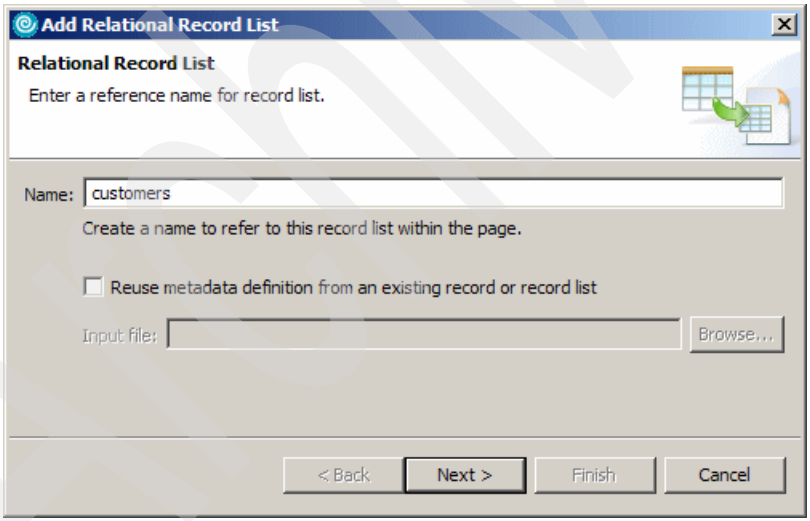


Figure 13-46 Add Relational Record List wizard

6. When the Record List Properties dialog appears, do the following:
   a. The first time this dialog appears the Connection name drop-down list will be empty even though the actual database connection named Bank Connection exists. Click **New**.

b. When the Select a Database dialog appears, select **Use Live Connection**, select either BankJSF_Con1 connection or the BANK database for the BankJSF_Con1 connection (as seen in Figure 13-47), and then click **Finish**.

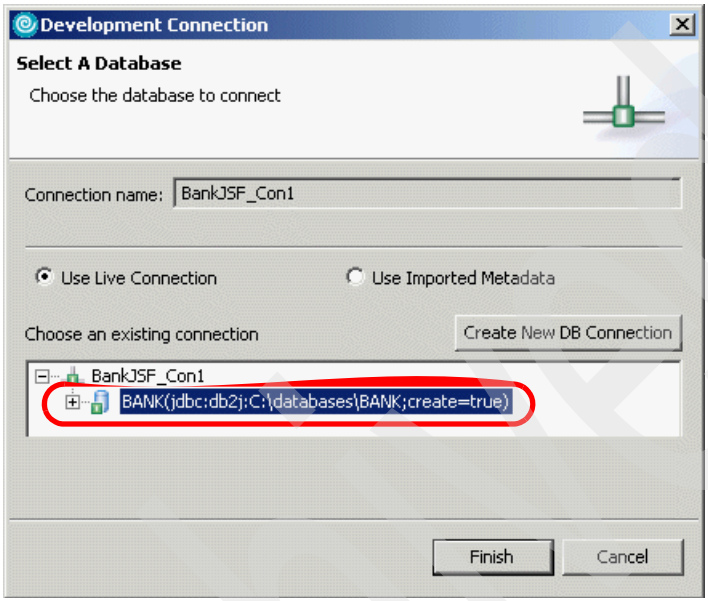


Figure 13-47 Add SDO to JSF page - Select a database

> **Note:** In our example, the BANK database did not always show up under the BankJSF_Con1 connection. We found that in these cases, we could just select the BankJSF_Con1 connection.

c. As a result of selecting the database in the previous step, the Record List Properties page will be populated with the database tables. Select the **ITSO.CUSTOMER** table (as seen in Figure 13-48 on page 722), and then click **Next**.

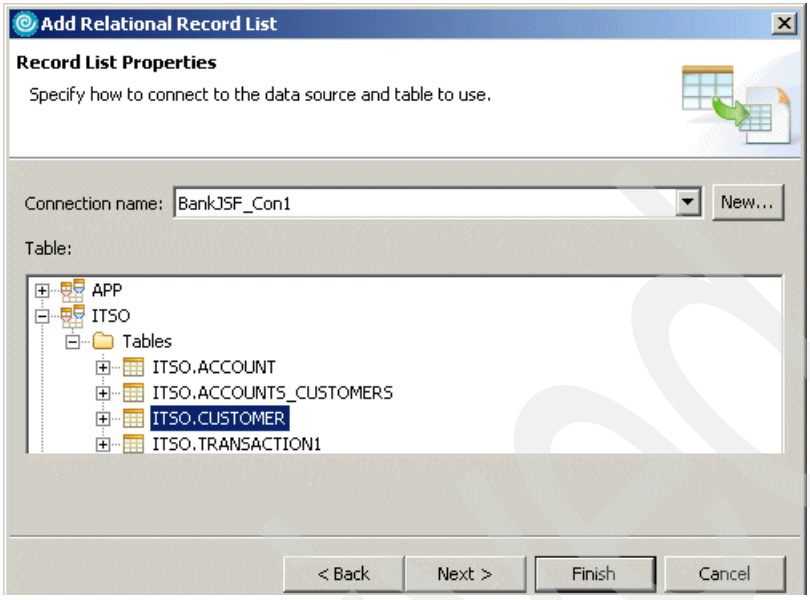


Figure 13-48 Select a Connection and Table

7. When the Column Selection and Other Task dialog appears, do the following (as seen in Figure 13-49 on page 723), and then click the **Filter results** link:
   - Check **TITLE**.
   - Check **FIRSTNAME**.
   - Check **LASTNAME**.

Archive

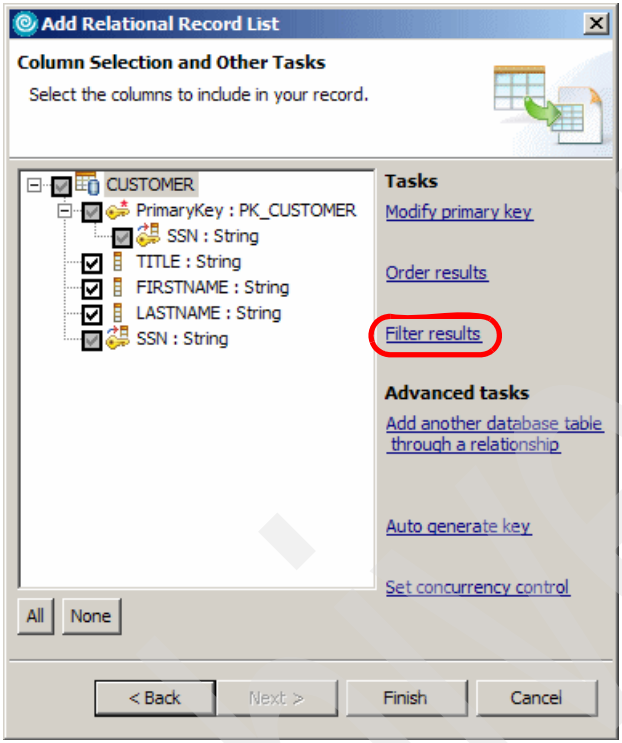


Figure 13-49 Column selection

8. When the Filters dialog appears, click **+** (add) to add a new filter rule, as shown in Figure 13-50 on page 724.

Archived

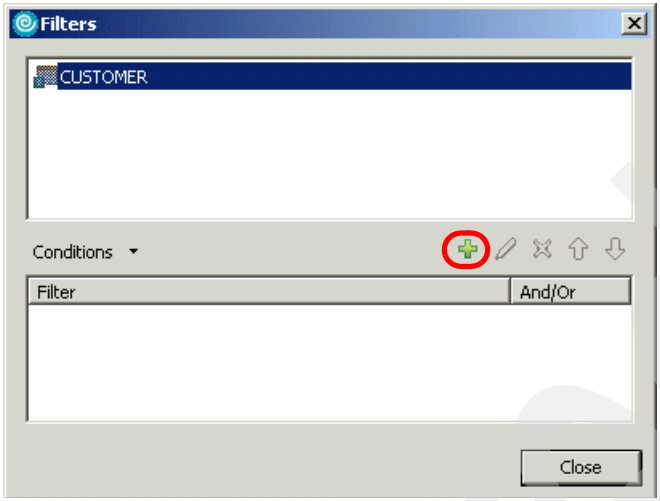


Figure 13-50 Add new filter

9. When the Conditions dialog appears, select **SSN** from the Column drop-down list, and click the **....** (more) button to the right of Value, as seen in Figure 13-51.

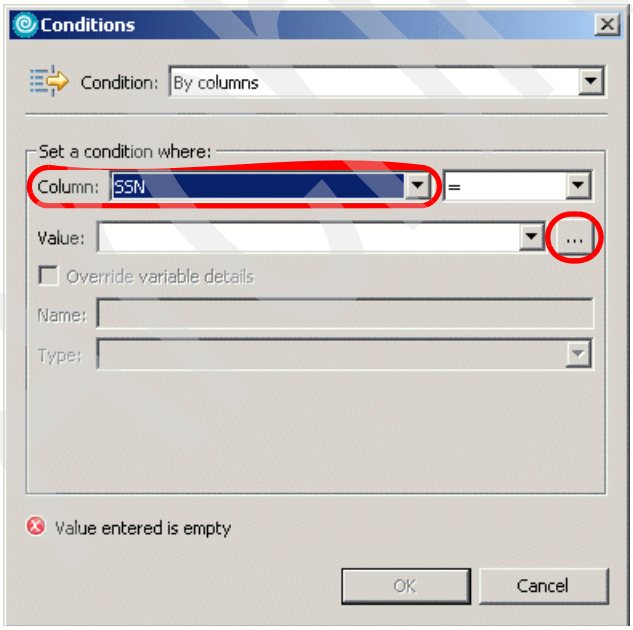


Figure 13-51 The Conditions dialog

10. When the Select Page Data Object dialog appears, expand **sessionScope**, select **customerSSN** (as seen Figure 13-52), and then click **OK**.

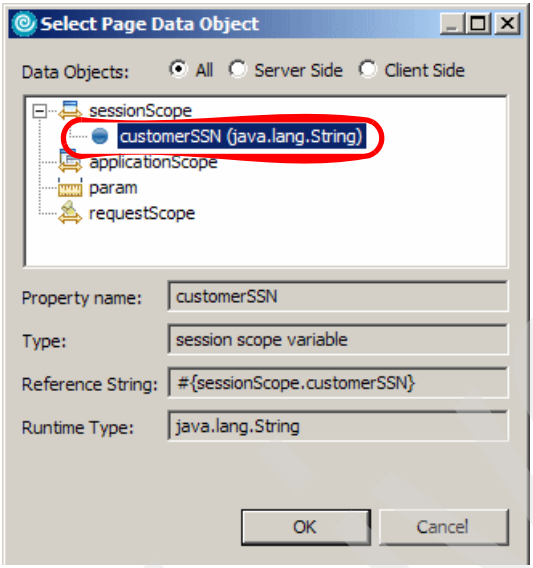


Figure 13-52 Select Page Data Object dialog

11. When the Conditions dialog appears, the proper values are displayed, as seen in Figure 13-53 on page 726. Click **OK**.

Archived

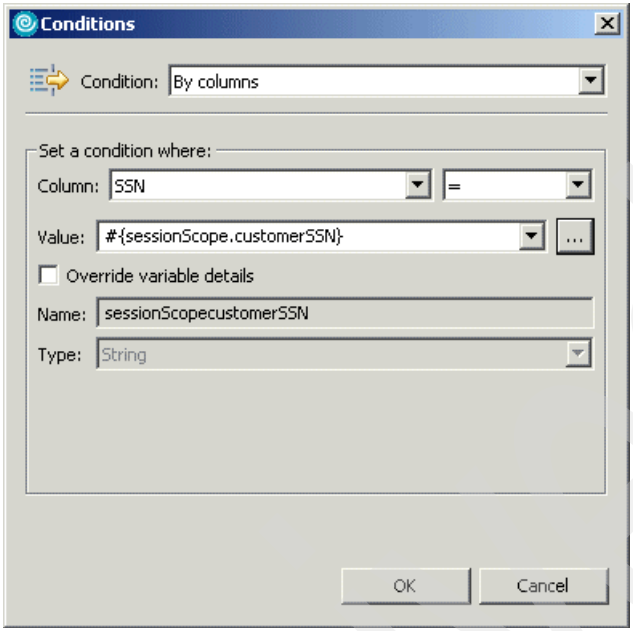


Figure 13-53 Conditions dialog with values

12. The Filters dialog should now look like Figure 13-54. Click **Close**.

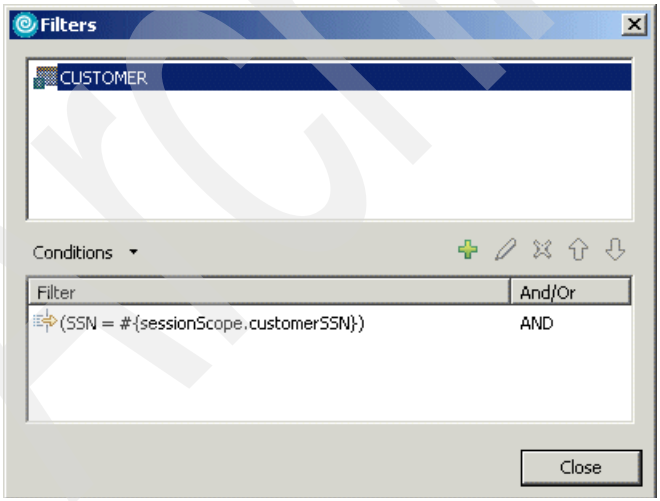


Figure 13-54 Filter dialog with values

13. When you return to the Column Selection and Other Tasks dialog, click **Finish**.

14. Save the BankJSPLogin.jsp.

At this point, we have added an SDO Relational Record List object to the BankJSFLogin.jsp, and we can use the object to validate the customer SSN that is entered.

As a result of creating the relational record list, several files were generated. The customer.xml file is of most interest to us in that it contains the configuration details entered in the wizard.

```
/WebContent/WEB-INF/wdo/customers.xml
```

Add custom validation and dynamic navigation

To add custom validation and dynamic navigation we will have to add some Java code to our managed bean.

1. Expand **Dynamic Web Project** → **BankJSF** → **Java Resources** → **JavaSource** → **pagecode**.
2. Double-click **BankJSFLogin.java** to open the file in an editor.
3. Insert the code in Example 13-4 to the BankJSFLogin.java at the end of the file.

   Note: To save time, the source code can be copied from c:\6449code\jsf\BankJSFSource.jpage.

4. Add an import statement for ArrayList used by the code inserted in the previous step. Press Ctrl+Shift+O (Organize Imports) to add the following import.

   ```
   import java.util.ArrayList;
   ```

5. Save the BankJSFLogin.java file.

Example 13-4 Custom validation code to check customer's SSN

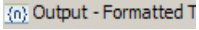
```
public static final String OUTCOME_FAILED = "failed";
public static final String OUTCOME_LOGIN = "login";

public String login(){
   String outcome = OUTCOME_FAILED;

   List custs = getCustomers();

   // A valid customer SSN should only return one record
   if (custs == null || custs.size() < 1) {
      addErrorMessage("Customer Record not found.");
   } else if (custs.size() == 1){
```

Archived

```
            // successfully entered a valid Customer SSN
            outcome = OUTCOME_LOGIN;
        } else {
            addErrorMessage("To many customer records found.");
        }
        return outcome;
    }

    protected void addErrorMessage(String error){
        if (errorMessages == null){
            errorMessages = new ArrayList();
        }

        errorMessages.add(error);
    }

    public String getErrorMessages(){
        StringBuffer messages = new StringBuffer();

        if (errorMessages != null && errorMessages.size() > 0){
            int size = errorMessages.size();
            for (int i=0;i<size;i++){
                messages.append(errorMessages.get(i));
            }
        }

        return messages.toString();
    }
    private ArrayList errorMessages = null;
```

Test the custom validation and dynamic navigation

To test the custom validation and dynamic navigation code, do the following:

1. Open the Web perspective Project Explorer view.
2. Expand **Dynamic Web Project** → **BankJSF** → **WebContent**.
3. Double-click **BankJSFLogin.jsp** to open it in the editor.
4. In the Design view, select the Edit button (Id button1).
5. In the Page Data view, expand **Actions**.
6. Right-click **login()** and select **Bind to 'button1'** from the context menu, where button1 is the ID of the Enter button.
7. From the Palette, expand **Faces Components**.
8. From the Faces Components Palette, select **Output - Formatted Text**  and drag it to the area between `Login` and `Enter Customer SSN` texts.

Archived

9. Select the **outputFormat** component in the Design view.
10. In the Properties view for the new outputFormat component, do the following:
    a. Enter `errorMessage` in the Id field.
    b. Click the icon next to Style: Props: to change the font, color, and size.
        i. Select the Font size **12**.
        ii. Select **Arial** font and click **Add**.
        iii. Select **sans-serif** font and click **Add**.
        iv. Select **Red** in the Color drop-down field.
        v. Click **OK**.
11. In the Page Data view, right-click the **errorMessages** bean and select **Bind to 'errorMessage'** from the context menu, where format1 is the ID of the output formatted text component.

Important: At the time of writing, the procedure described here will result in invalid code being generated for BankJSFLogin.jsp. The following is the code that has been added to BankJSFLogin.jsp:

```
<f:param id="param1" name="msg1" value="#{pc_BankJSFLogin.errorMessages.}">
</f:param>
```

This code instructs JSF to show the contents of the variable #{pc_BankJSFLogin.errorMessages.} in the outputFormat component.

Unfortunately, there is no such variable. The correct name is #{pc_BankJSFLogin.errorMessages} without the dot. It is thus necessary to change the code in BankJSFLogin.jsp to:

```
<f:param id="param1" name="msg1" value="#{pc_BankJSFLogin.errorMessages}">
</f:param>
```

This is done in the Source view. After modifying the code, save BankJSFLogin.jsp.

Now we have bound the Command button to run the login method, which will get the customers from the database using the SDO Relational Record List. It will check the number of records found. If it is not equal to 1 then it will return an error message and we will come back to the login screen. If it returns only one record, the login method will return an outcome of success. As this outcome is linked to a navigation rule, that navigation rule will be invoked and will present the customer details screen.

13.3.6 Completing the SDO example

This section describes how to complete the sample application by using SDO with the pages to display and access the database.

Complete BankJSFCustomerDetails.jsp

We will complete the following tasks to complete the BankJSFCustomerDetails.jsp to allow the user to update the customer details:

- Add relational record.
- Add the relational record component to the page.
- Link update button to doCustomerUpdateAction (database update).
- Add relational record list to display accounts.
- Add the relational record list to the page.
- Change the formatting of the balance field.
- Add a row action.
- Add reusable JavaBean.
- Add logout method.
- Add a logout button.

Add relational record

To add a relational record, do the following:

1. Double-click **BankJSFCustomerDetails.jsp** to open it in the editor.
2. In the Page Data view, click the Create new data component ( ) button, select **Relational Record**, and then click **OK**.
3. When the Add Relational Record dialog appears, enter `customer` in the Name filed (as seen in Figure 13-55 on page 731), and click **Next**.

Archived

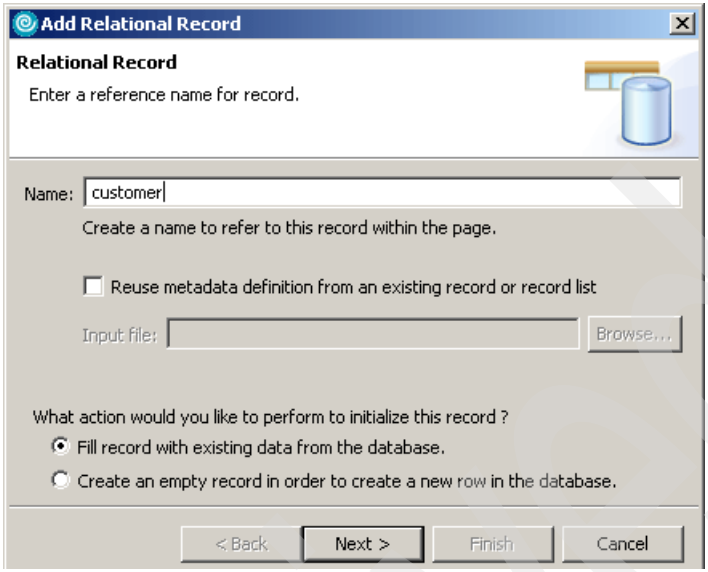


Figure 13-55 Adding a Relational Record

4. Select the **ITSO.CUSTOMER** table, as seen in Figure 13-56, and click **Next**.

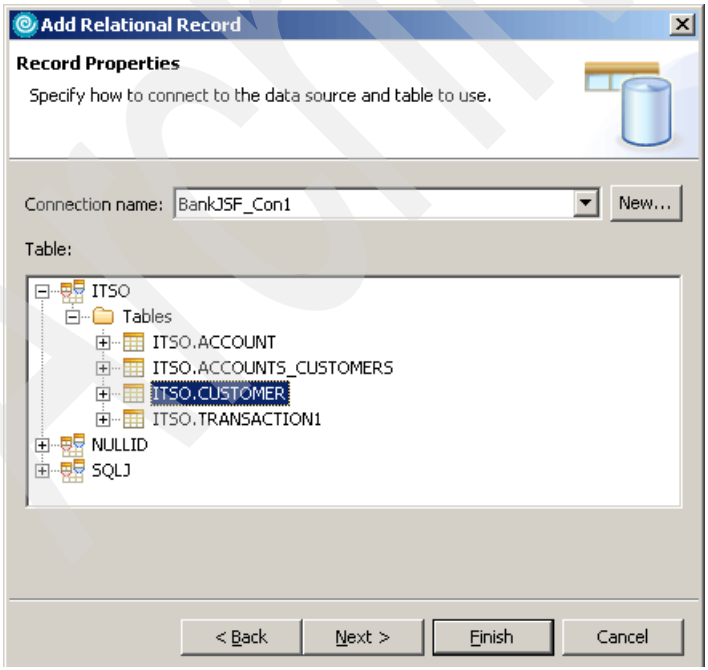


Figure 13-56 Selecting the table for the Relational Record

5. When the Column Selection and Other tasks dialog appears, click **Filter Results**.
6. When the Filters dialog appears, select the filter rule as seen in Figure 13-57, and click the Edit button ( ).

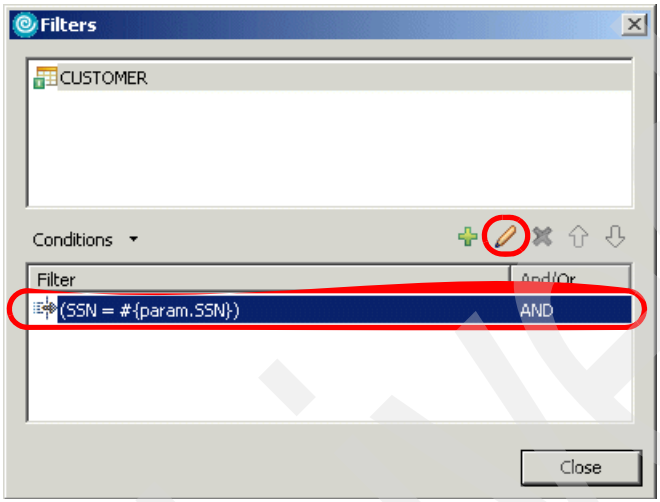


Figure 13-57 Filter rule selection

7. When the Conditions dialog appears, click the more button ( ) next to the Value field, as seen in Figure 13-58 on page 733.
8. When the Select Page Data Object dialog appears, expand **sessionScope**, select the **customerSSN**, and click **OK**.

Archived

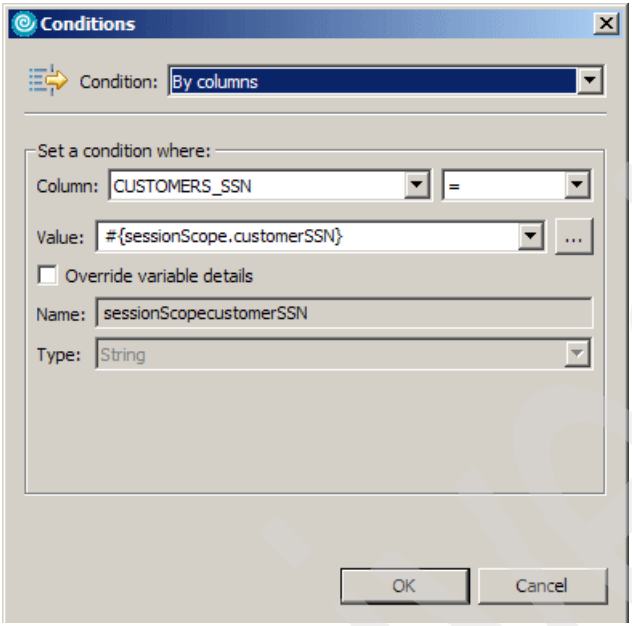


Figure 13-58 Add sessionScope variable for customerSSN

9. When you return to the Conditions dialog, the values should be set as seen in Figure 13-58. Click **OK**.
10. When you return to the Filters dialog, click **Close**.
11. When you return to the Column Selection and Other Tasks dialog, click **Finish**.

We have now created the relational record.

Add the relational record component to the page

To add the relational record component to the page, do the following:

1. In the Design view, select the text **Default content of bodyarea**, right-click, and select **Delete**.
2. In the Page Data view, expand **customer (Service Data Object)**, select the **customer (CUSTOMER)** relational record component, and drag it onto the content area of the page.
3. When the Insert Record - Configure Data Controls dialog appears, select **Updating an existing record**, uncheck the **SSN** field as seen in Figure 13-59 on page 734, and then click the **Options ...** button.

Archived

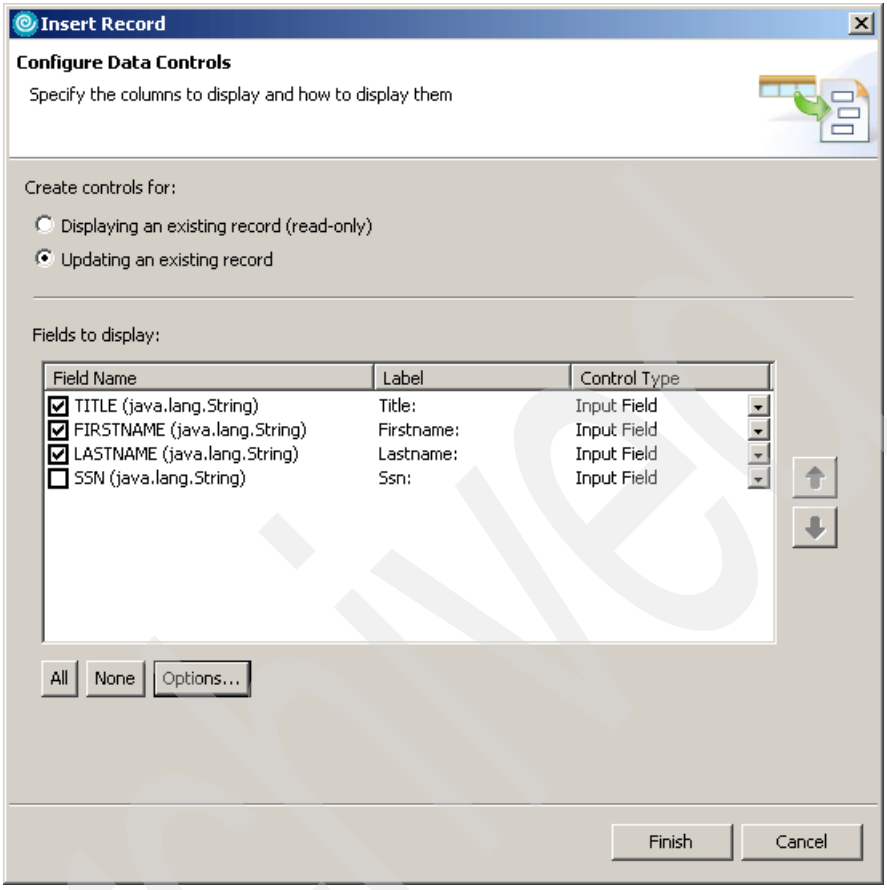


Figure 13-59 Insert Record wizard

4. When the Options dialog appears, uncheck the **Delete button** check box, enter `Update` in the Label field as seen in Figure 13-60 on page 735, and then click **OK**.

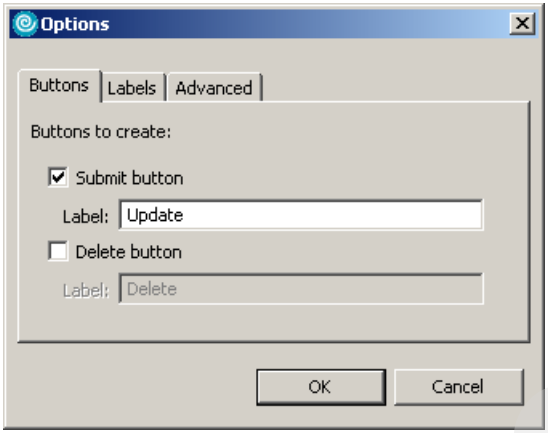


Figure 13-60 Insert Record Options dialog

5. When you return to the Insert Record - Configure Data Controls dialog, click **Finish**.
6. Save the BankJSFCustomerDetails.jsp.

The customer details can now be displayed on the page, with the input fields available to enter data.

Link update button to doCustomerUpdateAction (database update)

In this step, we provide the ability to update the database with the customer input field data, by linking the doCustomerUpdateAction method to the update button.

1. In the Design view, select the **Update** button on the page.
2. In the Page Data view, expand **customer (Service Data Object)**.
3. Right-click **doCustomerUpdateAction** and select **Bind to 'button1'** from the context menu.

Add relational record list to display accounts

In this section we demonstrate how to add the relational record list for accounts to the page, so that we can display the accounts for the customer.

To add the relational record list for the accounts, do the following:

1. In the Page Data view, click the button.
2. Select the **Relational Record List** from the New data component dialog and click **OK**.
3. When the Add Relational Record List dialog appears, enter `accounts` in the Name field and then click **Next**.

Archived

4. Select the **ITSO.ACCOUNTS_CUSTOMERS** table and then click **Next**.
5. Click the **Filter results** under the Tasks on the right of the dialog.
6. In the Filters dialog click the button.
   a. When the Conditions dialog appears, do the following:
      i. Select the **CUSTOMERS_SSN** from the Column drop-down list.
      ii. Click the **...** (more) button next to the Value field.
      iii. When the Select Page Data Object dialog appears, select the **customerSSN** session scope variable and click **OK**.
      iv. When you return to the Conditions dialog, click **OK**.
   b. When you return to the Filters dialog, click **Close**.
7. When you return to the Column Selection and Other tasks dialog, click **Add another database table through relationship** under Advanced tasks.
   a. When the Choose Your Relationship dialog appears, select the **ITSO.ACCOUNT** table (as seen in Figure 13-61 on page 737), and then click **Next**.

Archived

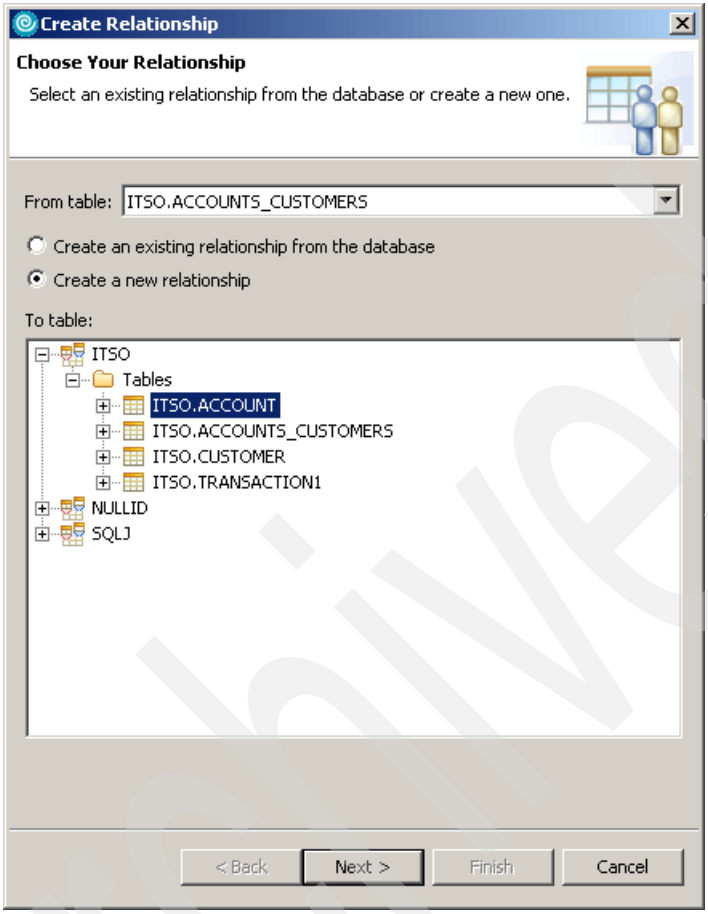


Figure 13-61 Create Relationship dialog

b. When the Edit Your Relationship dialog appears, enter the following, as seen in Figure 13-62 on page 738:

   i. Select ***->1 FOREIGN KEY -> PRIMARY KEY** from the Multiplicity drop-down list.

   ii. Select **ACCOUNTS_ID** from the ACCOUNT_CUSTOMERS primary key drop-down list.

   iii. Select **ITSO.ACCOUNTS_CUSTOMER->ITSO.ACCOUNT** in the Relationships panel to gain focus, and then click **Finish**.

> **Note:** We found that if we did not highlight this field, the Finish button was not available to click.

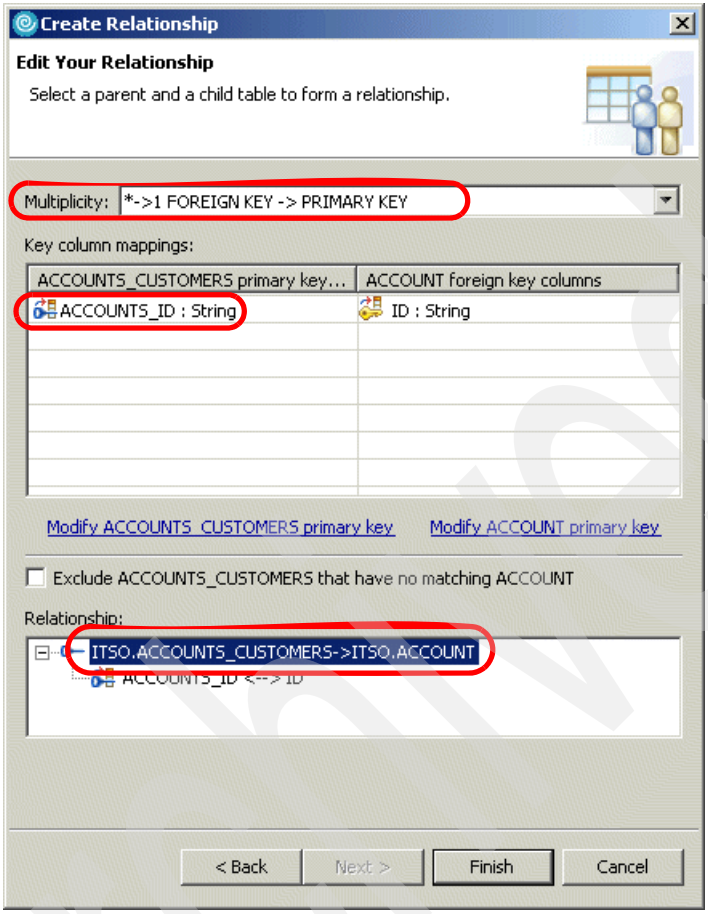


Figure 13-62 Create Relationship conditions page

8. When you return to the Column Selection and Other Tasks dialog, click **Finish**.
9. Save the BankJSFCustomerDetails.jsp.

Add the relational record list to the page

To add the relational record list to the page, do the following:

1. In the Page Data view, expand **accounts (Service Data Object)**.
2. Select the **accounts (ACCOUNT_CUSTOMERS)** relational record list component and drag it onto the area to the right of the Update button.

3. When the Insert Record List - Configure Data Control dialog appears, do the following, as seen in Figure 13-63 on page 739:
   a. In the Label field for the ACCOUNTS_ID enter `Account ID`.
   b. Uncheck **CUSTOMERS_SSN**.
   c. Uncheck **ACCOUNTS_CUSTOMERS_ACCOUNT.ID**.
   d. Click **Finish**.

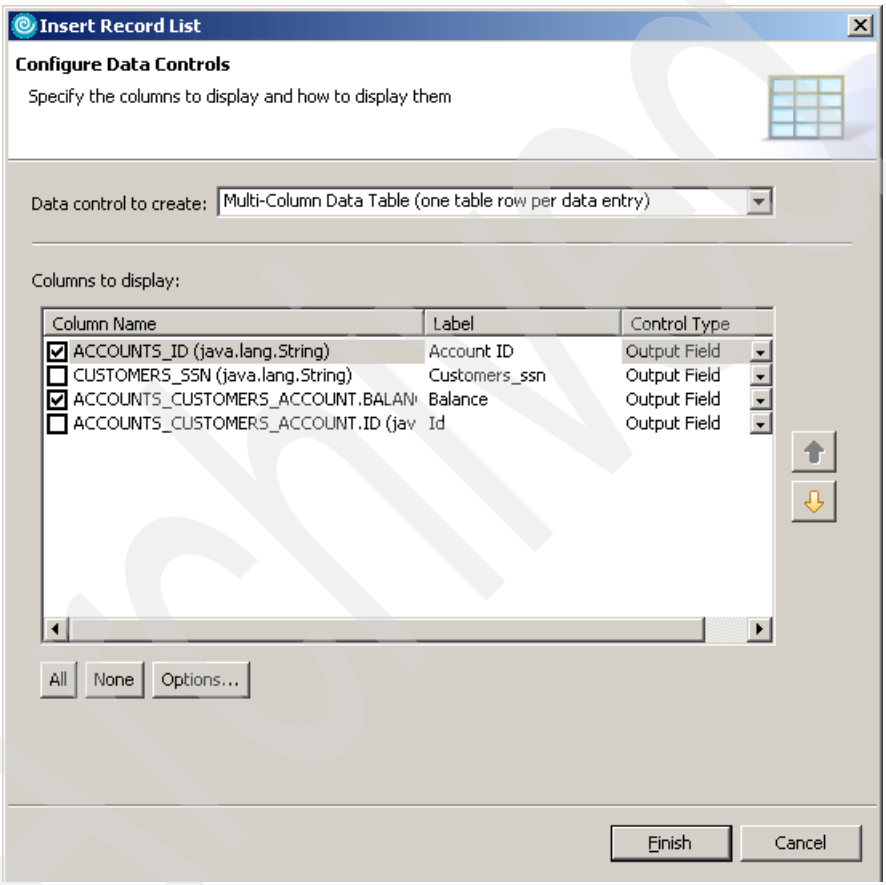


Figure 13-63 Insert Record List

4. Save the BankJSFCustomerDetails.jsp.

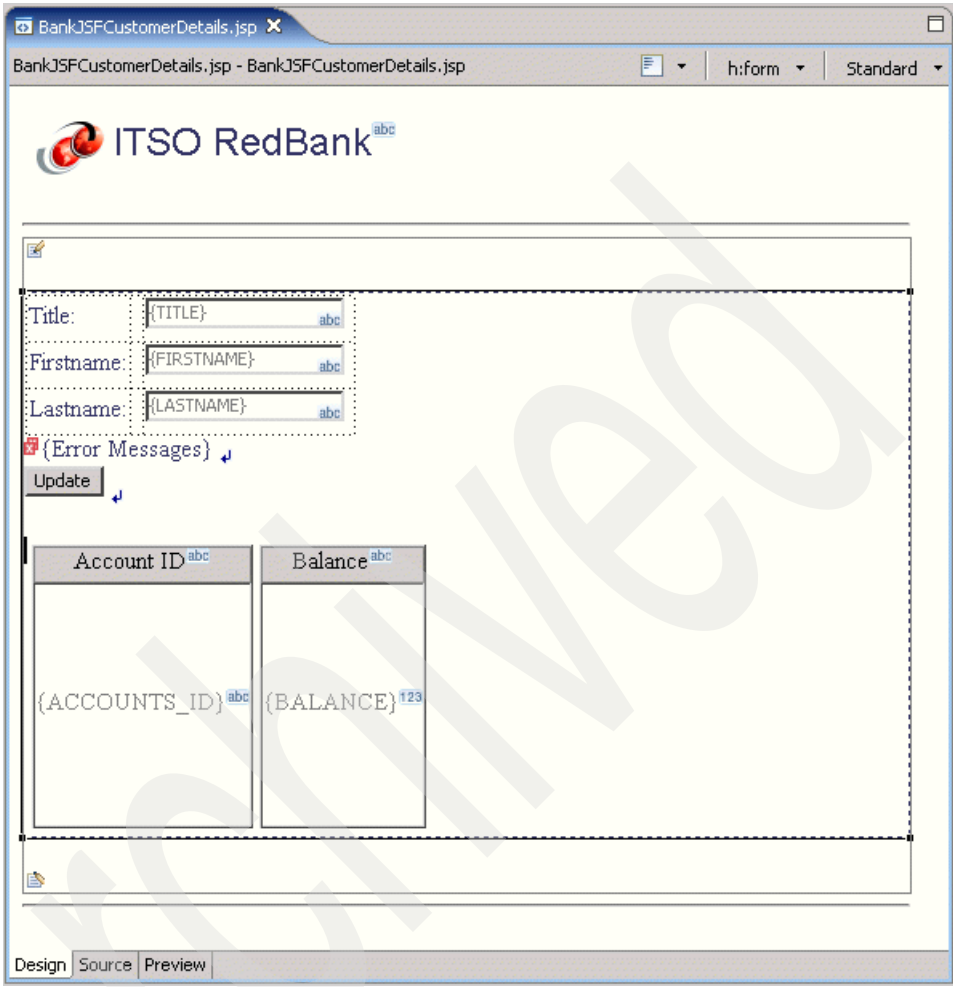


Figure 13-64 Design view of showing the table for customer accounts

Figure 13-64 displays the Design view of BankJSFCustomerDetails.jsp representing the table of the accounts that belong to the customer.

Change the formatting of the balance field

To change the formatting of the balance field, do the following:

1. Select the {BALANCE} output component in the Design view.
2. In the Properties view, select **Currency** from the Type drop-down list.
3. Save the BankJSFCustomerDetails.jsp.

Archived

Add a row action

We will now add a row action that will be activated when the user clicks a row of the table.

To add a row action, do the following:

1. In the Design view, select the data table component for accounts.
2. In the Properties view, select **h:dataTable →Row actions**.
3. Click **Add** next to `Add an action that's performed when a row is clicked`.

   The page should now look like Figure 13-65 in the Design view.

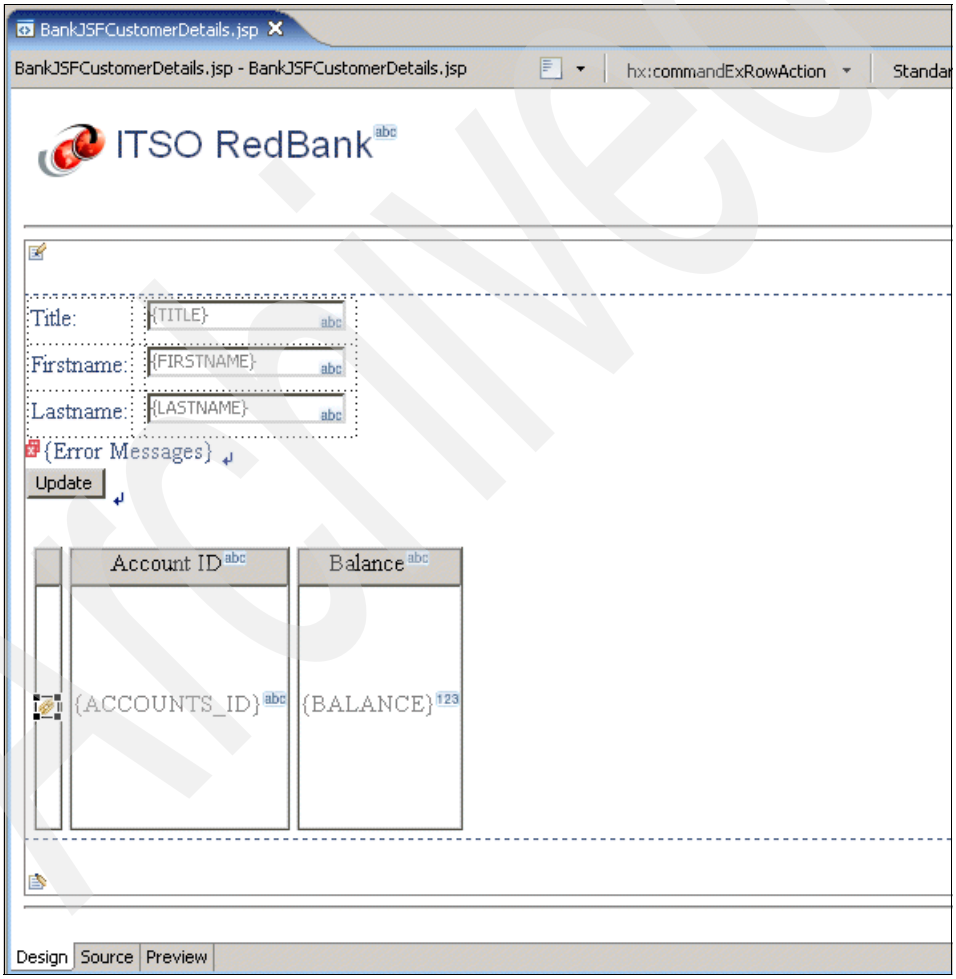


Figure 13-65 Add row action

4. Select the **Quick Edit** view. Click the icon in the Properties view if there is no Quick Edit tab.
5. Copy the code from Example 13-5 in to the Quick Edit view.

> **Note:** To save time, the source code can be copied from c:\6449code\jsf\BankJSFSource.jpage.

Example 13-5 Code for accounts row action

```
      try {
         int row = getRowAction1().getRowIndex();
         Object keyvalue =
((DataObject)getAccounts().get(row)).get("ACCOUNTS_ID");
         getSessionScope().put("accountId", keyvalue);
      } catch(Exception e){
         e.printStackTrace();
      }

      return "accountDetails";
```

6. Create a new Session scope variable named `accountId`, which is used by the code that was added.
   a. In the Page Data view click the Create new data component button ( ) (upper right of view).
   b. When the Add Data Component dialog appears, select the **Session Scope Variable**, and then click **OK**.
   c. When the Add Session Scope Variable dialog appears, enter `accountId` in the Variable name field, enter `java.lang.String` in the Type field, and then click **OK**.

Add reusable JavaBean

We will now add the reusable JavaBean to the page to handle the logout process.

To add a reusable java bean, do the following:

1. In the Page Data view, click the Create new data component button ( ).
2. Select **JavaBean** and click **OK**.
3. When the Add JavaBean dialog appears, do the following:
   a. Select **Add existing reusable JavaBean (JSF ManagedMean)**.
   b. Select the **Logout** bean, as seen in Figure 13-66 on page 743.
   c. Click **Finish**.

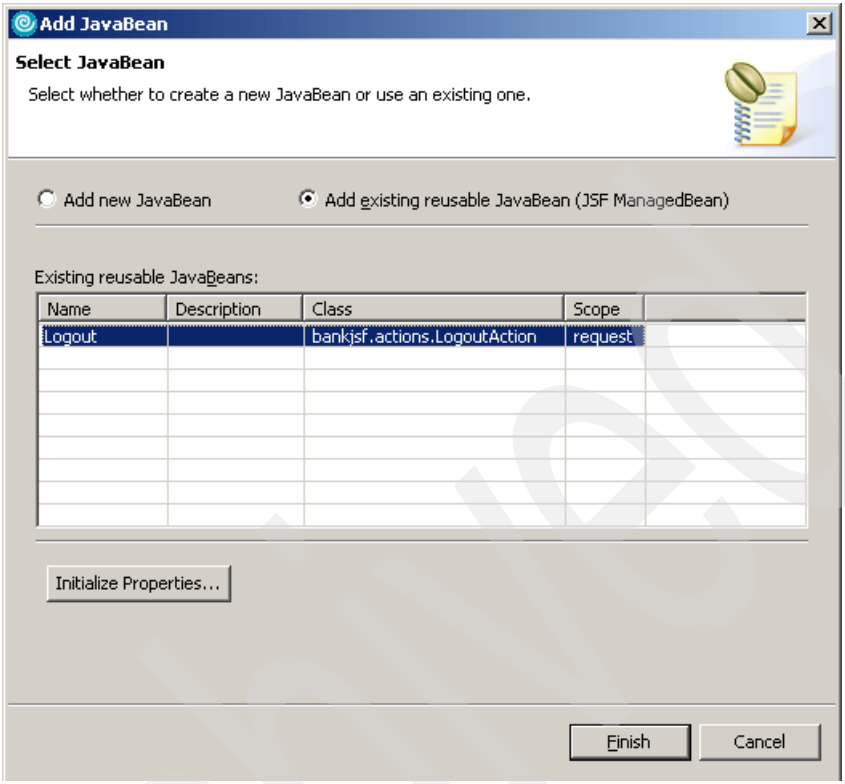


Figure 13-66 Add JavaBean wizard

Add logout method

To add the logout method from the Logout JavaBean, do the following:

1. In the Page Data view, right-click the **Logout** JavaBean and select **Add New JavaBean Method** from the context menu.
2. When the Select Method dialog appears, select **logout()** and click **OK**.

> **Note:** This will add a new method on the page called doLogoutLogoutAction(). This method will call the JavaBean method logout. Since the default code returns a null as the outcome, you will have to change this method to return the logoutLogoutResultBean.

3. Modify the return of the doLogoutLogoutAction() method.
   a. In the Page Data view, expand **logout (com.ibm.bankjsf.actions.LogoutAction)** → **logout()**.

b. Double-click **doLogoutLogoutAction()** to open the file BankJSFCustomerDetails.java in the editor.

c. Change the return from null to the `logoutLogoutResultBean`.

d. Save and close BankJSFCustomerDetails.java.

4. Save BankJSFCustomerDetails.jsp.

Add a logout button

To add a command button to the page and bind the logout method to the command button, do the following:

1. Open BankJSFCustomerDetails.jsp.
2. From the Design view, select the **Command - Button** from the Faces component palette and drag it onto the page.
3. Select the newly added Command Button in the Design view.
4. In the Properties view, click the **Display options** tab, and change the Button label to `Logout`.
5. In the Page Data view, right-click the **logout()** method of the Logout JavaBean and select **Bind to 'button2'** from the context menu.
6. Save the BankJSFCustomerDetails.jsp.

The BankJSFCustomerDetails.jsp page is now complete.

Complete the BankJSFAccountDetails.jsp

This section describes the steps needed to complete the BankJSFAccountDetails.jsp. In many cases, the details for these steps can be found in the procedures used to complete the BankJSFCustomerDetails.jsp.

To complete the BankJSFAccountsDetails.jsp, do the following:

1. In the Project Explorer view, expand **Dynamic Web Projects** → **BankJSF** → **WebContent**.
2. Double-click the **BankJSFAccountDetails.jsp** to open it in the editor.
3. Remove the text `Default content of bodyarea` like on previous pages.
4. Create a New Relational Record called account.

   For details refer to "Add relational record" on page 730.

   – Select the **ACCOUNT** table.
   – Add a filter condition of ID equal to the Session scope variable accountId.
5. From the Page Data view drag the Relational Record account onto the page.

Archived

For details refer to “Add the relational record component to the page” on page 733.

On the Configure Data Controls page, do the following:

- Ensure that `Display an existing record (read-only)` is selected.
- Use the up and down arrows to change the column order such that the ID column is displayed before the BALANCE column.

6. Change the Balance output component type to `Currency`.

   For details refer to “Change the formatting of the balance field” on page 740.

7. Add a command button that:
   - Has the display text `Customer Details`
   - Returns the outcome of `customerDetails`

   Tip: Use the Quick Edit view to change the outcome of a command button.

8. Add the reusable Logout Java Bean as we did in the previous section.
9. Add a command button and bind the logout method to the command button.

   Tip: Remember to return the logoutLogoutResultBean in the doLogoutLogoutAction().

10. Add a column to the account information table containing the customer’s SSN:
    a. Place the cursor in the Id: cell and select **Table** → **Add Row** → **Add Above**.
    b. Enter `SSN:` in the left-most cell of the new row.
    c. Drag an outputFormat component to the right-most cell of the new row.
    d. Drag the customerSSN session variable to the new outputFormat component.

The resulting page should look like Figure 13-67 on page 746.

Archived

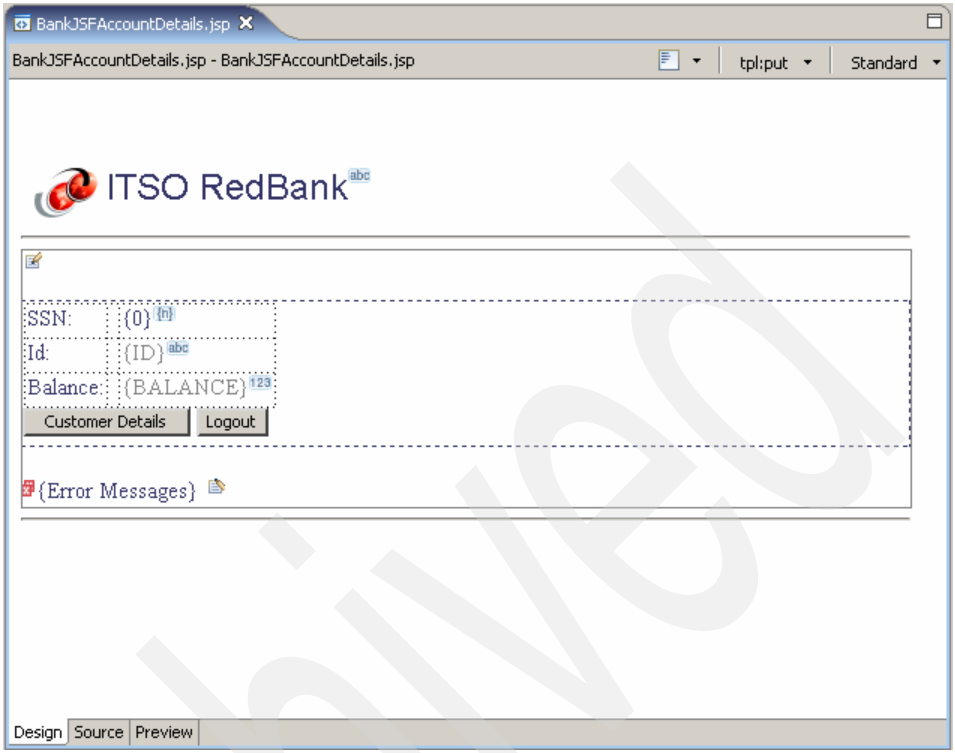


Figure 13-67 Finished BankJSFAccountDetails.jsp

13.4 Run the sample Web application

This section demonstrates how to run the sample Web application built using JSF and SDO.

13.4.1 Prerequisites to run sample Web application

In order to run the Web application you will need to have completed the following:

1. Sample code.
   - Complete the sample following the procedures described in 13.3, “Develop a Web application using JSF and SDO” on page 684. Or do the following.
   - Import the completed sample c:\6449code\jsf\BankJSF.zip Project Interchange file. Refer to “Import sample code from a Project Interchange file” on page 1398 for details.
2. Set up the sample database.

Archived

For details refer to 13.2.2, “Set up the sample database” on page 681.

3. Configure the data source.

   The details can be found in 13.2.3, “Configure the data source via the enhanced EAR” on page 681.

13.4.2 Run the sample Web application

To run the sample Web application in the test environment, do the following:

1. Open the Web perspective Project Explorer view.
2. Expand **Dynamic Web Projects** → **BankJSF** → **WebContent**.
3. Right-click **BankJSFLogin.jsp**, and select **Run** → **Run on Server**.
4. When the Server Selection dialog appears, select **Choose an existing server**, select **WebSphere Application Server v6.0**, and click **Finish**.

The Web application Logon page should be displayed in a Web browser.

13.4.3 Verify the sample Web application

Once you have launched the application by running it on the test server, there are some basic steps that can be taken to verify that the Web application using JSF and SDO is working properly.

1. From the ITSO RedBank login page, enter 111-11-1111 in the customer SSN field (as seen in Figure 13-68 on page 748), and then click **Enter**.

Archived

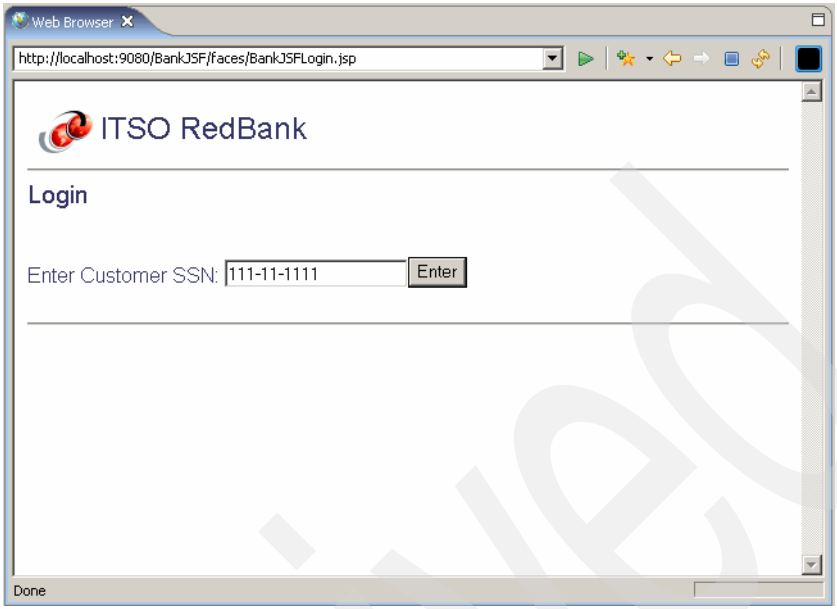


Figure 13-68 ITSO RedBank Login page

The resulting page should look like Figure 13-69 on page 749.

Note: We noticed that under some circumstances, the JSF runtime would not initialize until the Enter button had been clicked. This resulted in the behavior that the first time the button was clicked, nothing happened. Retrying the login would then work as expected.

Archived

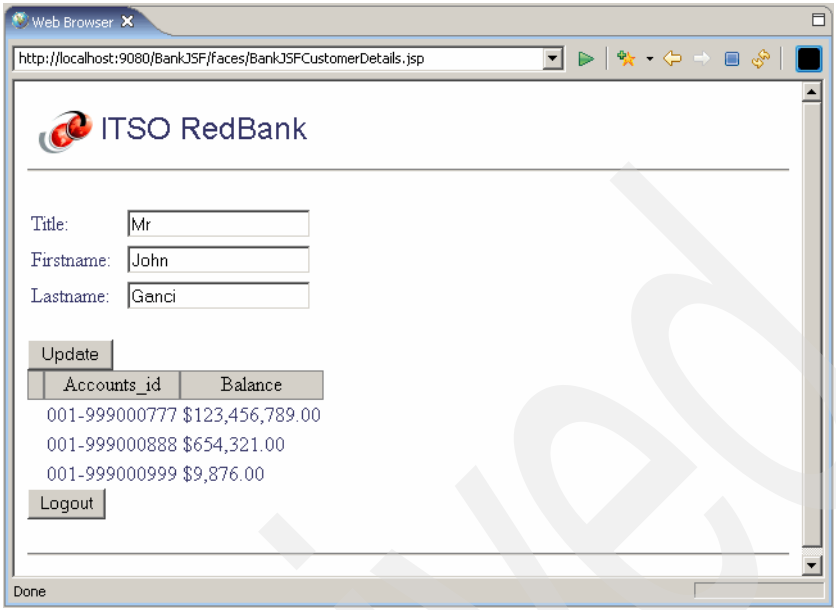


Figure 13-69 Display of customer accounts

2. From the page displayed in Figure 13-69, you can do one of the following:
   - Change the fields and then click **Update**. This verifies write access to the database using SDO. For example, change Title to Sir and then click Update.
   - You can click **Logout**, which will perform a logout and return to the home page.
   - You can click the accounts to display the accounts information, resulting in the page displayed in Figure 13-70 on page 750.

Archived

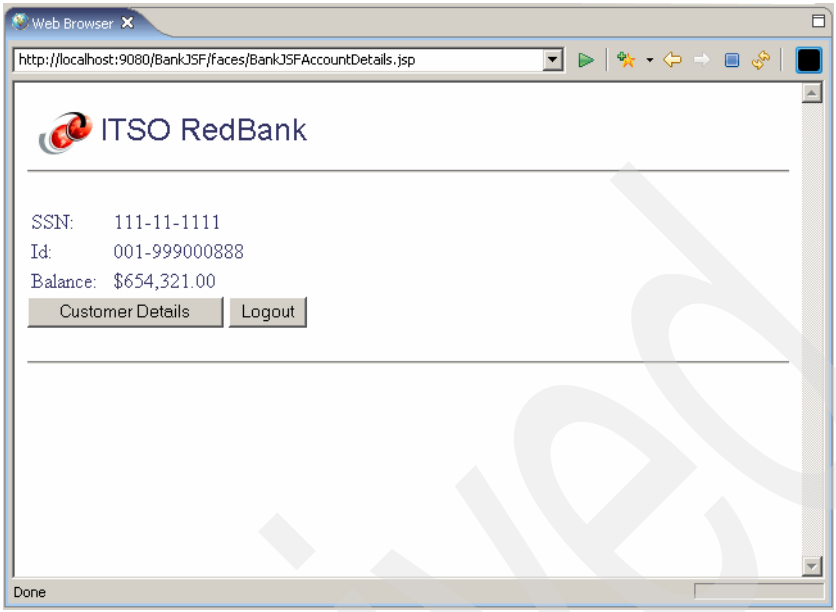


Figure 13-70 Details for a selected account

3. From the page displayed in Figure 13-70, you can do one of the following:
   - You can click **Customer Details**, which will return you to the page displayed in Figure 13-69 on page 749.
   - You can click **Logout**, which will perform a logout and return you to the home page.

Archived

14

Develop Web applications using EGL

IBM Enterprise Generation Language (EGL) is a business application centric procedural programming language and environment used to develop batch, text user interface (TUI), and Web applications. When developing an EGL Web application, the developer creates the EGL source files using wizards and the source editor. Java/J2EE source is generated from the EGL source files so that the application can then be deployed to WebSphere Application Server.

In this chapter we introduce the concepts, benefits, and architecture of the IBM Enterprise Generation Language. We have included a working example that describes how to develop Web applications using EGL and JSF with the tooling provided in IBM Rational Application Developer V6.0.

The chapter is organized into the following sections:

- Introduction to EGL
- IBM EGL tooling in Rational Developer products
- Prepare for the sample application
- Develop the Web application using EGL
- Import and run the sample Web application
- Considerations for exporting an EGL project

© Copyright IBM Corp. 2005. All rights reserved.

14.1 Introduction to EGL

Throughout this chapter, we explore the features of EGL and tooling provided in IBM Rational Application Developer V6.0 used to develop EGL applications.

This section is organized as follows:

- Programming paradigms
- IBM Enterprise Generation Language
- IBM EGL and Rational brand software
- IBM EGL feature enhancements
- Where to find more information on EGL

14.1.1 Programming paradigms

The objective of this section is to describe the classes of programming paradigms to better understand why EGL is well suited for business application development.

System vs. business application programming

From a granular perspective, there are system programs and business application programs. Examples of systems programming include creating operating systems, spreadsheets, or word processors. Some examples of business application programming include creating an employee time and attendance system, customer order entry system, or a bill payment system.

Generally it can be said that business application programming involves slightly less software theory or invention, and more focus on applying the technology with the purpose of achieving a tangible business goal.

Computer programming languages that can provide for systems programming are general purpose in that they have fewer if any higher level constructs. For example, if you wanted to provide the end user with a data entry screen form and functions using the C language, you might have to write 2000 lines of program source code. When using a business application programming language, the same user interface might be developed in 200 lines of source code. Using the appropriate programming language can dramatically increase productivity.

Declarative programming languages

A declarative programming language describes what output is desired, not how to generate it. Examples of declarative languages include Structured Query Language (SQL), Extensible Markup Language (XML), and Hyper Text Markup Language (HTML).

Archived

Procedural programming languages

Procedure programming languages are modular in that they allow the programmer to group the code into modules, procedures, and/or functions. Examples of procedural languages include C, Fortran, Pascal, COBOL, and Basic.

Object-oriented programming languages

An object-oriented programming language supports the following constructs:

- Objects
- Abstraction
- Encapsulation
- Polymorphism
- Inheritance

Examples of object-oriented programming languages include Java, Smalltalk, C++, and C#. Object-oriented programming languages generally offer a language with many advanced capabilities over structured languages. These capabilities aid in software re-use, ease of maintenance, and numerous other desirable features. If there is a caveat to object-oriented programming languages, it is that they are considered to have a longer and more expensive learning curve than procedural programming languages.

Fourth generation programming languages (4GL)

Fourth generation programming languages, also known as 4GLs, are focused on business application programming. Using a 4GL minimizes programming skill requirements, development time, and total development cost.

14.1.2 IBM Enterprise Generation Language

IBM Enterprise Generation Language is a procedural language with a number of higher level 4GL language constructs. For example, EGL contains a two-word command verb that allows the developer to produce a dynamically retrieved array of tabular data for display, update, or other purposes.

IBM EGL provides the ability to develop a wide range of business application programs including applications with no user interface, a text user interface (TUI), or a multi-tier graphical Web interface. The IBM EGL compiler will generate Java source code for either Java 2 Platform Standard Edition or Java 2 Platform Enterprise Edition, as required.

Additionally, IBM EGL supports software re-use, ease of maintenance, and other features normally associated with object-oriented programming languages by following the Model-View-Controller (MVC) design pattern.

In summary, EGL offers the benefits of J2EE, while providing a simple procedural programming environment for non-Java developers, which facilitates rapid Web application development.

Target audience of IBM EGL

Since IBM EGL is geared toward business application development, the developers tend to have a greater focus on understanding the business needs and less so on technology. Java 2 Platform Enterprise Edition offers many benefits and is extremely powerful; however, this platform requires developers to have extensive programming knowledge, which can be an impediment for developers who are new to Java. EGL provides the ease of use of a procedural programming language with the power of Java 2 Platform Enterprise Edition, which is generated by the EGL tooling.

We have listed some common types of developers that use EGL:

- Business application programmers in need of higher productivity
- Programmers needing to deploy to diverse platforms
- Business-oriented developers
  - 4GL developers (Oracle Forms, Natural, CA Ideal, Mantis, Cool:Gen, reports)
  - Visual Basic developers
  - RPG developers
  - COBOL/PL1 developers
  - VisualAge® Generator developers
  - IBM/Informix 4GL developers

History of EGL

In 1981 IBM introduced the Cross System Product (CSP). This programming language began the concept of a platform neutral, singular programming language and environment that would allow enterprise-wide application development and support.

Generally speaking, the objectives of a cross-platform solution are as follows:

- Abstraction: Hide the platform-specific differences from the developer and end user.
- Code generation: Using code generation to bridge abstract and concrete applications
- Platform and language neutrality.
- Rich client and debugger support.

Archive

The benefits of a cross-platform strategy include:

- Less program code to write (not platform specific)
- Reduction of training requirements
- Provide easier transition to new technologies
- Provide tested performance and code quality

In 1994 the IBM VisualAge Generator product (VisualGen®) product was released. VisualAge Generator V4.0 allowed for the creation of Web programs without knowledge of Java. Later releases of VisualAge introduced the EGL-like ability to output COBOL.

In 2001, the WebSphere Studio tools were introduced. In 2003, IBM acquired Rational software, which was known as a leader in software engineering technologies, methodologies, and software. In 2004, IBM officially transferred the WebSphere Studio development tooling to the IBM Rational division to consolidate software under one brand supporting the entire software development life cycle. During this same time period, IBM joined forces with many strategic companies to contribute to the Eclipse project. Today Eclipse 3.0 is used as the base for which the IBM Rational Software Development Platform is built. Rational Application Developer, as well as much other Rational tooling, shares this common base. For more information on Eclipse and Rational products refer to 1.3.3, “Eclipse and IBM Rational Software Development Platform” on page 19.

The IBM Enterprise Generation Language emerged from many IBM 4GL predecessors (for example, VisualAge Generator and CSP). Today IBM EGL incorporates the ability to generate source output for Java 2 Platform Standard Edition, Java 2 Platform Enterprise Edition, COBOL, and PL1. It also includes migration tools for other 4GL languages.

IBM EGL value proposition

The IBM Enterprise Generation Language increases the developer’s productivity and value to the business in three key areas. First, EGL is an easy-to-learn procedural programming language. Second, EGL is a business application centric programming language, which allows you focus and complete business applications more rapidly than other languages. Third, since the Rational Application Developer EGL tooling can generate Java and Java 2 Platform Enterprise Edition source code, you can take advantage of the benefits of the open, scalable, and potent Java and J2EE programming environments.

When evaluating infrastructure choices for Web applications, there are two major competing models, Java 2 Platform Enterprise Edition and Microsoft .NET. Java 2 Platform Enterprise Edition is the platform of choice of IBM and many businesses. Although Java 2 Platform Enterprise Edition does deliver on its great promises and is an excellent platform, it can take developers a great deal of time

to master and be proficient. EGL addresses the learning curve issue of Java 2 Platform Enterprise Edition by offering a simple procedural and business application centric programming model, which with the use of the EGL tooling provided by Rational Developer products, can be generated into Java or Java 2 Platform Enterprise Edition resources.

As IBM EGL outputs Java/J2EE, IBM EGL acts as an abstract layer from changes in the Java/J2EE platform. Parts of Java/J2EE are mature. Parts of Java.J2EE are emerging and changing. IBM EGL offers to insulate programmers from this volatility.

In summary, IBM EGL provides the great productivity gains of a procedural and business centric programming language, with the many benefits of Java and Java 2 Platform Enterprise Edition.

Application architecture

IBM EGL can be used to deliver business application programs with no user interface (batch jobs), text user interface programs, and Web applications. In this section, we explore the architecture of Web applications.

Figure 14-1 displays the architecture of a standard Web application. While Web applications can be delivered via a number of computing infrastructure choices, we focus on J2EE.

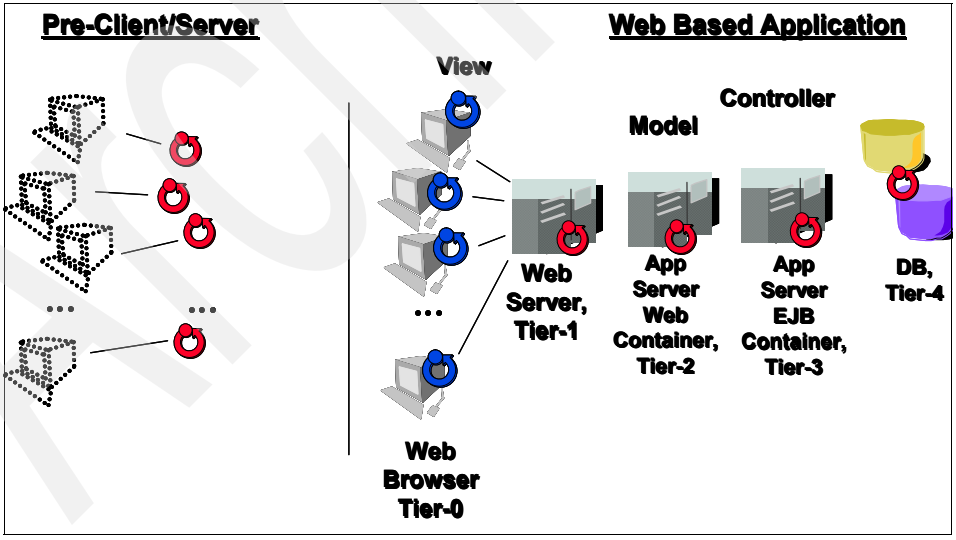


Figure 14-1 Architecture for a standard Web-based application

Archived

Using Java/J2EE, the following is held to be true:

- The end user operates from Tier-0. Tier-0 has no runtime or software requirements other than a Web browser and access to a network.
- Tier-1 supports a Web server and communicates with the end user using an HTTP communication protocol (for example, IBM HTTP Server).
- Generally, the Web server handles requests for static resources—GIF files, JPEG files, static HTML, and more. The Web server maintains a list of file types, if you will, and is configured to forward requests it does not understand to another agent. This agent, also referred to as a Web server plug-in, is a dynamic linked library (DLL) or shared object (a "*.so" file). Generally, the Web server plug-in forwards requests to Tier-2.
- Tier-2 supports a Java/J2EE-compliant Web container, a Java/J2EE-compliant application server such as WebSphere Application Server.
- Tier-2 receives inbound communication requests, parses the input, and formulates a response. Within Java/J2EE, the primary objects being programmed here are Java filters, Java servlets, and Java server pages. In short, Java servlets handle inbound communication, and Java server pages handle output.
- In the Model-View-Controller design pattern, Java servlets are the controller, and JSPs are used for the view.
- Tier-3 supports a Java/J2EE-compliant EJB container, a Java/J2EE application server such as IBM WebSphere Application Server.
- As seen in Figure 14-1 on page 756, Tier-3 is used to provide an advanced data persistence layer; access to a relational database server such as DB2 Universal Database.
- In the Model-View-Controller design pattern, Java beans or Enterprise Java Beans are models.
- Tier-4 supports the database server.

Java/J2EE is fabulous in that it is open, proven to scale, and offers numerous other advantages. If there is one criticism with Java/J2EE, it is the steep learning curve. First one must learn object-oriented programming, then Java, then Java/J2EE. There are numerous and distinct objects one must learn inside Java/J2SE and Java/J2EE—Java filters, Java servlets, Java server pages, Java beans, and/or Enterprise Java beans. IBM EGL is procedural and measurably less difficult to lean than Java/J2EE.

IBM EGL outputs Java/J2SE and Java/J2EE. For a Web application, IBM EGL uses one procedural language with no objects with 4GL level productivity. The

Archived

interface is developed using Faces JSPs with small EGL page handlers to respond to page events, page loads, and action buttons.

14.1.3 IBM EGL and Rational brand software

The IBM Rational software brand includes products focused on design and construction, process and portfolio management, software configuration management, and software quality. The IBM EGL tooling and environment are included with the Rational Developer products on Windows and Linux platforms for the following editions (see Figure 14-2):

- IBM Rational Web Developer V6.0
- IBM Rational Application Developer V6.0
- IBM Rational Software Architect V6.0
- IBM WebSphere Studio Enterprise Developer
- IBM WebSphere Studio Integration Developer

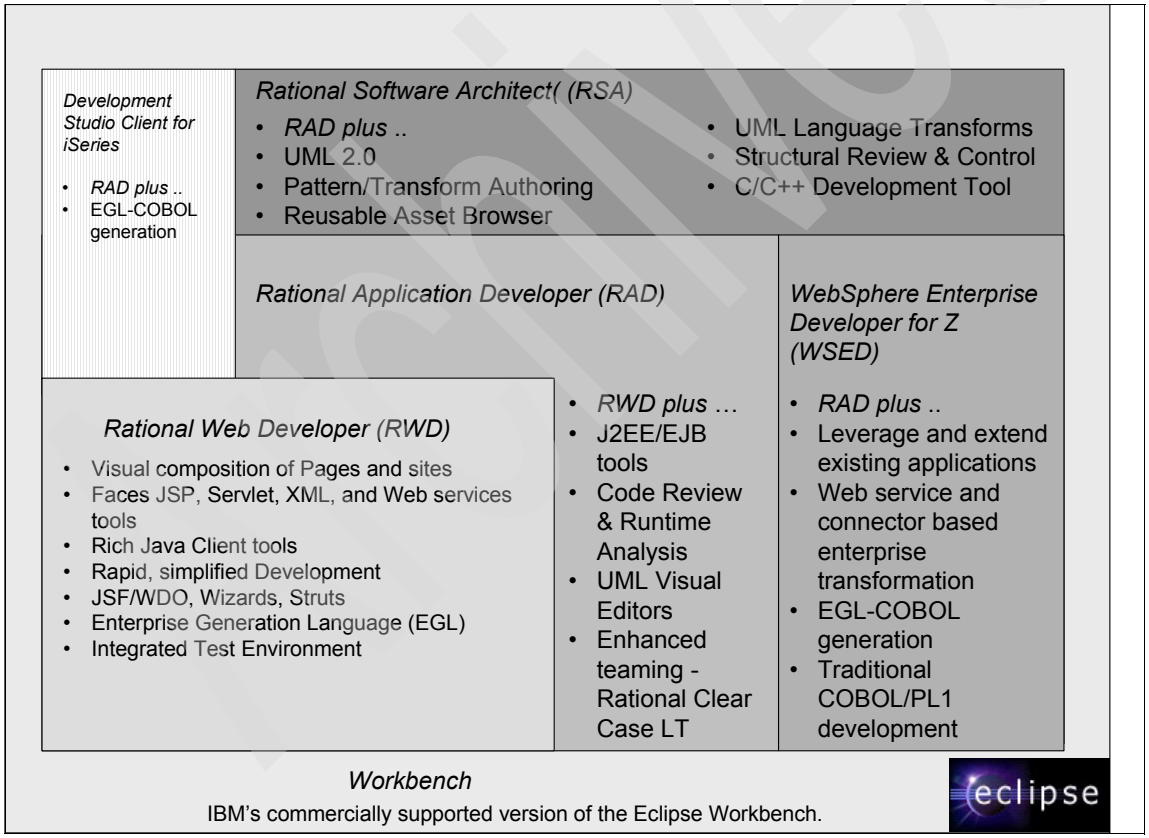


Figure 14-2 IBM Rational software containing IBM EGL

To keep the Rational Developer product base install image as small as possible, EGL is installable as an optional component. Figure 14-5 on page 769 displays a picture of the IBM Enterprise Generation Language (EGL) component in the IBM Rational Application Developer V6.0 Installer.

IBM WebSphere Studio Enterprise Edition has additional EGL capabilities including the ability to output COBOL program code that can execute on the IBM zSeries® (zOS) and IBM iSeries platforms (OS/400®), in addition to generate Java source output.

IBM EGL Web applications can be deployed to J2EE-compliant application servers, including the WebSphere Application Server V6.0, Express, Base, and Network Deployment Editions, as well as IBM WebSphere Application Server Enterprise Edition (V5.x).

14.1.4 IBM EGL feature enhancements

At the time of completing this book, we learned of EGL feature enhancements that would be delivered point releases of IBM Rational Web Developer V6.x and IBM Rational Application Developer V6.x. Although we were not able to test these new features, we want to make people aware of this new functionality.

Version 6.0.01 feature enhancements

The following section describes the key feature enhancements to EGL included in IBM Rational Developer products V6.0.01. This list is by no means exhaustive.

- TUI Editor for EGL: This is an Eclipse-based WYSIWYG editor for the construction of Text User Interfaces (TUIs) for EGL. This allows EGL customers to define EGL TUIs that will deploy as 5250 or 3270 applications running against iSeries or zSeries, respectively. The EGL TUIs can also be run on distributed Java platforms. This is useful for VisualAge Generator customers migrating their TUI applications from these platforms to EGL.
- EGL runtime support for HP/UX and SUN Solaris: Support for HP/UX V11.0, V11.11, V11.23, and SUN Solaris V7, V8, V9.
- Debugger support for Jasper Reports: EGL provides the ability to debug an EGL ReportHandler, including breakpoints, variable inspection, etc.
- i4GL Migration Utility: This utility provides enhanced tooling integrated with IBM Rational's Software Development Platform to increase productivity of Informix application developers. The Informix 4GL to EGL Conversion Utility is available as an iFix004 iFeature with the Rational Developer V6.0 products.

This release of IBM Informix 4GL to EGL Conversion Utility offers the following key features and benefits:

- Uniform Conversion from 4GL to EGL to retain the look-and-feel of your 4GL program with the equivalent EGL program after conversion.
- Graphical Conversion Wizard steps you through each step of the conversion process.
- Command Line Conversion - Scripted or automated usage.

Informix 4GL to EGL Conversion Utility iFix004 iFeature should be installed via the Update Manager functionality with any of the following products:

- IBM Rational Web Developer V6.0
- IBM Rational Application Developer V6.0
- IBM Rational Software Architect V6.0
- IBM WebSphere Application Server Express V6.0

Version 6.0.1 feature enhancements

This section describes the key feature enhancements to EGL included in IBM Rational Developer products V6.0.1. This list is by no means exhaustive.

- EGL Parts Reference View Enhancements:
  - Part wizard: This wizard allows you to (optionally) create a part that does not exist. Currently a message saying the part does not exist will be displayed if the part is not defined.
  - Flat layout: This layout lists all the parts that are referenced in a flat table. We show part name, part type (icon), project, package, and file name in the table. The customer can toggle between viewing in the flat layout versus the existing hierarchical layout.
  - Search declarations and references: Allows you to search declarations and references to a part within a given scope. This is similar to the Java Development Tooling (JDT) in Eclipse.
  - Find part: Ability to search the view (find capability) for a given part name.
- EGL Language Enhancements for IMS™ and DLI Support: This includes the addition of part types to allow access to data stored in DL/I databases, which includes parts to represent PCBs and SSPs, in addition to a DL/I record.
- EGL Support for MS SQL Server: EGL supports Microsoft SQL Server 2000.
- EGL Data Item Part Source Assistant: This assistant provides a dialog that groups the various properties available on EGL Data items and facilitates the entry of appropriate values via a graphical dialog instead of the EGL Source editor.

Archived

14.1.5 Where to find more information on EGL

For more information on IBM EGL, refer to the following:

- IBM developerWorks EGL home page:

  http://www.ibm.com/developerworks/rational/products/egl/

- *Generating Java using EGL and JSF with WebSphere Studio Site Developer V5.1.2*, white paper found at:

  http://www.ibm.com/developerworks/websphere/library/techarticles/0408_barosa/0408_barosa.html

- *Transitioning: Informix 4GL to Enterprise Generation Language (EGL)*, SG24-6673, redbook; expected publish date is June 2005

14.2 IBM EGL tooling in Rational Developer products

This section highlights the IBM EGL tooling and support included in Rational Developer products.

Important: After installing the Rational Application Developer EGL component, you must enable the EGL development capability in order to access the EGL preferences and features.

- For details on installation refer to 14.3.1, “Install the EGL component of Rational Application Developer” on page 768.
- For details on enabling the EGL development capability refer to 14.3.2, “Enable the EGL development capability” on page 771.

14.2.1 EGL preferences

The EGL preferences allow you to define EGL development environment settings.

Once you have enabled the EGL development capability from the Workbench preferences, you can access the EGL preferences. Select **Window** → **Preferences**. Click the **EGL** tab. From the EGL preferences you will be able to customize the settings highlighted in Figure 14-3 on page 762.

Archived

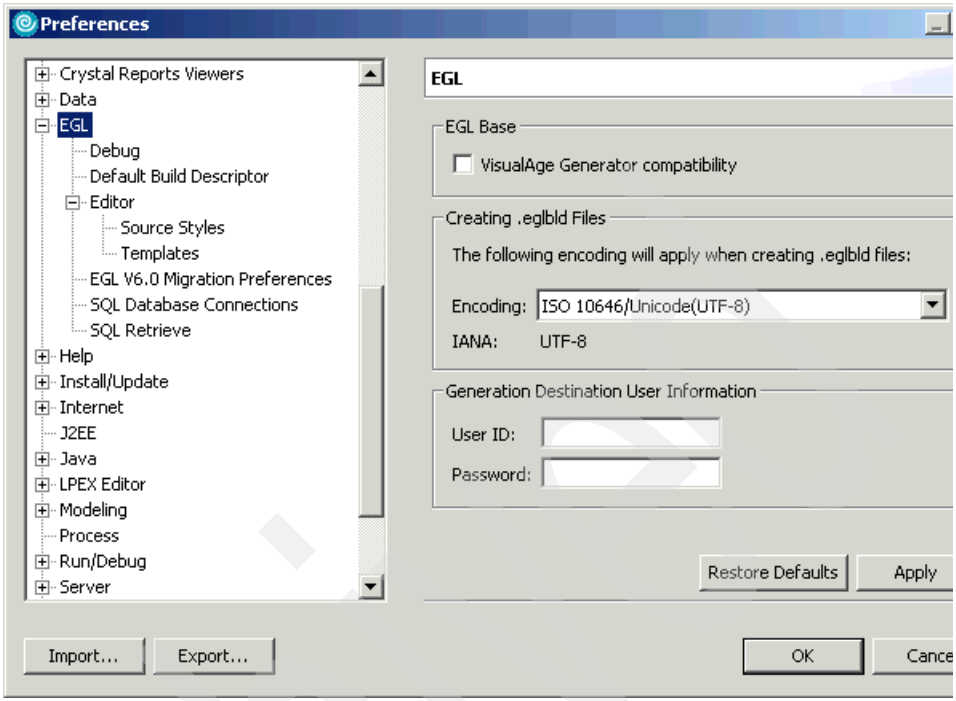


Figure 14-3 EGL preferences

14.2.2 EGL perspective and views

Within IBM Rational Application Developer V6, there is an EGL perspective and supporting views for EGL application development.

To open the EGL perspective, select **Window** → **Open Perspective** → **Other**. When the Select Perspective dialog appears, check **Show all**, select **EGL**, and then click **OK**.

EGL includes the following views, which can be accessed by selecting **Window** → **Show View**:

- EGL Debug Validation Errors
- EGL Generation Results
- EGL Parts Reference
- EGL SQL Errors
- EGL Validation Results

14.2.3 EGL projects

When working with EGL, it is important to understand the difference between an EGL Project and an EGL Web Project.

EGL Project

An EGL Project is used to develop EGL batch and text user interface (TUI) applications.

EGL Web Project

An EGL Web Project is a Dynamic Web Project with EGL support enabled and is used to develop applications with a Web interface. For the working example in this chapter, we demonstrate how to create an EGL Web Project in preparation for developing a Web application using EGL.

For details on creating an EGL Web Project, refer to 14.3.4, “Create an EGL Web Project” on page 773.

14.2.4 EGL wizards

IBM Rational Application Developer V6 includes several EGL wizards that can be used to generate the EGL source with the objective of speeding development tasks. For example, if you have created an EGL Web Project, right-click the project in the EGL perspective. Select **New** → **EGL** and then select one of the following EGL wizards:

- EGL Build File: The build descriptor controls the generation process. Within the context of Rational Application Developer, the EGL output type is Java/J2EE. The build descriptor is part of the EGL build file.
- EGL Source Folder (EGLSource): The EGLSource folder is the location where the EGL source files will be stored within the a project.
- EGL Package: EGL packages are used to group common types of code (for example, data, libraries, pagehandlers).
- EGL Source File: When using the EGL Source File wizard, the EGL file is created with the given name in the EGLSource folder.
- Faces JSP File: When creating a Faces JSP within an EGL Web Project, an EGL page handler source file is created in addition to the Faces JSP Java source being created.
- Program: An EGL program part is the main logical unit used to generate a Java program, Java wrapper, or Enterprise JavaBean session bean.
- Library: An EGL Library Part contains a set of functions, variables, and constructs that can be used by programs, page handlers, and other libraries.

Archived

- Data Table: An EGL Data Table Part associates a data structure with an array of initial values for the structure.
- Form Group: An EGL Form Group Part defines a collection of text and print forms.
- EGL Data Parts: The EGL Data Parts wizard is used to create SQL records, as well as data-item parts and library-based function parts, from one or more relational database tables or pre-existing views. The working example demonstrates how to use this wizard.
- EGL Data Parts and Pages: The EGL Data Parts and Pages wizard provides a method to create an EGL Web Project, data parts, and Faces JSPs for a given application in one simplified process.

14.2.5 EGL migration

IBM Rational Application Developer V6 includes EGL migration tooling for the following:

- EGL migration to V6.0

  The EGL migration to V6.0 tooling is used to migrate previous EGL versions to EGL V6.0 with Interim Fix 0004.

  This is enabled in through **Workbench** → **Capabilities** → **EGL Developer** → **EGL V6.0 Migration**. Once the capability is enabled, you can right-click on the project and select **Migrate** → **EGL V6.0 Migration** to migrate the EGL source code to the new level.

  Note: If you have developed an EGL application with IBM Rational Application Developer V6 .0 (original release), you will need to migrate the source code and manually copy the runtime libraries in your project after installing Rational Application Developer V6.0 - Interim Fix 0004. There are significant changes in the EGL language syntax that require that you migrate.

- VisualAge Generator to EGL migration

  The VisualAge Generator to EGL migration is used to migrate code developed in VisualAge Generator to EGL V6.0 with Interim Fix 004.

  Enabled this by selecting **Workbench** → **Capabilities** → **EGL Developer** → **VisualAge Generator to EGL Migration**. Once the capability is enabled, you can right-click the project and select **VisualAge Generator to EGL Migration** to migrate to EGL.

- Informix 4GL to EGL Conversion Utility

  The Informix 4GL migration is used to migrate Informix 4GL to EGL.

This feature requires that the *Informix 4GL to EGL Conversion Utility* be installed using the Rational Product Updater Optional Features tab.

Once this feature is installed, you will need to enable the *Informix 4GL to EGL Conversion* capability by selecting **Window** → **Preferences** → **Workbench** → **Capabilities** → **EGL Developer**, and checking the Informix 4GL to EGL Conversion check box.

For details on the Informix 4GL to EGL Conversion Utility refer to the following:

– *IBM Informix 4GL to EGL Conversion Utility User's Guide*, G251-2485, found in the following directory after installing the optional feature:

```
<rad_home>\egl\eclipse\plugins\com.ibm.etools.i4gl.conversion_6.0.0.2
```

– *Transitioning: Informix 4GL to Enterprise Generation Language (EGL)*, SG24-6673, redbook (expected to be published June 2005)

14.2.6 EGL debug support

IBM Rational Application Developer V6 includes support for debugging applications developed in EGL. Like other development environments and languages within Rational Application Developer, EGL can be run and debugged on the WebSphere Application Server V6.0 or WebSphere Application Server V5.1 Test Environments.

From the Server view, right-click the server started in debug mode, and select **Enable/Disable EGL Debugging**.

14.2.7 EGL Web application components

In the previous sections we highlighted the EGL tools and features included in Rational Application Developer used to develop an EGL application. In this section, we describe the components generated by the wizards that make up an EGL Web application.

Figure 14-4 on page 767 displays the components of an EGL Web Project in the Project Explorer view. We have outlined the key resources to provide a better understanding of an EGL Web application.

- BankEGLEAR: The BankEGLEAR is created in the Enterprise Applications folder.
- Deployment Descriptor: BankEGLEAR: We will modify the enhanced EAR settings to define the deployment database in a later step.
- BankEGL: The BankEGL EGL Web Project is created in the Dynamic Web Projects folder.

Archive

- EGLSource: This folder will contain the EGL resources created for the application.
  - data: Folder includes EGL source files containing data records.
  - libraries: Folder includes EGL source files containing functions.
  - pagehandlers: Folder includes EGL page handler source files used to define the EGL functionality within a Faces JSP. For example, the page handler will contain the code to be executed when a button within the Faces JSP, such as Logout, is clicked.
- BankEGL.eglbld: The BankEGL.eglbld is the EGL Build Descriptor that was generated when we created the new EGL Web Project and selected to create new project build descriptor(s) automatically. This file contains the preferences that will be used for the EGL Web Project.
- JavaSource: This folder contains Java source files that have been generated from the EGL source files found in the EGLSource folder.
- WebContent: This folder contains Faces JSPs with EGL references (page handlers, libraries, data).

Archived

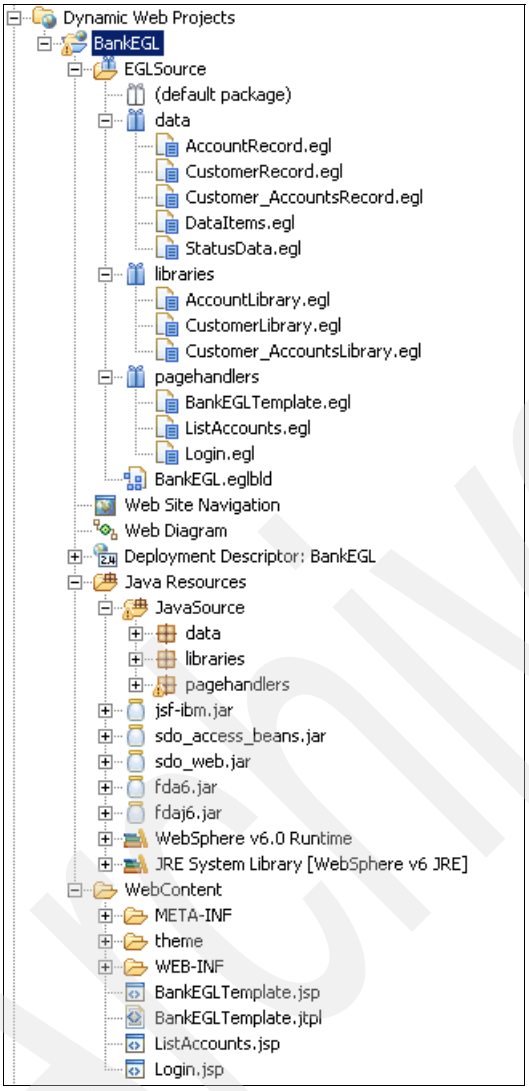


Figure 14-4 Components of a sample EGL Web application

14.3 Prepare for the sample application

Prior to developing the sample application using EGL, you will need to complete the following tasks described in this section:

- Install the EGL component of Rational Application Developer.
- Enable the EGL development capability.

Archived

- Install DB2 Universal Database.
- Create an EGL Web Project.
- Set up the sample database.
- Configure EGL preferences for SQL database connection.
- Configure the data source.

Note: A completed version of the ITSO RedBank Web application built using EGL can be found in the c:\6449code\egl\BankEGL.zip Project Interchange file. If you do not wish to create the sample yourself, but want to see it run, follow the procedures described in 14.5, “Import and run the sample Web application” on page 816.

14.3.1 Install the EGL component of Rational Application Developer

This section describes the configuration requirements for using IBM EGL within IBM Rational Application Developer V6.0.

Install IBM Enterprise Generation Language

The IBM Enterprise Generation Language is optionally installed as a component of the Rational Application Developer installation, as seen in Figure 14-5 on page 769.

Note: Within the context of this book, the installation of EGL requires that you have installed IBM Rational Application Developer V6.0 with the WebSphere Application Server V6.0 Integrated Test Environment.

Archive

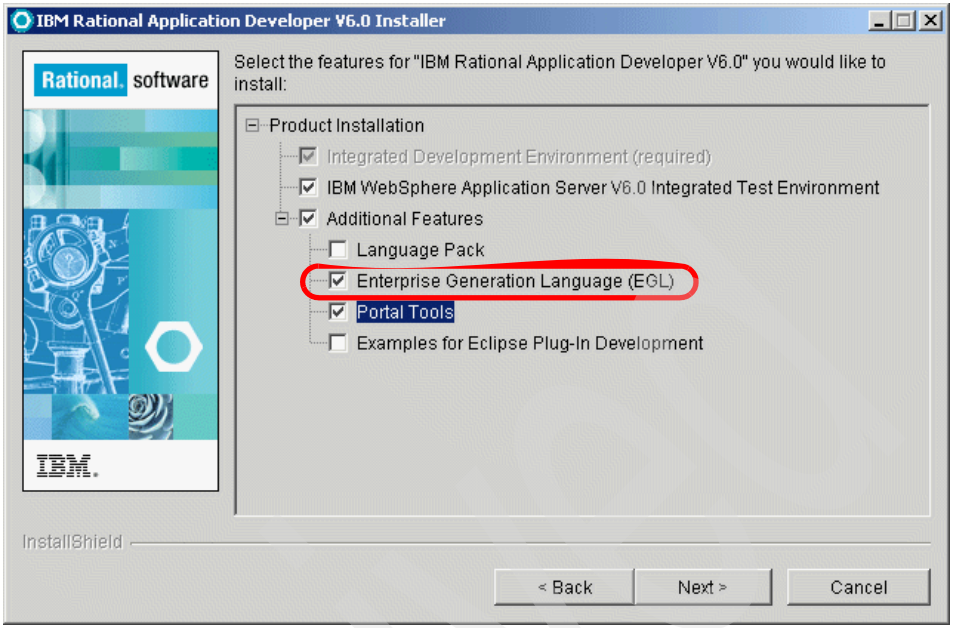


Figure 14-5 EGL component of Rational Application Developer installation

Interim fix

At the time of writing this chapter and sample application, we used IBM Rational Application Developer V6.0 - Interim Fix 0004. We suggest that you install the latest available interim fix level.

Refer to the following URL for details on the contents and instructions for installing the latest IBM Rational Application Developer V6.0 - Interim Fix:

`http://www.ibm.com/support/search.wss?rs=2043&tc=SSRTLW%2BSSRTLW&q=interim+fix`

Archive

Attention: EGL applications developed with IBM Rational Application Developer V6.0 need to be migrated for the source code to adhere to the new EGL language syntax included in Interim Fix 0004 (or 0001).

1. Migrate the EGL source.
   a. Enable EGL V6.0 Migration capability by selecting **Window** → **Preferences**. Expand **Workbench** and select **Capabilities**. Expand **EGL Developer** and check **EGL V6.0 Migration**. Click **OK**.
   b. Right-click the **EGLSource** folder, and select **EGL V6.0 Migration** → **Migrate**.
2. Manually copy the new EGL runtime libraries listed in Table 14-2 on page 820 from a newly created EGL Web Project (contains correct level of runtime files) to the same folder of the project being migrated.

Verify EGL plug-ins exist

After installing the Enterprise Generation Language component, verify that the EGL plug-ins are configured.

1. From the menu bar of Rational Application Developer, select **Help** → **About IBM Rational Software Development Platform**.
2. Click **Plug-in Details**.
3. Scroll down the page until you see plug-ins with **IBM** in the Provider column.
4. Find the plug-ins that begin with `EGL`. You should see something like Figure 14-6 on page 771.

Archive

Figure 14-6 Enterprise Generation Language plug-ins

14.3.2 Enable the EGL development capability

The Rational Application Developer Workbench hides certain features by default. The EGL development capability is disabled by default.

To enable the EGL development capability, do the following:

1. From the menu bar, select **Window** → **Preferences**.
2. Expand **Workbench** → **Capabilities**.
3. Select and expand EGL Developer. Depending on your needs, check the appropriate sub features, as seen in Figure 14-7.
   - Check **EGL Developer** - Required for EGL development
   - Check **EGL V6.0 Migration** - Required for V6.0 to V6.0 + Interim Fix 0004

     Note: We needed this feature to migrate an EGL Web Project created with the IBM Rational Application Developer V6.0, since Interim Fix 0001 and 0004 contain an EGL version that has new EGL language syntax.

   - VisualAge Generator to EGL Migration (as needed)
4. Click **OK**.

Archive

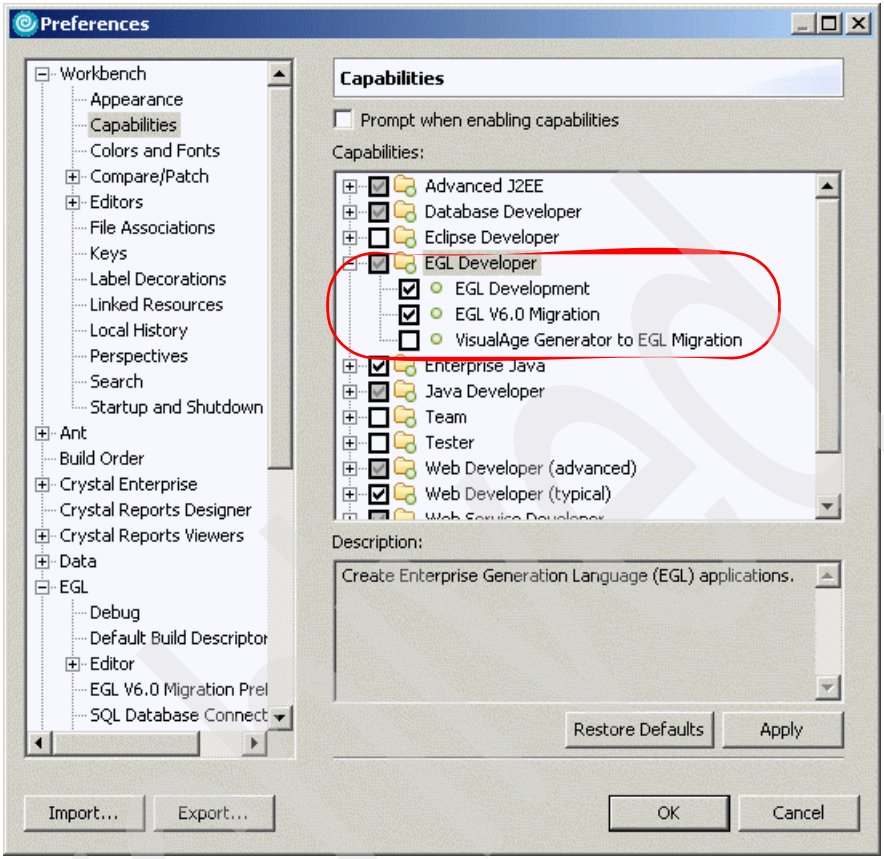


Figure 14-7 Enable EGL Developer capability

14.3.3 Install DB2 Universal Database

For the working example, we installed IBM DB2 Universal Database V8.2 Express Edition included with the IBM Rational Application Developer V6.0 packaging.

Refer to “IBM DB2 Universal Database V8.2 installation” on page 1387 for details on installing DB2 Universal Database.

14.3.4 Create an EGL Web Project

To create an EGL Web Project for the redbook sample application, do the following:

1. Open the EGL perspective.

a. From the menu bar, select **Window** → **Open Perspective** → **Other**.

b. When the Select Perspectives dialog appears, check **Show All**.

c. Select **EGL** and click **OK**.

2. From the menu bar, select **File** → **New** → **Project**.
3. When the New Project dialog appears, expand **EGL**, select **EGL Web Project**, and then click **Next**.
4. When the New EGL Web Project dialog appears, do the following (as seen in Figure 14-8 on page 775), and then click **Next**:

   – Name: `BankEGL`

   – Build Descriptor Options: Select **Create new project build descriptor(s) automatically**.

   – JNDI name for SQL connection: `jdbc/BankDS`

     This value should match the value entered in 14.3.6, “Configure EGL preferences for SQL database connection” on page 779.

     Note: Should you need to change the JNDI name after creating the EGL Web Project, you will need to also modify the sqlJNDIName in the EGL build descriptor (for example, EGLSource\BankEGL.eglbld).

   – Click **Show Advanced**.

     If other EAR projects exist prior to creating the EGL Web Project, you may need to specify the EAR project by clicking **New**. In our example, no other EAR projects existed; thus, the wizard supplied BankEGLEAR as the EAR project name.

Archived

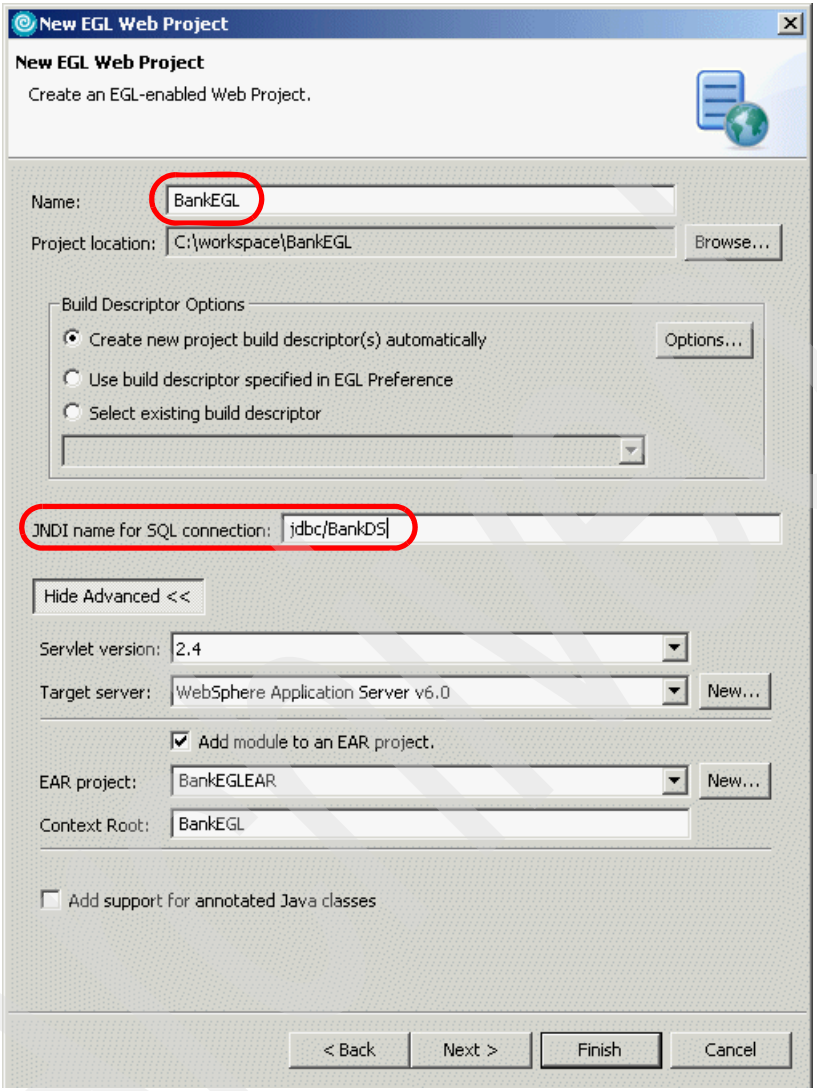


Figure 14-8 New EGL Web Project

5. When the New EGL Web Project - Features dialog appears, notice that Add EGL Support and JSP Standard Tag Library are checked, as seen in Figure 14-9 on page 776. Click **Finish**.

Note: An EGL Web Project is a Dynamic Web Project with Add EGL Support and JSP Standard Tag Library enabled.

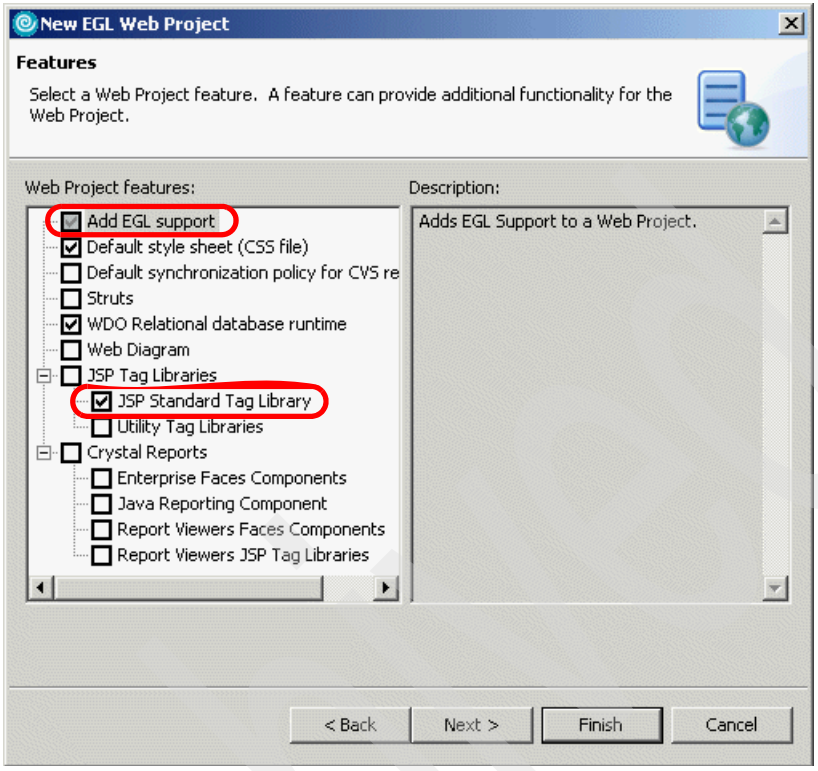


Figure 14-9 New EGL Web Project - Features

Add EGL support to an existing Dynamic Web Project

You can also add EGL support to an existing Dynamic Web Project by doing the following:

1. Open the Web perspective Project Explorer view.
2. Right-click the Dynamic Web Project that you wish to add EGL support to and select **Properties**.
3. In the Properties dialog select **Web Project Features**.
4. From the Available Web Project Features select **Add EGL Support** and **JSP Standard Tag Library**.
5. Click the **Apply** button.

14.3.5 Set up the sample database

This section provides instructions for deploying the BANK sample database and populating the database with sample data. For example, we chose to use IBM DB2 Universal Database V8.2 Express Edition.

To create the database, create the connection, and create and populate the tables for the BANK sample from within Rational Application Developer, do the following:

1. Create the BANK database in DB2 Universal Database.

   For details refer to “Create DB2 UDB database via a DB2 command window” on page 346.

2. Create the database connection from within Rational Application Developer to the BANK database.

   For details refer to “Create a database connection” on page 347.

   Note: When using DB2 or Oracle you will have to provide the user information by providing the user ID and password. Also, you may need to update the class location. Once the user ID and password have been entered, you can click **Test Connection**.

3. Create the BANK database tables from within Rational Application Developer.

   For details refer to “Create DB2 UDB database tables via a DB2 command window” on page 351.

4. Populate the BANK database tables with sample data from within Rational Application Developer.

   For details refer to “Populate the tables via a DB2 UDB command window” on page 354.

After creating the BANK database, tables, and loading the sample data, view the database tables by opening the Database Explorer in the Data perspective, as seen in Figure 14-10 on page 778.

Archived

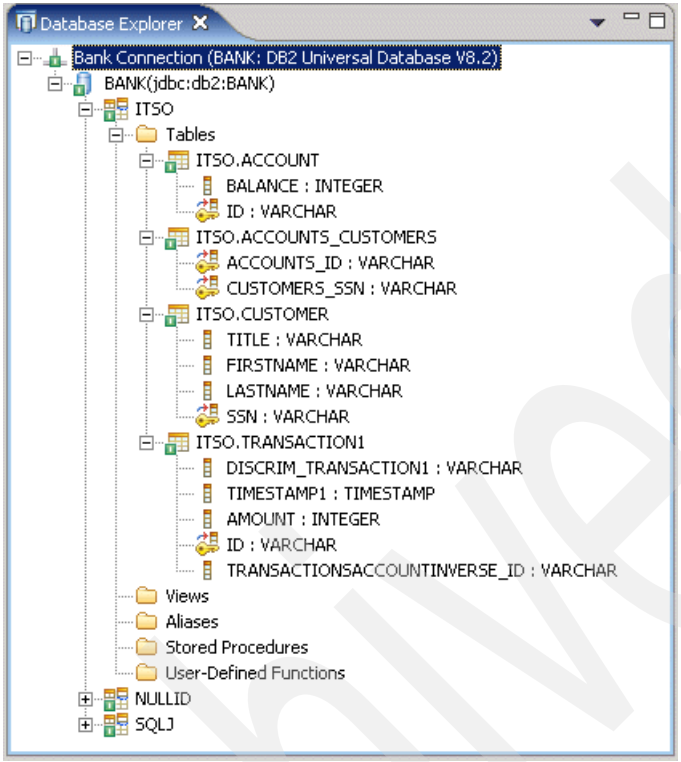
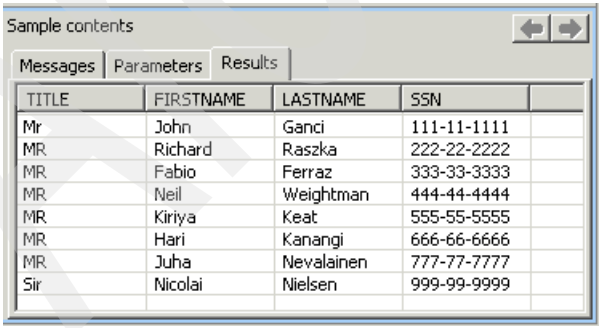


Figure 14-10 BANK database schema in the Database Explorer view

Right-click the **ITSO.CUSTOMER** table, and select **Sample contents**. You should see results like Figure 14-11 in the DB Output view.



TITLE	FIRSTNAME	LASTNAME	SSN
Mr	John	Ganci	111-11-1111
MR	Richard	Raszka	222-22-2222
MR	Fabio	Ferraz	333-33-3333
MR	Neil	Weightman	444-44-4444
MR	Kiriya	Keat	555-55-5555
MR	Hari	Kanangi	666-66-6666
MR	Juha	Nevalainen	777-77-7777
Sir	Nicolai	Nielsen	999-99-9999

Figure 14-11 ITSO.CUSTOMER table sample data in DB Output view

Archived

14.3.6 Configure EGL preferences for SQL database connection

The EGL wizards rely upon the SQL database connection being configured. To configure the EGL preferences with the appropriate DB2 SQL database connection, do the following:

1. Select **Window** → **Preferences**.
2. When the Preferences dialog appears, expand **EGL and select SQL Database Connection**.
3. When the SQL Database Connection dialog appears, enter the following (as seen in Figure 14-12 on page 780), and then click **OK**:
   - Connection URL: `jdbc:db2:BANK`
   - Database: `BANK`
   - Database vendor type: Select **DB2 Universal Database Express V8.2**.
   - JDBC driver: Select **IBM DB2 App Driver**.
   - JDBC driver class: `COM.ibm.db2.jdbc.app.DB2Driver`
   - Class location: `C:\Program Files\IBM\SQLLIB\java\db2java.zip`
   - Connection JNDI name: `jdbc/BankDS`

   Note: This value should match the value entered when creating the EGL Web Project in 14.3.4, “Create an EGL Web Project” on page 773.

Archived

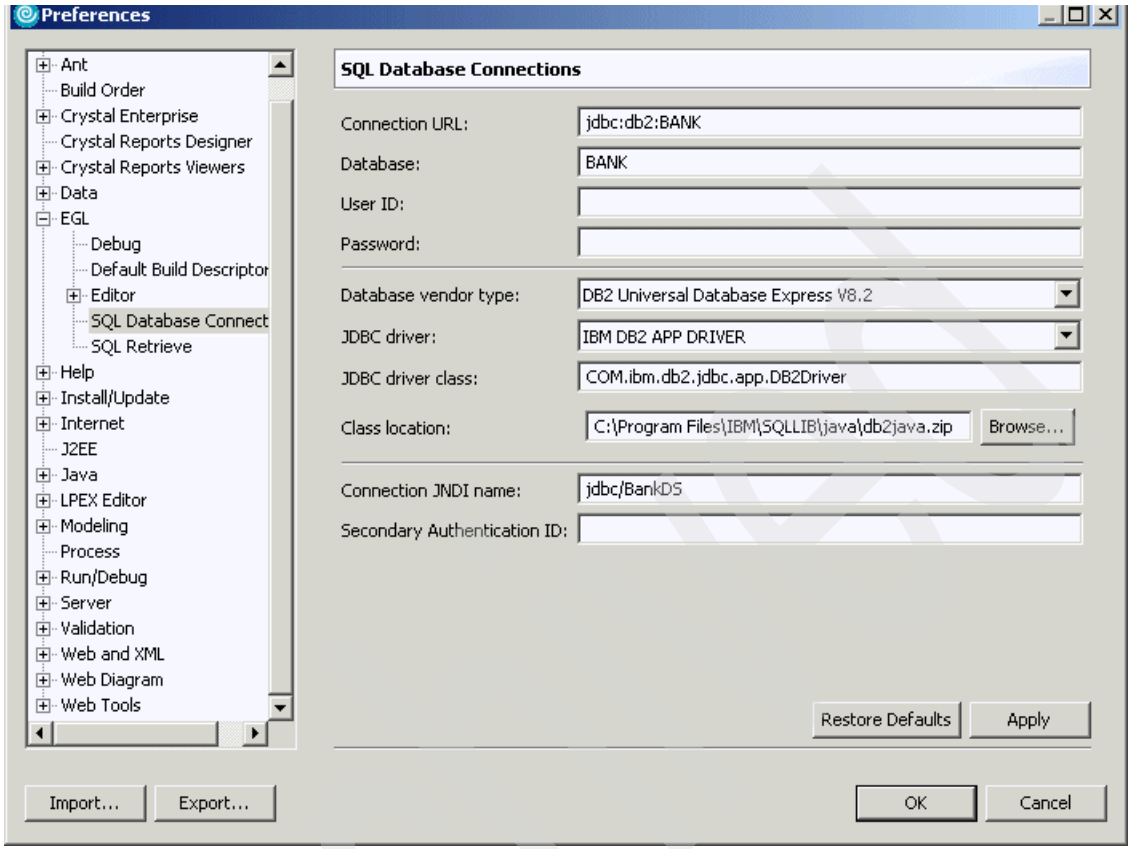


Figure 14-12 EGL preferences

14.3.7 Configure the data source

There are a couple methods that can be used to configure the datasource, including using the WebSphere Administrative Console or using the WebSphere Enhanced EAR, which stores the configuration in the deployment descriptor and is deployed with the application.

This section describes how to configure the datasource using the WebSphere Enhanced EAR capabilities. The enhanced EAR is configured in the Deployment tab of the EAR deployment descriptor.

The procedure found in this section considers two scenarios for using the enhanced EAR:

- If you choose to import the complete sample code, you will only need to verify that the value of the databaseName property in the deployment descriptor matches the location of your database.
- If you are going to complete the working example Web application found in this chapter, you will need to create the JDBC provider and datasource, and update the databaseName property.

Note: For more information on configuring data sources and general deployment issues, refer to Chapter 23, “Deploy enterprise applications” on page 1189.

Configure authentication

To configure the authentication settings in the enterprise application deployment descriptor where the enhanced EAR settings are defined, do the following:

1. Open the Web perspective Project Explorer view.
2. Expand **Enterprise Applications** → **BankEGLEAR**.
3. Double-click **Deployment Descriptor : BankEGLEAR** to open the file in the Deployment Descriptor Editor.
4. Click the **Deployment** tab.
5. Click **Authentication** (lower left of page).
6. Click **Add**.
7. When the Add JASS Authentication Entry dialog appears, enter the following and then click **OK**:
   - Alias: `dbuser`
   - User ID: `db2admin`
   - Password: `<your_db2admin_password>`

Configure a new JDBC provider

To create a new JDBC provider for DB2 Universal Database, do the following:

1. From the Deployment tab of the Application Deployment Descriptor, click **Add** under the JDBC provider list.
2. When the Create a JDBC Provider dialog appears, select **IBM DB2** as the Database type, select **DB2 Universal Database JDBC Driver Provider (XA)** as the JDBC provider type, and then click **Next**.
3. Enter `DB2 Universal JDBC Driver Provider (XA)` in the Name field (as seen in Figure 14-13 on page 782), and then click **Finish**.

Note: Our example only requires the db2jcc.jar and db2jcc_license_cu.jar.

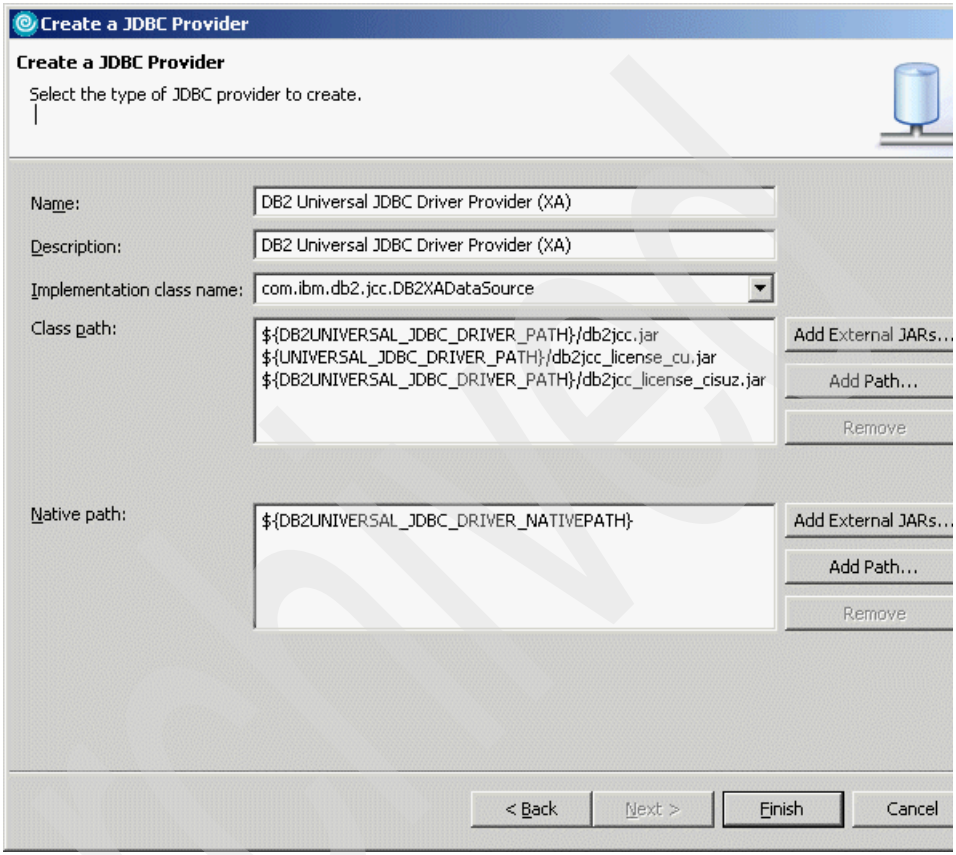


Figure 14-13 Add new JDBC provider

Configure the data source

To configure a new data source using the enhanced EAR capability in the deployment descriptor, do the following:

1. From the Deployment tab of the Application Deployment Descriptor, click the JDBC provider created in the previous step.
2. Click **Add** next to data source.
3. When the Create a Data Source dialog appears, select **DB2 Universal Database JDBC Driver Provider (XA)** under the JDBC provider, select **Version 5.0 data source**, and then click **Next**.

4. When the Create a Data Source dialog appears, enter the following and click **Finish**:
   - Name: `BankDS`
   - JNDI name: `jdbc/BankDS`
   - Component managed authentication alias: Select **dbuser**.

   Note: The user is configured in “Configure authentication” on page 781.

Configure the databaseName property

To configure the databaseName in the new data source using the enhanced EAR capability in the deployment descriptor to define the location of the database for your environment, do the following:

1. Select the data source created in the previous section.
2. Select the **databaseName** property under the Resource properties.
3. Click **Edit** next to Resource properties to change the value for the databaseName.
4. When the Edit a resource property dialog appears, enter `BANK` in the Value field and then click **OK**.
5. Save the Application Deployment Descriptor.

14.3.8 Configure the DB2 JDBC class path environment variables

The JDBC provider for DB2 Universal Database, which we configured in “Configure a new JDBC provider” on page 781, depends upon environment variables being defined. We chose to update the environment variable values for the WebSphere Application Server environment in which the application will be deployed via the WebSphere Administrative Console.

Table 14-1 DB2 Universal Database JDBC driver classpath environment variables

Variable name	Path value
DB2UNIVERSAL_JDBC_DRIVER_PATH	C:\Program Files\IBM\SQLLIB\java
UNIVERSAL_JDBC_DRIVER_PATH	${WAS_INSTALL_ROOT}/universalDriver/lib

To configure the DB2 Universal Database JDBC driver classpath environment variables listed in Table 14-1, do the following:

1. Open the Web perspective.
2. From the Servers view, right-click the WebSphere Application Server v6.0 test server, and select **Start**.

3. After the server is started, right-click the WebSphere Application Server V6.0 test server, and select **Run administrative console**.
4. When prompted for a user ID, you can enter an ID or simply click **Log in** since WebSphere security is not enabled.
5. Expand **Environment**.
6. Click **WebSphere Variables**.
7. Ensure that the variables listed in Table 14-1 on page 783 are configured correctly for your environment at the node level. If they are not configured, enter the appropriate path values. Click **OK**.
8. Click **Save**, and then **Save to Master**.
9. Click **Logout**.
10. Restart the WebSphere Application Server V6.0 test server.

14.4 Develop the Web application using EGL

This section provides a working example describing how to develop a Web application using EGL. The application will use EGL for database access (records, libraries) and Faces JSPs with EGL page handlers for the user interface.

This section includes the following tasks:

- Create the EGL data parts.
- Create and customize a page template.
- Create the Faces JSPs using the Web Diagram tool.
- Add EGL components to the Faces JSPs.

When developing a Web application using EGL, there are several options that can be used to get started, including:

- EGL Data Parts and Pages wizard.

  When using the EGL Data Parts and Pages wizard, you will be guided through a sequence of dialogs that result in creating an EGL Web Project.

  Within minutes you will have a simple working application, customized for your database schema, ready for you to further customize.

  The down side to this approach is that you have less control of what is being created. Furthermore, the wizard can only be used to initially create a project, not update the contents of an existing project.

- Create EGL Web Project, Data Parts, and Faces JSPs individually.

Archived

When using this approach, we will still take advantage of the EGL wizards and tooling provided, with more control with what resources are created.

In our working example, we chose this approach. We will first create an EGL Web Project. Then we will create the EGL source for database access using the EGL Data Parts wizard. Finally we will create the Faces JSPs and EGL page handlers. We will then generate the Java resources and test the Web application on the WebSphere Application Server V6.0 test server.

14.4.1 Create the EGL data parts

There are a few methods of creating EGL data parts. In our example, we chose to use the EGL Data Parts wizard to create the EGL source files that define an SQL record type and reusable functions associated for the record type.

This section is organized as follows:

- Create records and libraries via the EGL Data Parts wizard.
- Summary of code created by the EGL Data Parts wizard.
- Generate the Java code for the EGL file.
- Create the Customer to Account relationship.
- Modify the SQL in the EGL source code.

Create records and libraries via the EGL Data Parts wizard

To create the records and libraries for customers and account, using the EGL Data Parts wizard, do the following:

1. Open the EGL perspective, Project Explorer view.
2. Expand **Dynamic Web Projects** → **BankEGL**.
3. Right-click the **EGLSource** folder, and select **New** → **EGL Data Parts**.
4. When the Generate EGL Components dialog appears, do the following:
   a. EGL project name: Select **BankEGL**.
   b. Database connection: Click **Add**.

Archived

Note: At the time of writing, when using the EGL Data Parts wizard, only database connections for *DB2 aliases* worked. If a database connection was created with the *Database Manager and JDBC driver* option, you will not be able to use the database connection in the EGL Data Parts wizard (once selected, the wizard will not allow you to click Next).

Additionally, we found that once a database connection is used by the EGL Data Parts wizard, it cannot be used again to create a new record with the EGL Data Parts wizard.

You can manage database connections in the Database Explorer view of the Data perspective.

c. When the Establish a connection to a database dialog appears, select **Choose a DB2 alias**, enter `EGL Bank Connection` in the Connection name field, and then click **Next**.

d. When the Specify connection parameters dialog appears, do the following:

   i. For the JDBC driver, select **IBM DB2 Universal**.

   ii. For the Alias, select **BANK**.

   iii. Click **Test Connection** to verify the settings. You should see the message `Connection to BANK is successful.` Click **OK**.

   iv. Click **Finish**.

e. When you return to the Generate EGL Components dialog, select **ACCOUNT** and **CUSTOMERS** under the Select your data column, and click the **>** to add to the column on the right of the dialog. When done the dialog should look like Figure 14-14 on page 787. Click **Next**.

Archive

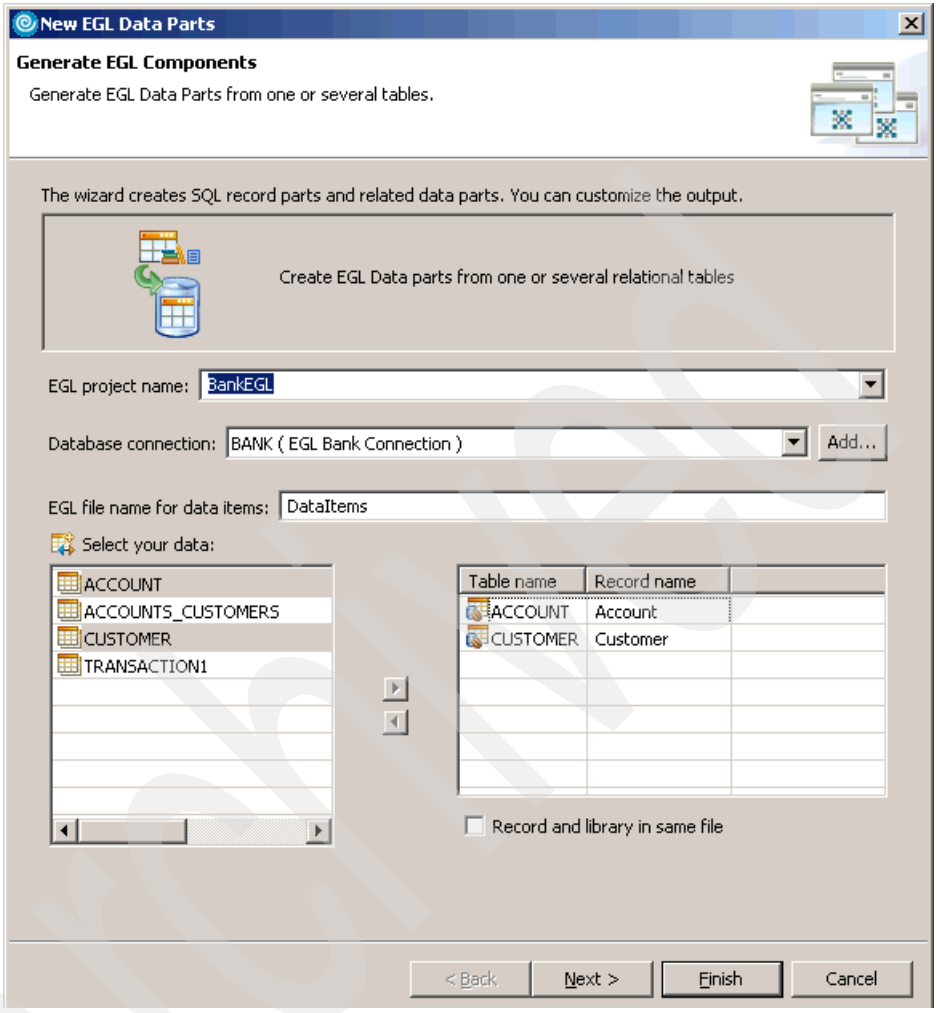


Figure 14-14 New EGL Data Parts - Generate EGL Components

Note: The "Record and library in same file" check box seen in Figure 14-14 on page 787 allows us to keep the record and library content in the same file. In our example, we chose to keep them separate (default).

5. When the Define the Fields dialog appears, as seen in Figure 14-15 on page 788, we accepted the default settings and clicked **Next**.

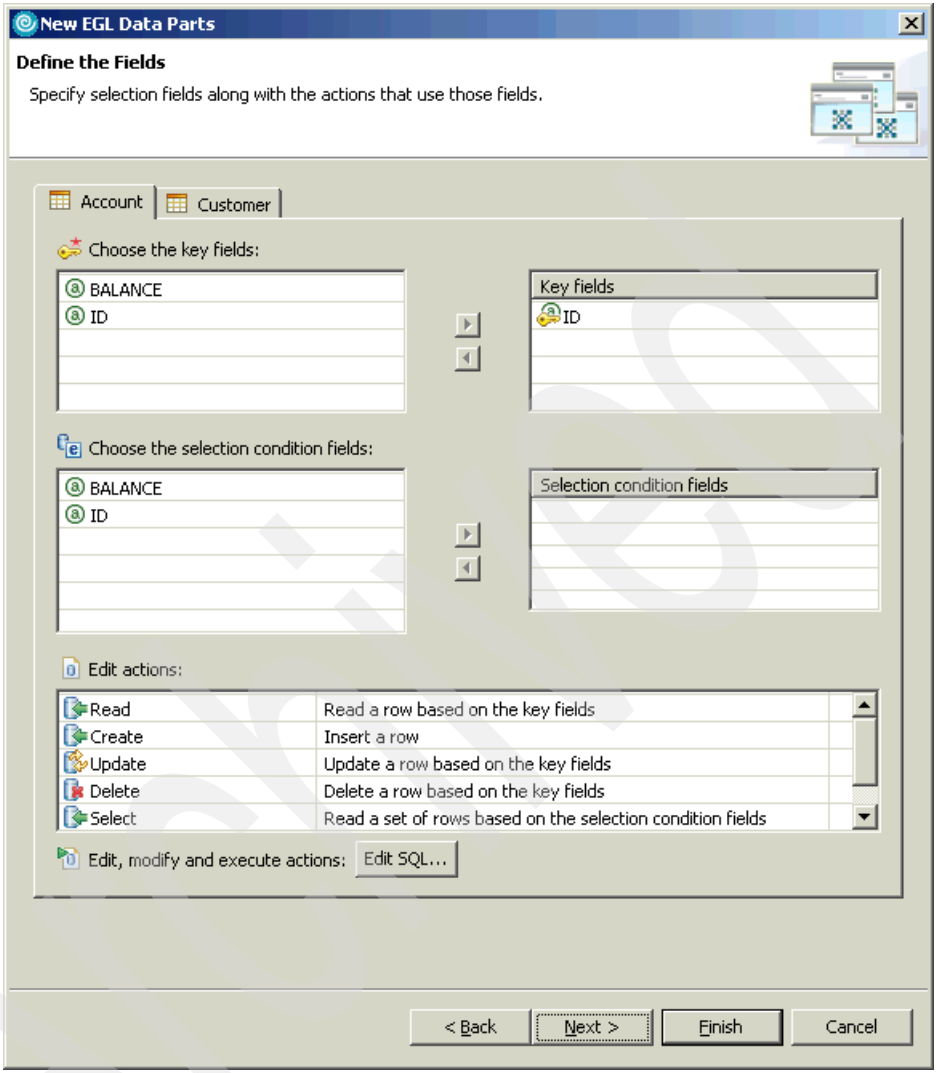


Figure 14-15 New EGL Data Parts - Define the Fields

6. When the Generate EGL Data Parts Summary dialog appears, as seen in Figure 14-16 on page 789, we accepted the default settings and clicked **Finish**.

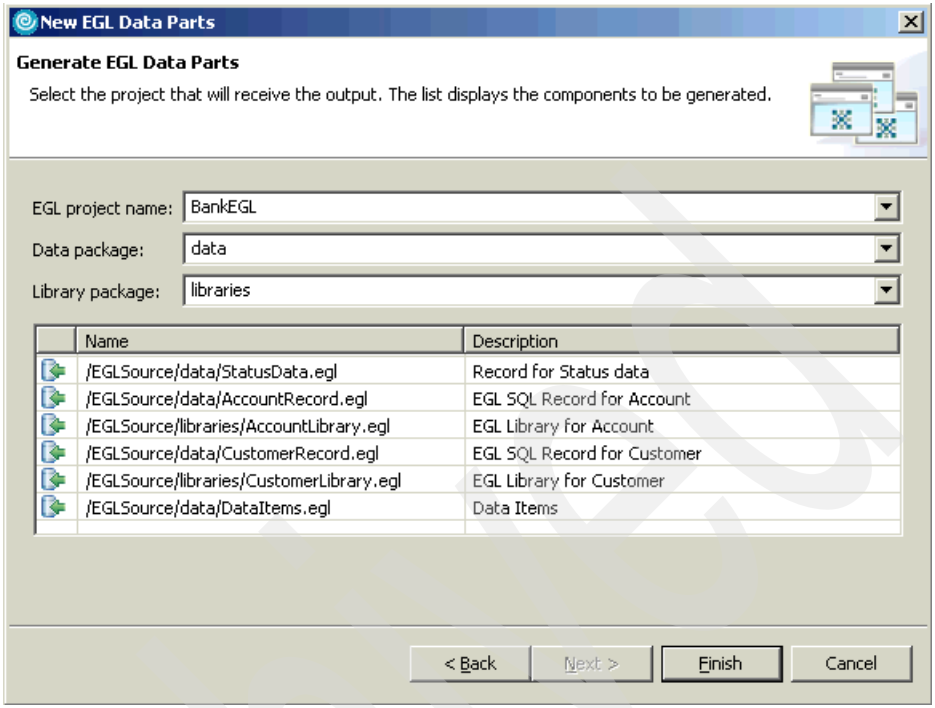


Figure 14-16 New EGL Data Parts - Summary

7. You should see a dialog with the message `Successful Generation`. Click **OK**.

Summary of code created by the EGL Data Parts wizard

The objective of this section is to provide a basic understanding of what was created by the EGL Data Parts wizard.

Figure 14-17 on page 790 displays the EGL and Java source files generated by the EGL Data Parts wizard for the sample. Notice there are data and libraries package folders under the EGLSource folder. Also, notice there is a JavaSource folder that has corresponding packages for data and libraries. When the EGL Generate feature is executed on the EGL source, the Java source will be generated in the appropriate JavaSource sub folder.

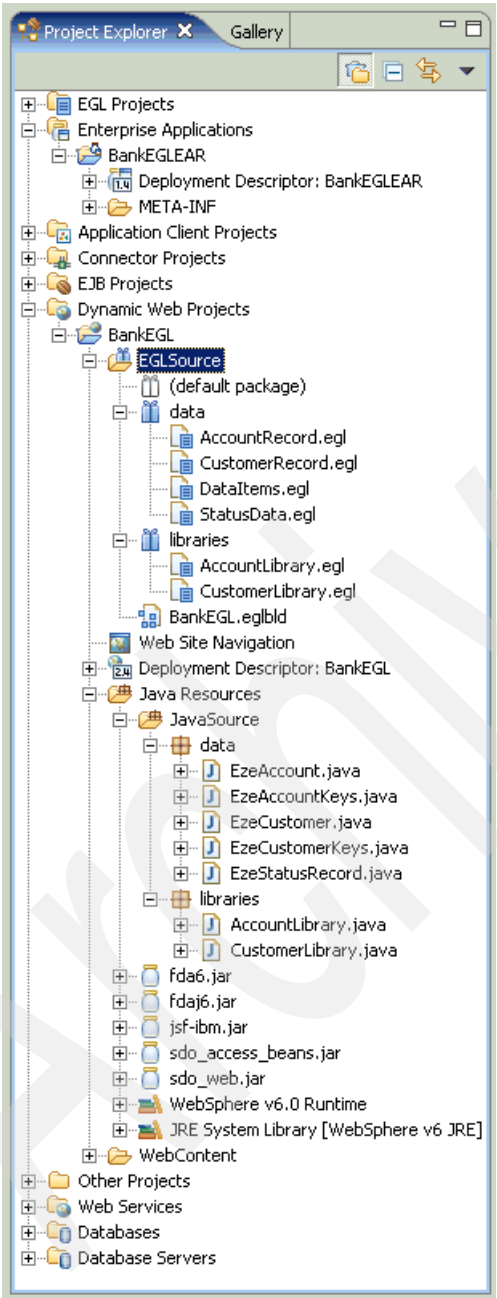


Figure 14-17 EGL and Java source files generated by the EGL Data Parts wizard

Archived

The data folder in the EGLSource contains EGL source files with record definitions for the database schema. For example, the CustomerRecord.egl file contains the record definition for the Customer table, as seen in Example 14-1.

Example 14-1 Sample CustomerRecord.egl

```
package data;

Record Customer type SQLRecord

   { tableNames = [["ITSO.CUSTOMER"]],
   keyitems = ["ssn"] }

   title TITLE {column = "TITLE", sqlVariableLen = yes, maxlen = 250, isNullable = yes};
   firstname FIRSTNAME {column = "FIRSTNAME", sqlVariableLen = yes, maxlen = 250, isNullable =
yes};
   lastname LASTNAME {column = "LASTNAME", sqlVariableLen = yes, maxlen = 250, isNullable =
yes};
   ssn SSN      {column = "SSN", sqlVariableLen = yes, maxlen = 250};

// used in arrays: index of the row
   indexInArray int {persistent = no};
end

// define the data to store in the session for a detail of Customer
 Record CustomerKeys type BasicRecord
  ssn SSN;
end

// define the data to store in the session for a list of Customer
 Record CustomerSessionListData type BasicRecord
// index of the first item of the current page
   indexOfCurrentPage int;
end
```

The record definition in Example 14-1 uses the type names SSN, TITLE, FIRSTNAME, and LASTNAME to define the types for the fields of the record. These are not built-in types, but defined in the file DataItems.egl, placed in the same folder as the record definition. Example 14-2 displays the contents of the DataItems.egl file. In addition to the type definitions for the Customer record, Example 14-2 also contains type definitions for the fields ID and BALANCE from the Account record.

Example 14-2 Sample DataItems.egl

```
package data;

DataItem BALANCE int {displayName = "BALANCE"} end
```

Archived

```
DataItem ID string {displayName = "ID"} end
DataItem SSN string {displayName = "SSN"} end
DataItem LASTNAME string {displayName = "LASTNAME"} end
DataItem FIRSTNAME string {displayName = "FIRSTNAME"} end
DataItem TITLE string {displayName = "TITLE"} end
```

To access the record, functions were created in the CustomerLibrary.egl file found in the libraries directory. Example 14-3 lists the contents of the CustomerLibrary.egl containing functions to manipulate the Customer record.

Example 14-3 Sample CustomerLibrary.egl

```
package libraries;
import data.StatusRecord;
import data.Customer;
import data.CustomerKeys;

Library CustomerLibrary

   /* Pass ssn in via Customer argument.
   Customer is returned if found. Status is returned with success or failure */
   Function readCustomer (customer Customer, sqlStatusData StatusRecord)

      try
         get customer;
         sqlStatusData.sqlStatus = 0;
      onException
         sqlStatusData.sqlStatus = sysvar.sqlCode;
         sqlStatusData.description = syslib.currentException.description;
      end
   end

   /* Pass ssn in via CustomerKeys argument.
   Customer is returned if found. Status is returned with success or failure */
   Function readCustomerFromKeyRecord (customer Customer, CustomerKeys customerKeys,
sqlStatusData StatusRecord)

      customer.ssn = customerKeys.ssn;
      readCustomer (customer, sqlStatusData);
   end

   /* Pass Customer to be created argument.
   Status is returned with success or failure */
   Function createCustomer (customer Customer, sqlStatusData StatusRecord)

      try
         add customer;
         sqlStatusData.sqlStatus = 0;
      onException
```

```
      sqlStatusData.sqlStatus = sysvar.sqlCode;
      sqlStatusData.description = syslib.currentException.description;
   end
end

/* Pass Customer to be deleted argument.
Status is returned with success or failure */
Function deleteCustomer (customer Customer, sqlStatusData StatusRecord)

   try
      execute #sql{
       DELETE FROM ITSO.CUSTOMER WHERE CUSTOMER.SSN = :customer.ssn
      };
      sqlStatusData.sqlStatus = 0;
   onException
      sqlStatusData.sqlStatus = sysvar.sqlCode;
      sqlStatusData.description = syslib.currentException.description;
   end
end

/* Pass Customer to be updated argument.
Status is returned with success or failure */
Function updateCustomer (customer Customer, sqlStatusData StatusRecord)
   try
      execute #sql{
       UPDATE ITSO.CUSTOMER SET TITLE = :customer.title, FIRSTNAME = :customer.firstname,
        LASTNAME = :customer.lastname WHERE CUSTOMER.SSN = :customer.ssn
      };
      sqlStatusData.sqlStatus = 0;
   onException
      sqlStatusData.sqlStatus = sysvar.sqlCode;
      sqlStatusData.description = syslib.currentException.description;
   end
end

/* Pass Customer[] dynamic array to be returned with data.
Status is returned with success or failure */
Function selectCustomer (customer Customer[], sqlStatusData StatusRecord)

   try
      get customer;
      sqlStatusData.sqlStatus = 0;
   onException
      sqlStatusData.sqlStatus = sysvar.sqlCode;
      sqlStatusData.description = syslib.currentException.description;
   end
end
```

Archived

Generate the Java code for the EGL file

In our example, the Java source code was generated from the EGL source files automatically by the EGL Data Parts wizard. After modifying the EGL source files, the files are typically generated automatically. On occasion, you will need to run the Generate feature manually to generate the corresponding Java source.

Generate Java from an individual EGL library source file

To generate Java from an individual EGL library source file, do the following:

1. Open the EGL perspective Project Explorer view.
2. Expand **Dynamic Web Projects** → **BankEGL** → **EGLSource**.
3. Right-click the EGL library source file, and select **Generate**.

   Alternatively, in the Source Editor view of an EGL source file, right-click and then select **Generate** (or press CTRL+G).

Attention: Java source for EGL data (records) source files are only generated when a EGL library source file that references the record is being generated.

Although it is possible to select an EGL record source file and run the Generate task, it will not generate Java source. You must run the Generate task on an EGL library source file that references the record.

Generate all EGL files in a EGL Web Project

To generate EGL files in a EGL Web Project, do the following:

1. From the EGL perspective, expand **Dynamic Web Projects**.
2. Right-click **BankEGL**, and select **Generate With Wizard**.
3. When the Generate wizard appears, click **Select All**, and click **Next**.
4. Select **Use one build descriptor for all parts**, select **BankEGLWebBuildOptions <BankEGL/EGLSource/BankEGL.eglbld>** from the drop-down list, and then click **Finish**.

Create the Customer to Account relationship

In our example, we needed to create an EGL record that joins the Accounts and Accounts_Customers tables, so that we can retrieve all account information for a customer. At the time of writing, the EGL tooling did not provide this capability; therefore, we needed to add the EGL source manually.

Create the Customer_AccountsRecord.egl

To create the Customer_AccountsRecord.egl, do the following:

1. Open the EGL perspective.
2. Expand Dynamic **Web Projects** → **BankEGL** → **EGLSource**.
3. Right-click **data**, and select **New** → **EGL Source File**.
4. When the New EGL Source file dialog appears, enter `Customer_AccountsRecord` in the EGL source file name field, and click **Finish**.
5. Open the Customer_AccountsRecord.egl file.
6. Enter the contents of Example 14-4 into the Customer_AccountsRecord.egl file.

> **Note:** The code in Example 14-4 can be found in c:\6449code\egl\Customer_AccountsRecordSnippet.txt.

Example 14-4 Add contents to Customer_AccountsRecord.egl

```
package data;

Record Customer_Accounts type SQLRecord {
   tableNames = [["ITSO.ACCOUNTS_CUSTOMERS", "AC"], ["ITSO.ACCOUNT", "A"]],
   defaultSelectCondition = #sqlCondition{
         AC.ACCOUNTS_ID=A.ID
         and AC.CUSTOMERS_SSN=:customers_ssn }}
end
```

7. Place the cursor inside the record, and press Ctrl+Shift+R to retrieve the record data.

   After running the SQL retrieve, the Customer_AccountsRecord.egl file should look similar to Example 14-5 (SQL formatted for readability).

> **Note:** The code in Example 14-5 can be found in c:\6449code\egl\Customer_AccountsRecord.egl.

Example 14-5 After SQL retrieve - Customer_AccountsRecord.egl

```
package data;

Record Customer_Accounts type SQLRecord {
   tableNames = [["ITSO.ACCOUNTS_CUSTOMERS", "AC"], ["ITSO.ACCOUNT", "A"]],
   defaultSelectCondition = #sqlCondition{
         AC.ACCOUNTS_ID=A.ID
         and AC.CUSTOMERS_SSN=:customers_ssn },
   keyItems=["ACCOUNTS_ID", "CUSTOMERS_SSN", "ID"]}
```

```
   ACCOUNTS_ID string {
         column="AC.ACCOUNTS_ID", isReadOnly=yes,
         sqlVariableLen=yes, maxLen=250};
   CUSTOMERS_SSN string {
         column="AC.CUSTOMERS_SSN", isReadOnly=yes,
         sqlVariableLen=yes, maxLen=250};
   BALANCE int {
         column="A.BALANCE", isReadOnly=yes};
   ID string {
         column="A.ID", isReadOnly=yes,
         sqlVariableLen=yes, maxLen=250};
end
```

8. Save the Customer_AccountsRecord.egl file.

Create the Customer_AccountsLibrary.egl

To create the Customer_AccountsLibrary.egl, do the following:

1. Open the EGL perspective.
2. Expand Dynamic **Web Projects** → **BankEGL** → **EGLSource**.
3. Right-click **library**, and select **New** → **EGL Source File**.
4. When the New EGL Source file dialog appears, enter `Customer_AccountsLibrary` in the EGL source file name field, and click **Finish**.
5. Open the Customer_AccountsLibrary.egl file.
6. Enter the contents of Example 14-6 into the Customers_AccountsLibrary.egl file.

> **Note:** The contents of Example 14-4 can be found in the c:\6449code\egl\Customer_AccountsLibrary.egl.

Example 14-6 Add contents to Customer_AccountsLibrary.egl

```
package libraries;
import data.StatusRecord;
import data.Customer;
import data.Customer_Accounts;
import data.Customer_AccountsKeys;

Library Customer_AccountsLibrary


   Function getCustomerAccountsByCustomer(
            customer Customer,
```

```
            customerAccounts Customer_Accounts[],
            sqlStatusData StatusRecord)
        returns (int)

        customers_ssn char(250);

        customers_ssn = Customer.ssn;

        try
           get customerAccounts usingkeys customers_ssn;
           sqlStatusData.sqlStatus = 0;
        onException
           sqlStatusData.sqlStatus = sysvar.sqlCode;
           sqlStatusData.description = syslib.currentException.description;
        end
    end
end
```

7. Save the Customers_AccountsLibrary.egl file.
8. Generate the Java source from the EGL library.

   For details refer to “Generate Java from an individual EGL library source file” on page 794.

Modify the SQL in the EGL source code

At the time of writing, we found that the EGL Data Parts wizard prefixed table names with the schema name in the FROM clause; however, the wizard prefixed the column names with the table name, without the schema name. This results in an invalid SQL statement, which requires that you modify the code manually.

Note: While creating our sample, we only tested with the following software:

- IBM Rational Application Developer V6.0 with Interim Fix 0004
- IBM DB2 Universal Database V8.2 Express Edition (included with Rational Application Developer)

We were not able to verify if this issue applies to other database platforms.

While testing our sample application, we found that our database update function for customer data did not work properly due to this issue. We have included a sample of the generated and modified source to resolve this issue:

- Generated EGL library source for deleting a customer (formatted for readability):

```
DELETE FROM ITSO.CUSTOMER
WHERE CUSTOMER.SSN = :customer.ssn
```

- Corrected EGL library source for deleting a customer:

```
DELETE FROM ITSO.CUSTOMER
WHERE ITSO.CUSTOMER.SSN = :customer.ssn
```

- Generated EGL library source for updating a customer (formatted for readability):

```
UPDATE ITSO.CUSTOMER
SET    TITLE = :customer.title,
       FIRSTNAME = :customer.firstname,
       LASTNAME = :customer.lastname
WHERE CUSTOMER.SSN = :customer.ssn
```

- Corrected EGL library source for updating a customer:

```
UPDATE ITSO.CUSTOMER
SET    TITLE = :customer.title,
       FIRSTNAME = :customer.firstname,
       LASTNAME = :customer.lastname
WHERE ITSO.CUSTOMER.SSN = :customer.ssn
```

To work around this issue for our sample application, we modified the EGL library source file generated by the EGL Data Parts wizard to include the schema name.

1. Open the Web perspective.
2. Expand the **Dynamic Web Projects** → **BankEGL** → **EGLSource** → **libraries**.
3. Double-click **CustomerLibrary.egl** to open in the source editor.
4. Modify the generated SQL code to match the corrected SQL that includes the schema name (ITSO) as follows:
   - Generated:

```
UPDATE ITSO.CUSTOMER
SET    TITLE = :customer.title,
       FIRSTNAME = :customer.firstname,
       LASTNAME = :customer.lastname
WHERE CUSTOMER.SSN = :customer.ssn
```

   - Corrected:

```
UPDATE ITSO.CUSTOMER
SET    TITLE = :customer.title,
       FIRSTNAME = :customer.firstname,
       LASTNAME = :customer.lastname
WHERE ITSO.CUSTOMER.SSN = :customer.ssn
```

5. Save the CustomerLibrary.egl file.
6. Generate the CustomerLibrary.egl by pressing Ctrl+G.

Archived

14.4.2 Create and customize a page template

This section describes how to create and customize a page template. We will use the page template in a later section to provide a common look and feel for the JSF pages.

Note: For more detailed information on creating and customizing page templates used by JSF pages, refer to the following:

- 13.3.1, “Create a page template” on page 684
- 13.3.2, “Useful views for editing page template files” on page 687
- 13.3.3, “Customize the page template” on page 695

Create a page template

To create a page template containing JSF components, do the following:

1. Open the Web perspective Project Explorer view.
2. Expand **Dynamic Web Projects**.
3. Right-click the **BankEGL** project, and select **New** → **Page Template File** from the context menu.
4. When the New Page Template wizard appears, enter the following and then click **Finish**:
   - Folder: `/BankEGL/WebContent`
   - File name: `BankEGLTemplate`
   - Model: Select **Template containing Faces Components**.
5. When prompted with the message `A page template must contain at least one Content Area which is later filled in by the pages that use the template`, click **OK**. We will add a content area as part of our page template customization.

Customize the page template

Now that you have created a page template, it is likely that you will want to customize the page. This section demonstrates how to make the following common customizations to a page template:

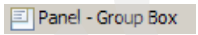
- Customize the logo image and title of the page template.
- Customize the style of the page template.
- Add the content area to the page template.

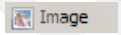
Customize the logo image and title of the page template

To customize the page template to include the ITSO logo image and ITSO RedBank title, do the following:

1. Open the Web perspective.

Archived

2. Import the itso_logo.gif image.
   a. Expand **Dynamic Web Projects** → **BankEGL** → **WebContent** → **theme**.
   b. Right-click the **theme** folder, and select **Import**.
   c. Select **File System** and click **Next**.
   d. Enter `c:\6449code\web` in the From directory, check **itso_logo.gif**, and then click **Finish**.
3. Expand **Dynamic Web Projects** → **BankEGL** → **WebContent**.
4. Double-click **BankEGLTemplate.jtpl** to open the file.
5. Click the **Design** tab.
6. Select the text **Place content here**, right-click, and select **Delete**.
7. From the Palettes view, expand **Faces Components**.
8. Select the **Panel - Group Box** 

and drag it onto the page.
9. When the Select Type dialog appears, select **List** and then click **OK**.
10. You should now see a box on your page with the text `box1: Drag and Drop Items from the palette to this area to populate this region`. From the Faces Components, select **Image** 

and drag it to the panel.
11. Update the image values in the Properties view.
    a. Select the image on the page to highlight the image.
    b. In the Properties view, enter `headerImage` in the Id field.
    c. Click the folder icon next to File and select **Import**. Enter the path to the image and click **Open**. In our example, we entered /WebContent/themes/itso_logo.gif to import the image.
    d. You will notice that the absolute reference has been entered. We need to make it a relative reference by removing the /BankEGL/ from the File field. After making the change, it will be theme/itso_logo.gif without any leading slash.
12. From the Faces Components palette, select **Output**

and drag it under the image.
13. In the Properties view enter `ITSO RedBank` into the Value field.
14. Select the Output box (ITSO RedBank) and drag it to the right of the image.

Customize the style of the page template

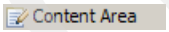
To customize the style of the page template, do the following:

1. Select the Output text box on the page.
2. In the Properties view, click the button next to the Style: Props: field.

3. Change the Size field value to 18.
4. Select **Arial** for the Font and click **Add**.
5. Select **sans-serif** (rule of thumb) for the Font and click **Add**.
6. Click **OK**.

Add the content area to the page template

To add the required content area to the page template, do the following:

1. Right-click under the Output field and from the context menu select **Insert** → **Horizontal Rule**.
2. Expand **Page Template** in the Palette view.
3. From the Page Template, select the **Content Area** 

and drag it under the horizontal rule.
4. When the Insert Content Area for Page Template dialog appears, accept the default name (bodyarea) and click **OK**.
5. Right-click under the content area and from the context menu select **Insert** → **Horizontal Rule**.
6. Save the page template file.

The customized page template file should look similar to Figure 14-18.

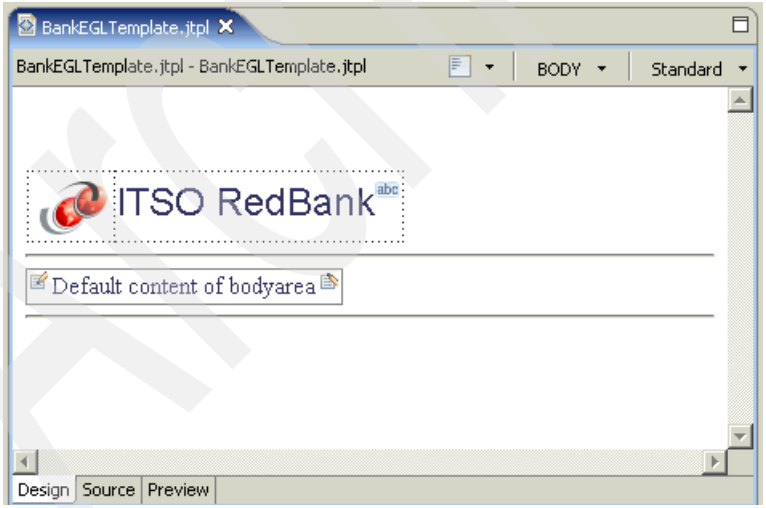


Figure 14-18 Customized page template - BankEGLTemplate.jtpl

Archived

14.4.3 Create the Faces JSPs using the Web Diagram tool

This section demonstrates how to create the Faces JSPs using the Web Diagram tool, including the following tasks:

- Create a Web diagram.
- Create a Web page using the Web Diagram tool.
- Create a Faces JSP file.
- Create connections between Faces JSPs.

The sample application consists of the following pages:

- Login page (Login.jsp): Validate the entered CustomerSSN. If it is valid it will then display the Customer details for the entered customer.
- Customer account details page (ListAccounts.jsp): Display the customer's account details.

Create a Web diagram

By default, and EGL Web Project does not include a Web diagram like other Web Projects. To add a Web diagram to the EGL Web Project, do the following:

1. Open the Web perspective.
2. Expand Dynamic **Web Projects**.
3. Right-click **BankEGL**, and select **New** → **Other**.
4. Expand **Web**, select **Web Diagram**, and click **Next**.
5. When the Web Diagram dialog appears, enter `BankEGL Web Diagram` in the File name field, and then click **Finish**.

Note: Although at a file system level the Web diagram is named Bank EGL Web Diagram.gph, within the Project Explorer it will be named Web diagram if there is only one diagram. After creating more than one Web diagram, they will be displayed with the actual file name in a folder called Web diagrams.

Create a Web page using the Web Diagram tool

To create a page using the Web Diagram tool, do the following:

1. Open the Web perspective Project Explorer view.
2. Expand Dynamic **Web Projects** → **BankEGL**.
3. Double-click **Web Diagram** to open.
4. When the Web diagram appears in the Web Diagram Editor, select **Web Page** from the Web Parts palette and drag it onto the page.

Archived

5. In the Properties view change Web Page Path value to `/Login.jsp`, and change the description to `The login page`.

   Note: Now that we have created a Web page in the Web Diagram tool, you may notice that the BankJSFLogin.jsp icon has a gray-blue tint. The reason for this is that it is not linked to an actual JSP file. We will use this diagram to create the actual JSP file that this icon will link to in a later step.

6. Repeat the process to create a Web page for /ListAccounts.jsp.

Create a Faces JSP file

To create the Faces JSP file from a page template using the Web diagram, do the following:

1. Double-click the **Login.jsp** in the Web diagram.
2. When the New Faces JSP File wizard appears, enter the following and then click **Next**:
   - Folder: `/BankEGL/WebContent`
   - File name: `Login.jsp`
   - Options: Check **Create from page template**.

   Note: If you have not already created the page template, refer to 14.4.2, “Create and customize a page template” on page 799, to create one.

3. When the Page Template File Selection page appears, select **User-defined page template**, select **BankEGLTemplate.jtpl**, and then click **Finish**.
4. The new Faces JSP file will be loaded into the Page Designer. At this point, the newly create JSF page should look like the page template.
5. Double-click **Web Diagram** to repeat the process to create the ListAccounts.jsf Faces JSPs using the BankEGLTemplate.jtpl page template.
6. Save the Web diagram.

   Now that the pages have been realized, the Web page icons should have color and the title icons displayed in bold.

Figure 14-19 on page 804 displays the EGL page handlers and corresponding Faces JSPs in the Project Explorer view. The EGL page handlers are created automatically when the Faces JSPs are created (realized in the Web diagram) within the EGL Web Project.

Archived

The EGL page handlers are used to define the EGL functionality within a Faces JSP. For example, the page handler will contain the code to be executed when a button within the Faces JSP, such as Logout, is clicked.

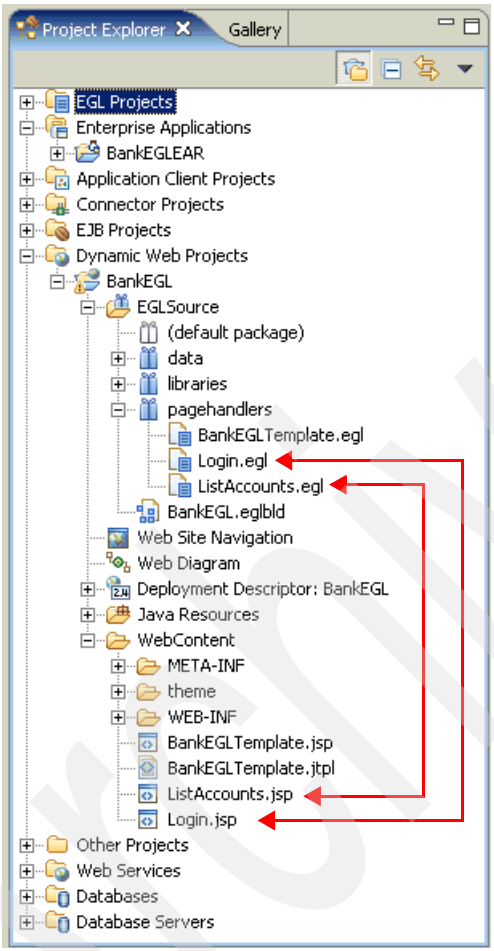


Figure 14-19 Creation of EGL pagehandlers when creating JSF page

Create connections between Faces JSPs

Now that the pages have been created, we can create connections between the pages.

To add connections between pages, do the following:

1. Create a connection from Login.jsp to ListAccounts.jsp.
   a. Click **Connection** Connection from the Palette. Click **Login.jsp**, and then click **ListAccounts.jsp**.

Archived

Note: The Connection palette item is not dragged to the Web diagram like the remaining palette items.

b. When the Choose a Connection dialog appears, select **Faces Outcome** → **ListAccounts** (defaults to target name), as seen in Figure 14-20, and then click **OK**.

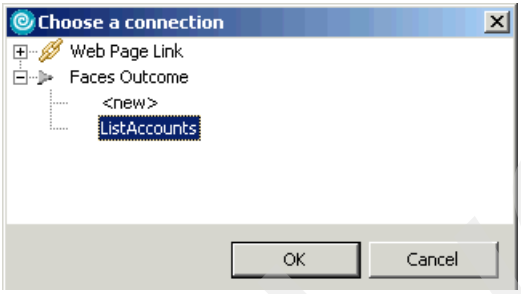


Figure 14-20 Choose a Connection dialog

2. Create a connection from ListAccounts.jsp to Login.jsp.
   a. Click **Connection** Connection from the Palette, click **ListAccounts.jsp**, and then click **Login.jsp**.
   b. When the Choose a Connection dialog appears, select **Faces Outcome** → **<new>** and then click **OK**.
   c. Click **<new>** and change the name to `Logout`.
   d. Double-click **connection** and click **OK**.

When done, your Web diagram should look like Figure 14-21 on page 806.

Archived

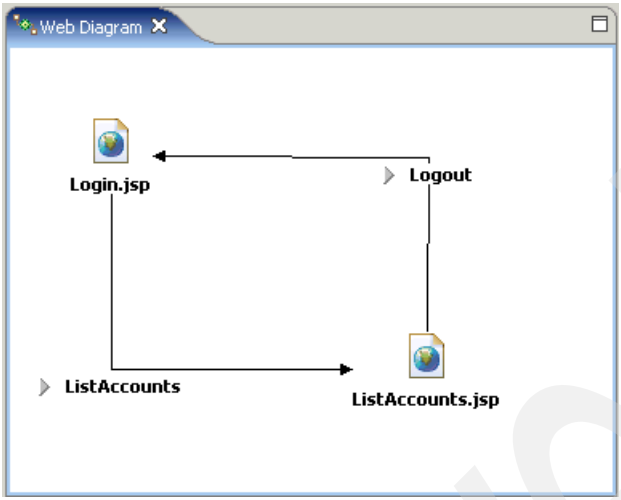


Figure 14-21 Web diagram

14.4.4 Add EGL components to the Faces JSPs

This section describes how to add the EGL components created in 14.4.1, “Create the EGL data parts” on page 785, to the Faces JSPs such that the application will have database access to read and update the database.

Add the content to the Login.jsp

To add the content to the JSP file, do the following:

1. Open the Web perspective, Project Explorer view.
2. Expand **Dynamic Web Projects** → **BankEGL** → **WebContent**
3. Double-click **Login.jsp** to open in the editor.
4. Click the **Design** tab.
5. Select the text **Default content of bodyarea**, and press Delete.
6. In the Palette, expand **EGL**.
7. Click **Record** and drag it to the page data view.
8. When the Select a Record Part dialog appears, do the following (as seen in Figure 14-22 on page 807), and then click **OK**:
   - Select **Customer (data/CustomerRecord.egl)**.
   - Enter the name of the field: `customer` (default)
   - Uncheck **Add controls to display the EGL element on the Web page**.

Archived

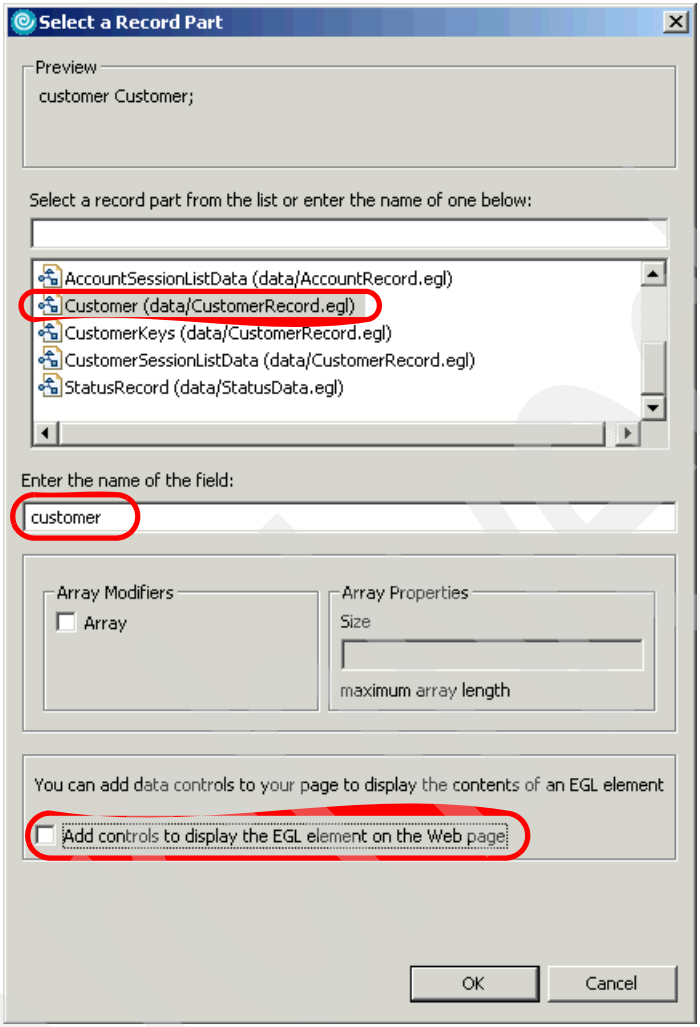


Figure 14-22 Select a Record Part dialog

The Page Data view will now display the customer, as shown in Figure 14-23 on page 808.

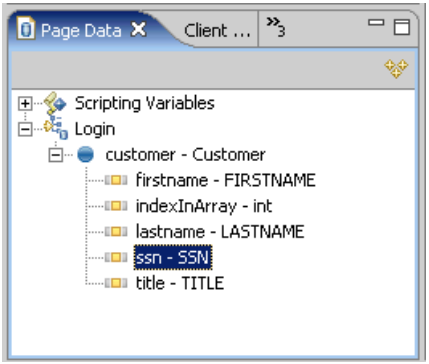


Figure 14-23 Page Data view

9. In the Page Data view, expand **Login** → **customer - Customer**.
10. Select the **ssn - SSN** field from the customer record and drag it onto the page.
11. When the Insert Control dialog appears, do the following:
    a. Select **Updating an existing record**.
    b. For the Label enter `Enter Customer SSN:`
    c. Select **Input field** for the Control Type.
    d. Click **Options**.
    e. When the Options dialog appears, uncheck **Delete button**, enter `Login` in the Label field (as seen in Figure 14-24), and then click **OK**.

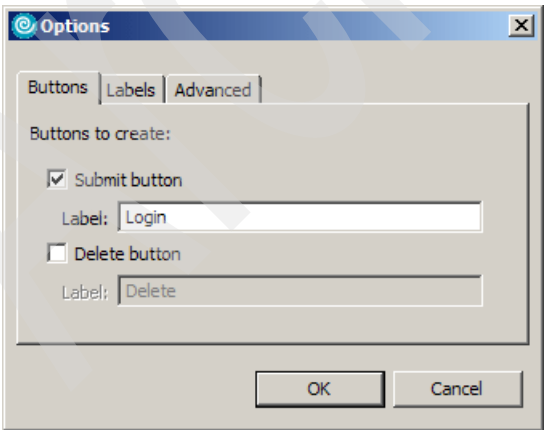


Figure 14-24 The insert control option dialog

Archived

12. The Insert Control page should look like Figure 14-25 on page 809. Click **Finish**.

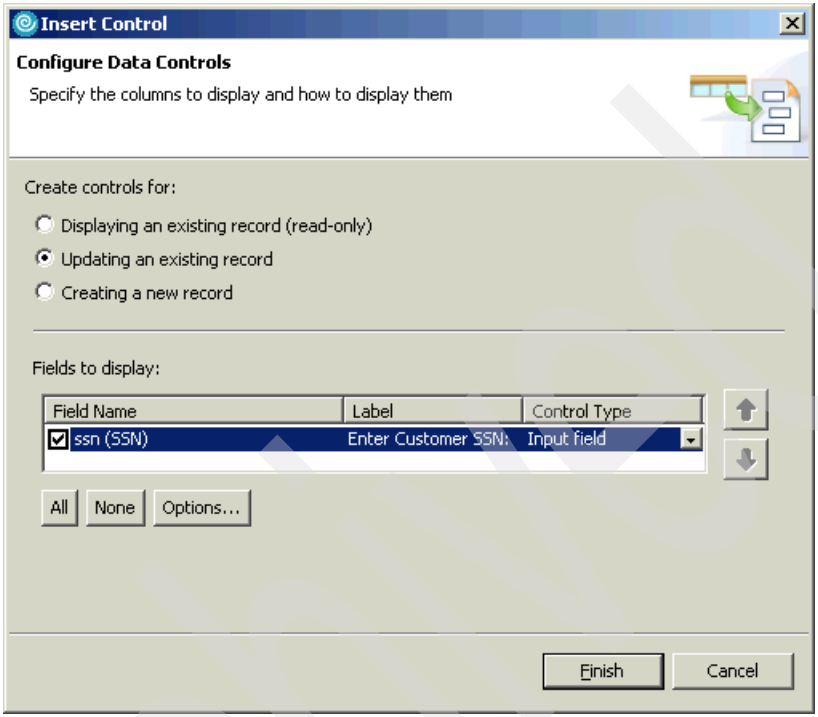


Figure 14-25 Insert Control

The resulting Login page should look similar to Figure 14-26 on page 810 in the Design view.

Archive

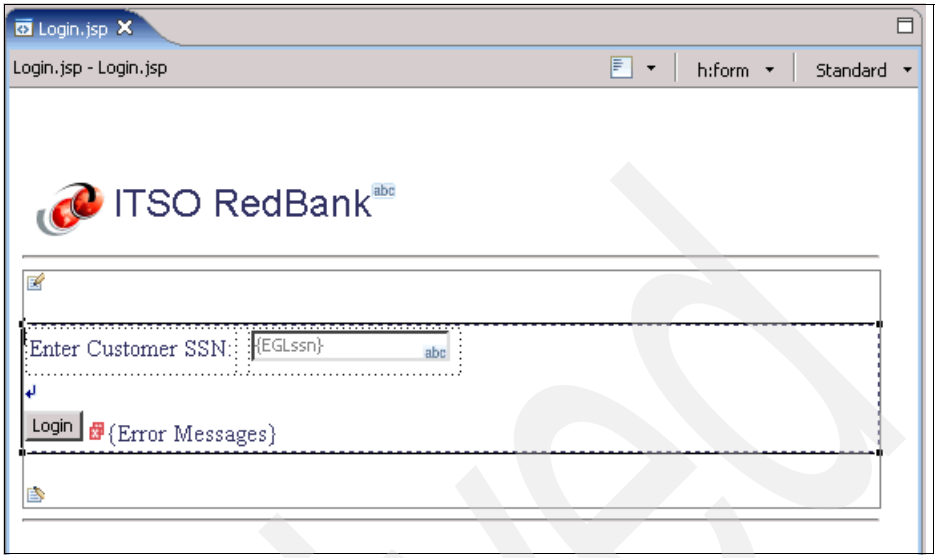


Figure 14-26 Login.jsp page

13. To add an EGL action to the login button:
    a. Select the **Login** button on the page.
    b. Open the Quick Edit view.
    c. Insert the code from Example 14-7 into the quick edit view.

> **Note:** The Login source code found in Example 14-7 can be found in the c:\6449\egl\login_handler.txt.

Example 14-7 Login button code

```
// define SQL status area
sqlStatusData StatusRecord;

// lookup customer
libraries.CustomerLibrary.readCustomer(customer, sqlStatusData);

if (sqlStatusData.sqlStatus == 0)
   // SQL code 0 = No error. Record found.
   // Add the customer's SSN to the session variable customerSSN
   // and forward to the accounts page.
   J2EELib.setSessionAttr("customerSSN", customer.SSN);
   forward to "ListAccounts";
else
   if (sqlStatusData.sqlStatus == 100)
```

Archived

```
      // SQL code 100 = No row found. Set error code to display.
      setError("Customer does not exist");
   else
      // Unknown database error
      setError("Customer retrieval failed - database error");
   end
end
```

Before we can test the login screen we must first create a session scope variable, a navigation rule, and a new jsp file to be forwarded to.

14. Add a new Session Scope variable called `customerSSN`.
    a. In the Page Data view, right-click and select **New** → **ScriptingVariable** → **Session Scope Variable**.
    b. When the Add Session Scope Variable dialog appears, enter `customerSSN` in the Variable name field, enter `java.lang.String` in the Type field, and then click **OK**.

Add content to the ListAccounts.jsp

This section describes the steps required to complete the ListAccounts.jsp.

Add customer data

To add the content to the JSP file, do the following:

1. Open the Web perspective, Project Explorer view.
2. Expand **Dynamic Web Projects** → **BankEGL** → **WebContent**
3. Double-click **ListAccounts.jsp** to open in the editor.
4. Click the **Design** tab.
5. Select the text **Default content of bodyarea**, and press Delete.
6. In the Palette, expand **EGL**.
7. Click **Record** and drag it to the Page Data view.
8. When the Select a Record Part dialog appears, do the following (as seen in Figure 14-22 on page 807), and then click **OK**:
   – Select **Customer (data/CustomerRecord.egl)**.
   – Enter the name of the field: `customer`(default)
   – Check **Add controls to display the EGL element on the Web page**.
9. When the Configure Data Controls dialog appears, do the following:
   a. Select **Updating a existing record**.
   b. Uncheck **indexInArray**.
   c. Change the labels as seen in Figure 14-27 on page 812.

d. In the Control Type drop-down for SSN, select **Output field**.

e. Click **Options**.

   i. Uncheck **Delete**.

   ii. Change the label to `Update`.

   iii. Click **OK**.

f. Click **Finish**.

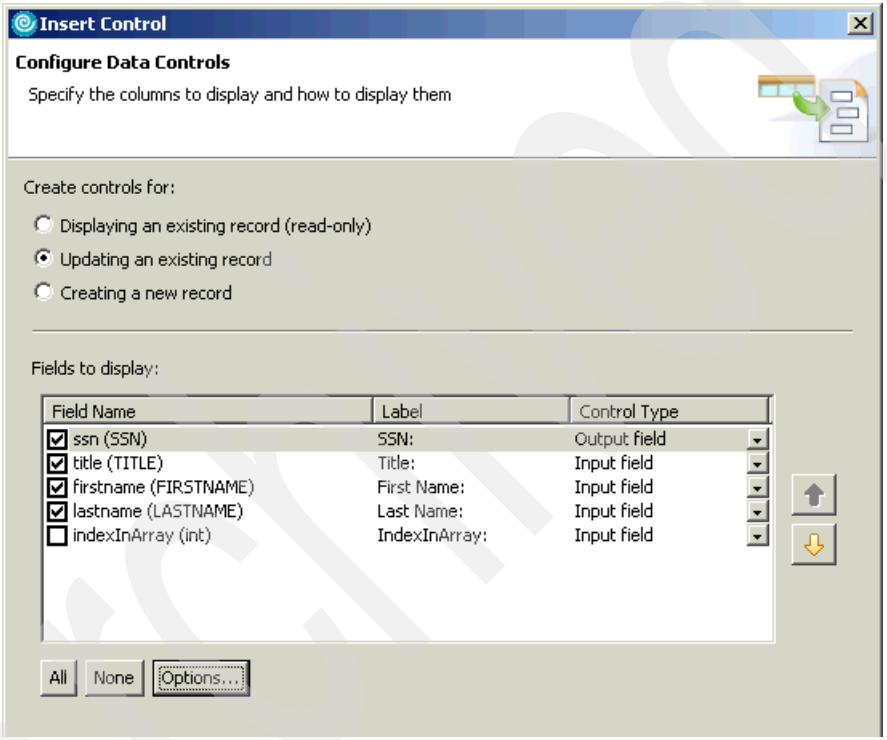


Figure 14-27 Insert customer record

10. Define action code for the Update button.

a. Select the **Update** button.

b. Enter the code found in Example 14-8 into the Quick Edit view.

Example 14-8 Add code for Update button to the page

```
// define SQL status area
sqlStatusData StatusRecord;

CustomerLibrary.updateCustomer(customer, sqlStatusData);
```

```
if (sqlStatusData.sqlStatus != 0)
   setError("Error updating customer");
end
```

Add the Logout button and action code

To add the Logout button and action code, do the following:

1. In the Design tab of the ListAccounts.jsp, expand **Faces Components**.
2. Click **Command - Button** and drag it to the area below the Account data list.
3. In the Properties, enter **Logout** in the Button label field.
4. Define action code for the Logout button.
   a. Switch to the Quick Edit view.
   b. Select the **Logout** button.
   c. Enter the code found in Example 14-9 into the Quick Edit view.

Example 14-9 Add code for Logout button to the page

```
j2eelib.clearSessionAttr(“customerSSN”);
forward to label “Logout”;
```

Add account data

To add the account data to be displayed for a customer on the ListAccounts.egl, do the following:

1. Open the Web perspective, Project Explorer view.
2. Expand **Dynamic Web Projects** → **BankEGL** → **EGLSource** → **pagehandlers**.
3. Double-click **ListAccounts.egl** to open in the editor.
4. Click the **Design** tab.
5. Enter the contents of Example 14-10 into the ListAccounts.egl file.

> **Note:** The contents of Example 14-10 can be found in the c:\6449code\egl\ListAccounts.egl.

Example 14-10 Add contents to ListAccounts.egl

```
package pagehandlers;

import data.*;
import libraries.*;
```

Archived

```
PageHandler ListAccounts {view="ListAccounts.jsp",
onPageLoadFunction=onPageLoad}

   customer Customer;
   customerAccounts Customer_Accounts[];

   Function onPageLoad()
      // define SQL status area
      sqlStatusData StatusRecord;

      J2EELib.getSessionAttr("customerSSN", customer.ssn);
      try
         get customer;
         sqlStatusData.sqlStatus = 0;
      onException
         sqlStatusData.sqlStatus = sysVar.sqlcode;
      end

      if (sqlStatusData.sqlStatus == 0)
         Customer_AccountsLibrary.getCustomerAccountsByCustomer(customer,
               customerAccounts, sqlStatusData);
         case (sqlStatusData.sqlStatus)
            when (0)
            ;
            when (100)
            ;
            otherwise
               setError("Error retrieving accouts for customer");
         end
      else
         setError("Error retrieving customer");
      end
   End
End
```

6. Save the ListAccounts.egl file.
7. Expand **Dynamic Web Projects** → **BankEGL** → **WebContent**.
8. Double-click **ListAccounts.jsp** to open in the editor.
9. Click the **Design** tab.
10. In the Page Data view, expand **ListAccounts** and drag **customerAccounts - Customer_Accounts[]** to the right of the Error Messages tag.

Archived

Note: We found that in some cases, the tooling allowed dropping the CustomerAccounts - Customer_Accounts on the page but did not add the code after clicking Finish. Ensure that you drop the record in the proper location of the content area.

11. When the Configure Data Controls dialog appears, do the following:
    a. Uncheck **CUSTOMERS_SSN** and **ID**.
    b. Change the labels as seen in Figure 14-28.

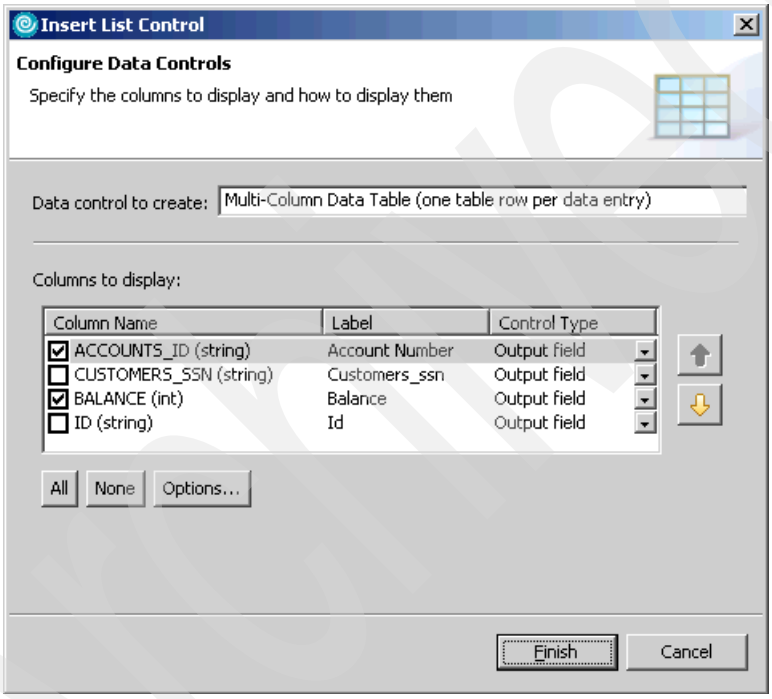


Figure 14-28 Configure Data Controls

    c. Click **Finish**.

12. Test the sample Web application.

    Refer to 14.5.4, “Run the sample EGL Web application” on page 818.

14.5 Import and run the sample Web application

This section describes how to import the completed Web application built using EGL, and run it within the WebSphere Application Server V6.0 Test Environment.

14.5.1 Import the EGL Web application sample

The imported sample project interchange file will use the project name BankEGL.

To import the completed sample EGL Web application, do the following:

1. Open the EGL perspective Project Explorer view.
2. Select **File** → **Import**.
3. When the Import Wizard wizard appears, select **Project Interchange** and click **Next**.
4. In the Import Projects screen, browse to the c:\6449code\egl folder and select **BankEGL.zip**. Click **Open**.
5. Check the **BankEGL** and **BankEGLEAR** projects, and click **Finish**.

Rational Application Developer will import the BankEGL and BankEGLEAR projects. Since the projects were packaged without the EGL runtime libraries to reduce the file of the sample code zip file, you will have a number of errors in your workspace after the import. These will be fixed when you generate in 14.5.3, “Generate Java from EGL source” on page 817, since the runtime libraries are automatically added back to the appropriate folder during the Generate task.

14.5.2 Prerequisites

If you have worked your way through this chapter, it is likely that you have completed the following prerequisite steps.

Ensure the you have completed the following prerequisite steps:

1. Ensure that the IBM Rational Application Developer V6.0 EGL component is installed, and that you have installed Interim Fix 0004.

   For details refer to “Install the EGL component of Rational Application Developer” on page 768.
2. Ensure that the EGL development capability is enabled under the Workbench preferences.

   For details refer to “Enable the EGL development capability” on page 771.
3. Ensure that IBM DB2 Universal Database V8.2 is installed, as this is a required for the EGL Web application sample.

Archived

For details refer to “Install DB2 Universal Database” on page 773.

4. Ensure that the BANK sample database and tables have been created, the sample data has been loaded, and a connection has been created in Rational Application Developer.

   For details refer to “Set up the sample database” on page 777.

5. Ensure that the EGL preferences for the SQL database connection defining the database and datasource have been configured.

   For details refer to “Configure EGL preferences for SQL database connection” on page 779.

   Note: This step is not necessary if you simply want to run the sample application in the test environment. If you wish to further customize the sample you will need this option configured.

6. Ensure that the JDBC provider and datasource have been configured in the application deployment descriptor (enhanced EAR feature). This information will be used during deployment to create the datasource on the target test server.

   For details refer to “Configure the data source” on page 780.

7. Ensure that the DB2 Universal Database JDBC class path environment variable variables have been defined for the WebSphere Application Server V6.0 test server via the WebSphere Administrative Console.

   For details refer to “Configure the DB2 JDBC class path environment variables” on page 783.

14.5.3 Generate Java from EGL source

The BankEGL.zip Project Interchange file shipped with the redbook sample code was modified manually to remove some EGL runtime libraries. This was done to reduce the size of the BankEGL.zip for distribution purposes. For more information regarding this issue refer to 14.6, “Considerations for exporting an EGL project” on page 820.

We will run the Generate task on the EGL source to generate the Java used for deployment. This also has the effect of copying the EGL runtime libraries back to the \WebContent\WEB-INF\lib folder.

1. From the EGL perspective, expand **Dynamic Web Projects**.
2. Right-click **BankEGL**, and select **Generate With Wizard**.
3. When the Generate wizard appears, click **Select All**, and click **Next**.

4. Select **Use one build descriptor for all parts**, select **BankEGLWebBuildOptions <BankEGL/EGLSource/BankEGL.eglbld>** from the drop-down list, and then click **Finish**.

The Generate task should resolve all the errors for the BankEGL Web Project. Some warnings may still exist.

14.5.4 Run the sample EGL Web application

To run the Web application sample built using EGL, do the following:

1. Open the Web perspective Package Explorer view.
2. Expand **Dynamic Web Projects** → **BankEGL** → **WebContent**.
3. Right-click **Login.jsp**, and select **Run** → **Run on Server** from the context menu.
4. When the Server Selection dialog appears, select **Choose and existing server**, select **WebSphere Application Server v6.0**, and click **Finish**.

   This operation will start the server and publish the application to the server.
5. When the Login page appears, enter 111-11-1111 in the Customer SSN field (as seen in Figure 14-29 on page 819), and then click the **Login** button.

Note: At the time of writing, we found that we had to enter the Customer SSN input again after clicking Login, due to a problem with JSF input validation. This is only necessary after the first time the application is run.

Archived

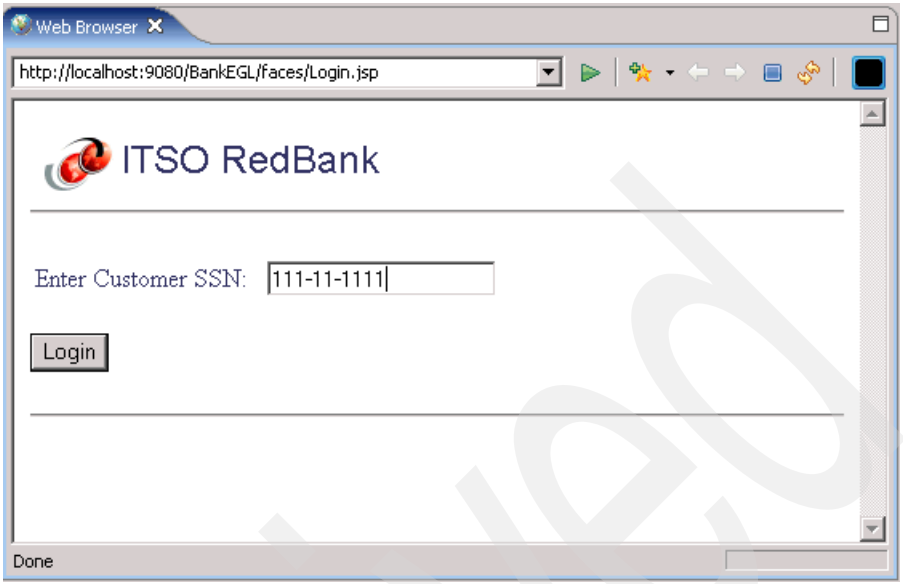


Figure 14-29 EGL Web application sample: Login page

6. The Accounts page should be displayed (as seen in Figure 14-30 on page 820) with the customer account information retrieved from the BANK database.

   From the page you can do the following:

   – You can modify the values in the Title, Firstname, and Lastname fields and then click **Update**.

   – You can log out by clicking **Logout**.

Archived

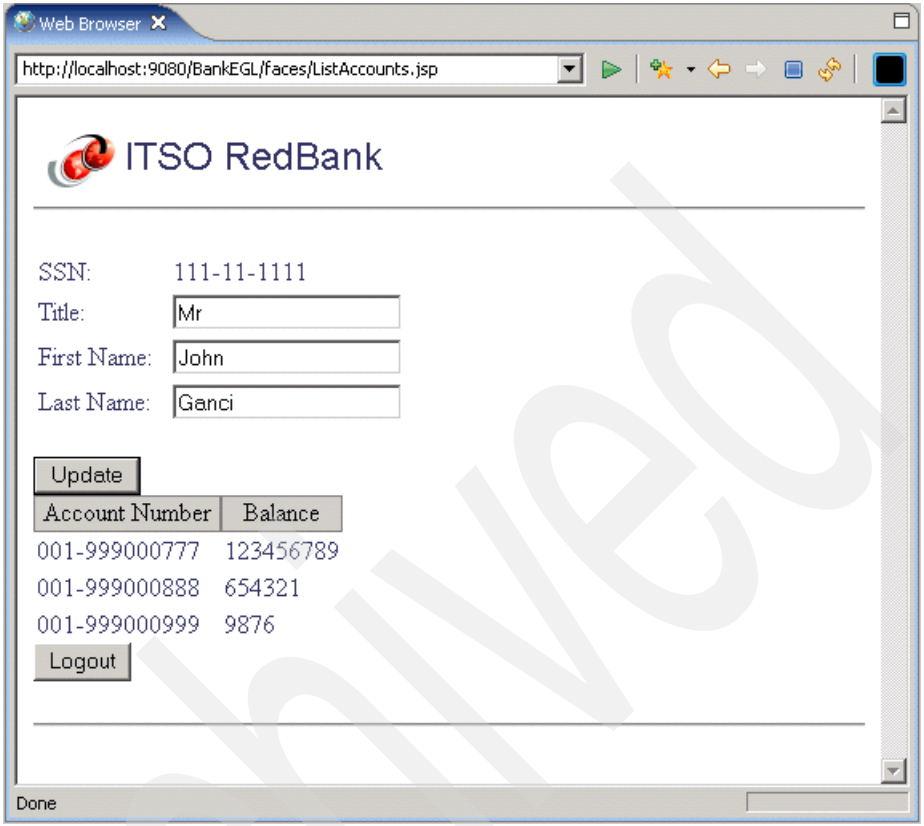


Figure 14-30 EGL Web application sample - Customer accounts

14.6 Considerations for exporting an EGL project

Table 14-2 lists the EGL runtime libraries added to the \WebContent\WEB-INF\lib folder of an EGL Web Project.

Table 14-2 EGL and supporting runtime libraries

EGL and supporting runtime libraries	Time of addition to EGL Web Project
jsf-ibm.jar	When project is created
sdo_access_beans.jar	When project is created
sdo_web.jar	When project is created
fda6.jar	When project is created and when Generate is executed

Archived

EGL and supporting runtime libraries	Time of addition to EGL Web Project
fdaj6.jar	When project is created and when Generate is executed
eglintdebug.jar	When 1st Faces JSP added and when Generate is executed
eglintdebugsupport.jar	When 1st Faces JSP added and when Generate is executed

At the time of writing, we discovered several issues when exporting an EGL Web Project to a Project Interchange file, WAR and EAR:

- Project Interchange and WAR file size: We found that a simple EGL Web Project when exported was fairly large. For example, the eglintdebugsupport.jar is 14 MB, when using IBM Rational Application Developer V6.0 and Interim Fix 0004, and is packaged in the Project Interchange file.

  Refer to 14.6.1, “Reduce the file size of the Project Interchange file” on page 821, for a work-around procedure to address this issue for Project Interchange files.

- Migration of runtime libraries: We found that the EGL V6.0 Migration tooling used to migrate EGL V6.0 to V6.0 with Interim Fix 0001 or 0004, did not migrate the EGL runtime libraries; thus, the migrated EGL source is out of synch with the EGL runtime libraries.

  Refer to 14.6.2, “Manually adding the runtime libraries after migration” on page 822, for a work-around procedure to address the issue of migrating the runtime libraries.

- Export WAR/EAR with source: We found that when checking Export source during the export of a WAR or EAR file containing an EGL Web Project, the EGL source was not exported. Only the generated Java source was exported. We were informed this is working as designed.

  Refer to “Export WAR/EAR with source” on page 823 for a work-around procedure to address the issue of including the EGL source in a WAR/EAR.

14.6.1 Reduce the file size of the Project Interchange file

Project Interchange files are a very useful means of packaging the contents of projects. Since the size of the EGL runtime files is likely to be much larger than the size of the EGL and generated Java source, it is desirable to remove some of the runtime files before exporting the EGL Web Project to a Project Interchange file.

We found that eglintdebug.jar, eglintdebugsupport.jar, fda6.jar, and fdaj6.jar were added back into the \WebContent\WEB-INF\lib folder when the EGL Generate task was run. Due to this behavior, we found that we could remove the noted files before exporting to a Project Interchange file, and thus greatly reduce the size of the file. For example, when using this technique our sample BankEGL.zip Project Interchange file was reduced from 15 MB to 800 KB (and that is a good thing).

14.6.2 Manually adding the runtime libraries after migration

When originally developing the EGL Web application sample, we used IBM Rational Application Developer V6.0, which includes EGL V6.0. We later upgraded to Interim Fix 0004. Interim Fix 0004 includes a significant number of changes to the EGL language syntax and requires that projects created in EGL V6.0 be migrated to EGL V6.0 with Interim Fix 0004.

We successfully migrated our sample source code by right-clicking the EGL Web Project and selecting **EGL V6.0 Migration → Migrate**.

After the migration, we discovered that the EGL runtime libraries listed in Table 14-2 on page 820 had not been migrated.

To manually update the EGL runtime libraries, do the following:

1. Expand **Dynamic Web Projects → BankEGL → WebContent → WEB-INF → lib**.

   Where *BankEGL* is the EGL Web Project that has been migrated and has the old level of the EGL runtime libraries.

2. Back up the files found in the lib directory. This step is only included for precautionary purposes.
3. Delete all files listed in Table 14-2 on page 820 from the lib directory.
4. Create a new EGL Web Project called EGLv601.

   The newly created EGL Web Project will contain the proper level of the EGL runtime files. We will use this project as a source of the desired EGL runtime files.

5. Expand **Dynamic Web Projects → EGLv601 → WebContent → WEB-INF → lib**.
6. Copy all files found in the lib folder EGLv601 project lib directory to the migrated project lib directory.
7. Generate the BankEGL project, which will automatically add the remaining EGL runtime libraries.

Archived

14.6.3 Export WAR/EAR with source

Although WAR/EAR files can be used to package source code, they are primarily intended as deployment packages. We recommend that developers use Project Interchange files as a method of exchanging project resources, such as source code. If you do decide to use WAR/EARs and would like to include EGL source in the EAR/WAR files, you will need to understand and complete the following work-around.

Note: When using this procedure, the EGL source files will always be included in the WAR file, regardless of the setting of the Export source files check box in the WAR/EAR export wizard. Once configured as described in "Adding EGL source to WAR" on page 823, you can temporarily disable this work-around by following the procedure outlined in "Disable export of EGL source" on page 824.

Adding EGL source to WAR

To enable the inclusion of EGL source code to exported WAR files, do the following:

1. Right-click the **BankEGL** project, and select **Properties**.

   Where *BankEGL* is the EGL Web Project you wish to configure.
2. Select **Java Build Path**.
3. Select the **Source** tab.
4. Check **Allow output folders for source folders**.
5. Click **Add Folder**.
6. Check **EGLSource** and click **OK**.
7. Expand the **BankEGL/EGLSource**, select **Output folder**, and click **Edit**.
8. When the Source Folder Output Location dialog appears, select **Specific output folder**, enter `WebContent/WEB-INF/EGLSource` (as seen in Figure 14-31 on page 824), and then click **OK**.

Archive

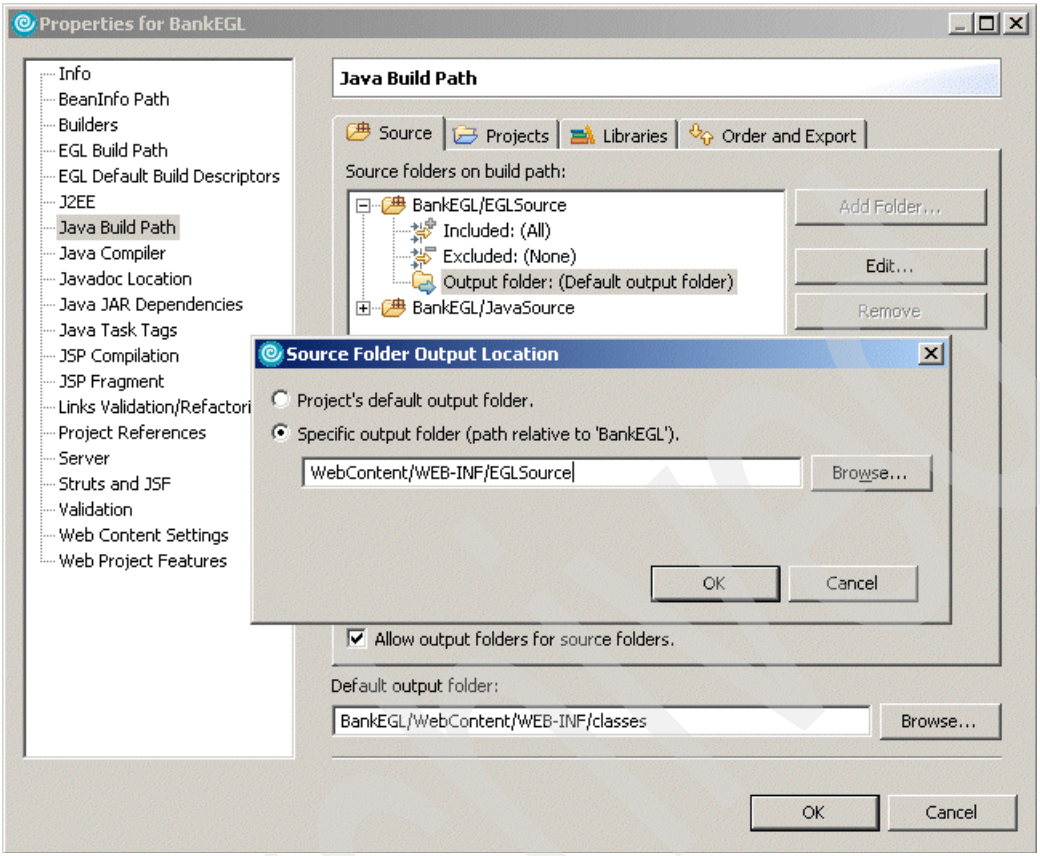


Figure 14-31 Source Folder Output Location

9. Click **OK**.

Now when you perform the Export of an EAR or WAR, the EGL source files will be found in the folder WEB-INF/EGLSource of the WAR file.

Disable export of EGL source

To exclude EGL source code from exported WAR files, do the following:

Note: When the work-around is disabled using the procedure mentioned here, the exported WAR files will still contain a folder named WEB-INF/EGLSource. However, this folder will be empty.

1. Right-click the ***BankEGL*** project, and select **Properties**.

   Where *BankEGL* is the EGL Web Project you wish to configure.

2. Select **Java Build Path**.
3. Select the **Source** tab.
4. Expand the **BankEGL/EGLSource**, select **Excluded**, and click **Edit**.
5. When the Inclusion and Exclusion Patterns window appears, click **Add** in the Exclusion patterns section.
6. When the Add Exclusion Pattern window appears, enter **/* and click **OK**.
7. Click **OK** to close the Inclusion and Exclusion Patterns window.
8. The Properties window should now look similar to Figure 14-32. Click **OK**.

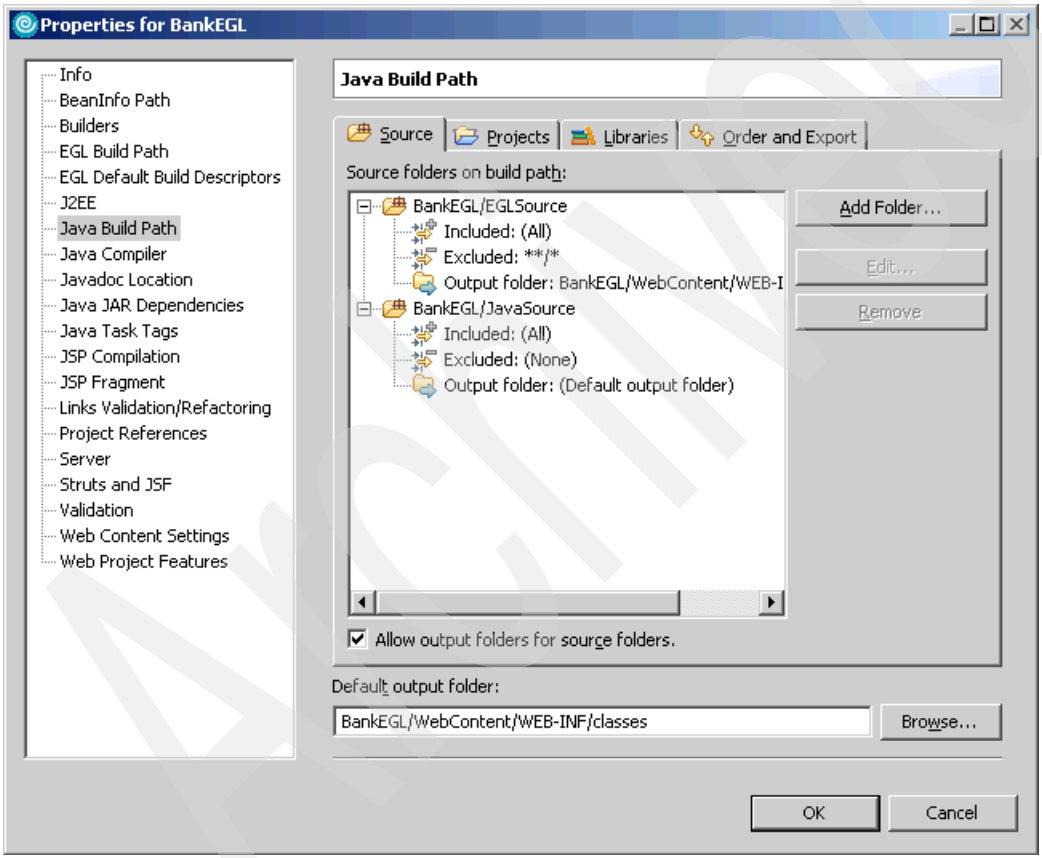


Figure 14-32 Disabled exporting of EGL source code to WAR files

Archived

15

Develop Web applications using EJBs

This chapter introduces Enterprise JavaBeans (EJB) and demonstrates by example how to create, maintain, and test such components.

We will describe how to develop entity beans, relationships between the entity beans, a session bean, and integrate the EJBs with a front-end Web application for the ITSO Bank sample. We will include examples for creating the EJBs using the Visual UML tooling as well as the Deployment Descriptor Editor.

The chapter is organized into the following topics:

- Introduction to Enterprise JavaBeans
- RedBank sample application overview
- Prepare for the sample
- Develop an EJB application
- Testing EJB with the Universal Test Client
- Adapting the Web application

© Copyright IBM Corp. 2005. All rights reserved.

15.1 Introduction to Enterprise JavaBeans

Enterprise JavaBeans (EJBs) is an architecture for server-centric, component-based, distributed object-oriented business applications written in the Java programming language.

Note: This chapter provides a condensed description of the EJB architecture and several coding examples. For more complete coverage on EJBs refer to the *EJB 2.0 Development with WebSphere Studio Application Developer*, SG24-6819. Although IBM Rational Application Developer V6 includes support for EJB 2.1, much of the information is still relevant.

15.1.1 What is new

The following features, supported by IBM Rational Application Developer V6.0, are new to the EJB 2.1 specification and require a J2EE 1.4 compatible application server, such as WebSphere Application Server V6.0:

- Stateless session beans can now implement a Web service endpoint.
- Enterprise beans of any type may utilize external Web services.
- The container-managed Timer service.
- Message-driven beans support more messaging types in addition to JMS.
- The EJB Query Language has been enhanced to include support for aggregate functions, ordering of results, and additional scalar functions. Also, the rules for allowing null values to be returned by finder and select methods have been clarified.

15.1.2 Enterprise JavaBeans overview

Since its introduction in December 1999, the technology has gained momentum among platform providers and enterprise development teams. This is because the EJB component model simplifies the development of business components that are:

- Secure: Certain types of applications have security restrictions that have previously made them difficult and time consuming to implement in Java. For example, an insurance application may have to restrict access to patient data in order to meet regulatory guidelines. Until the advent of enterprise beans, there was no inherent way to restrict access to an object or method by a particular user. Previously, restricting access at the database level and then catching errors thrown at the JDBC level, or restricting access at the application level by custom security code, would have been the only implementation options.

Enterprise JavaBeans allow the declaration of method-level security rules for any bean. Users and user groups can be granted or denied execution rights to any bean or method. In WebSphere, these same user groups can be granted or denied access to Web resources (servlets, JSPs, and HTML pages), and the user IDs can be in a seamless way passed from the Web resources to the EJBs by the underlying security framework. Not only that, but the authenticated credentials may also be forwarded to other systems, possibly legacy systems (compatible LTPA clients).

- Distributed: Enterprise JavaBeans automatically provide distribution capabilities to your application, allowing for the building of enterprise-scale systems. In short, this means that your system's modules can be deployed to many different physical machines and many separate OS processes to achieve your performance, scalability, and availability requirements. Better yet, you may start small with just a single process and grow to as many different machines as you want without ever having to touch your code.
- Persistent: Making an object persistent means preserving its persistent state (the values of its non-transient variables) even after the termination of the system that created that object.

  In most cases, the state of a persistent object is stored in a relational database. Unfortunately, the OO and relational paradigms differ a lot from each other. Relational models are less expressive than OO models because they provide no way to represent behavior, encapsulation, or complex relationships like inheritance. Additionally, SQL data types do not exactly match Java data types, leading to conversion problems. All these problems may be automatically solved when using EJBs.
- Transactional: Transactions give us four fundamental guarantees including atomicity, consistency, isolation, and durability (ACID):
  - *Atomicity* means that delimited sets of operations have to be executed as a single unit of work. If any single operation fails, the whole set must fail as well.
  - *Consistency* guarantees that no matter what the transaction outcome is, the system is going to be left in a consistent state.
  - *Isolation* means that even though you may have many transactions being performed at the same time, your system will be under the impression that these transactions occur one after the other.
  - *Durability* means that the effects of transactions are to be persistent. Once committed, they cannot be rolled-back.

  Enterprise beans support multiple concurrent transactions with commit and rollback capabilities across multiple data sources in a full two-phase commit-capable environment for distributed transactions.

Archive

- Scalable: Over the past several years customers have found that fat-client systems simply do not scale up, as Web-based systems do, to the thousands or millions of users that they may have. At the same time, software distribution problems have led to a desire to “trim down” fat clients. The 24-hour, 7-day-a-week nature of the Web has also made uptime a crucial issue for businesses. However, not everyone needs a system designed for 24x7 operation or that is able to handle millions of concurrent users. We should be able to design a system so that scalability can be achieved without sacrificing ease of development, or standardization.

  So, what customers need is a way to write business logic that can scale up to meet these kinds of requirements. WebSphere’s EJB support can provide this kind of highly scalable, highly available system. It does so by utilizing the following features:

  - Object caching and pooling: WebSphere Application Server automatically pools enterprise beans at the server level, reducing the amount of time spent in object creation and garbage collection. This results in more processing cycles being available to do real work.
  - Workload optimization: WebSphere Application Server Network Deployment provides advanced EJB workload optimization features. Servers can be grouped in clusters and then managed together using a single administration facility. Weights can be assigned to each server to account for their individual capabilities. When you install an application on a cluster, the application is automatically installed on all cluster members, providing for weighted load balancing. In addition, you can configure and run multiple application servers on one machine, taking advantage of multiprocessor architectures.
  - Automatic fail-over support: With several servers available in a cluster to handle requests, it is less likely that occasional hardware and software failures will produce throughput and reliability issues. In a clustered environment, tasks are assigned to servers that have the capacity to perform the task operations. If one server is unavailable to perform the task, it is assigned to another cluster member. No changes to the code are necessary to take advantage of these features.

- Portable: A strategic issue for businesses nowadays is achieving platform and vendor independence. The EJB architecture, which is an industry standard, can help achieve this goal. EJBs developed for the J2EE platform can be deployed to any compliant application servers. This promise has been demonstrated at the June 1999 JavaOne conference, where the same car dealer application was deployed on multiple application servers, from multiple vendors. While in the short-term it is often easier and faster to take advantage of features that may precede standardization, standardization provides the best long-term advantage.

The EJB architecture depicted in Figure 15-1 reduces the complexity of developing business components by providing automatic (non-programmatic) support for such system level services, thus allowing developers to concentrate on the development of business logic. Such focus can bring a competitive advantage to a business.

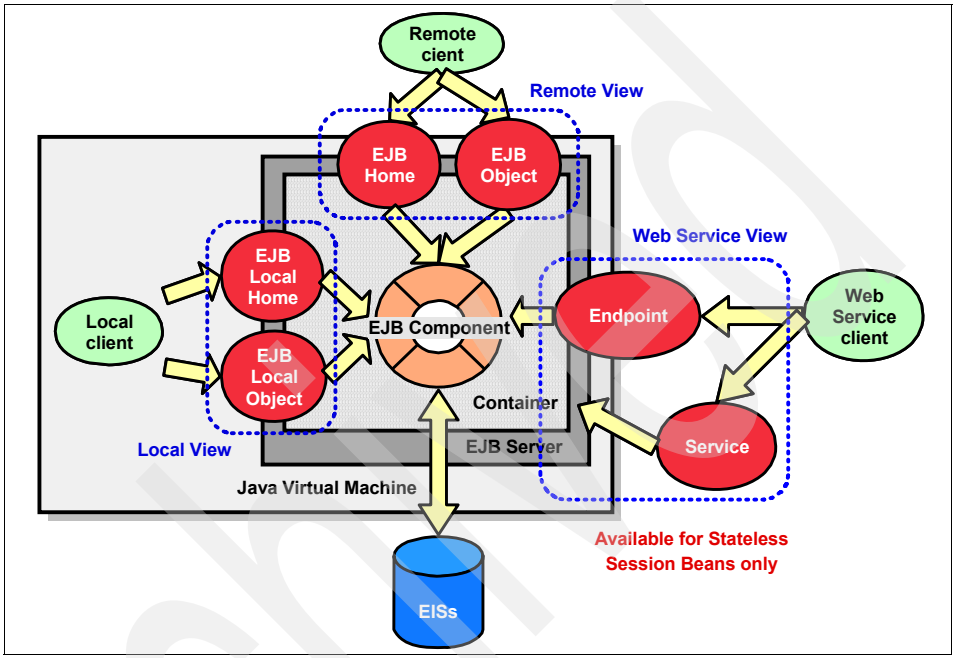


Figure 15-1 EJB architecture overview

In the following sections we briefly explain each of the EJB architecture elements depicted in:

- EJB server
- EJB container
- EJB components

15.1.3 EJB server

An EJB server is the part of an application server that hosts EJB containers. It is sometimes referred to as an Enterprise Java Server (EJS).

The EJB server provides the implementation for the common services available to all EJBs. The EJB server's responsibility is to hide the complexities of these services from the component requiring them. The EJB specification outlines eight services that must be provided by an EJB server:

- Naming
- Transaction
- Security
- Persistence
- Concurrency
- Life cycle
- Messaging
- Timer

Bear in mind that the EJB container and the EJB server are not very clearly separated constructs from the component point of view. EJBs do not interact directly with the EJB server (there is no standard API for that), but rather do so through the EJB container. So, from the EJBs' perspective, it appears as if the EJB container is providing those services, when in fact it might not. The specification defines a bean-container contract, but not a container-server contract, so determining who actually does what is somewhat ambiguous and platform dependent.

15.1.4 EJB container

The EJB container functions as a runtime environment for enterprise beans by managing and applying the primary services that are needed for bean management at runtime. In addition to being an intermediary to the services provided by the EJB server, the EJB container will also provide for EJB instance life cycle management and EJB instance identification. EJB containers create bean instances, manage pools of instances, and destroy them.

Containers are transparent to the client in that there is no client API to manipulate the container, and there is no way for a client to tell in which container an enterprise bean is deployed.

One of the container's primary responsibilities is to provide the means for remote clients to access components that live within them. Remote accessibility enables remote invocation of a native component by converting it into a network component. EJB containers use the Java RMI interfaces to specify remote accessibility to clients of the EJBs.

The responsibilities that an EJB container must satisfy can be defined in terms of the primary services. Specific EJB container responsibilities are as follows:

- Naming

- Transaction
- Security
- Persistence
- Concurrency
- Life cycle
- Messaging
- Timer

Note the similarity to the list in 15.1.3, “EJB server” on page 831. This is due to the unspecified division of responsibilities between the EJB server and container.

Naming

The container is responsible for registering the unique lookup name in the JNDI namespace when the server starts up, and binding the appropriate object type into the JNDI namespace.

Transaction

The EJB container may handle the demarcation of transactions automatically, depending on the EJB type and the transaction type attribute, both described in the EJB module’s deployment descriptor. When the container demarcates the transactions, applications can be written without explicit transaction demarcation code (for example, begin, commit, rollback).

Security

The container provides security realms for enterprise beans. It is responsible for enforcing the security policies defined at the deployment time whenever there is a method call, through access control lists (ACL). An ACL is a list of users, the groups they belong to, and their rights, and it ensures that users access only those resources and perform those tasks for which they have been given permission.

Persistence

The container is also responsible for managing the persistence of a certain type of bean (discussed later in this chapter) by synchronizing the state of the bean’s instance in memory with the respective record in the data source.

Concurrency

The container is responsible for managing the concurrent access to components, according to the rules of each bean type.

Archived

Life cycle

The container controls the life cycle of the deployed components. As EJB clients start sending requests to the container, the container dynamically instantiates, destroys, and reuses the beans as appropriate. The specific life cycle management that the container performs is dependent upon the type of bean. The container may ultimately provide for some resource utilization optimizations, and employ techniques for bean instance pooling.

Messaging

The container must provide for the reliable routing of asynchronous messages from messaging clients (JMS or otherwise) to message-driven beans (MDBs). These messages can follow either the peer-to-peer (queue-based) or publish/subscribe (topic-based) communication patterns.

Timer

Enterprise applications may model business processes that are dependent on temporal events. To implement this characteristic, the container must provide a reliable and transactional EJB Timer Service that allows callbacks to be scheduled for time-based events. Timer notifications may be scheduled to occur at a specific time, after a specific elapsed duration, or at specific recurring intervals. Note that this service is not intended for the modeling of real-time events.

15.1.5 EJB components

EJB components run inside an EJB container, their runtime environment. The container offers life-cycle services to these components, and provides them with an interface to the EJB server. It also manages the connections to the enterprise information systems (EISs), including databases and legacy systems.

Client views

For client objects to send messages to an EJB component, the component must provide a view. A view is a client interface to the bean, and may be local or remote:

- A local view can be used only by local clients (clients that reside in the same JVM as the server component) to access the EJB.
- A remote view allows any client (possibly distributed) to access the component.

The motivation for local interfaces is that remote calls are more expensive than local calls. Which one to use is influenced by how the bean itself is to be used by its clients, because local and remote depict the clients' view of the bean. An EJB

client may be a remote client, such as a servlet running on another process, or may be a local client, such as another EJB in the same container.

Note: Even though a component may expose both a local and a remote view at the same time, this is typically not the case. EJBs that play the role of facades usually offer only a remote interface. The rest of the components generally expose only a local interface.

In remote invocation, method arguments and return values are passed by value. This means that the complete objects, including their non-transient reference graphs, have to be serialized and sent over the network to the remote party, which reconstructs them as new objects. Both the object serialization and network overhead can be a costly proposition, ultimately reducing the response time of the request.

On the other hand, remote interfaces have the advantage of being location independent. The same method can be called by a client that is inside or outside of the container.

Additionally, Web Service clients may access stateless session beans through the Web Service client view. The view is described by the WSDL document for the Web Service the bean implements, and corresponds to the bean's Web Service endpoint interface.

Which interfaces to use, classes to extend, and other rules of bean construction are governed by the type of bean you choose to develop. A quick introduction to the types of enterprise beans and their uses is presented here.

Archive

EJB types

There are three main types of EJBs: Entity beans, session beans, and message-driven beans (see Figure 15-2).

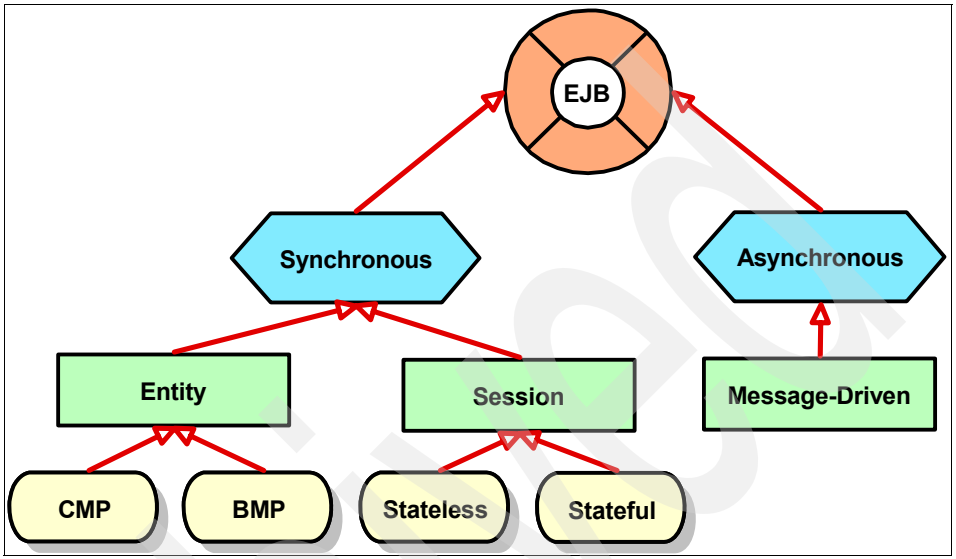


Figure 15-2 EJB types

- Entity beans: Entity beans are modeled to represent business or domain specific concepts, and are typically the nouns of your system, such as *customer* and *account*. Entity beans are persistent; that is, they maintain their internal state (attribute values) between invocations and across server restarts. Due to their persistent nature, entity beans usually represent data (entities) stored in a database.

  While the container determines *when* an entity bean is stored in persistent storage, *how* the bean is stored can either be controlled by the container, through *container-managed persistence* (CMP), or by the bean itself, through *bean-managed persistence* (BMP). Container-managed persistence is typically obtained by defining the mapping between the fields of the entity bean and columns in a relational database to the EJB server.

- Session beans: A session bean is modeled to represent a task or workflow of a system, and to provide coordination of those activities. It is commonly used to implement the facade of EJB modules. Although some session beans may maintain state data, this data is not persistent, it is just conversational.

  Session beans can either be *stateless* or *stateful*. Stateless session beans are beans that maintain no conversational state, and are pooled by the container to be reused. Stateful session beans are beans that keep track of the

conversational state with a specific client. Thus, they cannot be shared among clients.

- Message-driven beans (MDB): Like session beans, message-driven beans may also be modeled to represent tasks. However, they are invoked by the receipt of asynchronous messages, instead of synchronous ones. The bean either listens for or subscribes to messages that it is to receive.

Note: Although the bean flavors (CMP versus BMP, stateful versus stateless) are often referred to as EJB types, this is not the case. EJBs with different persistent management or stateness are not different types in the sense that there are no new classes or interfaces to represent these types. They are still just entity or session beans. Rather, how the container manages these beans is what makes them different. All information regarding the way the container has to handle these different bean flavors is managed in the deployment descriptor.

Entity and session beans are accessed synchronously through a remote or local EJB interface method invocation. This is referred to as synchronous invocation, because when a client makes an invocation request, it will be blocked, waiting for the return. Clients of EJBs invoke methods on session and entity beans. An EJB client may be remote, such as a servlet, or local, such as another EJB within the same JVM.

Message-driven beans are not accessible through remote or a local interfaces. The only way for an EJB client to communicate with a message-driven bean is by sending a JMS message. This is an example of asynchronous communication. The client does not invoke the method on the bean directly, but rather, uses JMS constructs to send a message. The container delegates the message to a suitable message-driven bean instance to handle the invocation. EJBs of any type can also be accessed asynchronously by means of a timer event, fired by the EJB Timer Service.

Interfaces and classes

An EJB component consists of the following primary elements, depending on the type of bean:

- EJB bean class: Contains the bean's business logic implementation. Bean classes must implement one of the enterprise bean interfaces, depending on the bean type: javax.ejb.SessionBean, javax.ejb.EntityBean, javax.ejb.MessageDrivenBean. Beans willing to be notified of timer events must implement the javax.ejb.TimedObject, so that the container may call the bean back when a timer expires. Message-driven beans must also implement the javax.jms.MessageListener interface, to allow the container to register the

bean as a JMS message listener and to call it back when a new message arrives.

- EJB component interface: Declares which of the bean's business methods should be exposed to the bean's public interface. Clients will use this interface to access such methods. Clients may not access methods that are not declared in this interface. A bean may implement a local component interface, a remote component interface, or both, depending on the kinds of clients it expects to serve. The local component interface is also known as an EJB local object, because of the javax.ejb.EJBLocalObject interface that it extends. The remote component interface, in turn, is also known as EJB object, due to the javax.ejb.EJBObject interface that it extends.
- EJB home interface: Declares which bean's life-cycle methods (to create, find, and remove beans instances) are available to clients, functioning very much like a factory. Local beans have local home interfaces that extend javax.ejb.EJBLocalHome. Remote beans have remote home interfaces that extend javax.ejb.EJBHome.
- Primary key class: Entity beans must also have a primary key class. Instances of this class uniquely identify an instance of the entity type in the database. Even though not formally enforced, primary key classes must also correctly implement the equals and hashCode methods. As you will see, Rational Application Developer takes care of that for you. The primary key class may be an existing class, such as java.lang.Integer or java.lang.String, or a new class created specifically for this purpose.

Relationships

Relations are a key component of object-oriented software development. Non-trivial object models can form complex networks with these relationships.

The container automatically manages the state of CMP entity beans. This management includes synchronizing the state of the bean with the underlying database when necessary and also managing any container-managed relationships (CMRs) with other entity beans. The bean developer is relieved of the burden that is writing database-specific code and, instead, can focus on business logic.

Multiplicity is an important characteristic of relations. Relations can have the following multiplicities:

- One-to-one: In a one-to-one (1:1) relationship, a CMP entity bean is associated with a single instance of another CMP entity bean. If you come up with a one-to-one composition or aggregation relationship, remember to check if you are not, in fact, modeling the same concept as two different entities.

- One-to-many: In a one-to-many (1:m) relationship, a CMP entity bean is associated with multiple instances of another CMP entity bean. For example, **an `Account` bean could be associated with multiple instances of a `Transaction` bea**n, kept as a log of transactions.
- Many-to-many: In a many-to-many (m:m) relationship, multiple instances of a CMP entity bean **are associated with multiple instances of another CMP entity bean. For example, a `Customer` bean may be associated with multiple instances of an `Account` bean, and a single `Account` bean may, in turn, be associated with many `Customer` beans.**

There are also three different types of relationships:

- Association: An association is a loose relationship between two independent objects.
- Aggregation: Aggregation identifies that an object is made up of separate parts. That is, the aggregating object is dependent on the aggregated objects. The lifetime of the aggregated objects is not controlled by the aggregator. If the aggregating object is destroyed, the aggregated objects are not necessarily destroyed.
- Composition: Composition defines a stronger dependency between the objects. Composition is similar to aggregation, but with composition, the lifetime of the objects that make up the whole are controlled by the compositor.

It is the developer's task to implement the differences among the three kinds of relationships. These differences may require considerations of characteristics such as the navigation of the relationship and the encapsulation of the related objects.

Component-level inheritance is still not in the EJB 2.1 specification, even though it is planned for future releases. In want of standardized component-level inheritance, IBM WebSphere Application Server V6.0 and IBM Rational Application Developer V6.0 implements proprietary component-level inheritance.

EJB query language (EJB QL)

The EJB query language is a query specification language, similar to SQL, for entity beans with container-managed persistence. With it, the Bean Provider is able to specify the semantics of custom finder or EJB select methods in a portable way and in terms of the object model's entities, instead of the relational model's entities. This is possible because EJB QL is based on the abstract schema types of the entity beans.

Archived

Note: Both finder and EJB select methods are used to query the backend where the actual data is stored. The difference is that finder methods are accessible to the entity beans' clients, whereas select methods are internal to the implementation and not visible to clients.

An EJB QL query is a string consisting of the following clauses:

- A SELECT clause, which determines the type of the objects or values to be selected.
- A FROM clause, which provides declarations that designate the domain to which the specified expressions apply.
- An optional WHERE clause, which may be used to restrict the results that are returned by the query.
- An optional ORDER BY clause, which may be used to order the results that are returned by the query.
- The result type can be an EJBLocalObject, an EJBObject, a CMP-field value, a collection of any of these types, or the result of an aggregate function.

EJB QL queries are defined by the Bean Provider in the deployment descriptor. The SQL statements for the actual database access is generated automatically by the deployment tooling. As an example, this query retrieves customers that have accounts with a large balance:

```
select object(c) from Customer c, in(c.accounts) a where a.balance > ?1
```

As you can see, this EJB QL statement is independent of the database implementation. It follows a CMR relationship from customer to account and queries the account balance. Finder and EJB select methods specified using EJB QL are portable to any EJB 2.1 environment.

Note: These finder and EJB select methods may also be portable to EJB 2.0 environments if they do not use the new order and aggregate features defined by the EJB 2.1 specification.

15.2 RedBank sample application overview

In this chapter, we reuse the design of the RedBank application, described in Chapter 11, "Develop Web applications using JSPs and servlets" on page 499. The focus of this chapter is on implementing EJBs for the business model, instead of regular JavaBeans. The rest of the application's layers (control and view) still apply exactly as designed.

Archived

Note: While the sample application in this chapter extends the sample application developed in the Web application chapter (Chapter 11, “Develop Web applications using JSPs and servlets” on page 499), the content of this chapter does not strictly depend on it.

If your focus is on developing EJBs and testing them using the Universal Test Client, you can complete the sample in this chapter without knowledge of the sample developed in the Web application chapter.

Also, since the completed sample application from the Web application chapter is included with the additional material for this book, you can choose to import the finished sample application from the Web application chapter in order to get a running application at the end of this chapter.

Figure 15-3 depicts the RedBank application model layer design.

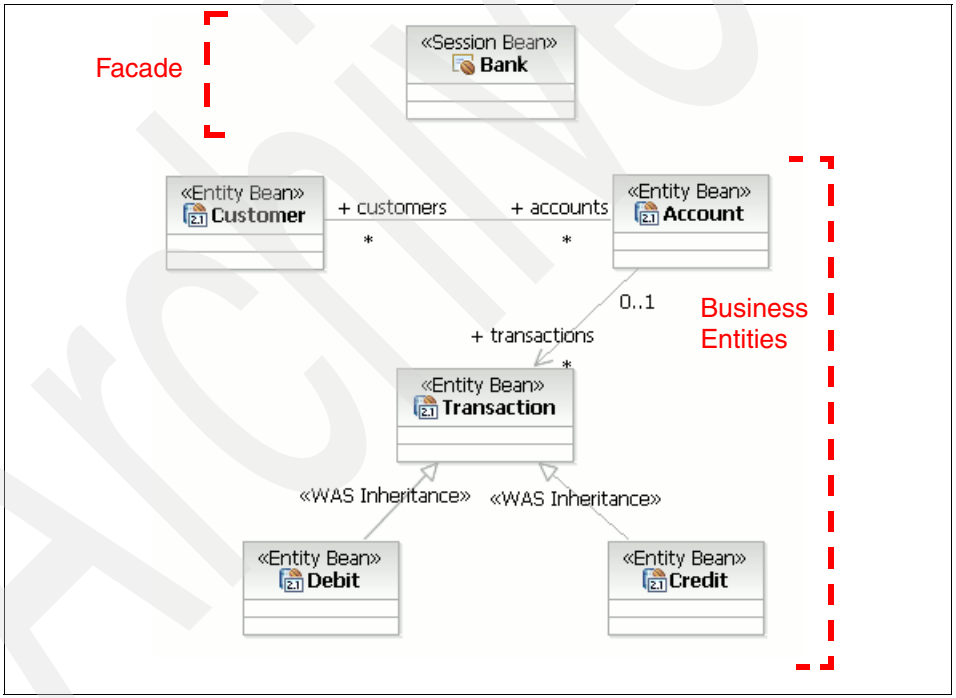


Figure 15-3 EJB module’s class diagram for RedBank application

If you compare the model depicted in Figure 15-3 on page 841 to the Web application model shown in Figure 11-8 on page 514, the Bank session bean will act as a facade for the EJB model. A new coordinator, EJBBank will replace the MemoryBank coordinator shown in Figure 11-8.

Our business entities (Customer, Account, Transaction, Credit, and Debit) are implemented as CMP entity beans with local interfaces, as opposed to regular JavaBeans. By doing so, we automatically gain persistence, security, distribution, and transaction management services. On the other hand, this also implies that the control and view layers will not be able to reference these entities directly, because they may be placed in a different JVM. Only the session bean (Bank) will be able to access the business entities through their local home interfaces.

You may be asking yourself then why we do not expose a remote interface for the entity beans as well? The problem with doing that is two-fold. First, in such a design, clients would probably make many remote calls to the model in order to resolve each client request. This is not a recommended practice because remote calls are much more expensive than local ones. Finally, allowing clients to see into the model breaks the layer's encapsulation, promoting unwanted dependencies and coupling.

As the control layer will not be able to reference the model's objects directly, we will reuse the Customer, Account, Transaction, Credit, and Debit from the Web application from Chapter 11, "Develop Web applications using JSPs and servlets" on page 499, as data transfer object, carrying data to the servlets and JSPs, but allowing no direct access to the underlying model.

Note: The data transfer object, also known as value object or transfer object, is documented in many J2EE architecture books. The objective is to limit inter-layer data sharing to serializable JavaBeans, thus avoiding remote references. The DTOs can be created by the session facade or by a builder object (according to the *Builder* design pattern) on its behalf, in case the building process is too complicated or needs validation steps.

Figure 15-4 shows the application component model and the flow of events.

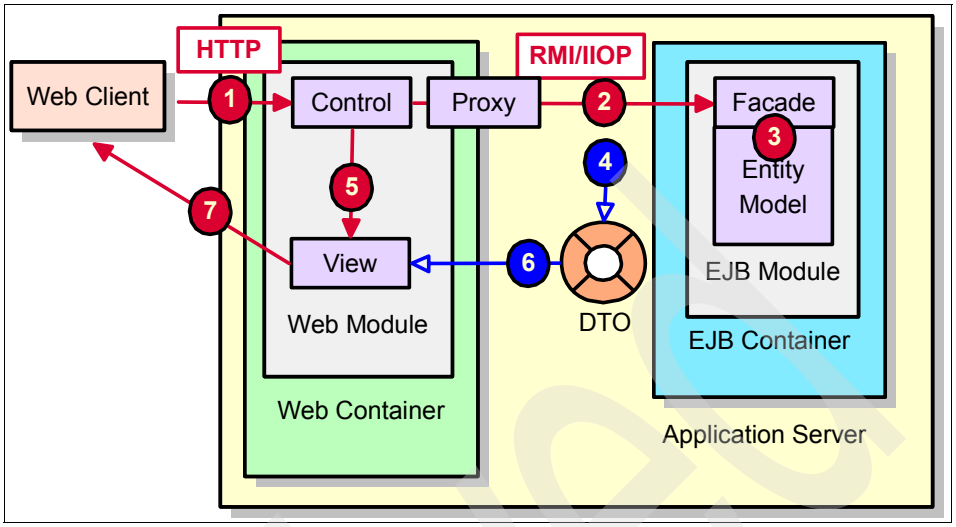


Figure 15-4 Application component model and workflow

The flow of events, as shown in Figure 15-4 on page 843, is:

1. The first event that occurs is the HTTP request issued by the Web client to the server. This request is answered by a servlet in the control layer, also known as the front controller, which extracts the parameters from the request. The servlet sends the request to the appropriate control JavaBean. This bean verifies whether the request is valid in the current user and application states.
2. If so, the control layer sends the request through the JavaBean proxy to the session EJB facade. This involves using JNDI to locate the session bean's home interface and creating a new instance of the bean.
3. The session EJB executes the appropriate business logic related to the request. This includes having to access entity beans in the model layer.
4. The facade creates a new DTO and populates it with the response data. The DTO is returned to the calling controller servlet.
5. The front controller servlet sets the response DTO as a request attribute and forwards the request to the appropriate JSP in the view layer, responsible for rendering the response back to the client.
6. The view JSP accesses the response DTO to build the user response.
7. The result view, possibly in HTML, is returned to the client.

Please note that the intent of this chapter is to introduce you to the Rational Application Developer tools that make the development of EJBs and enterprise applications possible. Together we will work only on a single session bean and three entity beans.

15.3 Prepare for the sample

This section describes the steps to prepare for developing the sample EJB application.

15.3.1 Required software

To complete the EJB development sample in this chapter, you will need the following software installed:

- IBM Rational Application Developer V6.0
- Database software:
  - Cloudscape V5.1 (installed by default with Rational Application Developer)

  Or:

  - IBM DB2 Universal Database V8.2

Note: For more information on installing the software, refer to Appendix A, "IBM product installation and configuration tips" on page 1371.

15.3.2 Create and configure the EJB projects

In Rational Application Developer, you create and maintain Enterprise JavaBeans and associated Java resources in EJB projects. The environment has facilities that help you create all three types of EJBs, define relationships (association and inheritance), and create resources such as access beans, converters, and composers. Within an EJB project, these resources can be treated as a portable, cohesive unit.

Note: Converters and composers are used for non-standard relational mapping. A converter allows you to transform a user-defined Java type to an SQL type back and forth. Composers are used when entity attributes have multi-column relational representations.

An EJB module typically contains components that work together to perform some business logic. This logic may be self-contained, or access external data and functions as needed. It should be comprised of a facade and the business entities. The facade is usually implemented using one or more remote session beans and message-driven beans. The model is commonly implemented with related local entity beans.

In this chapter we develop the entity beans shown in Figure 15-3 on page 841.

15.3.3 Create an EJB project

In order to develop EJBs, you must create an EJB project. When creating an EJB project, you need to create an EJB client project to hold the deployed code. It is also typical to create an Enterprise Application project that will be the container for deploying the EJB project.

To create a J2EE EJB project, do the following:

1. From the Workbench, select **File** → **New** → **Project**.
2. When the New Project dialog appears, select **EJB** → **EJB Project** and click **Next**.

   Tip: From the Project Explorer, it is possible to access the New EJB Project wizard by right-clicking **EJB Projects** and selecting **File** → **New** → **EJB Project**.

3. When the New EJB Project dialog appears, enter `BankEJB` in the Name field (as shown in Figure 15-5 on page 846), and click **Next**.

   If you click the **Show Advanced** button, additional options are displayed. The following lists the selections that should be done for advanced options:

   – EJB version: Select **2.1** (default).

   – Target server: Select **WebSphere Application Server V6.0.**

     Note: The New button allows you to define a new installed server runtime environment.

   – Check **Add module to EAR project** (default).

     You can choose to add the EJB module being created to an EAR project. The default behavior for that end is to create a new EAR project, but you can also select an existing one from the drop-down combo box. If you would like to create a new project and also configure its location, click the **New** button. For our example, we will use the given default value.

   – Check **Create an EJB Client JAR Project to hold client interfaces and classes** (default).

     You can also choose to create an EJB client JAR project, which is optional under the EJB 2.1 specification but considered a best practice. The EJB client jar holds the home and component interfaces of the project's enterprise beans, and other classes that these interfaces depend on, such as their superclasses and implemented interfaces, the classes and interfaces used as method parameters, results, and exceptions.

Archived

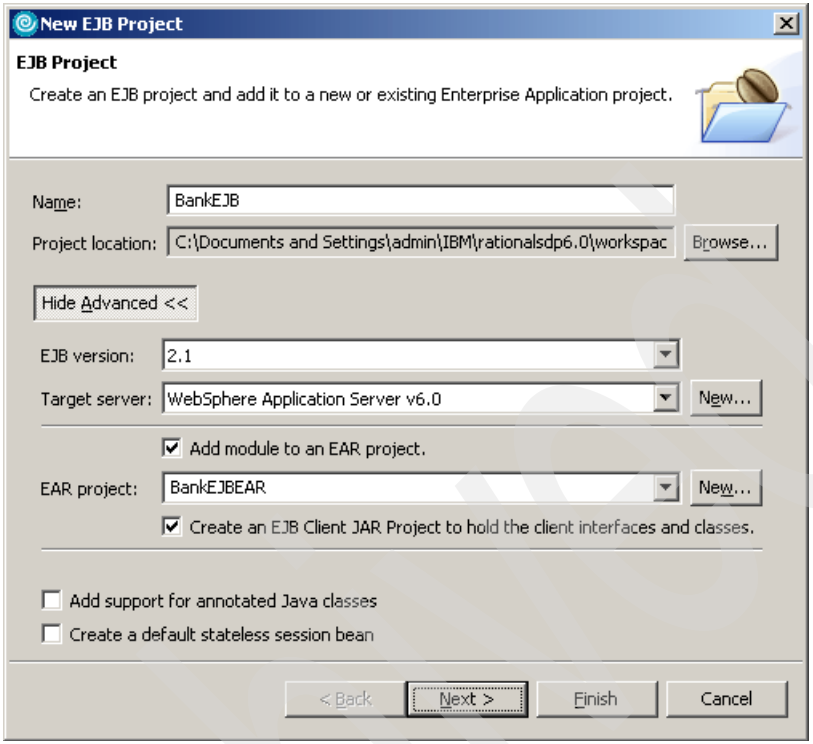


Figure 15-5 Create an EJB project wizard (page 1)

Finally, on the last section of the Figure 15-5 dialog, you can select whether you want to add support for annotated Java classes or create a default stateless session bean (always named DefaultSession).

Note: Annotation-based programming provides an extensible mechanism for generating application artifacts, packaging the application, and readying it for execution. Annotation-based programming offers a set of tags and a processing mechanism that allow you to embed additional metadata in your source code. This additional metadata is used to derive the artifacts required to execute the application in a J2EE environment.

The goal of annotation-based programming is to minimize the number of artifacts that you have to create and maintain, thereby simplifying the development process.

For example, consider a CMP entity EJB. With annotation-based programming, you simply create a simple Java source file containing the bean implementation logic, and a few tags indicating that you want to deploy this class as an EJB. Using this single artifact, Rational Application Developer can create:

- The home and remote interfaces and the key class
- The EJB deployment descriptor
- WebSphere-specific binding data

You can also use annotations to edit the bean's characteristics later, such as which methods should be exposed to the home and remote interfaces, which attributes the bean has, and which of those belong to the primary key.

Unfortunately, this extremely interesting capability is not yet fully integrated to the rest of the environment. If you choose to use it, you will not be able to edit some of the enterprise beans' characteristics through other means provided by the tool, such as the deployment descriptor or the graphical interface. Because of this, we chose not to use it at this time in the book.

Annotations are an integral part of J2SE 1.5 and will probably be integrated to a future J2EE 1.5 specification.

4. When the EJB client JAR Creation window appears, enter the following (as shown in Figure 15-6 on page 848), and then click **Finish**:
   - Client JAR URI: `BankEJBClient.jar`
   - Name: `BankEJBClient`

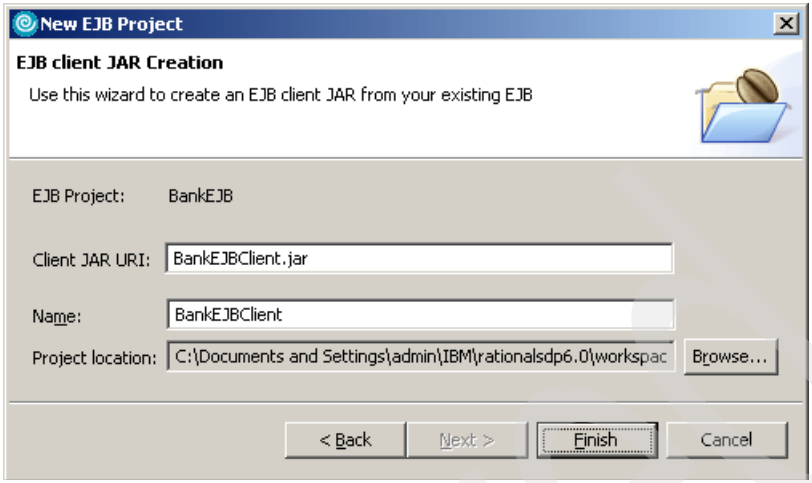


Figure 15-6 Create an EJB project wizard (page 2)

If the current perspective was not the J2EE Perspective when you created the project, Rational Application Developer will prompt if you want to switch to the J2EE Perspective. Click **Yes**.

5. Verify that when complete, the resulting workspace structure and created project components should look like Figure 15-7.

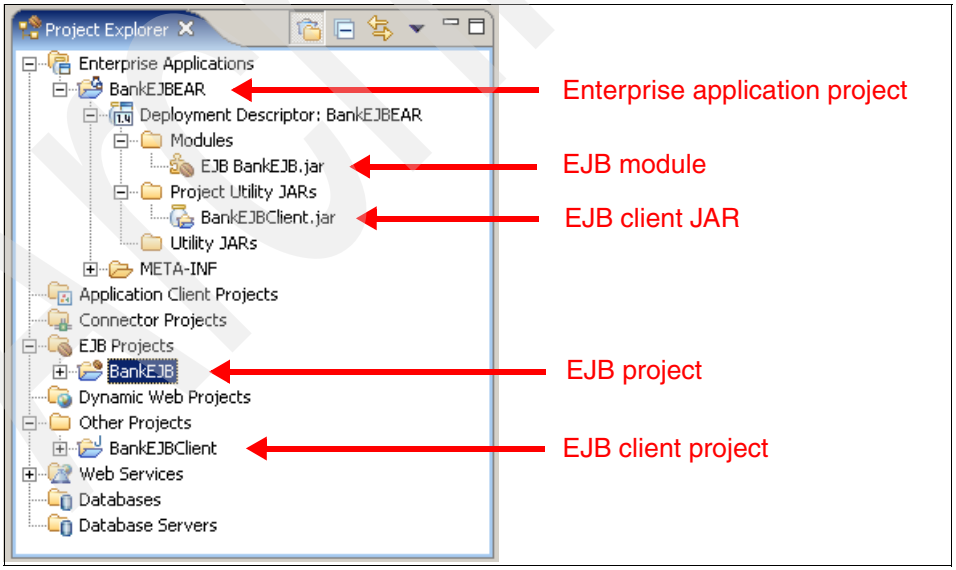


Figure 15-7 Resulting workspace structure after project creation

15.3.4 Configure the EJB projects

Before we can develop the EJB code, we need to prepare the project settings. The following needs to be done:

1. Add a new source folder, ejbDeploy, in the BankEJB project.
2. Set up the ned folder to be used for deployment source code.
3. Configure the BankEJB project to not produce warnings for unused imports.
4. Configure the BankEJBClient project to not produce warnings for unused imports.

Do the following to perform these changes to the workspace:

1. Configure the Java Build Path properties.

   We will now make some changes to the EJB project's structure in order to facilitate development and maintenance:

   a. From the Project Explorer view, right-click **EJB Project** → **BankEJB** and select **Properties**.

   b. When the Properties for BankEJB dialog appears, select **Java Build Path**.

   c. When the Java Build Path panel appears, click the **Source** tab and click **Add Folder**, as seen Figure 15-8.

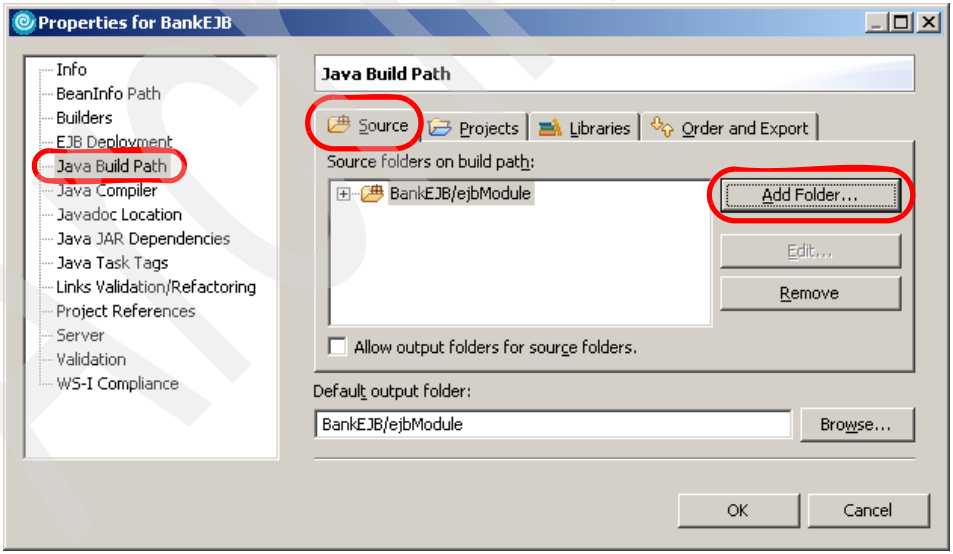


Figure 15-8 EJB project's properties

   d. When the Source Folder Selection dialog appears, click **Create New Folder**.

e. When the New Folder dialog appears, enter `ejbDeploy` as the folder name and then click **OK** (see Figure 15-9).

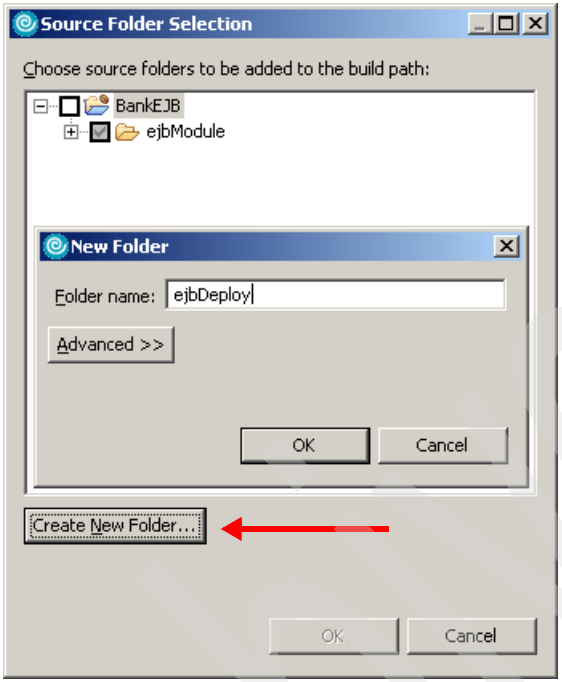


Figure 15-9 Source folder selection dialog

f. Click **OK** in the Source Folder Selection window to add the folder.

g. Click **OK** in the Properties for BankEJB window to apply the changes.

2. Configure the EJB Deployment properties.

   This step describes how to configure the Rational Application Developer folder where the deployment code will be generated and keep it independent of the code we will develop. This will make it easier to later develop and maintain the beans by keeping the ejbModule directory structure clean.

   a. From the Project Explorer view, select **EJB Project** → **BankEJB**, right-click, and select **Properties**.

   b. When the Properties for BankEJB dialog appears, select **EJB Deployment**.

   c. Check the **ejbDeploy** folder (only one folder can be checked) (as seen in Figure 15-10 on page 851), and click **OK**.

   This selects the source folder to store the EJBs deployment code to be the ejbDeploy folder that was created in the previous step.

Archived

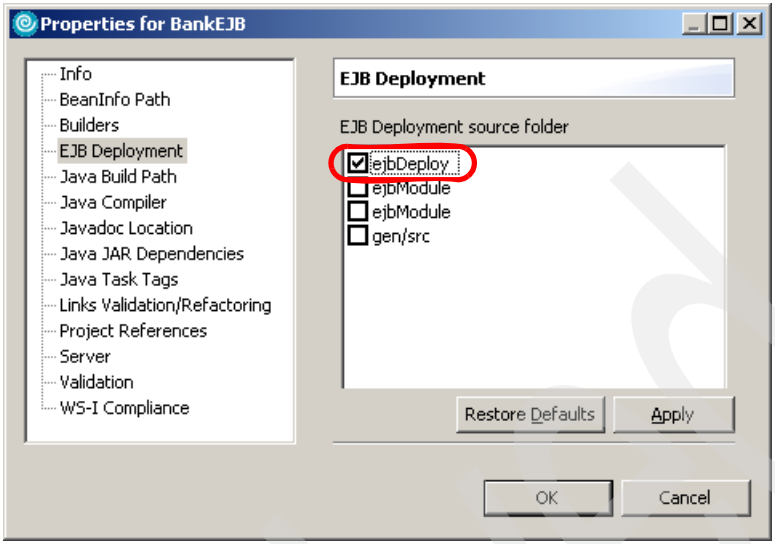


Figure 15-10 EJB Deployment properties

3. Configure the Java Compiler properties.
   a. From the Project Explorer view, select **EJB Project** → **BankEJB**, right-click, and select **Properties**.
   b. When the Properties for BankEJB window appears, select **Java Compiler**.
   c. Select **Use project settings**, as seen in Figure 15-11 on page 852.
   d. Select the **Unused Code** tab and select **Ignore** in the Unused imports drop-down.

> **Note:** Since Eclipse 3.0, the Java compiler's default behavior is to warn about unused imports. This is a very nice feature, but in EJB and EJB client projects this can be an annoyance. The code that Java's RMI compiler (RMIC) generates is full of unused imports, so if we do not disable the verification we will get a lot of warnings in the Problems view.

Archived

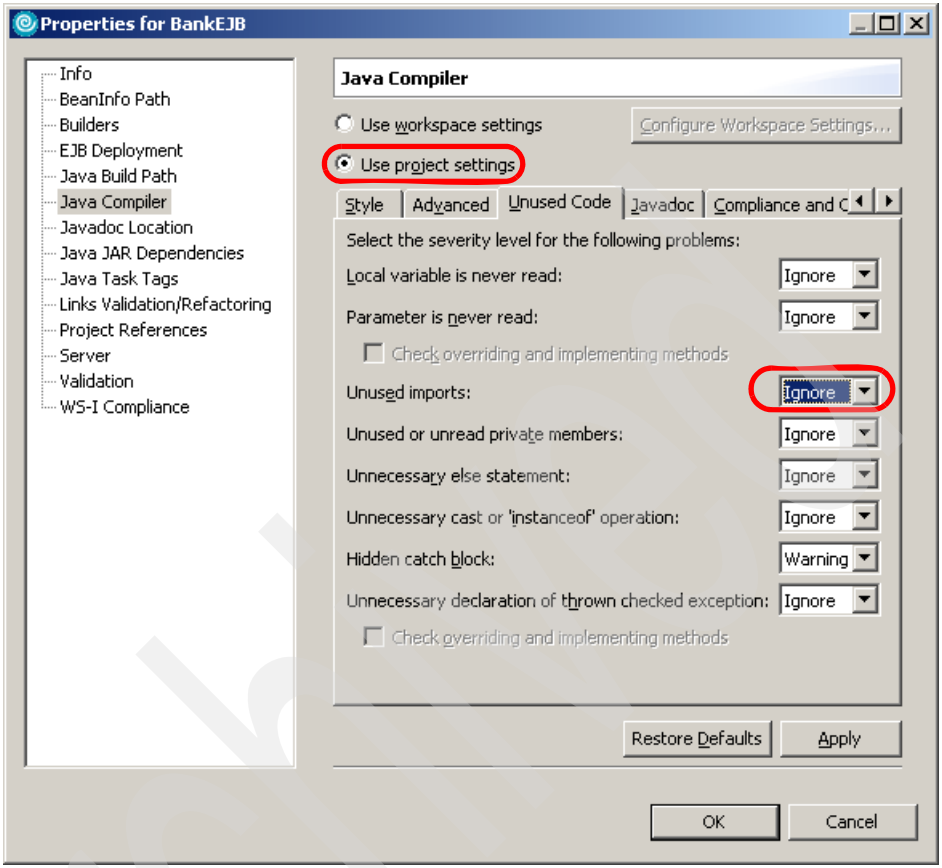


Figure 15-11 Java compiler settings

e. Click **OK** to close the properties dialog.

f. When the Compiler Settings Changed dialog appears, click **Yes**.

The dialog is displayed to let you know that the compiler settings have changed and that a project rebuild is required for the changes to take effect. Clicking **Yes** will perform the rebuild.

4. Repeat the previous step for the BankEJBClient project:

a. From the Project Explorer view, select **Other Projects** → **BankEJBClient**, right-click, and select **Properties**.

b. When the Properties for BankEJB window appears, select **Java Compiler**.

c. Select **Use project settings**.

d. Select the **Unused Code** tab and select **Ignore** in the Unused imports drop-down.

e. Click **OK** to close the properties dialog.

f. When the Compiler Settings Changed dialog appears, click **Yes** to rebuild the project.

15.3.5 Import BankBasicWeb Project

In order to finish the EJB application, we need resources from the Web Project, which was developed in Chapter 11, "Develop Web applications using JSPs and servlets" on page 499.

Do the following to import the BankBasicWeb project from the additional material (refer to Appendix B, "Additional material" on page 1395, for more information about the additional material for this Redbook):

1. From the Workbench, select **File** → **Import**.
2. When the Import dialog appears, select **Project Interchange** and click **Next**.
3. When the Import Projects dialog appears, enter `c:\6449code\web\BankBasicWeb.zip` in the From zip file field and select **BankBasicWeb** in the project list.

   Tip: It can take a while for the project list to populate. To force it to populate, click anywhere on the list box, or click **Back**, followed by **Next** to switch to the previous page, and **Back** again.

4. Click **Finish**.

Now we need to add the new BankBasicWeb project to the BankEJBEAR Enterprise Application. Do the following to add the Web Project to the EAR:

1. In the Project Explorer, expand **Enterprise Applications** → **BankEJBEAR**.
2. Double-click **Deployment Descriptor: BankEJBEAR** to open the Deployment Descriptor editor.
3. Click the **Module** tab.
4. In the Modules section you will see `EJB BankEJB.jar` listed; click **Add**.
5. When the Add Module dialog appears, select **BankBasicWeb** and click **Finish**.
6. Press Ctrl+S to save the deployment descriptor, and close the editor.

We need to reference the exceptions and data objects defined in the Web project from both the EJB project and EJB Client, as well as from the Web project itself. We have to move these classes to the EJB Client project.

1. In the Project Explorer, expand **Dynamic Web Projects** → **BankBasicWeb** → **Java Resources** → **JavaSource**.
2. Select **itso.bank.exception** and **itso.bank.model**, right-click, and select **Refactor** → **Move**.
3. When the Move dialog appears, expand and select **BankEJBClient** → **ejbModule** and click **OK**.
4. When the Confirm overwriting dialog appears, click **Yes To All**.

Several errors will appear in the Problems view. This is due to the fact that the BankBasicWeb project cannot see the classes that we just moved. To fix this, do the following:

1. In the Project Explorer, expand and right-click **Dynamic Web Projects** → **BankBasicWeb** and select **Properties**.
2. When the Properties for BankBasicWeb dialog appears, select **Java JAR Dependencies**.
3. Select **Use EJB client JARs**, check **BankEJBClient.jar**, and click **OK**.

You will notice that the build errors related to the exception and model classes have disappeared.

15.3.6 Set up the sample database

Before we can define the EJB to RDB mapping, we need to create and populate the database, as well as define a database connection within Rational Application Developer that the mapping tools will use to extract schema information from the database.

This section provides instructions for deploying the BANK sample database and populating the database with sample data. For simplicity we will use the built-in Cloudscape database.

To create the database, create the connection, and create and populate the tables for the BANK sample from within Rational Application Developer, do the following:

1. Create the database and connection to the Cloudscape BANK database from within Rational Application Developer.

   For details refer to “Create a database connection” on page 347.
2. Create the BANK database tables from within Rational Application Developer.

For details refer to “Create database tables via Rational Application Developer” on page 350.

3. Populate the BANK database tables with sample data from within Rational Application Developer.

   For details refer to “Populate the tables within Rational Application Developer” on page 352.

4. Import database metadata.

   Now that the database tables have been created, we need to import the metadata (database schema) into the EJB project. This has to be done in order for the mapping tools to be able to map the EJBs to the database.

   a. From from Database Explorer view, right-click the **Bank Connection**, and select **Copy to Project**.

   b. When the Copy to Project dialog appears, click **Browse**, expand and select **BankEJB** → **ejbModule** → **META-INF**, and click **OK**.

   c. Check **Use default schema folder for EJB projects**. The folder path is then automatically entered for the appropriate database type.

   d. Click **Finish** (see Figure 15-12 on page 855).

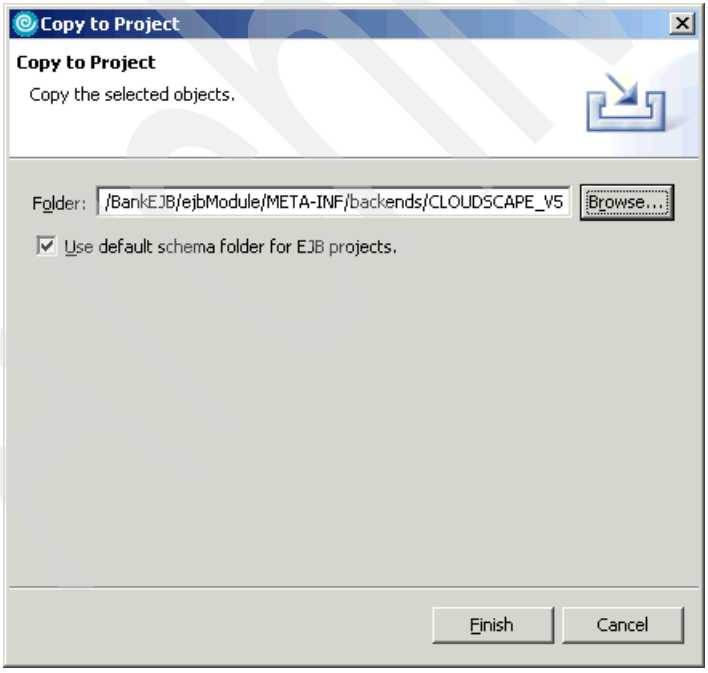


Figure 15-12 Copying schema information to EJB project

e. When the Confirm Folder Create dialog appears, click **Yes**.

15.3.7 Configure the data source

There are a couple of methods that can be used to configure the data source, including using the WebSphere Administrative Console or using the WebSphere Enhanced EAR, which stores the configuration in the deployment descriptor and is deployed with the application.

This section describes how to configure the data source using the WebSphere Enhanced EAR capabilities. The enhanced EAR is configured in the Deployment tab of the EAR deployment descriptor.

The procedure found in this section considers two scenarios for using the enhanced EAR:

- If you choose to import the complete sample code, you will only need to verify that the value of the databaseName property in the deployment descriptor matches the location of your database.
- If you are going to complete the working example Web application found in this chapter, you will need to create the JDBC provider and data source, and update the databaseName property.

Note: For more information on configuring data sources and general deployment issues, refer to Chapter 23, "Deploy enterprise applications" on page 1189.

Access the deployment descriptor

To access the deployment descriptor where the enhanced EAR settings are defined, do the following:

1. Open the J2EE Perspective Project Explorer view.
2. Expand **Enterprise Applications** → **BankEJBEAR**.
3. Double-click **Deployment Descriptor: BankEJBEAR** to open the file in the Deployment Descriptor Editor.
4. Click the **Deployment** tab.

Note: For JAAS authentication, when using Cloudscape, the configuration of the user ID and password for the JAAS authentication is not needed.

When using DB2 Universal Database or other database types that require a user ID and password you will need to configure the JAAS authentication.

Configure a new JDBC provider

To configure a new JDBC provider using the enhanced EAR capability in the deployment descriptor, do the following:

1. From the Deployment tab of the Application Deployment Descriptor, click **Add** under the JDBC provider list.
2. When the Create a JDBC Provider dialog appears, select **Cloudscape** as the Database type, select **Cloudscape JDBC Provider** as the JDBC provider type, and then click **Next**.

   Note: The JDBC provider type list for Cloudscape will contain two entries:

   - Cloudscape JDBC Provider
   - Cloudscape JDBC Provider (XA)

   Since we will not need support for two-phase commits, we choose to use the non-XA JDBC provider for Cloudscape.

3. Enter `Cloudscape JDBC Provider - BankEJB` in the Name field and then click **Finish**.

Configure the data source

To configure a new data source using the enhanced EAR capability in the deployment descriptor, do the following:

1. From the Deployment tab of the Application Deployment Descriptor, select the JDBC provider created in the previous step.
2. Click **Add** next to data source.
3. When the Create a Data Source dialog appears, select **Cloudscape JDBC Provider** under the JDBC provider, select **Version 5.0 data source**, and then click **Next**.
4. When the Create a Data Source dialog appears, enter the following and then click **Finish**:
   - Name: `BankDS`
   - JNDI name: `jdbc/BankDS`
   - Description: `Bank Data Source`

Configure the databaseName property

To configure the databaseName in the new data source using the enhanced EAR capability in the deployment descriptor to define the location of the database for your environment, do the following:

1. Select the data source created in the previous section.

Archive

2. Select the **databaseName** property under the Resource properties.
3. Click **Edit** next to Resource properties to change the value for the databaseName.
4. When the Edit a resource property dialog appears, enter `c:\databases\BANK` in the Value field and then click **OK**.

   Important: The Edit a resource property dialog allows you to edit the entire resource property, including the name. Ensure that you only change the value of the databaseName property, not the name.

   In our example, c:\databases\BANK is the database created for our sample application in 15.3.6, “Set up the sample database” on page 854.
5. Save the Application Deployment Descriptor.
6. Restart the test server for the changes to the deployment descriptor to take effect.

Set up the default CMP data source

Several data sources can be defined for an enterprise application. In order for the EJB container to be able to determine which data source should be used, we must configure the BankEJBEAR project to point to the newly created data source as follows:

1. Open the J2EE Perspective Project Explorer view.
2. Expand **Enterprise Applications** → **BankEJBEAR**.
3. Double-click **Deployment Descriptor: BankEJBEAR** to open the file in the Deployment Descriptor Editor.
4. On the Overview tab, scroll down to the JNDI - CMP Connection Factory Binding section.
5. Enter `jdbc/BankDS` in the JNDI name field.
6. Press **Ctrl+S** followed by **Ctrl+F4** to save and close the deployment descriptor.

15.4 Develop an EJB application

Our first step towards implementing the RedBank’s model with EJBs is creating the following entity beans (as seen in Figure 15-13 on page 859):

- Customer
- Account
- Transaction

Archive

- Debit
- Credit

In this section, we focus on defining and implementing the business logic for the entity beans. In 15.4.5, “Object-relational mapping” on page 892, we define the mapping to the relational database.

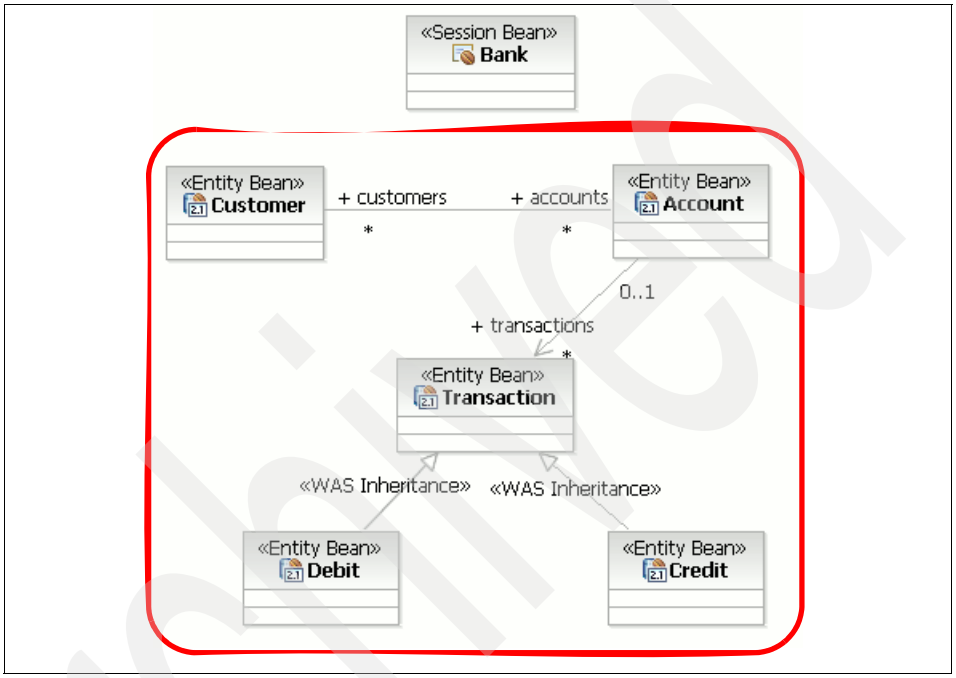


Figure 15-13 Business entities

15.4.1 Create the entity beans

This section describes how to implement the RedBank entity beans in the following sequence:

- Define the Customer bean.
- Define the Account and Transaction beans.
- Define the Credit and Debit derived beans.

Define the Customer bean

To define the Customer bean, do the following:

1. Select and expand **EJB Projects** → **BankEJB** from the Project Explorer view.

Archived

2. Select **File** → **New** → **Enterprise Bean**.

> **Tip:** This can also be done by right-clicking the Deployment Descriptor for the EJB project and selecting **New** → **Enterprise Bean**.

3. When the Create an Enterprise Bean dialog appears, do the following (as seen in Figure 15-14):
   - Select **Entity bean with container-managed persistence (CMP) fields**.
   - Bean name: `Customer`
   - Default package: `itso.bank.model.ejb`
   - Leave the remaining options with the default values and click **Next**.

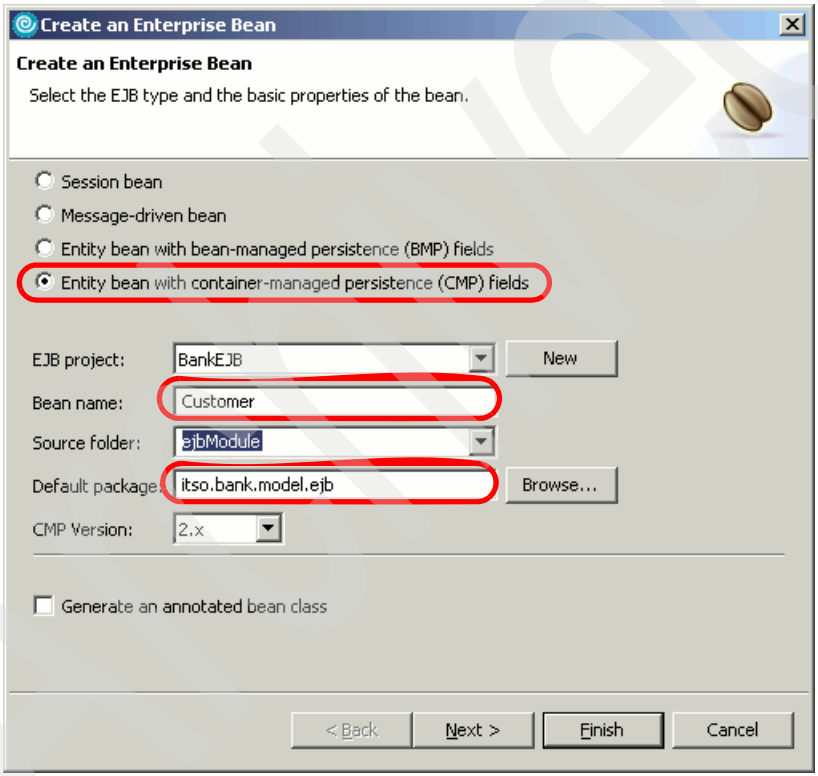


Figure 15-14 Create an enterprise bean (page 1)

4. When the Enterprise Bean Details dialog appears, we entered the information below (as seen in Figure 15-15 on page 862).

   This page lets you select the bean supertype, allowing you to define the inheritance structures. We will do this in “Define the Credit and Debit derived beans” on page 870. For now, we leave the supertype blank.

Additionally, you can define type names, which views you would like to create, and finally the key class and CMP attributes.

– *Do not* check Remote client view.

> **Note:** Most of the time the suggested values for the type names (derived from the bean name) are fine, so you do not have to worry about them. According to the design, entity beans should have only local interfaces, so make sure not to select the Remote client view check box. Rational Application Developer knows about this best practice, so it will only select Local client view by default.

– Check **Local client view** (default).

– CMP attributes: Select **id:java.lang.Integer** and then click **Remove**.

> **Note:** The Customer class will have a key field, but it will be named ssn and be of type java.lang.String.

– Add the CMP attributes by clicking **Add**.

Archived

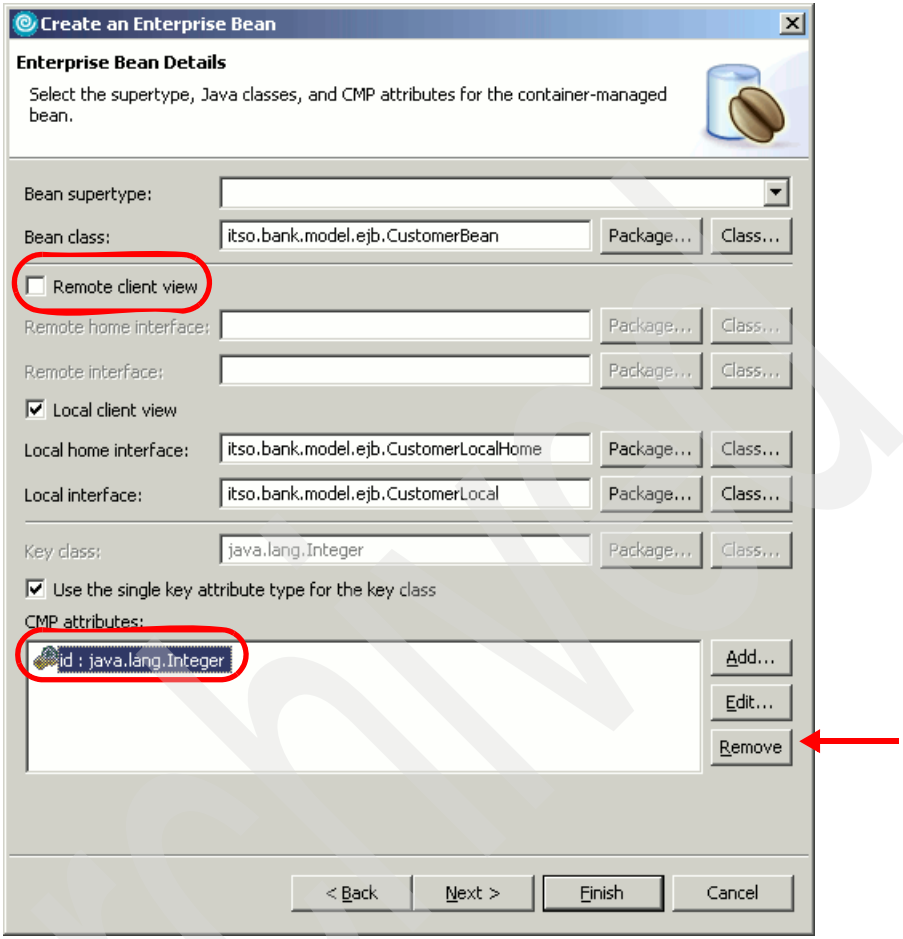


Figure 15-15 Create an entity bean (page 2)

5. When the Create CMP Attribute dialog appears, we entered the following, as seen in Figure 15-16 on page 863. This dialog lets you specify the characteristics of the new CMP attribute you would like to add to the entity bean.
   - Name: `ssn`
   - Type: Select **java.lang.String**.
   - Check **Key field**.
   - Click **Apply**.

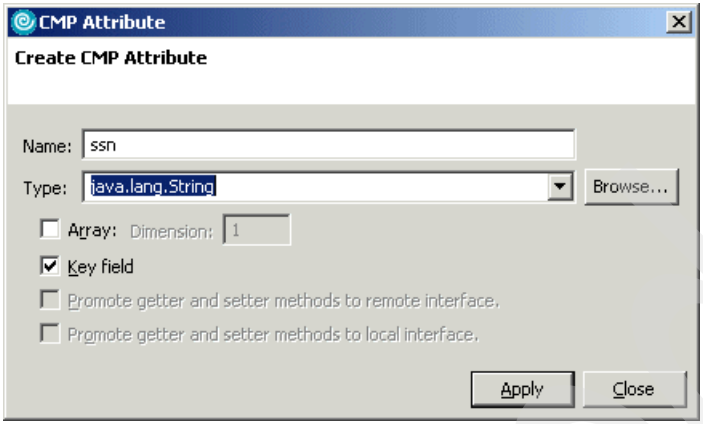


Figure 15-16 Create CMP attributes

Note: Create CMP Attributes.

If you do not define at least one key CMP attribute, you may not create the CMP entity bean.

- Array: If the attribute is an array, select the **Array** check box and specify the number of the dimensions for it.
- Key field: By selecting the **Key field** check box, you indicate that the new field should be part of the entity's unique identifier. You may declare as many attributes as you want to perform this role. Rational Application Developer is very smart here. If you specify just one key attribute of an object type, it will declare that type as the *key class*. If you select an attribute of a non-object type (like int or double), or if you select more than one key attribute, the environment will automatically create a new key class for you, implement all its methods (including equals and hashCode), and declare it as the key class.
- The two last check boxes let you indicate whether you want to promote the new attribute (through its getter and setter) to either the remote or the local interfaces, or to both. The availability of these options depends on which client views you selected, and if the attribute is a key field.

6. For the Customer bean, repeat the process of adding a CMP attribute for the fields listed in Table 15-1. Click **Apply** after adding each attribute. Click **Close** when done.

Tip: You can enter String, instead of java.lang.String, in the Type field. Rational Application Developer will automatically change the type to be java.lang.String.

Table 15-1 Customer bean's CMP attributes

Name	Type	Attribute type check box
ssn	java.lang.String	Key field
title	java.lang.String	Promote getter and setter methods to local interface
firstName	java.lang.String	Promote getter and setter methods to local interface
lastName	java.lang.String	Promote getter and setter methods to local interface

7. After closing the CMP Attribute dialog, the Enterprise Bean Details page should look similar to Figure 15-17 on page 865. Check **Use the single key attribute type for the key class**.
   - If you have one key attribute and it is of an object type, such as String, Integer, Float, BigDecimal, and so forth, then a separate key class is not required.
   - Key wrapper classes are required for the simple data types (int, float, char) or if there is more than one key attribute.

Archive

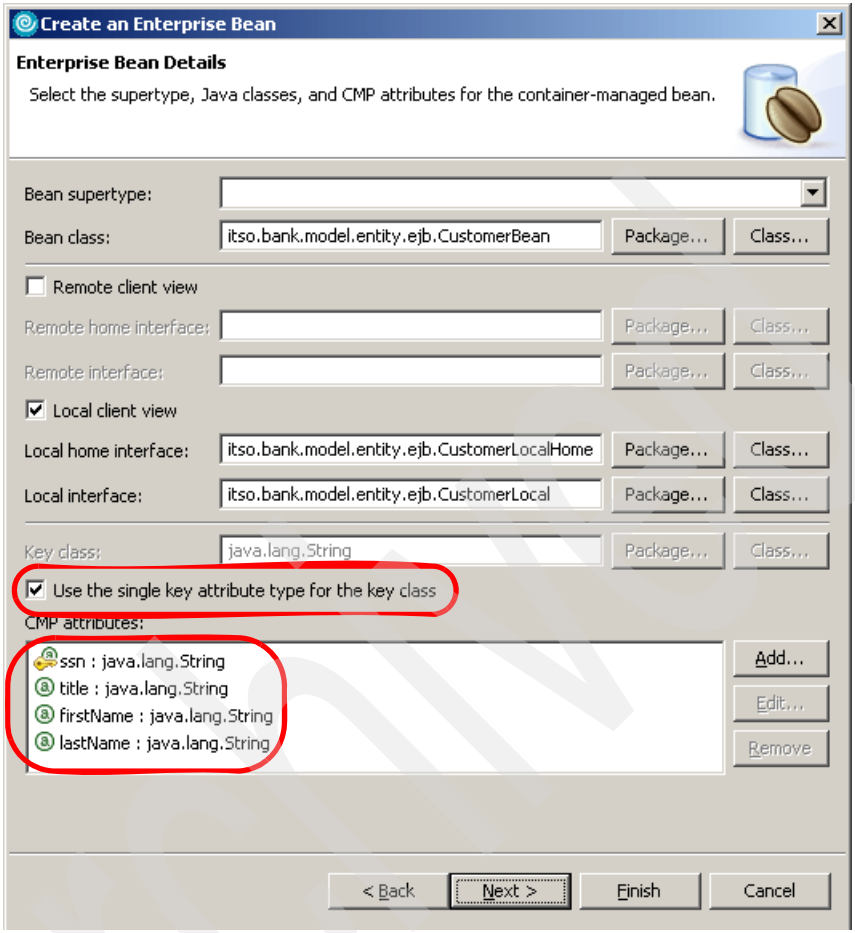


Figure 15-17 Creating an entity bean (page 2) after adding CMP attributes

8. When the EJB Java Class Details page appears, we accepted the defaults and clicked **Next**.

> **Important:** The Bean superclass on the EJB Java Class Detail, shown in Figure 15-18 on page 866, is not related to the Bean supertype field on the previous page, shown in Figure 15-17. The former is used to define Java class inheritance for the implementation classes that make up the EJB. The latter is used to define the EJB inheritance hierarchy. Refer to “Define a bean supertype” on page 871 for use of the Bean supertype and EJB inheritance feature.

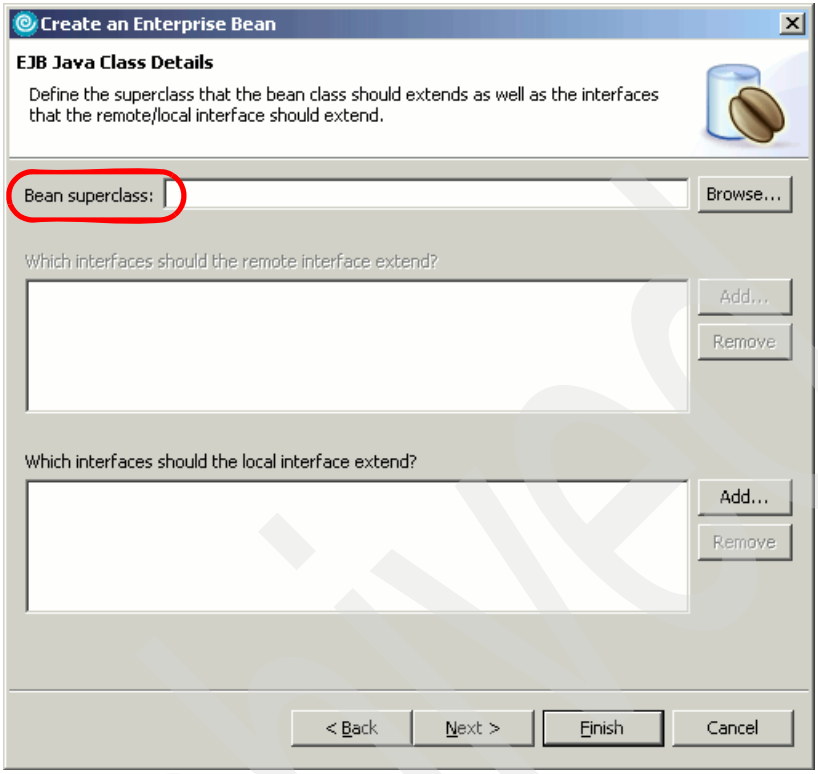


Figure 15-18 EJB Class Details page

9. When the Select Class Diagram for Visualization dialog appears, click **New**.
10. When the New Class Diagram dialog appears, enter `BankEJB/diagrams` in the Enter or select the parent folder field and `ejbs` in the File name field, as seen in Figure 15-19 on page 867, and click **Finish**.

Archive

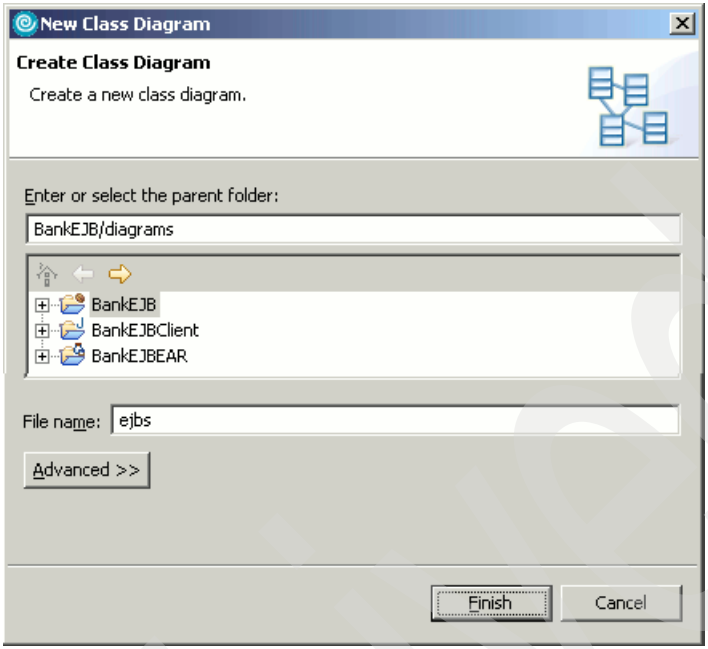


Figure 15-19 Creating a class diagram for EJBs

11. When the dialog closes, expand and select **BankEJB** → **diagrams** → **ejbs.dnx**, as shown in Figure 15-20 on page 868, and click **Finish**.

> **Note:** Although this page allows you to specify a class diagram, we found that it defaulted to use the diagram named default.dnx, located in the root of the BankEJB project. We chose to place all diagrams in a separate folder named diagrams.

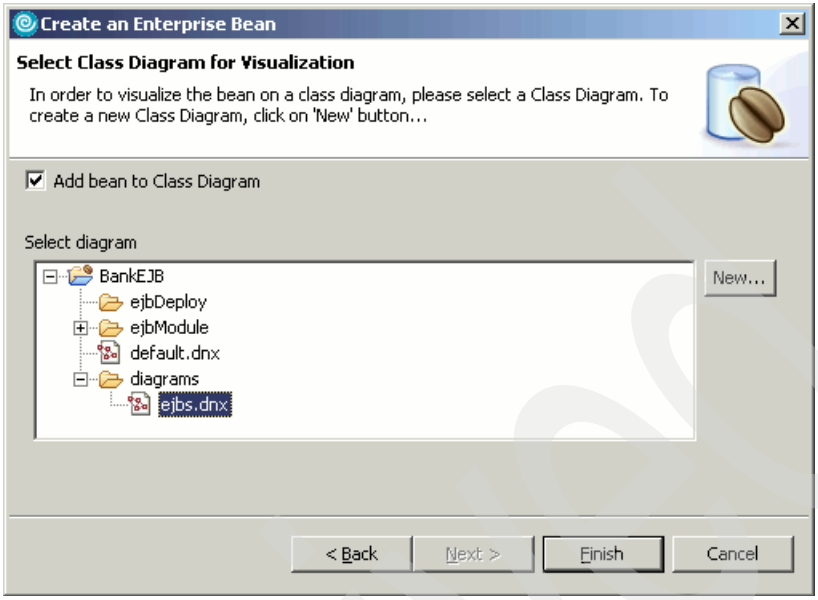


Figure 15-20 Creating an entity bean (page 4)

The new UML class diagram should be displayed with the Customer entity bean, as seen in Figure 15-21 on page 868.

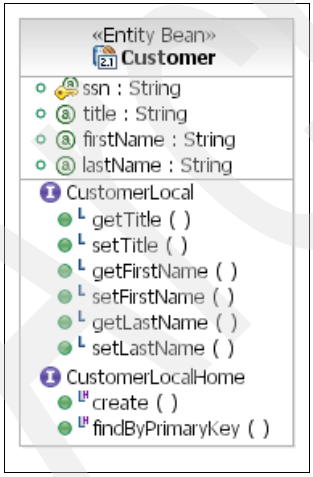


Figure 15-21 Class diagram with Customer entity bean

12. Select **File** → **Save**, or press Ctrl+S, to save the diagram, and then close the window.

Define the Account and Transaction beans

Repeat the same process for the next two CMP entity beans, Account and Transaction, according to the data on Table 15-2 and Table 15-3.

Important: Make sure to select **Use the single key attribute type for the key class**, as shown on Figure 15-17 on page 865.

Table 15-2 Account bean CMP attributes

Name	Type	Attribute type check box
id	java.lang.String	Key field
balance	int	Promote getter and setter methods to local interface.

Table 15-3 Transaction bean CMP attributes

Name	Type	Attribute type check box
id	java.lang.String	Key field
amount	int	Promote getter and setter methods to local interface
timestamp	java.util.Date	Promote getter and setter methods to local interface

Archived

Tip: Approaches to resolve no natural unique identifier follow.

It is common to find entities that do not have a natural unique identifier, as is the case with our Account and Transaction objects. The EJB 2.0 specification approached this problem when it introduced the unknown primary key class for CMP entity beans. Even though WebSphere Application Server has implemented this part of the specification since Version 5, IBM Rational Application Developer V6.0 does not include support for this.

There are basically two approaches to the problem. The first is to have the back-end database generate the unique identifiers. This is feasible because even though you may have as many application servers as you may like, the data pertinent to a single entity will hopefully be stored in just one database. The downside to this approach is that every time an entity is created, a database table must be locked in order to generate the ID, and thus becomes a bottleneck in the process. The upside is that sequential identifiers may be generated this way. This was our selected approach for the Account bean.

The second approach would be to generate the universally unique identifiers (UUIDs, unique even among distributed systems) in the application server tier. This is a little tricky, but can be accomplished. One way to do it is to come up with a utility class that generates the UUIDs based on a unique JVM identifier, the machine's IP address, the system time, and an internal counter. This technique is usually more efficient than having the back-end database generate UIDs because it does not involve table locking. This was our selected approach for the Transaction bean. We used the class com.ibm.ejs.util.Uuid to generate UUIDs. This class ships with Rational Application Developer and WebSphere Application Server.

Define the Credit and Debit derived beans

Complete the creation of our business entities by defining the last two beans: Credit and Debit. Both are subtypes of Transaction, so the process of creating them is slightly different.

1. Select and expand **EJB Projects** → **BankEJB** from the Project Explorer view.
2. Select **File** → **New** → **Enterprise Bean**.
3. When the Create an Enterprise Bean dialog appears, do the following to define the Credit bean (subtype of Transaction):
   - Select **Entity bean with container-managed persistence (CMP) fields**.
   - Bean name: `Credit`
   - Default package: `itso.bank.model.ejb`
   - Leave the remaining options with the default values and click **Next**.

4. When the Enterprise Bean Details dialog appears, select **Transaction** in the Bean supertype drop-down, as seen in Figure 15-22, and click **Next**.

   This page lets you select the supertype (allowing you to define the inheritance structures), type names, which views you would like to create, and finally the key class and CMP attributes.

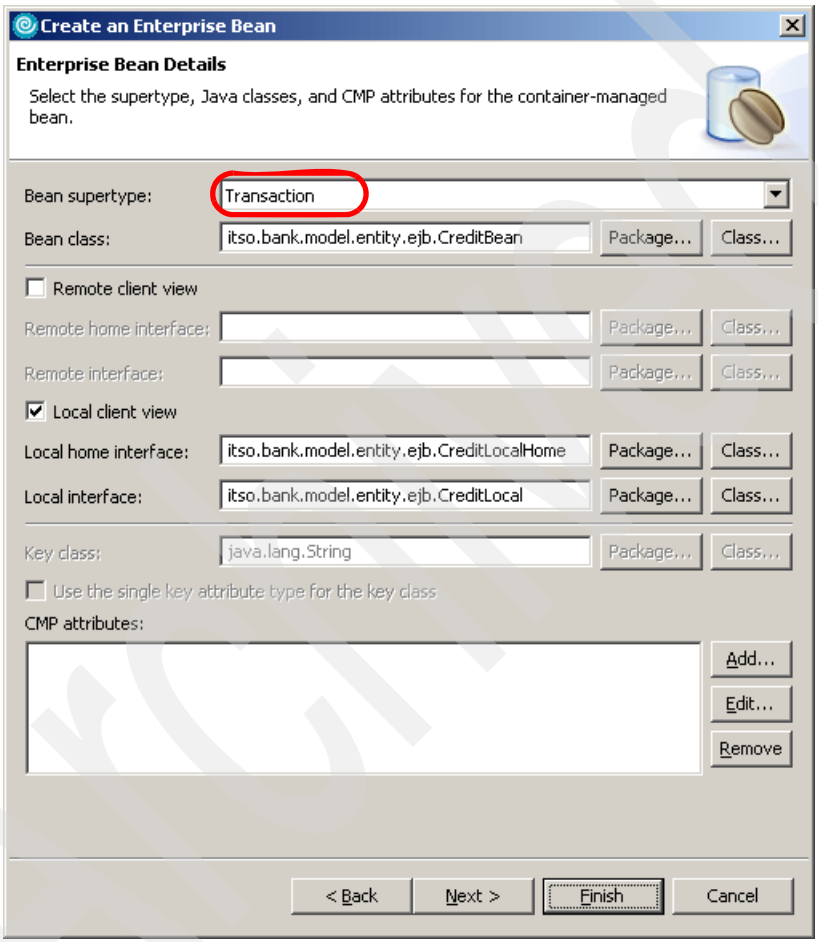


Figure 15-22 Define a bean supertype

5. When the EJB Java Class Details page appears, click **Next**.

> **Note:** As mentioned earlier, this page has nothing to do with EJB inheritance. As such, we leave the page blank.

6. When the Select Class Diagram for Visualization page appears, expand and select **BankEJB** → **diagrams** → **ejbs.dnx** and click **Finish**.
7. Repeat the process for the Debit bean. None of the beans have additional attributes, apart from the attributes inherited from the Transaction EJB. They only differ in behavior.
8. At this point, we have all the five entities created (Customer, Account, Transaction, Credit, and Debit). Your class diagram should look like Figure 15-23. As you can see, for each entity bean, we have a primary key attribute, regular attributes, and home and component interfaces.
9. Save the diagram, and the close the editor.

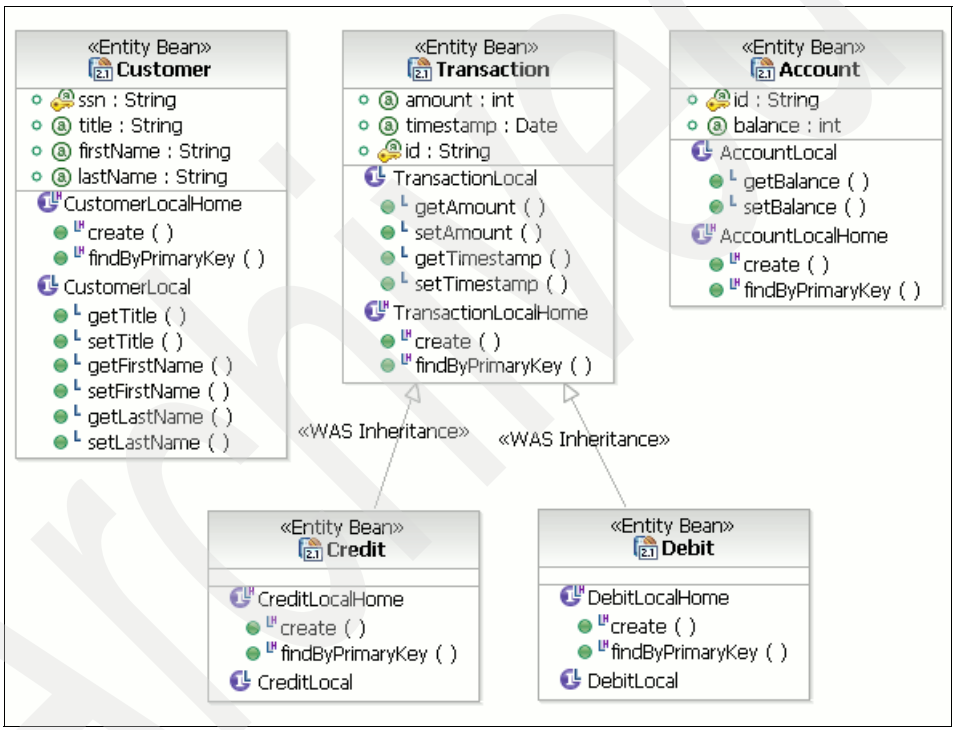


Figure 15-23 Class diagram at entity bean creation phase

15.4.2 Create the entity relationships

Now that the five business entities have been created, it is time to specify their relationships: A one-to-many unidirectional association, implementing an analysis composition, and a many-to-many bidirectional association (see Figure 15-24 on page 873).

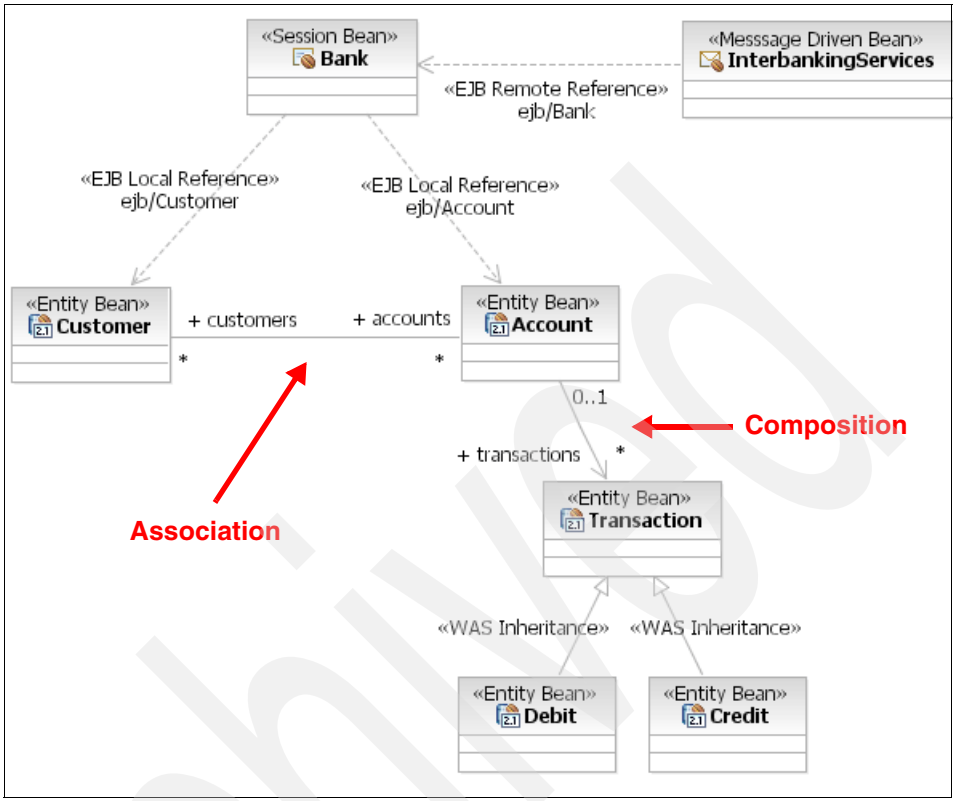


Figure 15-24 Association relationships in the model

Rational Application Developer offers a couple of facilities to streamline the process of both creating and maintaining container-managed relationships (CMRs). You can, for instance, visually create relationships with the UML Diagram Editor, and the environment will automatically generate all the necessary code and deployment descriptor changes. You may, alternatively, edit the deployment descriptor directly, and Rational Application Developer will also generate the appropriate code changes. Finally, you can use the environment's menus to accomplish the same task.

In the following sections, we will use the first two strategies described above to create the association and the composition relationships, respectively.

Archived

Customer Account association relationship

The first relationship that we will add is the association between the Customer bean and the Account bean. In this example, we demonstrate how to define relationships using the UML Diagram Editor.

1. From the Project Explorer view, expand **EJB Projects** → **BankEJB** → **diagrams**.
2. Double-click **ejbs.dnx** to open it with the UML Editor.
3. When the editor opens, click the down-arrow to the right of `0..1:0..1 CMP Relationship` in the Palette and select **0..*:0..* CMP Relationship**, as shown in Figure 15-25 on page 874.

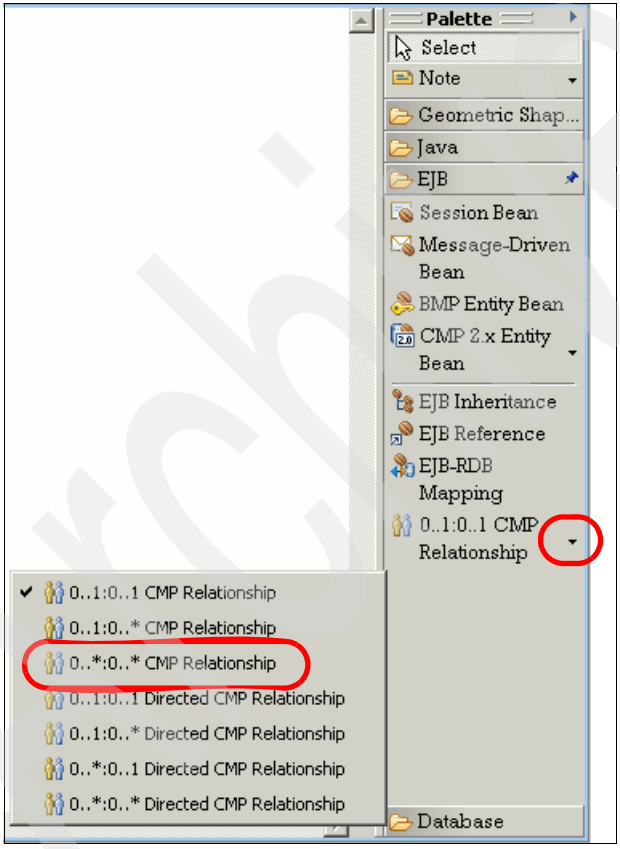


Figure 15-25 Selecting the relationship type and multiplicity

4. Left-click and hold the mouse arrow over the **Customer** bean, and then drag the mouse towards the **Account** bean. Release the button over the **Account** bean to create the association relationship, as shown in Figure 15-26.

Archived

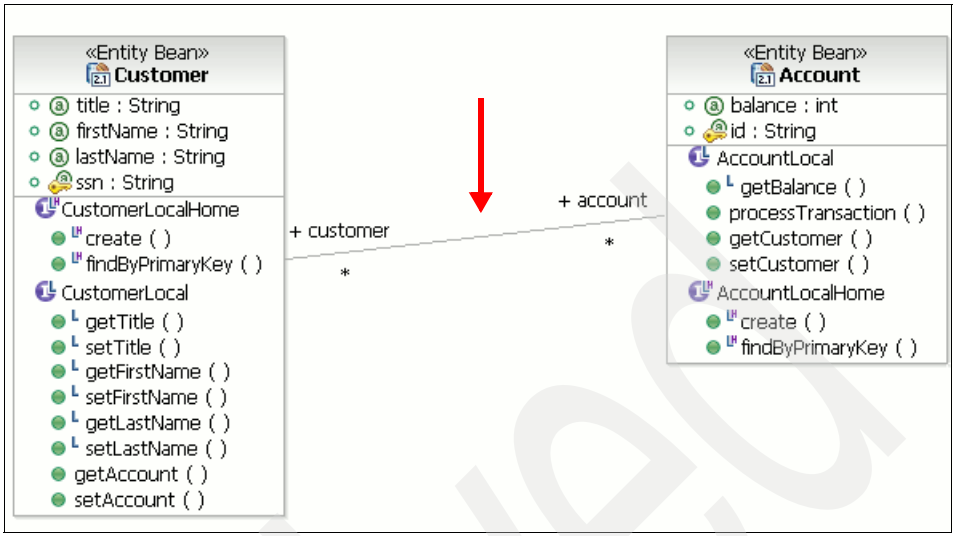


Figure 15-26 Association relationship between Customer and Account (part 1)

> **Note:** The names at the ends of the association (in this case customer and account) are used to generate accessor methods in the two beans' interface.
>
> The plus sign (+) means that the relationship is visible to the associated entity at the respective end.

5. Double-click **+ customer** and change to **+ customers**.
6. Double-click **+ account** and change to **+ accounts.**

   The resulting relationship and the added methods to support it should look similar to Figure 15-27.

Archived

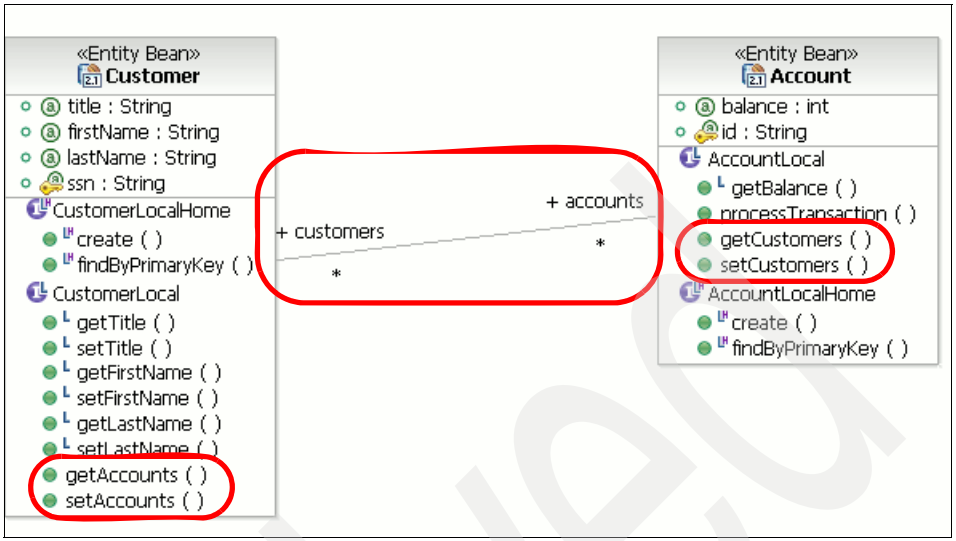


Figure 15-27 Finished customer account relationship

7. Save the changes and close the editor.

Important: If you ever need to delete the relationship, you need to select it, open its context menu, and select **Delete from Deployment Descriptor**.

If you select Delete from Diagram, or simply press Delete, the relationship is only removed from the diagram, but stays in the model.

If a relationship is deleted from the diagram but stays in the model, it can be redrawn by right-clicking one of the related entities and selecting **Filters** → **Show / Hide Relationships**. The resulting Show/Hide Relationships dialog can then be used to show or hide specific relationship types.

Account Transaction composition relationship

The second relationship that needs to be created is the composition between the Account and the Transaction beans. It represents the account's transaction log that has to be maintained over time. In this example, we create the relationship using the Deployment Descriptor Editor to demonstrate an alternative to the UML Diagram Editor.

1. In the Project Explorer view, expand **EJB Projects** → **BankEJB** → **Deployment Descriptor: BankEJB** → **Entity Beans**.

2. Double-click the **Account** bean to open the EJB Deployment Descriptor Editor. The Deployment Descriptor Editor will open on the Bean tab with the Account bean selected.
3. Scroll down to the Relationships section, as shown in Figure 15-28.

> **Tip:** There are alternative ways to open the EJB deployment descriptor. On the Project Explorer view, you can double-click **Deployment Descriptor: BankEJB**, or navigate **BankEJB** → **ejbModule** → **META-INF** and double-click **ejb-jar.xml**.

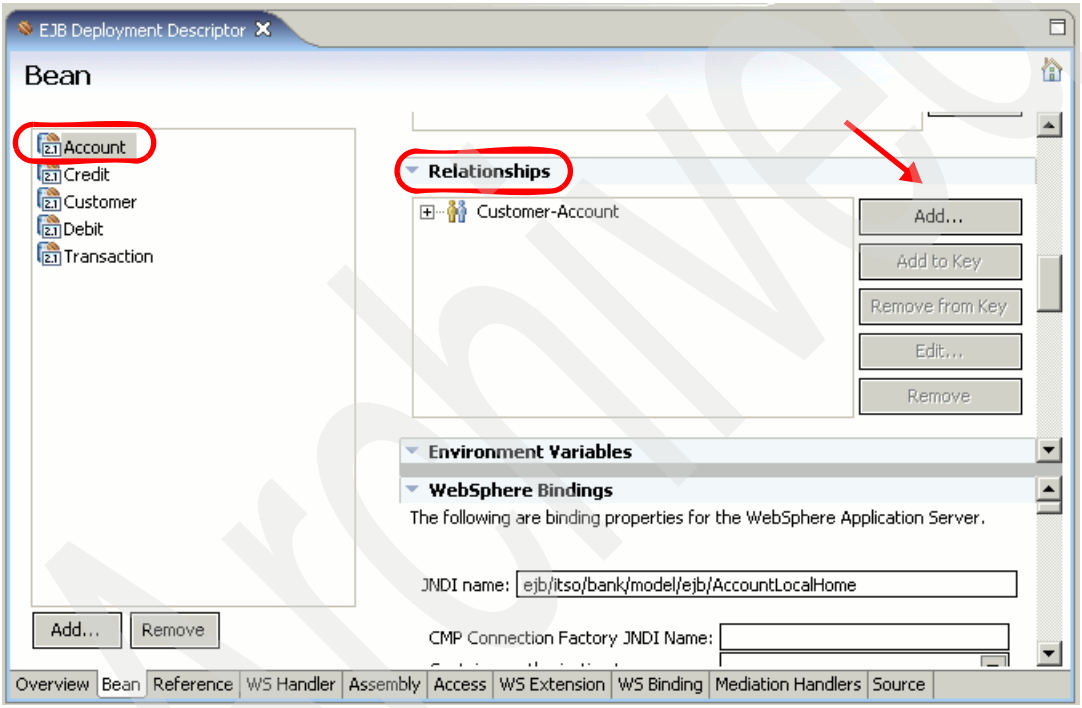


Figure 15-28 Defining relationships with the EJB Deployment Descriptor Editor

4. Click **Add** to create a new relationship for the Account bean.
5. The Add Relationship wizard opens, showing an UML view of the relationship, with the Account bean already displaying on the left-hand side.

   Select the **Transaction** bean on the right-hand side list box. The wizard should automatically fill in the remaining Role name field, as shown in Figure 15-29.

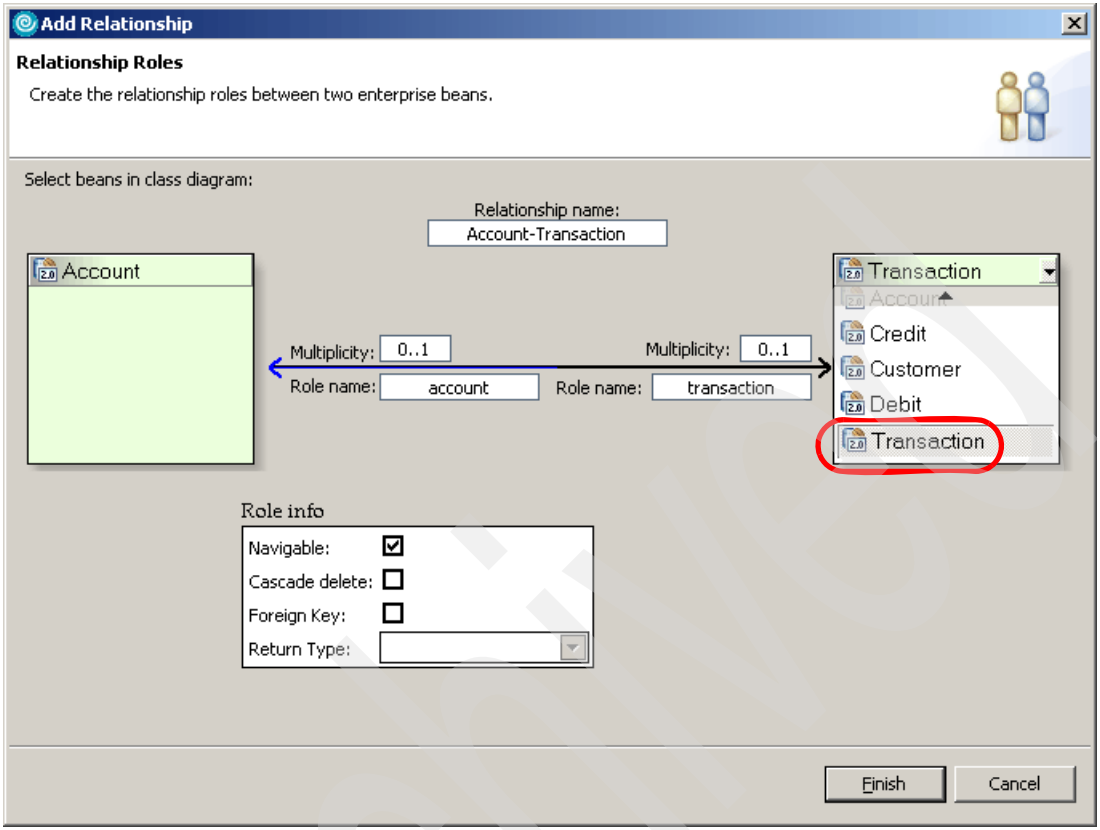


Figure 15-29 Add relationship wizard (part 1)

6. Modify the relationships.
   a. Modify the Transaction bean multiplicity. The Account bean's role name and multiplicity are already correct, but the Transaction's are not and need to be edited as follows:
      i. Modify the Transaction multiplicity by double-clicking **0..1** and changing it to `0..*`
      ii. Modify the Transaction role name to be plural, due to the relationship's multiplicity (one account may be associated with many transactions) by double-clicking the **transaction** role name and changing it to `transactions`.
   b. We also want to guarantee that the same transaction is not added to the account twice. Hover the mouse cursor over the Transaction's end of the relationship and select the return type of **java.util.Set**.

c. Hover the mouse cursor over the Account's end of the relationship to display the its role information. As this relationship is a composition relationship, the composed objects should have no knowledge about the composer. Uncheck the **Navigable** check box, as seen in Figure 15-30.

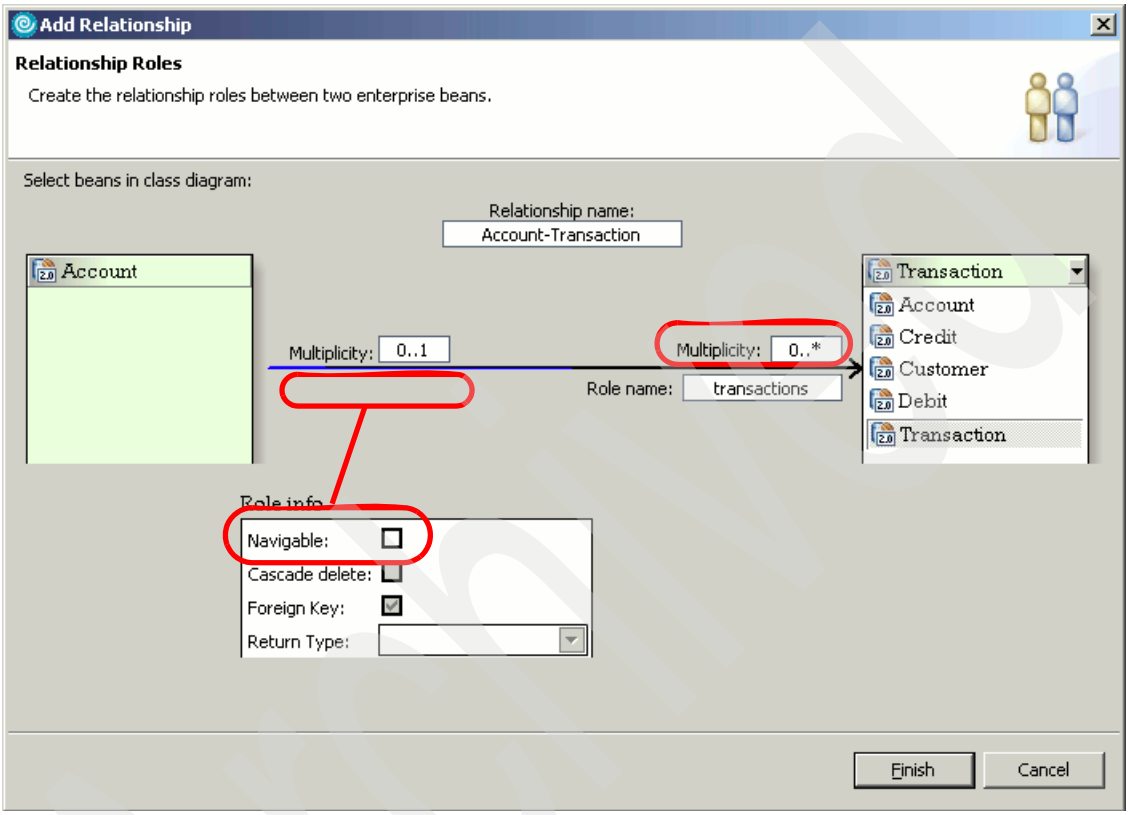


Figure 15-30 Add Relationship wizard (part 2)

7. Click **Finish** to complete the wizard.
8. Save the deployment descriptor to apply the changes just made and close it.
9. You may now inspect the changes the wizard made to your EJBs. Among them are the ones made to the `Account` bean.
   a. In the Project Explorer view, double-click the **AccountLocal** interface under the Account bean to open it with the Java editor.
   b. The following methods will have been added:

```
public java.util.Collection getCustomers();
public void setCustomers(java.util.Collection aCustomers);
public java.util.Set getTransactions();
```

```
public void setTransactions(java.util.Set aTransactions);
```

c. Close the editor.

15.4.3 Customize the entity beans and add business logic

Now that our five entity beans have been created, it is time for us to do a little programming. For each of the beans created, three types were generated: The bean class, the home interface, and the local component interface.

1. View the Account bean, for instance, to see the generated code, as these types are shown in Figure 15-31.

   From the Project explorer view, select and expand **EJB Projects** → **BankEJB** → **Deployment Descriptor: BankEJB** → **Entity Beans** → **Account**.

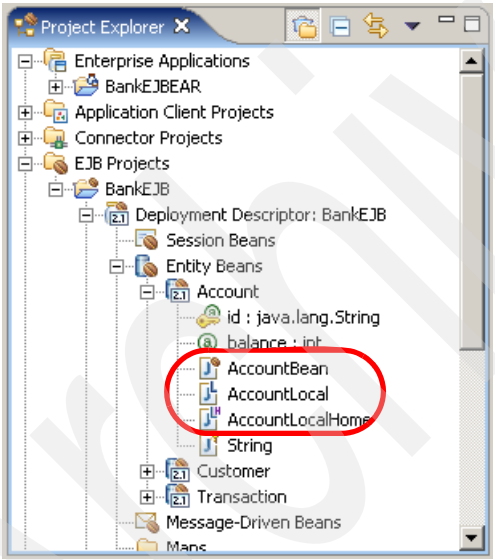


Figure 15-31 Generated types for the Account bean

> **Note:** There is a fourth type associated with the Account bean: String. It is the primary key class. If we had chosen a non-object key field or multiple key fields, a key class would also have been generated.

2. Double-click the **AccountBean** class.
3. Select the **Outline** view. It should look similar to Figure 15-32 on page 881.

Archived

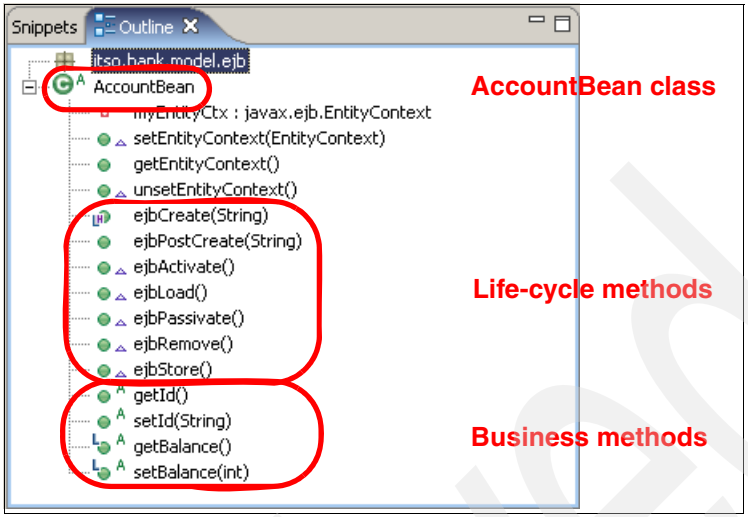


Figure 15-32 Outline view of the AccountBean class

There are two kinds of methods generated (see Figure 15-32):

- Life-cycle methods, which are the callback methods used by the EJB container at predefined events
- Business methods, which manipulate the CMP attributes

Manage the home and component interfaces

As you can see, Rational Application Developer has already generated the life-cycle methods and some business methods for us. The latest, of course, are the getters and setters. Some of these methods belong to the bean's home interface. Others belong to the local interface. Some of them are simply private to the component and should not be exposed.

Whether a method is exposed to the interface or not is a design choice. We will make some modifications to the generated interfaces in order to accommodate our design decisions.

Limit access to the Transaction bean

We do not want clients to instantiate the Transaction EJB or modify fields in an already created instance. The reason for the former is that in our model, this represents an abstract entity. While there is no such thing as an abstract EJB, in the same sense as an abstract Java class, we can obtain a similar behavior by removing any ejbCreate methods from the remote and home interfaces. The reason why we do not want clients to change a transaction object is that these should appear immutable, as they represent a log of historical transactions.

In order to block access for clients to the setter methods, as well as the constructor and setters for the Transaction EJB, do the following:

1. In the Outline view for the AccountBean class, right-click **setBalance** and select **Enterprise Bean → Demote from Local Interface**.

   Notice that the L-shaped icon next to `setBalance` disappears. The seBalance method is now inaccessible to clients.

2. In the Project Explorer, expand **EJB Projects → BankEJB → Deployment Descriptor: BankEJB → Entity Beans → Transaction**.
3. Double-click **TransactionBean** to open TransactionBean.java in the Java editor.
4. In the Outline view, right-click **ejbCreate(String)** and select **Enterprise Bean → Demote from Local Home Interface**.

   Notice that the LH-shaped icon next to ejbCreate(String) disappears.

5. In the Outline view, right-click **setAmount(int)** and select **Enterprise Bean → Demote from Local Interface**.
6. In the Outline view, right-click **setTimestamp(int)** and select **Enterprise Bean → Demote from Local Interface**.

Modify constructors for Transaction, Credit, and Debit

When creating a transaction object, we want to be able to specify the transaction amount and let the bean generate the identifier and timestamp automatically. Do the following to accomplish this.

Tip: The Java code for this section can be copied from the file c:\6449code\ejb\source\Transactions.jpage, included with the additional material for this book.

1. In the Project Explorer, expand **EJB Projects → BankEJB → Deployment Descriptor: BankEJB → Entity Beans → Transaction**.
2. Double-click **TransactionBean** to open TransactionBean.java in the Java editor.
3. Modify the ejbCreate and ejbPostCreate methods so they look like Example 15-1 on page 883. Save when done.

Note: As noted earlier, the class com.ibm.ejs.util.Uuid, referenced in Example 15-1, is used to generate unique identifiers. We use it here to automatically generate identifiers for any transaction object. Clients will thus not have to worry about this.

Archived

Example 15-1 TransactionBean ejbCreate and ejbPostCreate

```
public java.lang.String ejbCreate(int amount)
   throws javax.ejb.CreateException {

   setId((new com.ibm.ejs.util.Uuid()).toString());
   setAmount(amount);
   setTimestamp(new java.util.Date());
   return null;
}

public void ejbPostCreate(int amount)
   throws javax.ejb.CreateException {
}
```

Note: At this stage you will likely see the following two warning messages in the Problems view:

```
CHKJ2504W: The ejbCreate matching method must exist on itso.bank.model.ejb.CreditBean (EJB 2.0: 10.6.12)

CHKJ2504W: The ejbCreate matching method must exist on itso.bank.model.ejb.DebitBean (EJB 2.0: 10.6.12)
```

This is simply a message that we need to align the subtypes Credit and Debit with the new Transaction constructor. We will do this in the next two steps.

4. In the Project Explorer, expand **EJB Projects** → **BankEJB** → **Deployment Descriptor: BankEJB** → **Entity Beans** → **Transaction** → **Credit**.
5. Double-click **CreditLocalHome** to open CreditLocalHome.java in the Java editor.
6. Modify the create method signature to match Example 15-2. Save when done.

Example 15-2 Modify create method in CreditLocalHome.java

```
public itso.bank.model.ejb.CreditLocal create(int amount)
      throws javax.ejb.CreateException;
```

7. In the Project Explorer, expand **EJB Projects** → **BankEJB** → **Deployment Descriptor: BankEJB** → **Entity Beans** → **Transaction** → **Debit**.
8. Double-click **DebitLocalHome** to open DebitLocalHome.java in the Java editor.
9. Modify the create method signature to match Example 15-3. Save when done.

Example 15-3 Modify create method in DebitLocalHome.java

```
public itso.bank.model.ejb.DebitLocal create(int amount)
      throws javax.ejb.CreateException;
```

Add the business logic

Getters and setters are generated automatically, but the business methods must be implemented manually.

We will need to add business logic to the Transaction, Credit, Debit, and Account EJBs. We will add a new method, getSignedAmount, to the transaction EJBs. This method will return the transaction amount with a sign that denotes the transaction "direction" in such a way that the return value added to the pre-transaction account balance will yield the post-transaction account balance.

We will then add the method processTransaction to the Account EJB. This method will, given a Transaction instance, update the account balance, utilizing the new getSignedAmount method.

Tip: The Java code for this section can be copied from the file c:\6449code\ejb\source\Transactions.jpage, included with the additional material for this book.

Add business logic to the transaction EJBs

To add the getSignedAmount method to the transaction EJBs, do the following:

1. Open TransactionBean.java for editing.
2. Add the method getSignedAmount, shown in Example 15-4, to the TransactionBean class.

Example 15-4 TransactionBean getSignedAmount method

```
public int getSignedAmount()
      throws itso.bank.exception.ApplicationException
{
   throw new itso.bank.exception.ApplicationException(
         "Transaction.getSignedAmount invoked!");
}
```

The implementation of the getSignedAmount method is not relevant in the base Transaction bean, since the method is conceptually abstract. In a Java class hierarchy, this method would be made abstract, but this is not possible in an EJB hierarchy. If we were to define the getSignedAmount as abstract, the deployed code would have errors, and the EJB would not be deployable.

We thus choose to throw an exception from the method in case a programming error results in the execution of the method.

3. Promote the newly created getSignedAmount method to the bean's local interface.
   a. In the Outline view, right-click **getSignedAmount** and select **Enterprise Bean → Promote to Local Interface**.
   b. Save the changes and close the Java editor.
4. Modify the CreditBean class.
   a. Double-click the **CreditBean** class on the Project Explorer view to open the class in the Java editor.
   b. Insert the declarations in Example 15-5 to the CreditBean.java file.

Example 15-5 CreditBean extensions to TransactionBean

```
/**
 * Insert before the first method
 */
public static final String TYPE_KEY = "Credit";

/**
 * Insert after the last method
 */
public int getSignedAmount()
      throws itso.bank.exception.ApplicationException
{
   return getAmount();
}
```

Tip: As getSignedAmount is an inherited method, the code can be alternatively created by Rational Application Developer's sourcing facility, relieving you from having to type it.

1. Select the **CreditBean** class from the Outline view, right-click, and select **Source → Override/Implement Methods**.
2. On the resulting dialog, click **Deselect All** to make sure that no other methods have their signatures generated. Next, check the **getSignedAmount** method and click **OK**.
3. Manually add the following after the CreditBean class:

```
public static final String TYPE_KEY = "Credit";
```

   c. Save your changes and close the editor.
4. Modify the DebitBean class.

Archived

a. Double-click the **DebitBean** class on the Project Explorer view to open the class up on the Java editor.

b. Insert the declarations in Example 15-6 to the CreditBean.java file.

Example 15-6 DebitBean extensions to TransactionBean

```
/**
 * Insert before the first method
 */
public static final String TYPE_KEY = "Debit";

/**
 * Insert after the last method
 */
public int getSignedAmount()
       throws itso.bank.exception.ApplicationException
{
   return -getAmount();
}
```

c. Save the changes and close the editor.

Note: The TYPE_KEY constant defined in the Debit and Credit classes is used by the EJB container to determine what type a given record in the database refers to, since both Credit and Debit instances will be persisted to the same database table. The value of this constant is compared to the value of the DISCRIM_*<tablename>* column, where *<tablename>* is the name of the table that the EJB is being persisted to.

Add business logic to the Account bean

When we specified the account bean's CMP fields, for instance, we configured the wizard to expose both the setter and the getter for the balance attribute to the bean's local interface. This is why there is a small L-shaped icon next to the getBalance and setBalance methods. While it is fine for a client to retrieve an account's balance, we do not want the clients to change the balance by calling the setBalance method. Later we will implement and expose another business method, processTransaction, that manipulates the balance, adhering to business rules.

From a business perspective, it makes no sense to allow the creation of accounts that are not associated to any customers. Thus, the first modification that we want to make is to guarantee that accounts are not created unless a primary customer is specified.

Archived

Tip: The Java code for this section can be copied from the file c:\6449code\ejb\source\AccountBean.jpage, included with the additional material for this book.

1. Open the **AccountBean** class by double-clicking it from the Project Explorer view.
2. Refactor the ejbCreate(String) method signature.

   Tip: For refactoring, if you need to edit the signature of any method that already belongs to either the remote or home interface, the easiest way is as follows:

   - First demote the method from the interface.
   - Edit the signature.
   - Promote the method back to the interface.

   If you do not demote the method from the interfaces first, you will have to manually edit the method signatures in these interfaces. Using the approach mentioned here, you let Rational Application Developer update the method signatures in the interfaces when the method is promoted back to the interfaces.

   Since the *throws* clause for a method is part of the method signature, this procedure should also be followed when changing this clause.

   a. From the Outline view, select the existing **ejbCreate(String)** method.

   b. Right-click and select **Enterprise Bean → Demote from Local Home Interface** to remove the create method declaration from the bean's home interface.

   c. Modify the ejbCreate(String) and ejbPostCreate(String) methods so that they look like the definitions in Example 15-7.

Example 15-7 Account bean's ejbCreate and ejbPostCreate methods

```
public java.lang.String ejbCreate(CustomerLocal primaryCustomer)
      throws javax.ejb.CreateException
{
   setId((new com.ibm.ejs.util.Uuid()).toString());
   return null;
}

public void ejbPostCreate(CustomerLocal primaryCustomer)
      throws javax.ejb.CreateException
{
```

```
    getCustomers().add(primaryCustomer);
}
```

> **Important:** The getCustomers method was generated when we created the association relationship between the Account and the Customer beans. This method cannot be called from the Account's ejbCreate method. According to the specification, during ejbCreate the instance cannot acquire references to the associated entity objects.
>
> In the ejbPostCreate method, on the other hand, the instance may reference the associated entity objects. Thus, a call to getCustomers can be made.

d. Now add the create method back to the Account bean's home interface by selecting the **ejbCreate(CustomerLocal)** method on the Outline view, right-clicking it, and selecting **Enterprise Bean → Promote to Local Home Interface**.

3. When we created the relationship between the Account and the Customer beans, the method *setCustomers(Collection)* was created. It does not have to be available to the bean's clients. Right-click **setCustomers(Collection)** in the Outline view and demote it from the local interface by selecting **Enterprise Bean → Demote from Local Interface**.

4. Demote the accessors for the transactions relation.

   As you may recall, the relationship between the Account bean and the Transaction bean is a composition. This means that references to transaction objects cannot be exported to objects outside of the Account bean.

   When we created the relationship, getTransactions and setTransactions(Set) methods were generated and added to the local interface for the Account bean. We want clients to be able to check the transaction log. If we were to allow access to these methods, clients would be able not only to do so, but also to add transactions to the log without correctly processing them using the processTransaction method, which we will define later in this section. The result would be that the affected account object would be put into an invalid state. This would be breaching the object's encapsulation, and thus should not be allowed.

   Demote both getTransactions and setTransactions(Set) methods, then add the getLog method as follows:

   a. Right-click **getTransactions** from the Outline view, and select **Enterprise Bean → Demote from Local Interface**.

   b. Right-click **setTransactions(Set)** from the Outline view, and select **Enterprise Bean → Demote from Local Interface**.

c. Add the getLog method, as seen in Example 15-8, to the AccountBean.java.

Example 15-8 Account bean's getLog method

```
public java.util.Set getLog()
{
   return java.util.Collections.unmodifiableSet(getTransactions());
}
```

d. Right-click **getLog** from the Outline view, and select **Enterprise Bean** → **Promote from Local Interface**.

5. Add the processTransaction method to the AccountBean class.

   a. Double-click **AccountBean** in the Project Explorer view to open the class in the Java editor.

   b. Insert the declarations in Example 15-9 in to the AccountBean.java file to add the processTransaction method.

Example 15-9 AccountBean processTransaction method

```
public void processTransaction(TransactionLocal transaction)
      throws itso.bank.exception.ApplicationException
{
   setBalance(getBalance() + transaction.getSignedAmount());
   getTransactions().add(transaction);
}
```

The processTransaction method receives a transaction object local reference and changes the account's balance by adding the transaction's signed amount to it. Later in this chapter, we will alter this method to also add the transaction reference into a transaction log.

   c. Save the changes and close the editor.

6. Promote the processTransaction method to the bean's local interface.

   a. Add the method to the bean's local interface by right-clicking **processTransaction** from the Outline view.

   b. Select **Enterprise Bean** → **Promote to Local Interface**.

Archived

15.4.4 Creating custom finders

When you create an entity bean, you always get the findByPrimaryKey finder method on the home interface. Sometimes, though, you need to find an entity based on criteria other than just the primary key. For these occasions, the EJB 2.1 specification provides a query language called EJB QL. Custom finder methods are declared in the home interface and defined in the EJB deployment descriptor using the EJB QL.

Our example requires two similar simple custom finders: One for the Account bean and the other for the Customer bean. To add them, do the following:

1. From the Project Explorer, expand and double-click **EJB Projects** → **BankEJB** → **diagrams** → **ejbs.dnx** to open the class diagram in the UML Diagram Editor.
2. Modify the Account bean to create a findAll query.
   a. Right-click the **Account** bean to open its context menu and then select **Add EJB** → **Query**.
   b. When the Add Finder Descriptor wizard appears, do the following:
      - Method: Select **New**.

        The only finder we have in our AccountLocalHome interface is the default findByPrimaryKey. Rational Application Developerwill take care of updating the home interface for you by adding the declaration of the new finder method.
      - Method Type: Select **find method**.

        The Method Type field lets you select whether you want to create a descriptor for a finder method or for an ejbSelect method. The difference between the two is that finder methods get promoted to the bean's home interface, whereas ejbSelect methods do not. The ejbSelect methods are useful as internal helper methods, as they cannot be called by clients.
      - Type: Check **Local**.

        The Type field lets you select to which home interface, either local or remote, you would like the finder method promoted.

        Note: We found that in some cases, the Remote check box was enabled, since the Account bean only exposes a local view. If you check Remote, the query is only added to the deployment descriptor, but no changes are made to the home interface. Make sure that only **Local** is checked.

Archived

- Name: `findAll`
- Return type: Select **java.util.Collection**.

The dialog should look like Figure 15-33 on page 891 after making the selections. Click **Next** to continue.

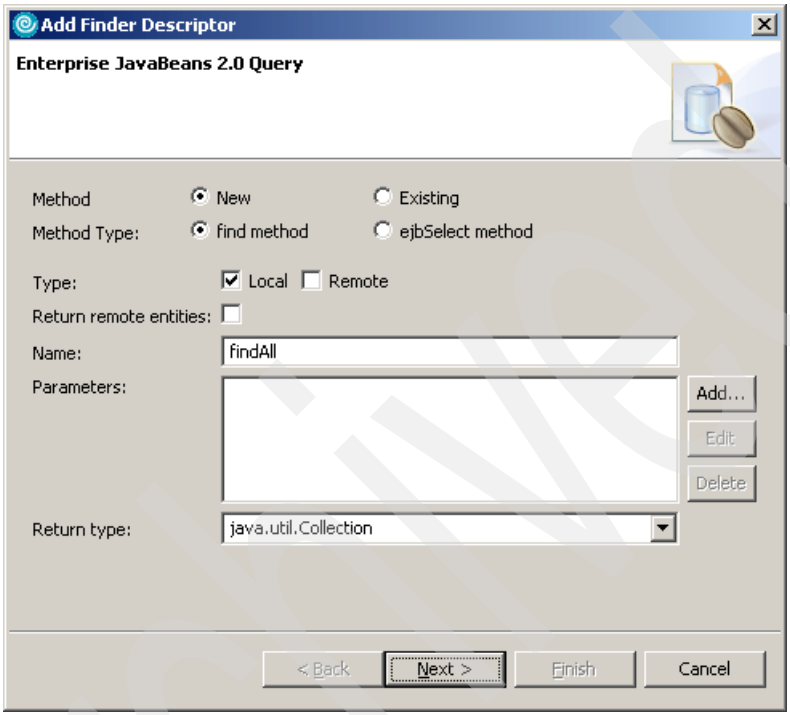


Figure 15-33 Adding a new finder descriptor (page 1)

c. When the Add Finder Descriptor dialog appears, select **Find All Query** and click **Finish** (see Figure 15-34 on page 892).

Archive

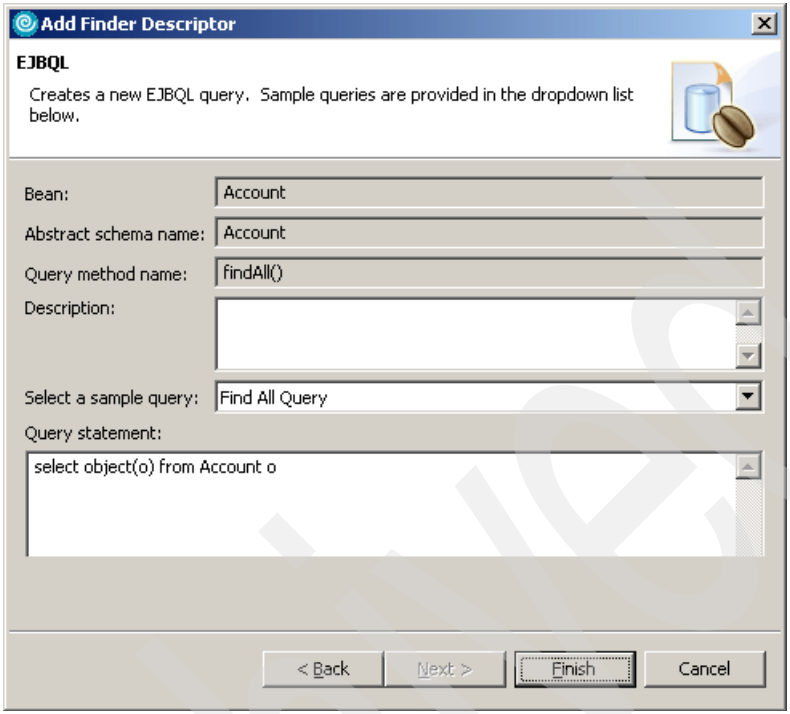


Figure 15-34 Adding a new finder descriptor (page 2)

3. Repeat the same steps for the Customer bean to create a findAll query.
4. Your changes should already be saved, so you may close the editor.

15.4.5 Object-relational mapping

Container-managed persistence (CMP) entity beans delegate their persistence to the container. As mentioned in “EJB types” on page 836, this means that it is the responsibility of the EJB container to handle the details of how to store the internal state of the EJBs.

In order for the container to be able to do this, we need to provide information about how the EJB fields should be mapped to the relational database. This information is stored in the deployment descriptor, and during deployment, the JDBC code to perform the operations is generated by the container.

When the beans are actually deployed, associations to real data sources can be made to dynamically bind the bean to the data. In this way, the CMPs are abstract classes that associate to data, but do not provide any implementation for accessing data themselves.

Archived

To facilitate development, deployment, and testing, Rational Application Developer contains the tools for the developer to both define the mappings and create deployment bindings.

The advantages to separating the development and persistence concerns are numerous. Apart from achieving database implementation independence, the developer is free to work with object views of the domain data instead of data views and writing SQL, and is allowed to focus on the business logic, instead of the technical details of accessing the database.

As mentioned, the CMP can be developed largely independently of the data source, and allows a clear separation of business and data access logic. This is one of the fundamental axioms of aspect-oriented programming, where the aspect of persistence can be removed from the development process and applied later, in this case at deployment time.

Rational Application Developer offers three different mapping strategies:

- *Top down* is when you start from an object-oriented model and let the environment generate the data model automatically for you, including the object-relational mapping and the DDL that you would use to create the tables in the database.

  Note: This strategy is preferred when the data backend does not exist and will be created from scratch.

- *Bottom up* is when you start from a data model and let Rational Application Developer generate the object model automatically for you, including the creation of the entity beans based on tables and columns that you select.

  Note: This strategy is not recommended for object-oriented applications, because the data model is less expressive than the object model. It should be used only for prototyping purposes.

- *Meet in the middle* is the compromise strategy, in which you keep both your existing object-oriented and data models, creating a mapping between the two. The mapping process is usually started by Rational Application Developer, based on cues like attribute names and types, and completed manually by the Developer.

For the ITSO Bank application, we will use the meet in the middle strategy because we do have an existing database for application data.

The BANK database schema

This chapter uses the same relational model created in Chapter 8, “Develop Java database applications” on page 333. Figure 15-35 shows the existing data model.

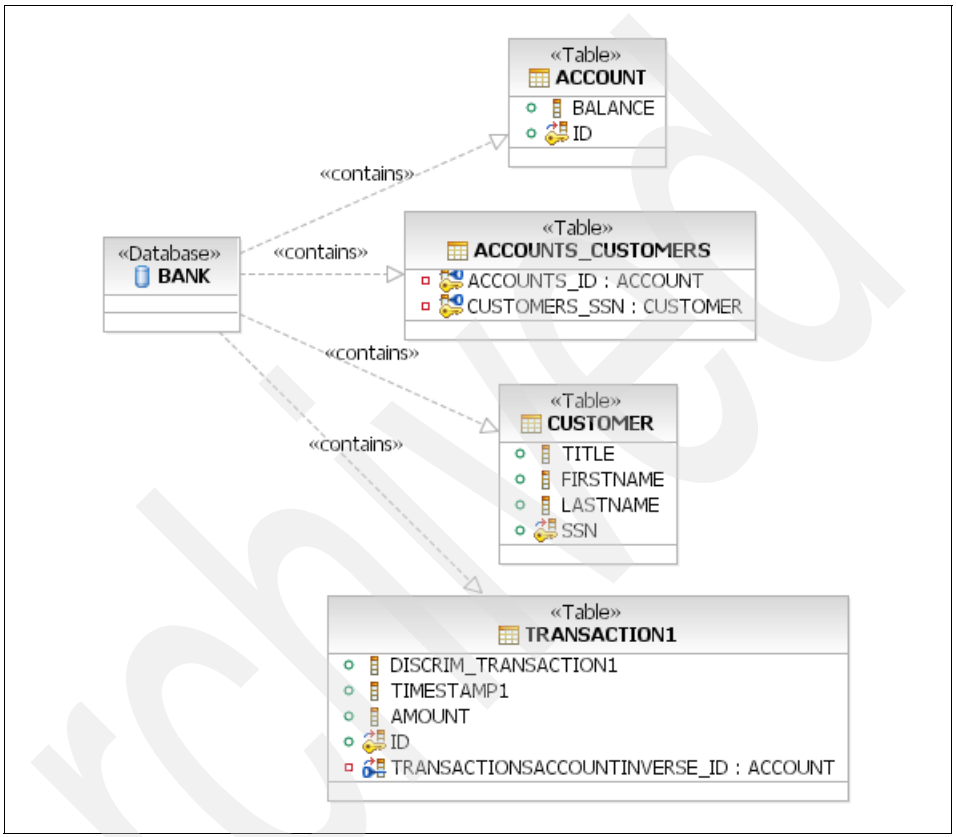


Figure 15-35 Existing relational model

Our objective is to map the object model created in 15.4, “Develop an EJB application” on page 858, to the existing relational model. As mentioned, this is a more realistic approach than creating a new top-down mapping from scratch, although a top-down mapping would be simpler and more convenient if a relational database model did not already exist.

Archived

Generate EJB-to-RDB mapping

We now create a mapping between the object-oriented EJB model and the relational database model, as defined by the BANK database schema. To generate the EJB-to-RDB mapping, do the following:

1. In the Project Explorer view of the J2EE perspective, right-click **EJB Projects** → **BankEJB** and select **EJB to RDB Mapping** → **Generate Map** from the context menu.
2. When the EJB to RDB Mapping wizard appears, select **Use an existing backend folder**. Select **CLOUDSCAPE_V51_1** and click **Next** (see Figure 15-36 on page 895).

> **Note:** This is the folder that was created when we copied the database metadata to the EJB project folder in 15.3.6, "Set up the sample database" on page 854.

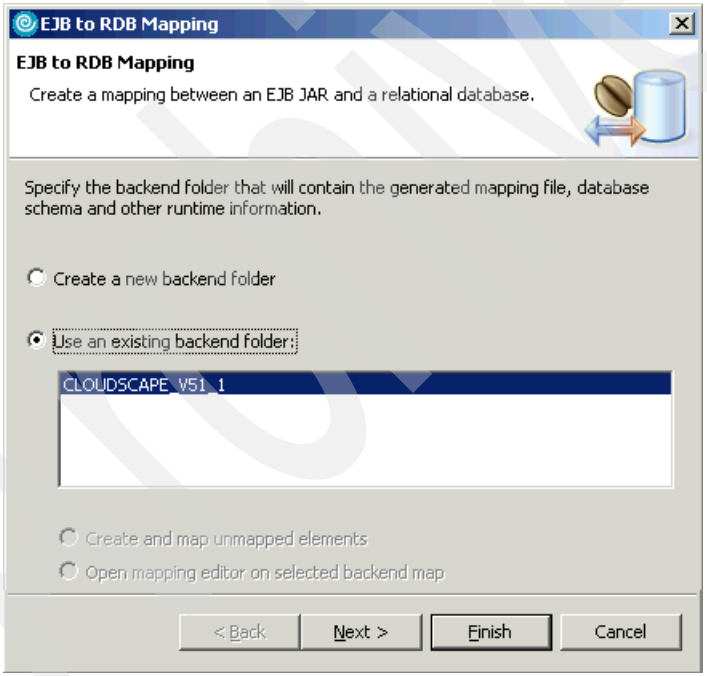


Figure 15-36 Using an existing backend folder

3. When the Create new EJB / RDB Mapping dialog appears, select **Meet In the Middle** and click **Next** (Figure 15-37 on page 896).

The Create new EJB / RDB Mapping page lets you select which kind of mapping you would like to create. The options displayed are the ones discussed earlier:

- Bottom-Up, which creates a new set of EJBs based on an existing database schema.
- Top-Down, which automatically generates a new database schema supporting the existing enterprise beans and the mapping between the existing object model and the new relational model. This option is greyed out since we chose to use an existing backend folder on the previous page.
- Meet-In-The-Middle, which generates a map between the existing object and relational models.

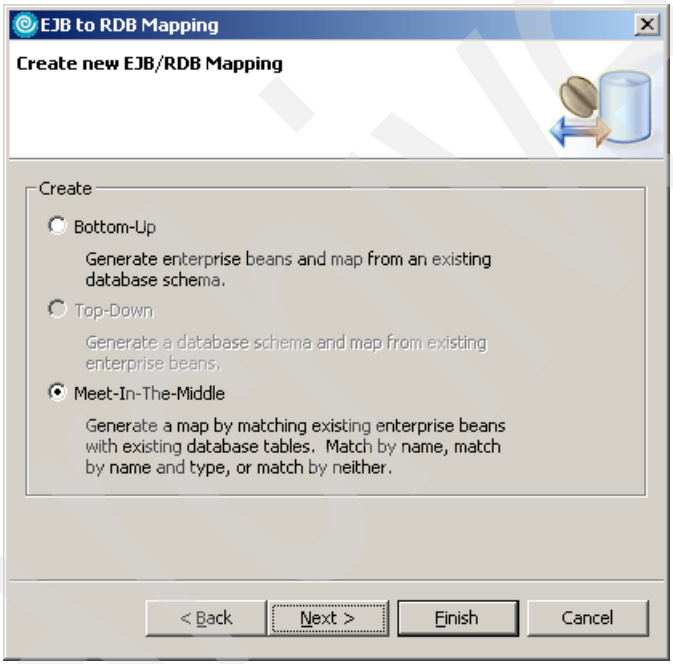


Figure 15-37 Creating a meet-in-the-middle EJB-to-RDB mapping

4. When the Select Meet-in-Middle Mapping options page appears, select **Match by Name (**as seen in Figure 15-39 on page 898) and click **Finish**.

   When selecting Match by Name, Rational Application Developer will attempt to match the entity beans to table names and entity bean field names to column names. When the EJBs and their fields are appropriately named, the amount of manual work will be minimized.

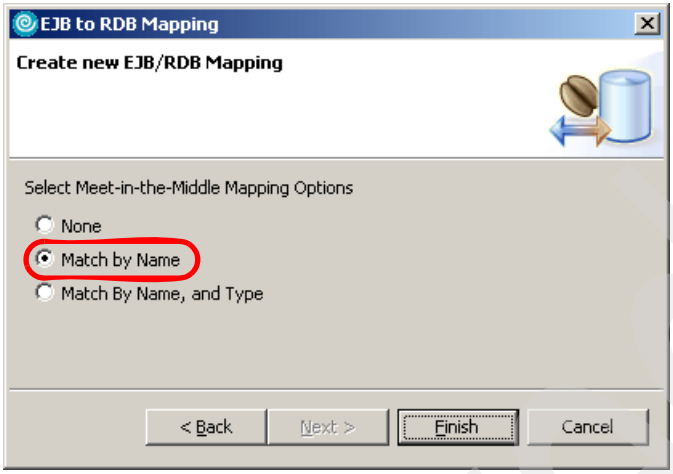


Figure 15-38 Select meet-in-the-middle option

Completing the EJB-to-RDB mapping

After completing the EJB to RDB wizard from the previous section, the Map.mapxmi editor will open. The display should look similar to Figure 15-39 on page 898.

As you can see, the wizard has already mapped the Customer and Account beans to the correct tables, and some of the fields are mapped to the correct columns. The mapped items carry a little triangle as an indicator, and they are listed in the bottom pane.

There are two possible methods of completing the mapping: Drag-and-drop, and using the context menus and choosing **Create Mapping**.

Archived

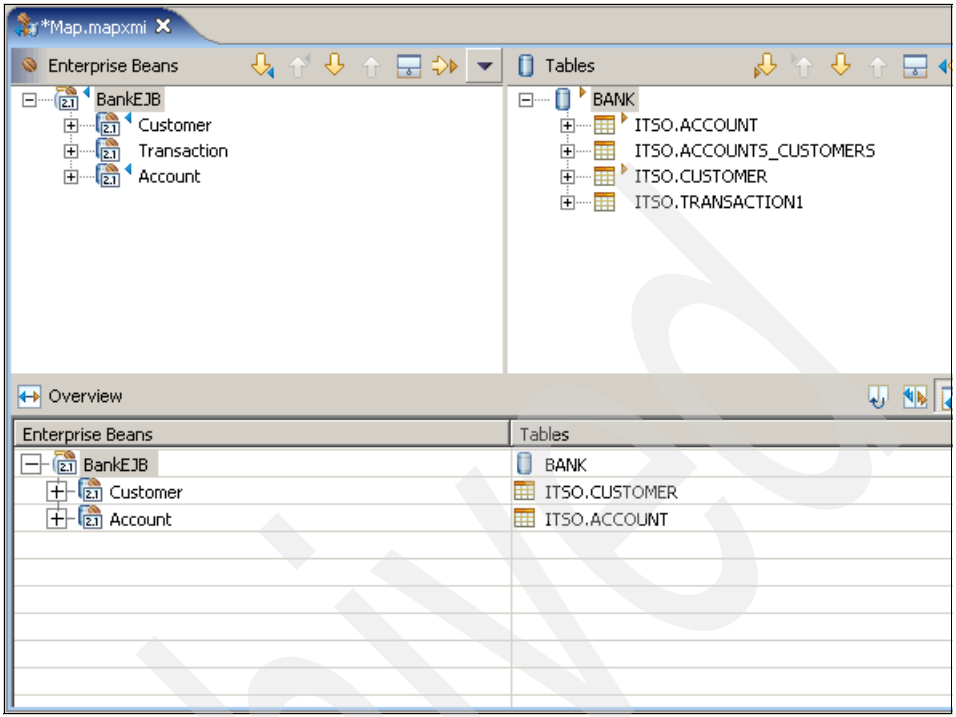


Figure 15-39 Generated object-relational mapping

To complete the EJB-to-RDB mapping using the drag-and-drop approach, do the following:

1. Map the EJBs to tables.

   A bean must be mapped to a table before you can map the attributes of the bean to the columns.

   a. Drag the **Transaction** EJB and drop it on the ITSO.TRANSACTION1 table.

   b. Expand the **Transaction** EJB.

   c. Drag the **Credit** EJB and drop it on the ITSO.TRANSACTION1 table.

   d. Drag the **Debit** EJB and drop it on the ITSO.TRANSACTION1 table.

Archived

Tip: The DISCRIM_TRANSACTION1 column of the ITSO.TRANSACTION1 table is the discriminator field. It is used to tell whether a transaction is a debit or a credit. The contents of this column correspond to the value of the TYPE_KEY constant field, which was added to the Credit and Debit beans in “Modify constructors for Transaction, Credit, and Debit” on page 882.

The DISCRIM_ prefix is reserved for use by the persistence manager.

2. Map the EJB attributes to the table columns.

   Some fields have not been matched automatically. We can perform the manual mapping by dragging and dropping attributes in the left pane to relational columns on the right, or vice-versa.

   a. Expand the **Transaction EJB** and the **TRANSACTION1** table.

   b. Drag the **id** attribute of the Transaction EJB to the ID column of the ITSO.TRANSACTION1 table.

   c. Drag the **amount** attribute of the Transaction EJB to the AMOUNT column of the ITSO.TRANSACTION1 table.

   d. Drag the **timestamp** attribute of the Transaction EJB to the TIMESTAMP1 column of the ITSO.TRANSACTION1 table.

3. Map the EJB container-managed relationships to the foreign key relationships between the database tables.

   a. Right-click the **ITSO.TRANSACTION1** table and select **Open Table Editor**.

   b. When the Table Editor opens, switch to the **Foreign Keys** tab.

   c. Click **Add Another**.

      Several edit fields, along with a Source Columns area that allows you to select columns for the foreign key, will appear in the right side of the page, as shown in Figure 15-40 on page 900.

      The value in the Foreign key name field is generated automatically and will have a different value for you.

   d. Enter `FK_TRANSACTION_ACCOUNT` in the Foreign key name field

   e. Select **ITSO.ACCOUNT** in the Target Table drop-down list.

   f. Select **TRANSACTIONSACCOUNTINVERSE_ID** on the source column and click **>** to add it to the list on the right.

   g. Save and close the Table Editor.

Archive

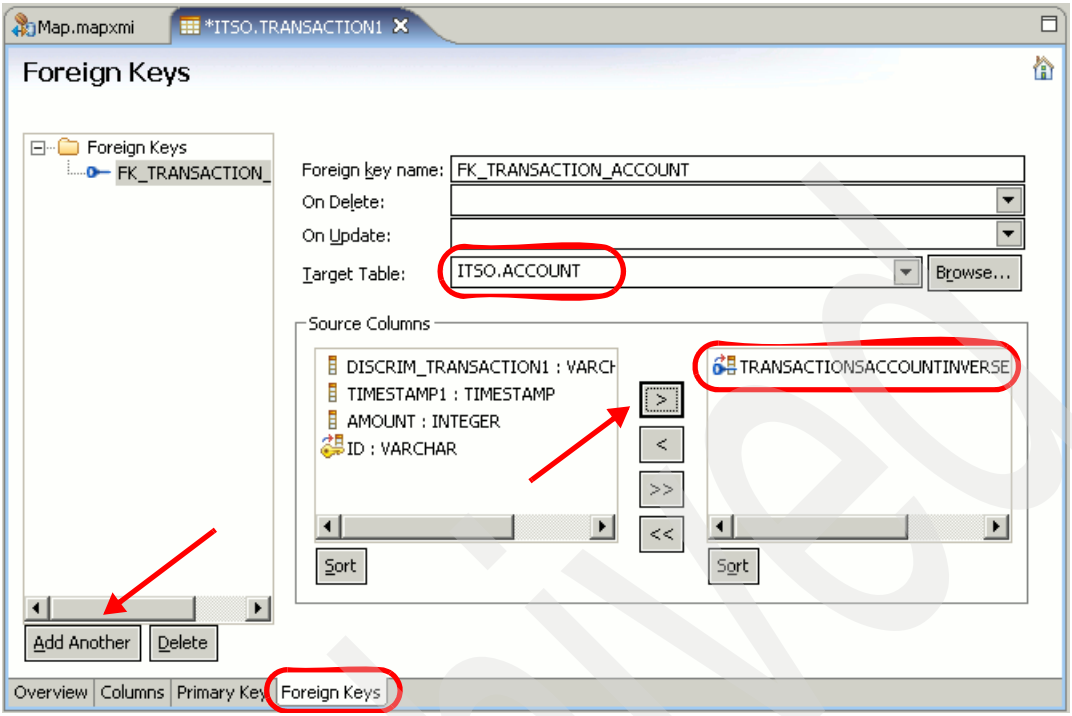


Figure 15-40 Creating a foreign key with the table editor

h. In the Map.mapxmi editor, drag **[0..1] account : Account** from the Transaction bean and drop it on FK_TRANSACTION_ACCOUNT.

i. Right-click the **ITSO.ACCOUNTS_CUSTOMERS** table and select **Open Table Editor**.

j. Create two foreign keys:

   i. Right-click the **ITSO.ACCOUNTS_CUSTOMERS** table and select **Open Table Editor**.

   ii. Switch to the **Foreign Keys** tab.

   iii. Click **Add Another**.

   iv. Enter `FK_CUSTOMER_ACCOUNT` in the Foreign key name field.

   v. Select **ITSO.ACCOUNT** in the Target Table drop-down list.

   vi. Select **ACCOUNTS_ID** on the source column and click **>** to add it to the list on the right.

   vii. Click **Add Another**.

   viii. Enter `FK_ACCOUNT_CUSTOMER` in the Foreign key name field.

ix. Select **ITSO.CUSTOMER** in the Target Table drop-down list.

x. Select **CUSTOMERS_SSN** on the source column and click **>** to add it to the list on the right.

xi. Save and close the Table Editor.

k. Expand the **Customer** EJB, and the **ITSO.ACCOUNT_CUSTOMERS** table. Drag **[0..*] accounts: Account** to FK_CUSTOMER_ACCOUNT.

l. Expand the **Account** EJB. Drag the **[0..*] customers: Customer** to FK_ACCOUNT_CUSTOMER.

The Outline view of the mapping editor summarizes our mapping activities (Figure 15-41).

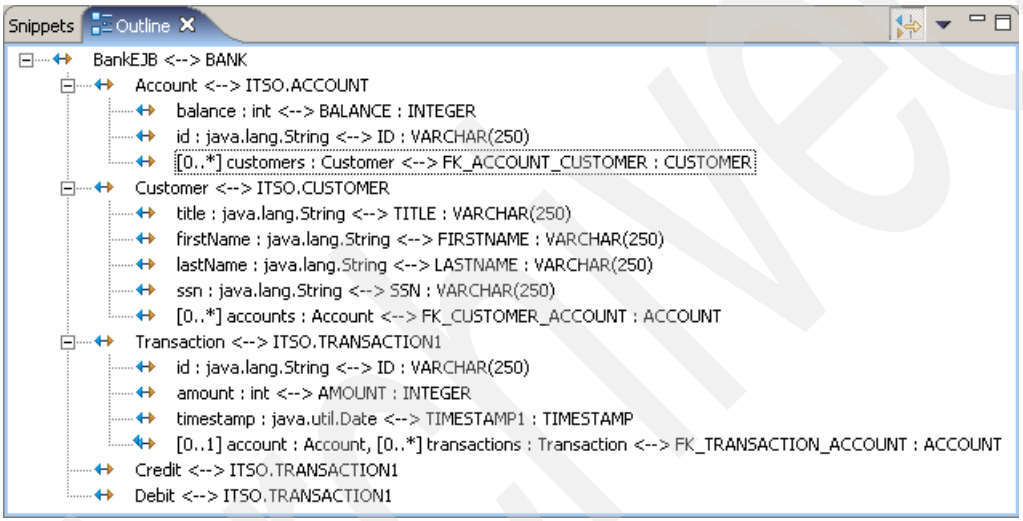


Figure 15-41 Outline view of the mapping editor

4. Save the mapping by pressing Ctrl+S and close the Map Editor.

15.4.6 Implement the session facade

The next EJB that we have to build is the session facade: The Bank stateless session bean (Figure 15-42).

Archived

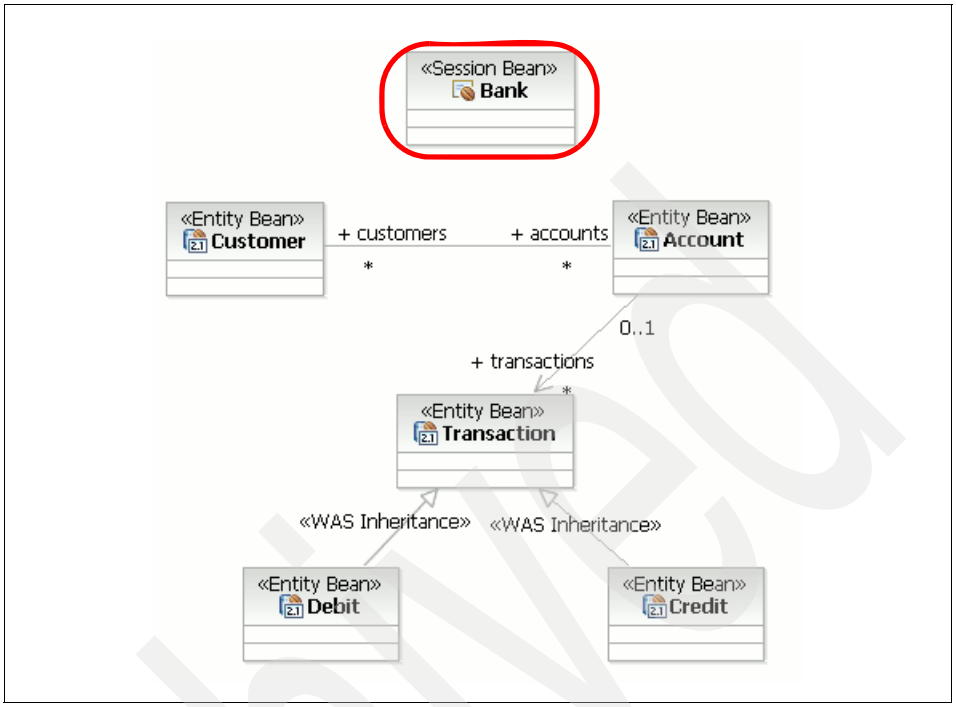


Figure 15-42 Business model session facade

Create the Bank session bean

To create the session bean, do the following:

1. In the Project Explorer view, expand **EJB Projects** → **BankEJB** → **diagrams**.
2. Double-click the **ejbs.dnx** diagram file to open it in the diagram editor.
3. In the Palette view, select **EJB** → **Session Bean** and click the canvas.
4. When the Create a new Session bean dialog appears, enter the following (as seen in Figure 15-43 on page 903), and then click **Next**:
   - EJB project: Select **BankEJB**.
   - Bean name: `Bank`
   - Source folder: `ejbModule`
   - Default package: `itso.bank.facade.ejb`

Archive

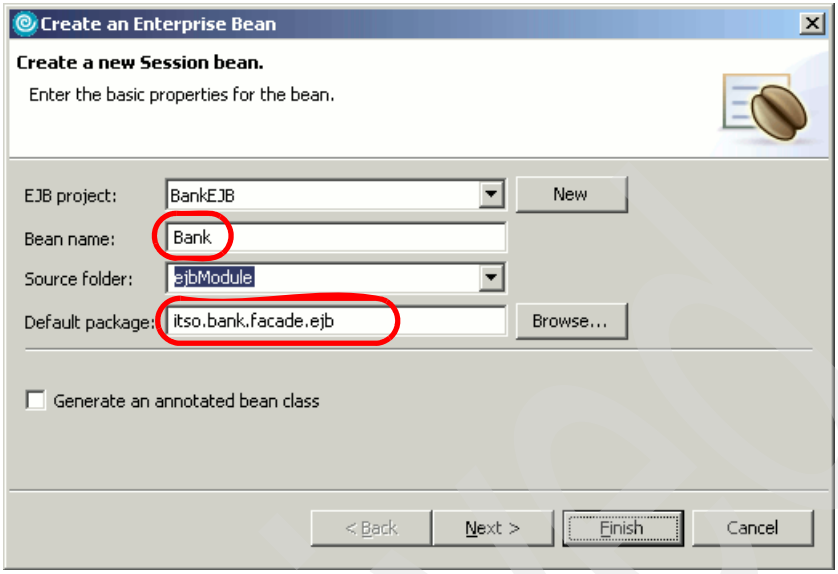


Figure 15-43 Creating a new session bean (page 1)

5. When the Enterprise Bean Details dialog appears, ensure that **Stateless** is selected in the Session type listbox and **Container** is selected in the Transaction type listbox, as shown in Figure 15-44 on page 904, and click **Finish**.

   Note that the default selection for the session bean is to create a remote client view instead of a local client view, as for the entity beans (see Figure 15-15 on page 862). This is because the environment knows that session beans are normally used to implement the model's facade and, as such, need remote interfaces as opposed to local ones.

6. Save the diagram to apply your changes, and close the UML Editor.

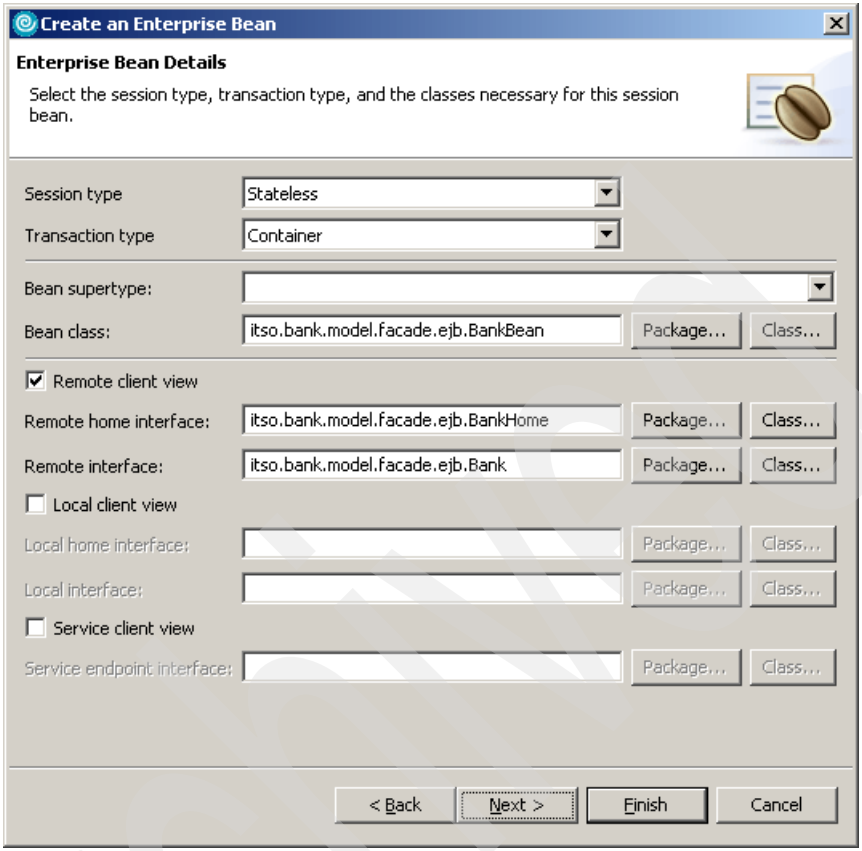


Figure 15-44 Creating a new session bean (page 2)

Edit the session bean

We will now add the facade methods that will be used by clients to perform banking operations.

Tip: The Java code for this section can be copied from the file c:\6449code\ejb\source\BankBean.jpage, included with the additional material for this book.

1. In the Project Explorer view of the J2EE Perspective, expand **EJB Projects** → **BankEJB** → **Deployment Descriptor: BankEJB** → **Session Beans** → **Bank**.
2. Double-click **BankBean** to open BankBean.java in the Java source editor.
3. Add the import statements from Example 15-10 to the file.

Example 15-10 Import statements for BankBean.java

```
import itso.bank.exception.*;
import itso.bank.model.*;
import itso.bank.model.ejb.*;

import java.util.Collection;
import java.util.Iterator;
import java.util.Set;
```

4. Complete the getCustomer method.

   a. Enter the method stub for the getCustomer method shown in Example 15-11 after the last method of the BankBean class.

Example 15-11 The getCustomer method stub

```
public Customer getCustomer(String ssn)
       throws UnknownCustomerException
{
}
```

   b. Place the cursor in the getCustomer method body.

   c. Switch to the Snippets view and double-click **EJB → Call an EJB "find" method**.

> **Tip:** The Snippets view is usually located in the same panel as the Outline view. If it is not visible, change to the Outline view and select **Window → Show View → Other**, and then select **Basic → Snippets** and click **OK**.

   d. When the Insert EJB Find wizard appears, click **New EJB Reference**.

   e. When the Add EJB Reference dialog appears, select **Enterprise Beans in the workspace**, expand and select **BankEJBEAR → BankEJB → Customer**, ensure that **Local** is selected in the Ref type drop-down, and click **Finish**.

> **Note:** The name defaults to ejb/*<BeanName>*, where *<BeanName>* is the name of the bean that you are adding a reference to. You can change this, but we choose to use the default.

Archived

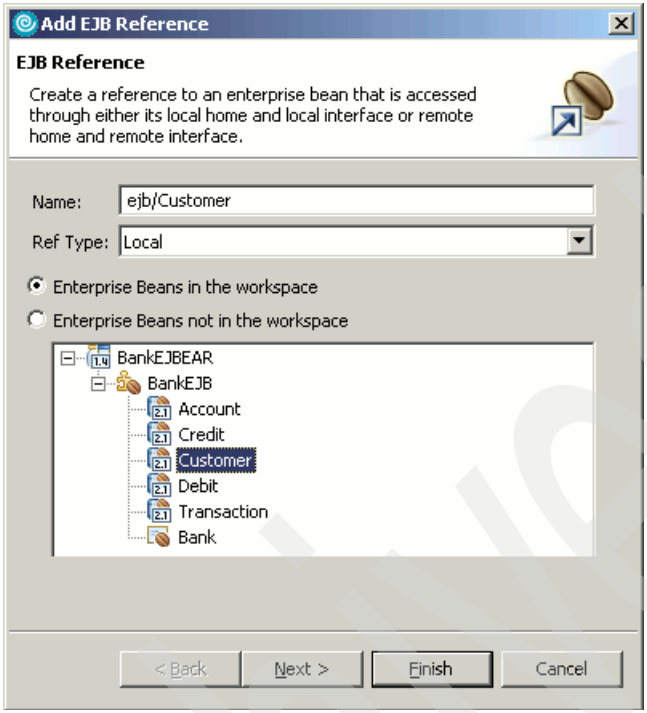


Figure 15-45 Adding the ejb/Customer EJB reference

f. When you return the Insert EJB Find wizard, click **Next**.

g. When the Select Method page appears, select **findByPrimaryKey(String primaryKey)** and click **Next**.

h. When the Enter Parameter Values page appears, enter `ssn` in the Value column of the first row and click **Finish**.

Several things will have happened when you are returned to the Java editor:

– A new EJB reference has been added to the Bank session bean, as shown in Figure 15-46 on page 908.

– A new line of code, shown in Example 15-12, has been added at the current cursor location.

Example 15-12 New code to find a Customer bean by its primary key

```
CustomerLocal aCustomerLocal = find_CustomerLocalHome_findByPrimaryKey(ssn);
```

– Example 15-12 invokes a new local method, find_CustomerLocalHome_findByPrimaryKey, which has been added after

Archived

the last method of the class. The code for this method is shown in Example 15-13.

Example 15-13 Generated find_CustomerLocalHome_findByPrimaryKey method

```
protected CustomerLocal find_CustomerLocalHome_findByPrimaryKey(
      String primaryKey) {
   CustomerLocalHome aCustomerLocalHome = (CustomerLocalHome) ServiceLocatorManager
         .getLocalHome(STATIC_CustomerLocalHome_REF_NAME,
               STATIC_CustomerLocalHome_CLASS);
   try {
      if (aCustomerLocalHome != null)
         return aCustomerLocalHome.findByPrimaryKey(primaryKey);
   } catch (javax.ejb.FinderException fe) {
      // TODO Auto-generated catch block
      fe.printStackTrace();
   }
   return null;
}
```

– Example 15-13 uses some constants, which have been added before the first method in the class. The definition for these is shown in Example 15-14.

Example 15-14 Constants to support the find_CustomerLocalHome_findByPrimaryKey method

```
private final static String STATIC_CustomerLocalHome_REF_NAME = "ejb/Customer";
private final static Class STATIC_CustomerLocalHome_CLASS = CustomerLocalHome.class;
```

– Additionally, the import statements, shown in Example 15-15, have been added at the top of the file.

Example 15-15 Imports added to the BankBean.java file

```
import itso.bank.model.ejb.CustomerLocal;
import com.ibm.etools.service.locator.ServiceLocatorManager;
import itso.bank.model.ejb.CustomerLocalHome;
```

– Finally, the JAR file serviceLocatorMgr.jar is added to the BankEJBEAR Enterprise Application, as shown in Figure 15-46 on page 908. This JAR, which contains the implementation of the ServiceLocatorManager class used in Example 15-13, is also added to the Java build path and the Java JAR Dependencies for the BankEJB project.

Archived

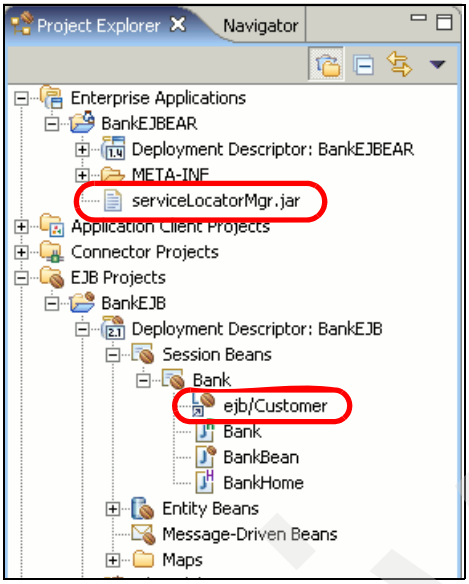


Figure 15-46 New JAR and EJB Reference to the Customer bean for the Bank bean

i. Modify the find_CustomerLocalHome_findByPrimaryKey method to match the code in Example 15-16. The difference, which consists of throwing an application-specific exception when the customer cannot be found, is highlighted in bold.

Example 15-16 Modified find_CustomerLocalHome_findByPrimaryKey method

```
protected CustomerLocal find_CustomerLocalHome_findByPrimaryKey(String primaryKey)
      throws UnknownCustomerException
{
   CustomerLocalHome aCustomerLocalHome = (CustomerLocalHome) ServiceLocatorManager
         .getLocalHome(STATIC_CustomerLocalHome_REF_NAME,
               STATIC_CustomerLocalHome_CLASS);
   try {
      if (aCustomerLocalHome != null)
         return aCustomerLocalHome.findByPrimaryKey(primaryKey);
   } catch (javax.ejb.FinderException fe) {
      // Customer not found
      throw new UnknownCustomerException(primaryKey);
   }
   return null;
}
```

j. Complete the getCustomer method, as shown in Example 15-17. The method uses the find_CustomerLocalHome_findByPrimaryKey method to

look up the customer in the database and then builds a data transfer object and returns that to the caller.

Example 15-17 Completed getCustomer method

```
public Customer getCustomer(String ssn)
      throws UnknownCustomerException
{
   CustomerLocal aCustomerLocal = find_CustomerLocalHome_findByPrimaryKey(ssn);
   Customer customer = new Customer();
   customer.setSsn(ssn);
   customer.setTitle(aCustomerLocal.getTitle());
   customer.setFirstName(aCustomerLocal.getFirstName());
   customer.setLastName(aCustomerLocal.getLastName());
   return customer;
}
```

5. Complete the getAccount method using a similar approach to the getCustomer method. The finished getAccount method is shown in Example 15-18.

Example 15-18 Completed getAccount method

```
public Account getAccount(String accountNumber)
      throws UnknownAccountException
{
   AccountLocal anAccountLocal = find_AccountLocalHome_findByPrimaryKey(accountNumber);

   Account account = new Account();
   account.setAccountNumber(accountNumber);
   account.setBalance(anAccountLocal.getBalance());

   return account;
}
```

Example 15-19 shows the find_AccountLocalHome_findByPrimaryKey method, modified to throw the application-specific exception UnknownAccountException.

Example 15-19 Completed find_AccountLocalHome_findByPrimaryKey method

```
protected AccountLocal find_AccountLocalHome_findByPrimaryKey(String primaryKey)
      throws UnknownAccountException
{
   AccountLocalHome anAccountLocalHome = (AccountLocalHome) ServiceLocatorManager
         .getLocalHome(STATIC_AccountLocalHome_REF_NAME,
               STATIC_AccountLocalHome_CLASS);
   try {
      if (anAccountLocalHome != null)
         return anAccountLocalHome.findByPrimaryKey(primaryKey);
```

Archived

```
   } catch (javax.ejb.FinderException fe) {
      // Account does not exist
      throw new UnknownAccountException(primaryKey);
   }
   return null;
}
```

6. Now that we have utility functions to look up accounts and customers, we can implement the methods for updating a customer (updateCustomer), retrieving the accounts for a customer (getAccounts), and the transactions for an account (getTransactions). These are shown in Example 15-20.

Example 15-20 Completed updateCustomer, getAccounts, and getTransactions methods

```
public void updateCustomer(String ssn, String title, String firstName, String lastName)
      throws UnknownCustomerException
{
   CustomerLocal aCustomerLocal = find_CustomerLocalHome_findByPrimaryKey(ssn);

   aCustomerLocal.setTitle(title);
   aCustomerLocal.setFirstName(firstName);
   aCustomerLocal.setLastName(lastName);
}

public Account[] getAccounts(String ssn)
      throws UnknownCustomerException
{
   CustomerLocal aCustomerLocal = find_CustomerLocalHome_findByPrimaryKey(ssn);

   Collection colAccounts = aCustomerLocal.getAccounts();
   Iterator itAccounts = colAccounts.iterator();
   Account[] arrAccount = new Account[colAccounts.size()];
   int i = 0;
   while (itAccounts.hasNext()) {
      AccountLocal accountLocal = (AccountLocal) itAccounts.next();
      Account account = new Account();
      account.setAccountNumber(accountLocal.getPrimaryKey().toString());
      account.setBalance(accountLocal.getBalance());
      arrAccount[i++] = account;
   }

   return arrAccount;
}

public Transaction[] getTransactions(String accountNumber)
      throws UnknownAccountException
{
   AccountLocal anAccountLocal = find_AccountLocalHome_findByPrimaryKey(accountNumber);
   Set setTransactions = anAccountLocal.getLog();
```

Proof

```
   Iterator itTransactions = setTransactions.iterator();
   Transaction[] arrTransaction = new Transaction[setTransactions.size()];
   int i = 0;
   while (itTransactions.hasNext()) {
      TransactionLocal transactionLocal = (TransactionLocal) itTransactions
            .next();
      Transaction transaction = null;
      if (transactionLocal instanceof CreditLocal) {
         transaction = new Credit();
      } else {
         transaction = new Debit();
      }
      transaction.setAccountNumber(accountNumber);
      transaction.setAmount(transactionLocal.getAmount());
      transaction.setTimestamp(transactionLocal.getTimestamp());
      arrTransaction[i++] = transaction;
   }

   return arrTransaction;
}
```

7. In order to implement the transaction methods, we need code to create a new transaction. The method is similar to creating code for an EJB lookup method:

   a. Create a method stub for the deposit method, as shown in Example 15-21, placing the cursor inside the method body.

Example 15-21 Method stub for the deposit method

```
public void deposit(String accountNumber, int amount)
      throws UnknownAccountException, ApplicationException
{
}
```

   b. In the Snippets view, double-click **EJB → Call an EJB "create" method**.

   c. When the Insert EJB Create wizard appears, click **New EJB Reference**.

   d. When the Add EJB Reference dialog appears, select **Enterprise Beans in the workspace**, expand and select **BankEJBEAR → BankEJB → Credit**, ensure that **Local** is selected in the Ref type drop-down, and click **Finish**.

   e. When you return the Insert EJB Create wizard, click **Finish**.

   Similar resources to the Insert EJB Find wizard are created for you:

   - An EJB reference named ejb/Credit is created for the Bank session bean.
   - A createCreditLocal method is created in the BankBean class.

Archived

- Constants STATIC_CreditLocalHome_REF_NAME and STATIC_CreditLocalHome_CLASS are added to the BankBean class.
- Import statements for itso.bank.model.ejb.CreditLocalHome and itso.bank.model.ejb.CreditLocal are added to BankBean.java.

f. Modify the generated createCreditLocal method to match the code shown in Example 15-22.

Example 15-22 Completed createCreditLocal method, customizations shown in bold

```
protected CreditLocal createCreditLocal(int amount)
      throws ApplicationException
{
   CreditLocalHome aCreditLocalHome = (CreditLocalHome) ServiceLocatorManager
         .getLocalHome(STATIC_CreditLocalHome_REF_NAME,
               STATIC_CreditLocalHome_CLASS);
   try {
      if (aCreditLocalHome != null)
         return aCreditLocalHome.create(amount);
   } catch (javax.ejb.CreateException ce) {
      throw new ApplicationException(
            "Unable to create credit transaction (amount="+amount+")", ce);
   }
   return null;
}
```

g. Complete the deposit method, as shown in Example 15-23.

Example 15-23 Completed deposit method

```
public void deposit(String accountNumber, int amount)
      throws UnknownAccountException, ApplicationException
{
   AccountLocal anAccountLocal = find_AccountLocalHome_findByPrimaryKey(accountNumber);

   CreditLocal aCreditLocal = createCreditLocal(amount);

   anAccountLocal.processTransaction(aCreditLocal);
}
```

8. Implement the withdraw method, using the same process as the deposit method, except that you use the Insert EJB Create wizard to generate code to create a Debit bean. The completed code for the withdraw, transfer, and createDebitLocal methods is shown in Example 15-24. Again, the modified parts of the createDebitLocal method are highlighted in bold.

Example 15-24 Completed withdraw, transfer, and createDebitLocal method

```
public void withdraw(String accountNumber, int amount)
```

```
      throws UnknownAccountException, InsufficientFundsException, ApplicationException
{
   AccountLocal anAccountLocal = find_AccountLocalHome_findByPrimaryKey(accountNumber);

   DebitLocal aDebitLocal = createDebitLocal(amount);

   anAccountLocal.processTransaction(aDebitLocal);
}

public void transfer(String debitAccountNumber, String creditAccountNumber, int amount)
      throws UnknownAccountException, InsufficientFundsException, ApplicationException
{
   withdraw(debitAccountNumber, amount);
   deposit(creditAccountNumber, amount);
}

protected DebitLocal createDebitLocal(int amount)
      throws ApplicationException
{
   DebitLocalHome aDebitLocalHome = (DebitLocalHome) ServiceLocatorManager
         .getLocalHome(STATIC_DebitLocalHome_REF_NAME,
               STATIC_DebitLocalHome_CLASS);
   try {
      if (aDebitLocalHome != null)
         return aDebitLocalHome.create(amount);
   } catch (javax.ejb.CreateException ce) {
      throw new ApplicationException(
            "Unable to create debit transaction (amount="+amount+")", ce);
   }
   return null;
}
```

9. Promote the facade methods to the remote interface, as follows:

   a. In the Outline view, highlight the following methods, as shown in Figure 15-47 on page 914:

      - getCustomer(String)
      - getAccount(String)
      - updateCustomer(String, String, String, String)
      - getAccounts(String)
      - getTransactions(String)
      - deposit(String, int)
      - withdraw(String, int)
      - transfer(String, String, int)

Archived

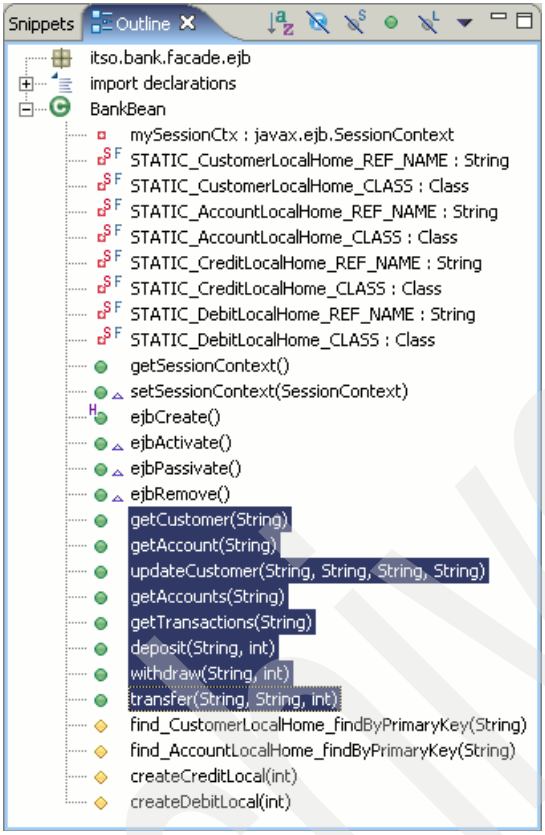


Figure 15-47 Promote Bank methods to the remote interface

b. Right-click one of the selected methods and select **Enterprise Bean** → **Promote to Remote Interface**.

> **Note:** We found that sometimes the R-shaped icons (which mean that the method exists in the remote interface) would not appear after promoting the methods. Closing and reopening the Java editor would fix this.

The Bank session bean is now complete. In the following sections we first test the EJBs using the Universal Test Client and then proceed to integrate the EJBs with the ITSO Web Bank application.

Archived

15.5 Testing EJB with the Universal Test Client

Before we integrate the EJB application with the Web application, imported in 15.3.5, “Import BankBasicWeb Project” on page 853, we will test the Bank session bean to see that it works as expected. We will use the enterprise application Universal Test Client (UTC), which is contained within Rational Application Developer.

In this section we describe some of the operations you can perform with the Universal Test Client. We will use the test client to find the Customer EJB home, find and create instances of the Customer bean, and send messages to those instances.

To test the EJBs, do the following:

1. From the Project Explorer of the J2EE Perspective, right-click **EJB Projects** → **BankEJB** and select **Run** → **Run on Server**.
2. When the Server Selection dialog appears, select **WebSphere Application Server v6.0** and click **Finish**.

   The server will be started and the EJB project will be deployed, if necessary. This may take a while.
3. When the Universal Test Client Welcome page appears, as shown in Figure 15-48, click **JNDI Explorer**.

> **Tip:** The default URL of the test client is http://localhost:9080/UTC/, so you can also access it through an external browser. If you want to access it from another machine, just substitute `localhost` with the hostname or IP address of the developer machine.

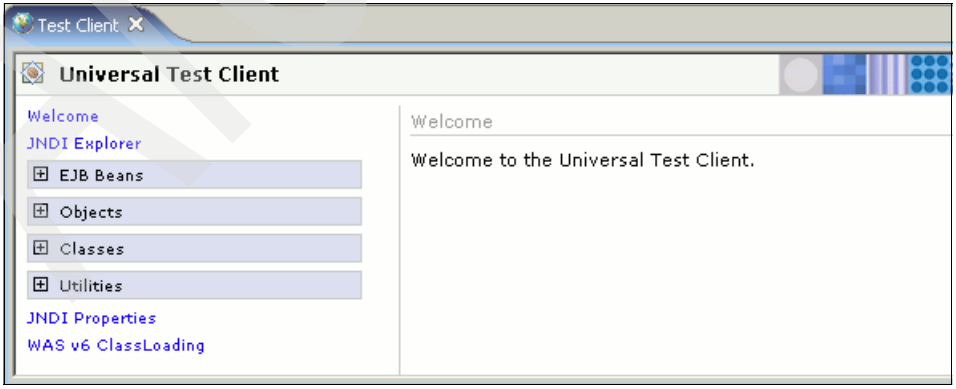


Figure 15-48 Universal Test Client home page

4. When the JNDI Explorer page appears, expand **[Local EJB Beans]** → **ejb** → **itso** → **bank** → **model** → **ejb**, **ejb** → **itso** → **bank** → **facade** → **ejb** and **jdbc**.

   The page should look similar to Figure 15-49. Notice that our five entity beans appear in the section for local EJBs, because they have no remote interface, while the Bank session bean appears in the remotely accessible scope. Also, the data source that we defined in 15.3.7, “Configure the data source” on page 856, appears in the JNDI explorer view. All EJBs appear as Web links.

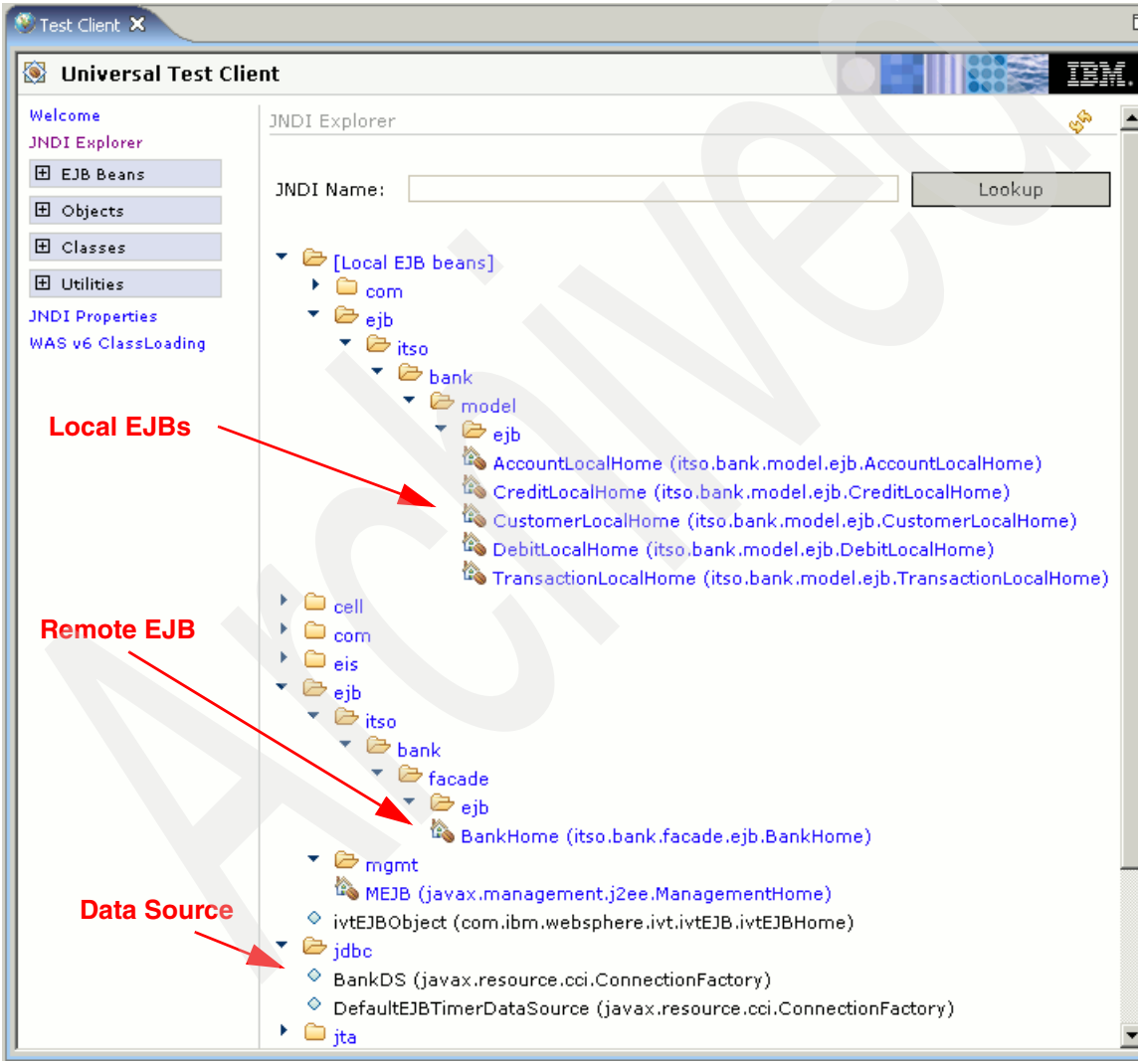


Figure 15-49 The UTC JNDI Explorer

Archive

Note: If the EJBs do not show up as links, we suggest that you unpublish the application from the server, restart the server, and start again by running the BankEJB project on the server.

5. Click **BankHome (itso.bank.facade.ejb.BankHome)**. The result is that the Bank bean is added to the EJB Beans list.
6. Expand **EJB Beans** → **BankHome** → **Bank** and click **Bank create()**.
7. The method signature for the create method is displayed. Click **Invoke**. The page should look similar to Figure 15-50.

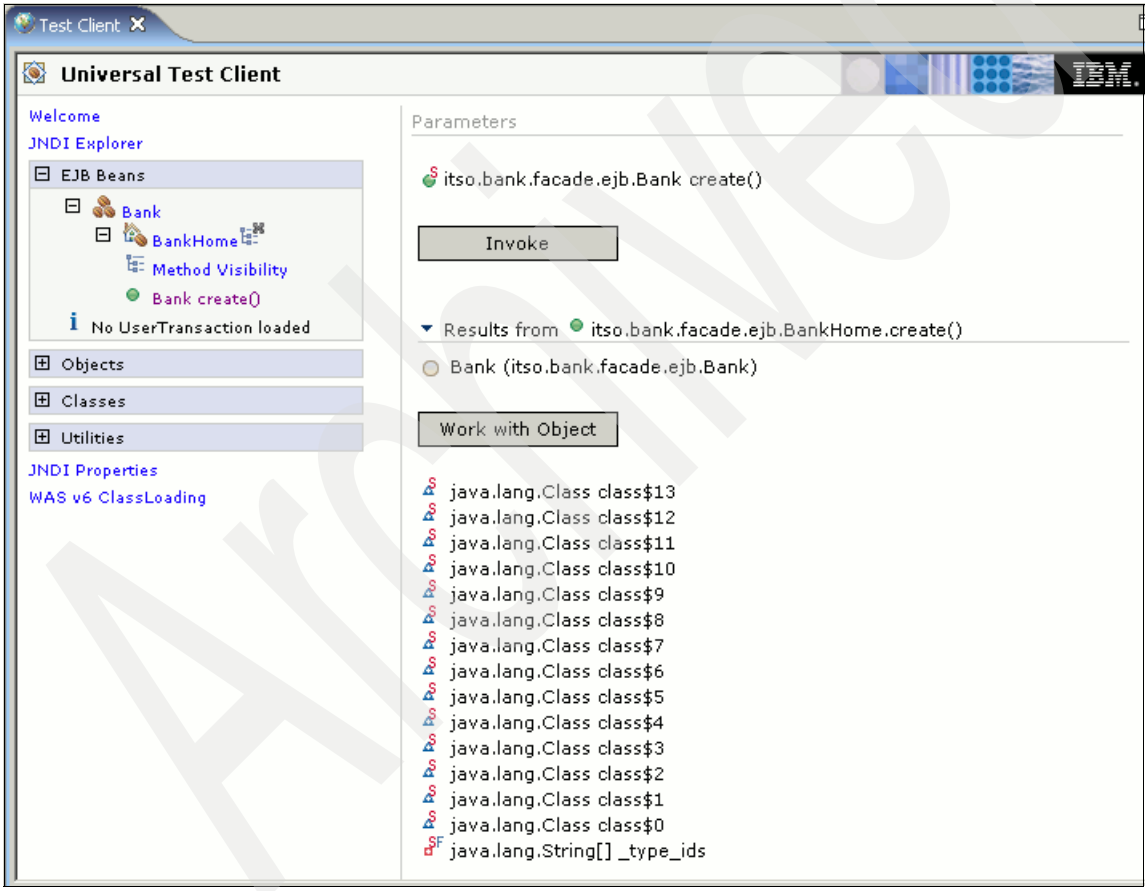


Figure 15-50 After invoking the create method for the Bank EJB

Archived

8. Click **Work with Object**. UTC will display the following message below the menu:

```
Recently added:
   Bank 1
```

   Additionally, the object `Bank 1` will appear below the existing BankHome object in the EJB Beans compartment of the menu.

9. Expand **EJB Beans** → **Bank 1** and click **Account[] getAccounts(String)**. The method signature for the getAccounts method will appear with a table, allowing you to specify the method parameters (in this case, only one String parameter).

10. Enter `111-11-1111` in the Value field and click **Invoke**. The resulting page should look similar to Figure 15-51. As you can see, three objects of the type itso.bank.model.Account were returned.

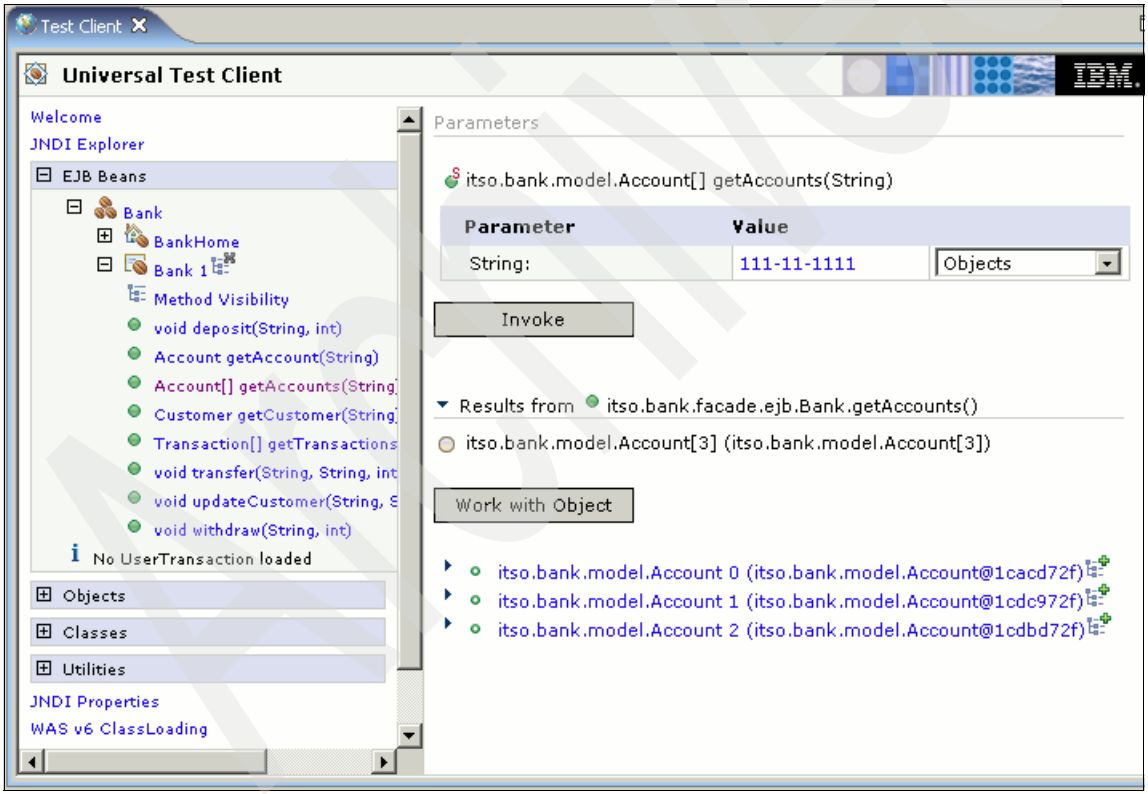


Figure 15-51 After invoking getAccounts for customer 111-11-1111

11. Click **Work with Object**. Notice how the object is added to the Objects compartment and not the EJB Beans.

The test client cannot inspect objects in arrays, so we will have to convert it to a java.util.List:

a. Expand **Utilities** and click **Object[] -> List**.

b. When the Object[] -> List page appears, select **Account[3]** and click **Convert**.

c. Click **Work with Contained Objects**.

Three Account objects will be added to the Objects compartment. Invoke their getAccountNumber and getBalance methods to inspect their contents.

You can play with the UTC to make sure all of your EJBs work. When you are done, close the browser window and stop the server in the Servers view.

15.6 Adapting the Web application

The Web application, developed in Chapter 11, "Develop Web applications using JSPs and servlets" on page 499, was imported and added to the BankEJBEAR project in 15.3.5, "Import BankBasicWeb Project" on page 853. We will now add client code to the Web application, in order to utilize the EJBs developed and tested in this chapter.

Tip: The Java code for the EJBBank class can be copied from the file c:\6449code\ejb\source\EJBBank.java, included with the additional material for this book.

1. In the Project Explorer, expand **Dynamic Web Projects** → **BankBasicWeb** → **diagrams**.
2. Double-click **model.dnx** to open the class diagram in the UML Editor.
3. The UML Editor will open.

   Notice that the model and exception classes and their relationship lines appear with crossed-out circles. This is the UML Editor's way of telling us that it cannot find the source for these resources.
4. Select the model and exception classes and press Delete to remove them from the diagram.
5. Select **itso.bank.facade** and press Delete to remove the package from the diagram.
6. In the Palette, select **Java** → **Class** and click the canvas.

Archived

7. When the New Java Class wizard appears, do the following and click **Finish**:
   - Name: `EJBBank`
   - Package: `itso.bank.facade`
   - Superclass: `Bank`
   - Select **Inherited abstract methods**.
8. When the new class has been added to the diagram, double-click it to open EJBBank.java in the Java source editor.
9. Add the code from Example 15-25 to the class.

Example 15-25 Method stub for getBankEJB

```
// add before the first method of the class
private itso.bank.facade.ejb.Bank bankEJB = null;

// add after the last method of the class
private itso.bank.facade.ejb.Bank getBankEJB()
      throws ApplicationException
{
   if (bankEJB == null) {
      // place the cursor on the next line

   }
   return bankEJB;
}
```

10. Place the cursor over the `if` statement and double-click **EJB → Call an EJB "create" method** in the Snippets view.
11. When the Insert EJB Create wizard appears, click **New EJB Reference**.
12. When the Add EJB Reference dialog appears, select **Enterprise Beans in the workspace**, expand and select **BankEJBEAR → BankEJB → Bank**, ensure that **Remote** is selected in the Ref type drop-down, and click **Finish**.
13. When you return the Insert EJB Create wizard, click **Finish**.
14. When the wizard is done adding configuration data and code, replace the getBankEJB and generated createBank methods with the code shown in Example 15-26.

Example 15-26 Completed getBankEJB method

```
private itso.bank.facade.ejb.Bank getBankEJB() throws ApplicationException {
   if (bankEJB == null) {
      BankHome aBankHome = (BankHome) ServiceLocatorManager.getRemoteHome(
            STATIC_BankHome_REF_NAME, STATIC_BankHome_CLASS);
      try {
         if (aBankHome != null)
            bankEJB = aBankHome.create();
```


Archived

```
      } catch (javax.ejb.CreateException ce) {
         throw new ApplicationException("Unable to create EJB: "+
                  STATIC_BankHome_REF_NAME, ce);
      } catch (RemoteException re) {
         throw new ApplicationException("Unable to create EJB: "+
                  STATIC_BankHome_REF_NAME, re);
      }
   }
   return bankEJB;
}
```

15. Remove the following import statement:

```
import itso.bank.facade.ejb.Bank;
```

This import statement was created by the Insert EJB Create wizard.

16. Replace the method stubs that were generated by the New Java Class wizard with the methods shown in Example 15-27. These methods will forward all calls to the Bank EJB.

Example 15-27 Completed EJBBank facade methods invoke the Bank EJB

```
public Account getAccount(String accountNumber)
      throws UnknownAccountException, ApplicationException {
   try {
      return getBankEJB().getAccount(accountNumber);
   } catch (RemoteException e) {
      throw new ApplicationException("Unable to retrieve account: "+
               accountNumber, e);
   }
}

public Account[] getAccounts(String customerNumber)
      throws UnknownCustomerException, ApplicationException {
   try {
      return getBankEJB().getAccounts(customerNumber);
   } catch (RemoteException e) {
      throw new ApplicationException("Unable to retrieve accounts for: "+
               customerNumber, e);
   }
}

public Customer getCustomer(String customerNumber)
      throws UnknownCustomerException, ApplicationException {
   try {
      return getBankEJB().getCustomer(customerNumber);
   } catch (RemoteException e) {
      throw new ApplicationException("Unable to retrieve accounts for: "+
               customerNumber, e);
```

Archived

```
    }
}

public Transaction[] getTransactions(String accountId)
       throws UnknownAccountException, ApplicationException {
    try {
       return getBankEJB().getTransactions(accountId);
    } catch (RemoteException e) {
       throw new ApplicationException("Unable to retrieve transactions for: "+
                 accountId, e);
    }
}

public void deposit(String accountId, int amount)
       throws UnknownAccountException, ApplicationException {
    try {
       getBankEJB().deposit(accountId, amount);
    } catch (RemoteException e) {
       throw new ApplicationException("Unable to deposit "+amount+
              " to "+accountId, e);
    }
}

public void withdraw(String accountId, int amount)
       throws UnknownAccountException, InsufficientFundsException,
       ApplicationException {
    try {
       getBankEJB().withdraw(accountId, amount);
    } catch (RemoteException e) {
       throw new ApplicationException("Unable to withdraw "+amount+" from
"+accountId, e);
    }
}

public void transfer(String debitAccountNumber, String creditAccountNumber,
       int amount)
       throws UnknownAccountException, InsufficientFundsException,
ApplicationException {
    try {
       getBankEJB().transfer(debitAccountNumber, creditAccountNumber, amount);
    } catch (RemoteException e) {
       throw new ApplicationException("Unable to transfer "+amount+
              " from "+debitAccountNumber+" to "+creditAccountNumber, e);
    }
}

public void updateCustomer(String ssn, String title,
       String firstName, String lastName)
       throws UnknownCustomerException, ApplicationException {
```

```
   try {
      getBankEJB().updateCustomer(ssn, title, firstName, lastName);
   } catch (RemoteException e) {
      throw new ApplicationException("Unable to update customer "+ssn, e);
   }
}
```

17. Now that we have completed the EJBBank facade, we need to update the abstract class itso.bank.facade.Bank to return an instance to the EJBBank class when its getBank method is called. Modify the getBank method of the Bank class as shown in Example 15-28.

Example 15-28 New getBank method for the Bank facade

```
public static Bank getBank() throws ApplicationException {
   if (singleton == null) {
      // no singleton has been created yet - create one
      singleton = new EJBBank();
   }

   return singleton;
}
```

Tip: The Java code for the getBank method can be copied from the file c:\6449code\ejb\source\Bank.jpage, included with the additional material for this book.

18. Restart the server.

19. Follow the instructions in 11.7, “Test the application” on page 610, to test the Web application.

Persistence: In the first implementation, every time you started the Web application you got the same data because it was created from memory. Now we are running with EJBs accessing the underlying BANK database. All the updates are persistent. The updated balance is stored in the database and the transaction records accumulate for each account.

When testing, try to restart the server to see the persistence.

Archived

Archived

16

Develop J2EE application clients

This chapter provides an introduction to J2EE application clients and the facilities supplied by the J2EE application client container. In addition, we highlight the features provided by Rational Application Developer for developing and testing J2EE application clients.

The chapter is organized into the following sections:

- Introduction to J2EE application clients
- Overview of the sample application
- Preparing for the sample application
- Develop the J2EE application client
- Test the J2EE application client
- Package the application client project

© Copyright IBM Corp. 2005. All rights reserved.

16.1 Introduction to J2EE application clients

A J2EE application server is capable of making several different types of resources available for remote access, such as:

- Enterprise JavaBeans (EJBs)
- JDBC Data Sources
- Java Message Service (JMS) resources (Queues and Topics)
- Java Naming and Directory Interface (JNDI) services

These resources are most often accessed from a component that is running within the J2EE application server itself, such as an EJB, servlet, or JSP. However, these resources can also be used from a stand-alone Java application running in its own Java Virtual Machine (JVM), possibly on a different computer from the server. This is known as a J2EE application client. Figure 16-1 depicts the resource access scenarios described.

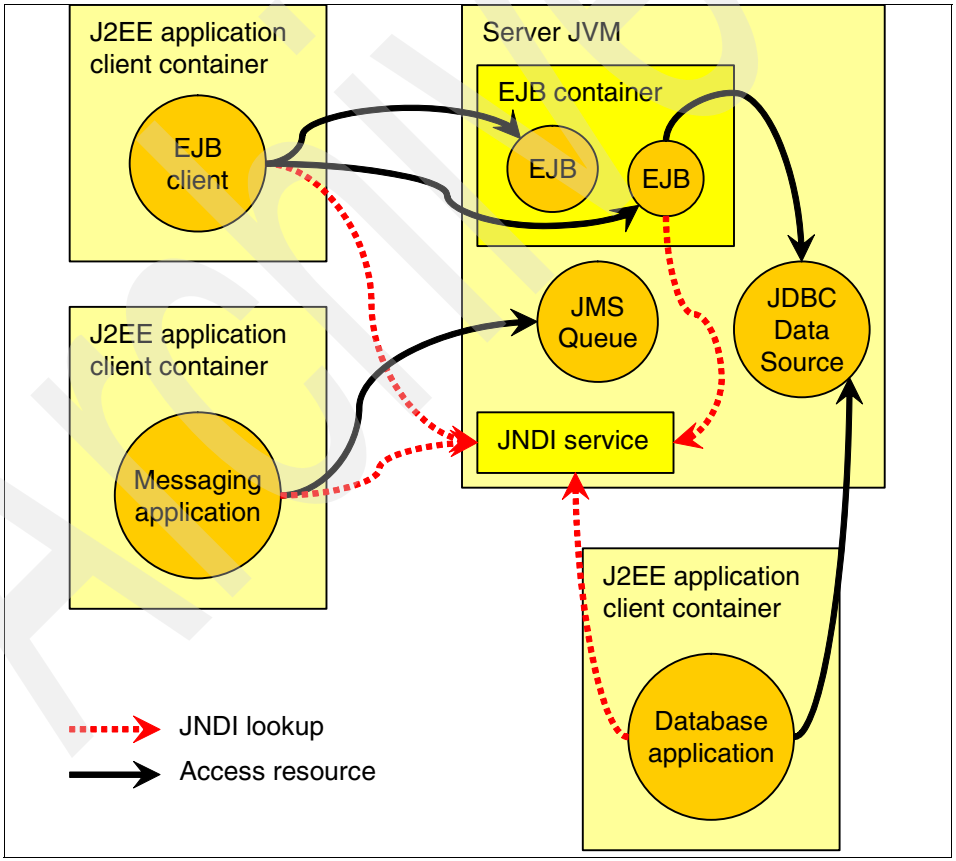


Figure 16-1 Java applications using J2EE server resources

Archived

Note: The clients shown in Figure 16-1 on page 926 may conceptually be running on the same physical node, or even in the same JVM as the application server. The focus in this chapter, however, is clients running in distributed environments. During the course of this chapter, we will develop an EJB client, invoking the EJBs from Chapter 15, “Develop Web applications using EJBs” on page 827, to provide a simple ITSO Bank client application.

Since a regular JVM does not support accessing such application server resources, additional setup for the runtime environment is required for a J2EE application. There are two methods to achieve this:

- Add the required packages to the Java Runtime Environment manually.
- Package the application according to the J2EE application client specification and execute the application in a J2EE application client container.

In this chapter, we focus on the second of these options. In addition to providing the correct runtime resources for Java applications wishing to access J2EE server resources, the J2EE application client container provides additional features, such as mapping references to JNDI names and integration with server security features.

Note: For more detailed information on J2EE application clients, refer to the J2EE specification (Chapter 9, “Application Clients”) found at:

`http://java.sun.com/j2ee/`

The specification defines the runtime requirements for J2EE application clients, particularly in the areas of security, transactions, and naming (JNDI). It also specifies which programming APIs are required in addition to those provided by Java 2 Standard Edition (J2SE):

- Enterprise JavaBeans (EJB) 2.1 (client-side APIs)
- Java Message Service (JMS) 1.1
- JavaMail 1.3
- JavaBeans Activation Framework (JAF) 1.0
- Java APIs for XML Processing (JAXP) 1.2
- Web Services 1.1
- Java API for XML-based Remote Procedure Call (JAX-RPC) 1.1
- SOAP with Attachments API for Java (SAAJ) 1.2
- Java API for XML Registries (JAXR) 1.0
- J2EE Management 1.0
- Java Management Extensions (JMX) 1.2

In addition, the J2EE specification describes the packaging format to be used for J2EE application clients (based on the JAR file format) and the contents of the the deployment descriptor for the J2EE application client.

IBM WebSphere Application Server V6.0 includes a J2EE application client container and a facility for launching J2EE application clients. The J2EE application client container, known as *Application Client for WebSphere Application Server*, can be installed separately from the WebSphere Application Server installation CDs, or downloaded from developerWorks, and runs a completely separate JVM on the client machine. When the JVM starts it loads the necessary runtime support classes to make it possible to communicate with WebSphere Application Server and to support J2EE application clients that will use server-side resources. Refer to the WebSphere Application Server Information Center for more information about installing and using the Application Client for WebSphere Application Server.

Note: Although the J2EE specification describes the JAR format as the packaging format for J2EE application clients, the Application Client for WebSphere Application Server expects the application to be packaged as a JAR inside an enterprise application archive (EAR). The Application Client for WebSphere Application Server does not support execution of a standalone J2EE client JAR.

IBM Rational Application Developer V6.0 includes tooling to assist with developing and configuring J2EE application clients and a test facility that allows J2EE application clients to be executed in an appropriate container. The focus of this chapter is on the Rational Application Developer tooling for J2EE application clients, so we will be looking only at this facility.

16.2 Overview of the sample application

The application that will be developed in the course of this chapter is a very simple client application. It will use the EJB application that was developed in Chapter 15, “Develop Web applications using EJBs” on page 827, to look up the customer information and account overview from a specified customer SSN.

The application will use a graphical user interface, implemented with Swing components, as shown in Figure 16-2, which displays the details for the customer with SSN 111-11-1111.

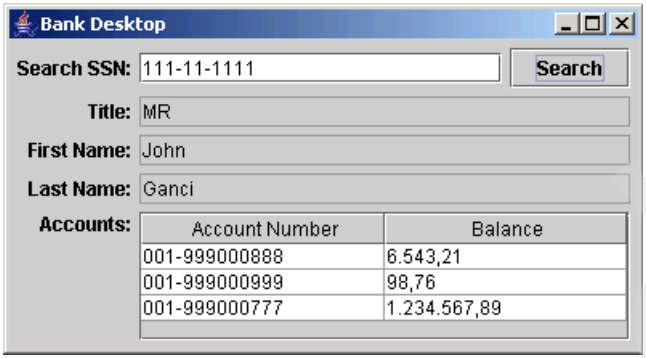


Figure 16-2 Interface for the sample application client

Referring to Figure 16-1 on page 926, the sample J2EE application client is an EJB client that uses the services of the Bank EJB, created in Chapter 15, “Develop Web applications using EJBs” on page 827, to access account information for customers in the ITSO Bank.

Figure 16-3 on page 930 shows a class diagram of the finished sample application. The classes on the right-hand side of the class diagram are classes from the EJB enterprise application, while the three left-hand classes are part of the application client. As the class diagram outlines, the application client controller class, BankDesktopController, uses the Bank EJB to retrieve Customer and Account object instances, representing the customer and associated account(s) that is being retrieved from the server database.

Archived

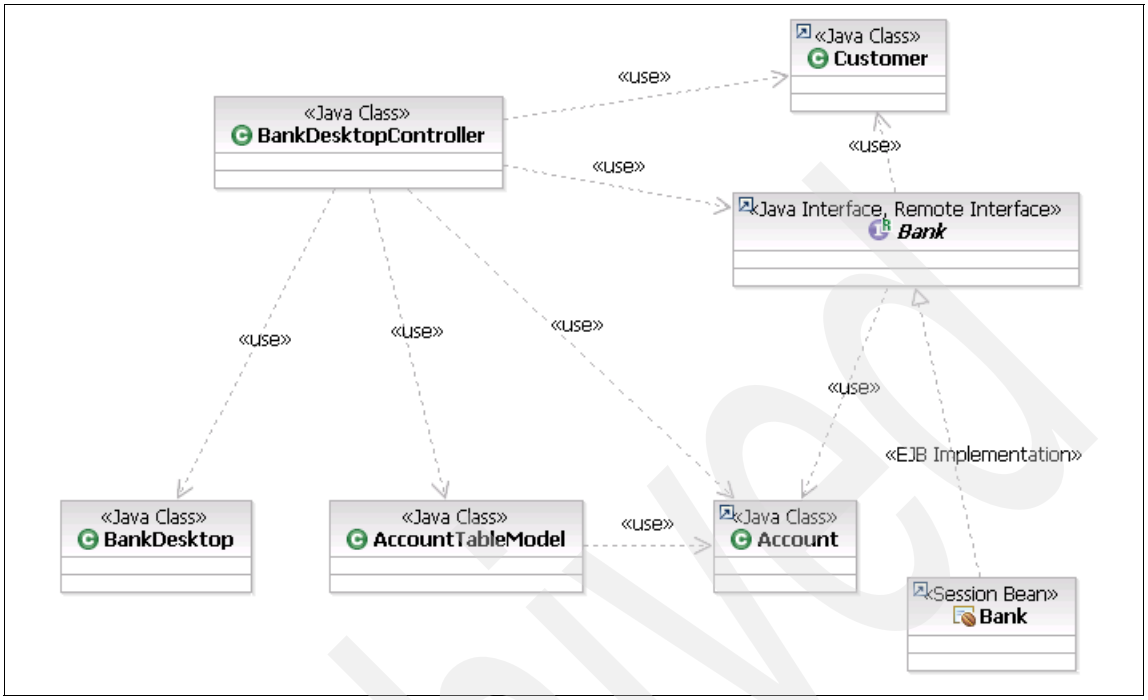


Figure 16-3 Class diagram for the ITSO Bank J2EE application client

To illustrate the deployment units for the finished application, Figure 16-4 on page 931 shows the Project Explorer view with all projects and the deployment descriptors for the two enterprise applications expanded. When deployed to a working environment, the BankAppClientEAR is deployed on the client node, while the BankEJBEAR is deployed on an application server.

As Figure 16-4 on page 931 shows, the two enterprise applications share the two utility JARs BankEJBClient.jar and serviceLocatorMgr.jar. The former is the client code necessary to use the EJBs from the BankEJB project, deployed within the BankEJBEAR enterprise application, while the latter contains utility classes for looking up and instantiating EJBs from a client application.

The projects BankAppClientEAR and BankAppClient are created during the course of this chapter, while the remaining projects were implemented in Chapter 15, “Develop Web applications using EJBs” on page 827.

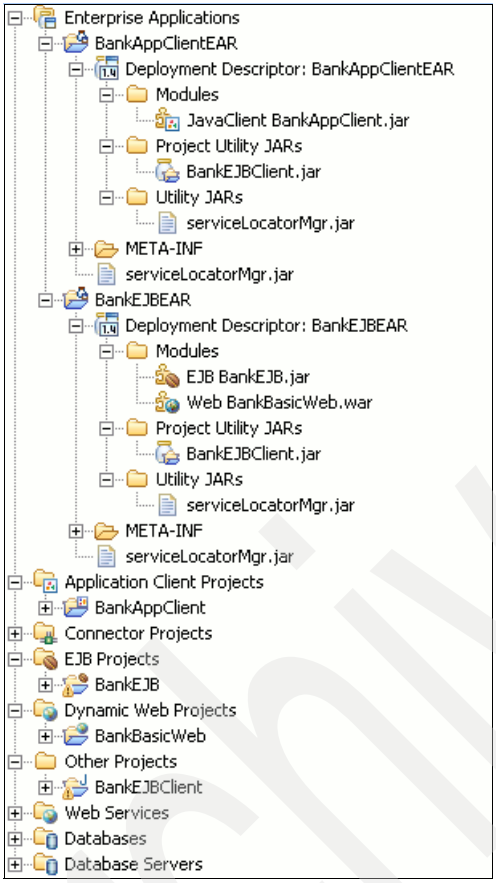


Figure 16-4 Project Explorer for the finished application client

16.3 Preparing for the sample application

Prior to working on the sample for this chapter, we need to set up the database for the sample application, import the EJB bank projects, and ensure that everything is working.

The preparation tasks are as follows:

- Import the base enterprise application sample.
- Set up the sample database.
- Configure the data source.
- Test the imported code.

Archived

16.3.1 Import the base enterprise application sample

To import the base enterprise application sample that we will use as a starting point for this chapter, do the following:

1. From the Project Explorer view in the J2EE perspective, select **File** → **Import...**.
2. When the Import dialog appears, select **Project Interchange** and click **Next**.
3. When the Import Project Interchange Contents dialog appears, enter `c:\6449code\ejb\BankEJB.zip` in the From zip file field, click **Select All**, and then **Finish**.

After the Import wizard has completed the import, the projects shown in Figure 16-5 should appear in the workspace.

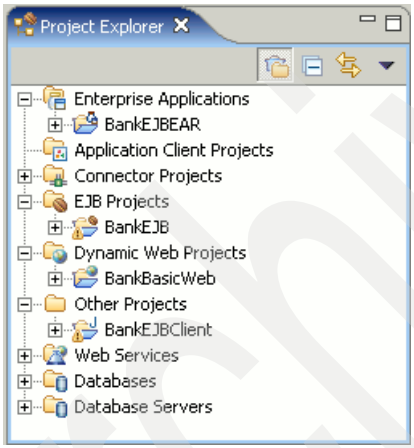


Figure 16-5 Projects imported from the BankEJB sample application

Note: You may notice that a number of warnings exist for the BankEJB and BankEJBClient projects. These are all related to unused import statements in code generated by Rational Application Developer, and can be safely ignored. To remove these warnings from the Problems view, you may follow the instructions in 15.3.4, “Configure the EJB projects” on page 849, for ignoring such warning messages.

The projects shown in Figure 16-5 are described as follows:

- BankEJBEAR: This is the deployable enterprise application, which functions as a container for the remaining projects. This enterprise application must be executed on an application server.

Archived

- BankEJB: This is the project, containing the EJBs, that makes up the business logic of the ITSO Bank. The Bank session bean acts as a facade for the EJB application.

  This project is packaged inside BankEJBEAR when exported and deployed on an application server.

- BankBasicWeb: This is a Web application that uses the EJBs to implement a Web interface for the ITSO Bank. During the course of this chapter, we develop a stand-alone J2EE client application with the same functionality as the ListAccounts servlet of this Web application.

  This project is packaged inside BankEJBEAR when exported and deployed on an application server.

- BankEJBClient: This is the client interface for the EJB application. This project is packaged with any application that will need to access the EJBs in the BankEJB project, including the BankBasicWeb project and the client application that will be developed during the course of this chapter.

16.3.2 Set up the sample database

If the Cloudscape database has already been configured for another sample in this book, you can skip the next step and go straight to 16.3.3, “Configure the data source” on page 934.

This section provides instructions for deploying the BANK sample database and populating the database with sample data. For simplicity we will use the built-in Cloudscape database.

To create the database, create the connection, and create and populate the tables for the BANK sample from within Rational Application Developer, do the following:

1. Create the database and connection to the Cloudscape BANK database from within Rational Application Developer.

   For details refer to “Create a database connection” on page 347.

2. Create the BANK database tables from within Rational Application Developer.

   For details refer to “Create database tables via Rational Application Developer” on page 350.

3. Populate the BANK database tables with sample data from within Rational Application Developer.

   For details refer to “Populate the tables within Rational Application Developer” on page 352.

Archived

16.3.3 Configure the data source

There are a couple of methods that can be used to configure the data source, including using the WebSphere Administrative Console or using the WebSphere Enhanced EAR, which stores the configuration in the deployment descriptor and is deployed with the application.

This section describes how to configure the data source using the WebSphere Enhanced EAR capabilities. The enhanced EAR is configured in the Deployment tab of the EAR deployment descriptor.

Access the deployment descriptor

To access the deployment descriptor where the enhanced EAR settings are defined, do the following:

1. Open the J2EE Perspective Project Explorer view.
2. Expand **Enterprise Applications** → **BankEJBEAR**.
3. Double-click **Deployment Descriptor: BankEJBEAR** to open the file in the Deployment Descriptor Editor.
4. Click the **Deployment** tab.

Note: For JAAS Authentication, when using Cloudscape, the configuration of the user ID and password for the JAAS Authentication is not needed.

When using DB2 Universal Database or other database types that require a user ID and password, you will need to configure the JAAS Authentication.

Configure a new JDBC provider

To configure a new JDBC provider using the enhanced EAR capability in the deployment descriptor, do the following:

1. From the Deployment tab of the Application Deployment Descriptor, click **Add** under the JDBC provider list.
2. When the Create a JDBC Provider dialog appears, select **Cloudscape** as the Database type, select **Cloudscape JDBC Provider** as the JDBC provider type, and then click **Next**.

Archived

Note: The JDBC provider type list for Cloudscape will contain two entries:

- Cloudscape JDBC Provider
- Cloudscape JDBC Provider (XA)

Since we will not need support for two-phase commits, we choose to use the non-XA JDBC provider for Cloudscape.

3. Enter `Cloudscape JDBC Provider - BankEJB` in the Name field and then click **Finish**.

Configure the data source

To configure a new data source using the enhanced EAR capability in the deployment descriptor, do the following:

1. From the Deployment tab of the Application Deployment Descriptor, select the JDBC provider created in the previous step.
2. Click **Add** next to data source.
3. When the Create a Data Source dialog appears, select **Cloudscape JDBC Provider** under the JDBC provider, select **Version 5.0 data source**, and then click **Next**.
4. When the Create a Data Source dialog appears, enter the following and then click **Finish**:
   - Name: `BankDS`
   - JNDI name: `jdbc/BankDS`
   - Description: `Bank Data Source`

Configure the databaseName property

To configure the databaseName in the new data source using the enhanced EAR capability in the deployment descriptor to define the location of the database for your environment, do the following:

1. Select the data source created in the previous section.
2. Select the **databaseName** property under the Resource properties.
3. Click **Edit** next to Resource properties to change the value for the databaseName.
4. When the Edit a resource property dialog appears, enter `c:\databases\BANK` in the Value field and then click **OK**.

Archived

Important: The Edit a resource property dialog allows you to edit the entire resource property, including the name. Ensure that you only change the value of the databaseName property, not the name.

In our example, c:\databases\BANK is the database created for our sample application in 15.3.6, “Set up the sample database” on page 854.

5. Save the Application Deployment Descriptor.
6. Restart the test server for the changes to the deployment descriptor to take effect.

Set up the default CMP data source

Several data sources can be defined for an enterprise application. In order for the EJB container to be able to determine which data source should be used, we must configure the BankEJBEAR project to point to the newly created data source as follows:

1. Open the J2EE Perspective Project Explorer view.
2. Expand **EJB Projects** → **BankEJB**.
3. Double-click **Deployment Descriptor: BankEJB** to open the file in the Deployment Descriptor Editor.
4. From the Overview tab, scroll down to the JNDI - CMP Connection Factory Binding section.
5. Enter `jdbc/BankDS` in the JNDI name field.
6. Press Ctrl+S followed by Ctrl+F4 to save and close the deployment descriptor.

16.3.4 Test the imported code

Before continuing with the sample application, we suggest that you test the imported code. Follow the instructions in 15.5, “Testing EJB with the Universal Test Client” on page 915, to test the EJBs.

16.4 Develop the J2EE application client

We will now use Rational Application Developer to create a project containing a J2EE application client. This application client will be associated with its own enterprise application.

Archived

Note: While it is possible to use the new client application with the existing BankEJBEAR enterprise application, this is not the recommended approach.

The reason for this is that the enterprise application that contains the EJBs and other server resources will typically contain information that should not be distributed to the clients, such as passwords or proprietary business logic.

To finish the J2EE application client sample you need to complete the following tasks:

- Create the J2EE application client projects.
- Configure the J2EE application client projects.
- Import the graphical user interface and control classes.
- Create the BankDesktopController class.
- Complete the BankDesktopController class.
- Register the BankDesktopController class as the Main class.

16.4.1 Create the J2EE application client projects

To create the J2EE application projects, do the following:

1. In the Project Explorer view of the J2EE perspective, right-click **Application Client Projects** and select **New** → **Application Client Project** from the menu.
2. When the New Application Client Project wizard appears, enter `BankAppClient` in the Name field and click **Show Advanced**.
3. From the Show Advanced options, do the following (as seen in Figure 16-6), and then click **Finish**:
   - Ensure that the J2EE version is set to 1.4.
   - Check **Add module to an EAR project**.
   - Select **BankAppClientEAR**.
   - Uncheck **Create a default Main class**.

Note: After the wizard has created the new projects, you will see the following error in the Problems view:

```
IWAE0035E The Main-Class attribute must be defined in the
application client module.
```

This is because we unchecked “Create a default Main Class” in the New Application Client Project wizard. We will create a main class, and thus resolve this problem in a subsequent step.

Archived

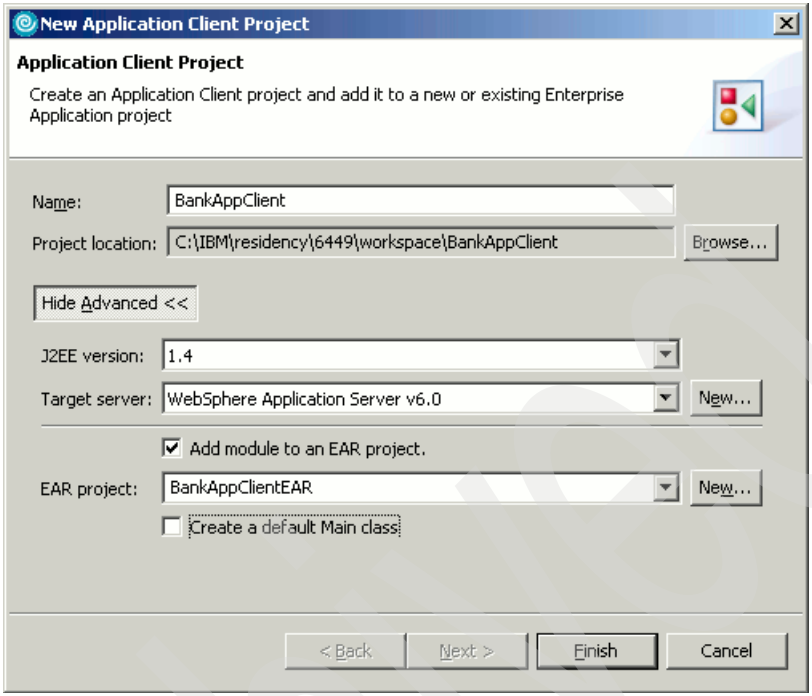


Figure 16-6 New Application Client Project wizard

When the wizard is complete, the following projects should have been created in your workspace:

- BankAppClientEAR: This is an enterprise application project that acts as a container for the code to be deployed on the application client node.
- BankAppClient: This project will contain the actual code for the ITSO Bank application client.

16.4.2 Configure the J2EE application client projects

As mentioned in 16.2, “Overview of the sample application” on page 928, the application client projects reference the BankEJBClient project that was imported in 16.3.1, “Import the base enterprise application sample” on page 932. In this section, we configure this dependency by completing the following tasks:

- Add BankEJBClient as a Utility JAR.
- Add BankEJBClient to the Java JAR Dependencies.

Add BankEJBClient as a Utility JAR

To add BankEJBClient as a Utility JAR for the BankAppClientEAR project, do the following:

1. In the Project Explorer view of the J2EE perspective, expand **Enterprise Applications** → **BankAppClientEAR**.
2. Double-click **Deployment Descriptor: BankAppClientEAR** to open in the Application Deployment Descriptor editor.
3. Select the **Module** tab.
4. Click **Add** in the Project Utility Jars section.
5. When the Add Utility JAR dialog appears, select **BankEJBClient** and click **Finish**.
6. Save and close the deployment descriptor.

Add BankEJBClient to the Java JAR Dependencies

To add BankEJBClient as a Java JAR Dependency for the BankAppClient project, do the following:

1. In the Project Explorer view of the J2EE perspective, expand **Application Client Projects**, right-click **BankAppClient**, and select **Properties** from the menu.
2. When the Properties for BankAppClient dialog appears, select **Java JAR Dependencies**.
3. Check **BankEJBClient.jar** and click **OK**.

16.4.3 Import the graphical user interface and control classes

In this section, we complete the graphical user interface (GUI) for the J2EE application client.

Since this chapter focuses on the aspects relating to development of J2EE application clients, we will import the finished user interface and focus on implementing the code for accessing the EJBs in the EJB project that was imported in 16.3.1, “Import the base enterprise application sample” on page 932.

To import the framework classes for the J2EE application client, do the following:

1. Expand **Application Client Projects** → **BankAppClient**, right-click **appClientModule**, and select **Import**.
2. When the Import dialog appears, select **Zip file** and click **Next**.
3. When the Zip file page appears, enter `c:\6449code\j2eeclt\BankAppClient_GUI.jar` in the From zip file field.

Archived

4. Expand **/** in the left panel and ensure that only **itso** is checked (as seen in Figure 16-7, and then click **Finish**.

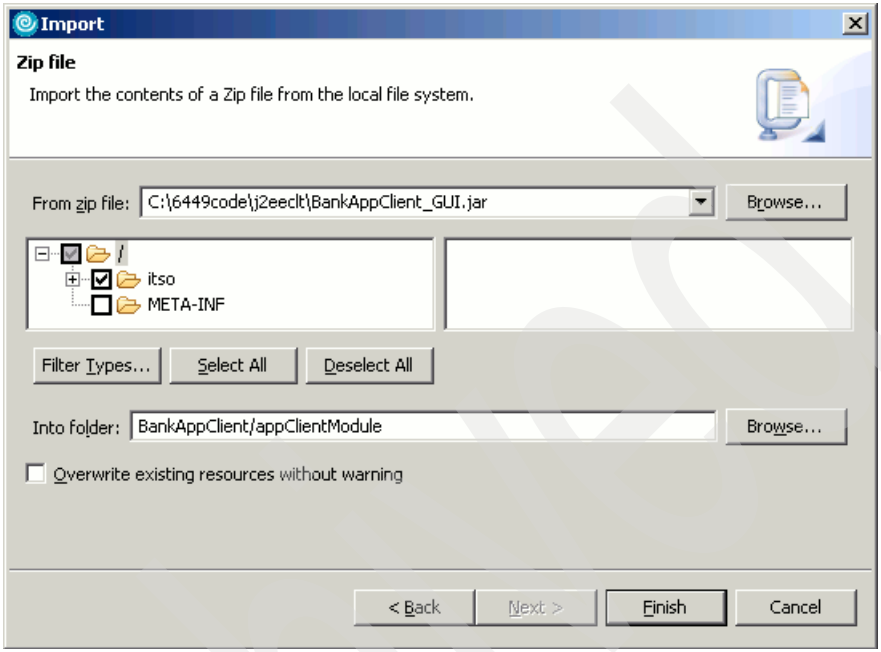


Figure 16-7 Import existing GUI class

When the wizard has completed the import, the following classes should have been added to the BankAppClient project:

- itso.bank.client.ui.BankDesktop

  This is a visual class, containing the view for the Bank J2EE application client.

- itso.bank.client.model.AccountTableModel

  This is an implementation of the interface javax.swing.table.TableModel. The class will provide the relevant TableModel interface, given an array of Account instances.

16.4.4 Create the BankDesktopController class

In this section, we create the controller class for the J2EE application client. This class will also be the main class for the application and will contain the EJB lookup code.

To create the BankDesktopController class, do the following:

1. Expand **Application Client Projects** → **BankAppClient**.

2. Right-click **appClientModule** and select **New** → **Class**.
3. When the New Java Class dialog appears, enter `itso.bank.client.control` in the Package field, `BankDesktopController` in the Name field, check **public static void main(String[] args)** and **Constructors from superclass**, and click **Add**.
4. When the Implemented Interfaces Selection dialog appears, enter `ActionListener` in the Choose interfaces field, select **java.awt.event** in the Qualifier list, and click **OK**.
5. When you return to the New Java Class wizard, it should look like Figure 16-8 on page 941. Click **Finish**.

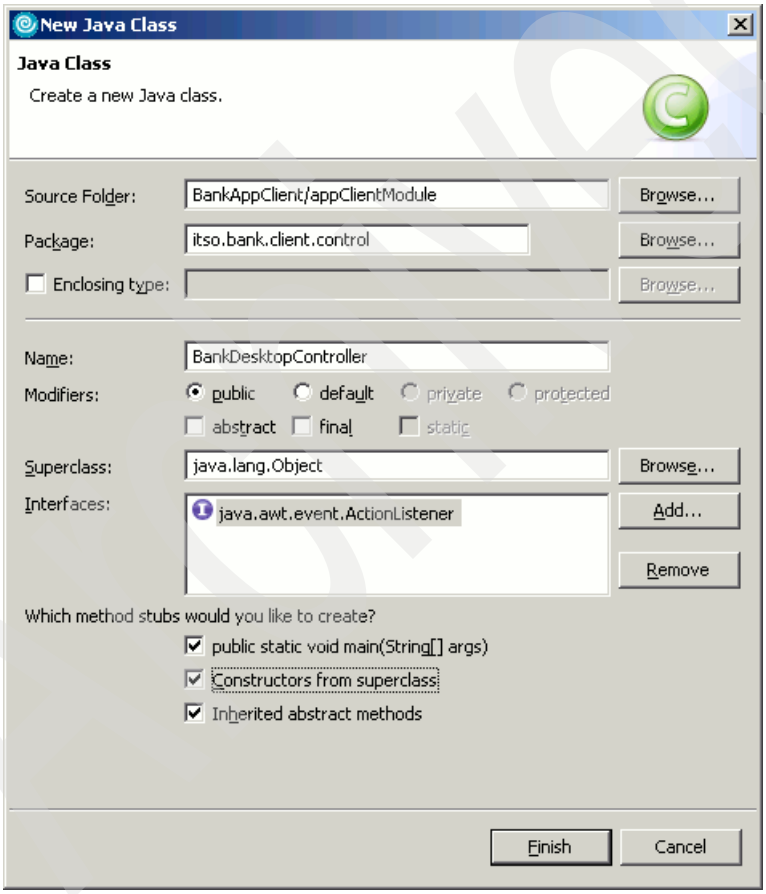


Figure 16-8 Create class BankDesktopController

The BankDesktopController class is now created. In the following section, you will add the logic to the class.

16.4.5 Complete the BankDesktopController class

The BankDesktopController class, which was created in the previous section, is an empty class with no logic. In this section we add control logic, as well as the code to look up customer and account information from the BankEJB enterprise application.

Note: The code found in this section can be copied from the complete BankDesktopController class that is supplied with the additional material for this book. The class can be found in the file c:\6449code\j2eeclt\BankDesktopController.java.

We suggest that you copy the sections as noted in our procedure from the completed BankDesktopController.java (step by step). If you simply import the BankDesktopController.java, the serviceLocatorMgr.jar will not be added as a Java JAR dependency to the BankAppClient project, and thus you will have *can not resolve* errors in the source code. The serviceLocatorMgr.jar is added by using the Insert EJB wizard.

To complete the BankDesktopController class, do the following:

1. In the Project Explorer view of the J2EE perspective, expand **Application Client Projects** → **BankAppClient** → **appClientModule** → **itso.bank.client.control**.
2. Double-click **BankDesktopController.java** to open the class in the Java editor.
3. Add the import statements shown in Example 16-1 to the Java file.

Example 16-1 Import statements to add to BankDesktopController.java

```
import itso.bank.client.ui.AccountTableModel;
import itso.bank.client.ui.BankDesktop;
import itso.bank.model.Account;
import itso.bank.model.Customer;
import itso.bank.exception.UnknownCustomerException;
```

4. Add the fields shown in Example 16-2 to the beginning of the class definition.

Example 16-2 Fields to add to the BankDesktopController class

```
private BankDesktop desktop = null;
private AccountTableModel accountTableModel = null;
```

5. Locate the constructor and modify it to look similar to Example 16-3. The new code is highlighted in bold.

Example 16-3 Constructor for the BankDesktopController class

```
public BankDesktopController() {

   desktop = new BankDesktop();

   desktop.getBtnSearch().addActionListener(this);

   desktop.setVisible(true);
}
```

6. Locate the main method stub and modify it to look similar to Example 16-4. The new code is highlighted in bold.

Example 16-4 Modify the main method

```
public static void main(String[] args) {

   BankDesktopController controller = new BankDesktopController();
}
```

7. Add the setAccounts method shown in Example 16-5 right before the main method.

Example 16-5 Add the setAccounts method

```
private void setAccounts(Account[] accounts) {
   if (accountTableModel == null) {
      // instantiate the model and associate it with the JTable, if it
      // hasn't been created yet.
      accountTableModel = new AccountTableModel();
      desktop.getTblAccounts().setModel(accountTableModel);
   }

   // update the JTable
   accountTableModel.setAccounts(accounts);
}
```

8. Locate the actionPerformed method stub and modify it to look similar to Example 16-6. The new code is highlighted in bold.

Example 16-6 Modify the actionPerformed method

```
public void actionPerformed(ActionEvent e) {
   // we know that we are only listening to action events from
   // the search button, so...

   String ssn = desktop.getTfSSN().getText();
```

Archived

```
}
```

9. Place the cursor on the last line of the actionPerformed method (the line between the ssn variable declaration and the ending curly brace).
10. In the Snippets view, expand **EJB** and double-click **Call an EJB "create" method**.
11. When the Insert EJB create dialog appears, click **New EJB Reference**.
12. When the Add EJB Reference dialog appears, select **Enterprise Beans in the workspace**, expand **BankEJBEAR** → **BankEJB**, select **Bank**, and click **Finish**.
13. When you return to the Insert EJB create dialog, click **Next**.
14. When the Enter Lookup Properties dialog appears, check **Use default context properties for doing a lookup on this reference** and click **Finish**.

Note: By specifying to use the default context properties, we do not need to hard-code the server name in the code, or in other ways make the code location-aware.

To allow the application client to run on a node, separate from the application server, just specify the server name when starting the J2EE client container.

In the Application Client for WebSphere Application Server, this can be done by using the -CCBootstrapHost parameter to the launchClient script. Refer to the WebSphere Application Server InfoCenter for more information about using the Application Client for WebSphere Application Server.

The Insert EJB create wizard will do the following:

- Adds the following line at the cursor location in the actionPerformed method:

```
Bank aBank = createBank();
```

- Adds private fields STATIC_BankHome_REF_NAME and STATIC_BankHome_CLASS to the class.
- Adds a private method, createBank, to the class.
- Adds the JAR serviceLocatorMgr.jar as a Utility JAR to the BankAppClientEAR enterprise application project.
- Creates an EJB reference, ejb/Bank, to the deployment descriptor for the BankAppClient project.

– Adds the following import statements to the Java class file:

```
import com.ibm.etools.service.locator.ServiceLocatorManager;
import java.rmi.RemoteException;
import itso.bank.facade.ejb.BankHome;
import itso.bank.facade.ejb.Bank;
```

15. Complete the actionPerformed to look similar to Example 16-7. The new code is highlighted in bold.

Example 16-7 Complete the actionPerformed method

```
public void actionPerformed(ActionEvent e) {
   // we know that we are only listening to action events from
   // the search button, so...

   String ssn = desktop.getTfSSN().getText();

   Bank aBank = createBank();

   try {
      // look up the customer
      Customer customer = aBank.getCustomer(ssn);
      // look up the accounts
      Account[] accounts = aBank.getAccounts(ssn);

      // update the user interface
      desktop.getTfTitle().setText(customer.getTitle());
      desktop.getTfFirstName().setText(customer.getFirstName());
      desktop.getTfLastName().setText(customer.getLastName());
      setAccounts(accounts);
   }
   catch (UnknownCustomerException x) {
      // unknown customer. Report this using the output fields...
      desktop.getTfTitle().setText("(not found)");
      desktop.getTfFirstName().setText("(not found)");
      desktop.getTfLastName().setText("(not found)");
      setAccounts(new Account[0]);
   }
   catch (RemoteException x) {
      // unexpected RMI exception. Print it to the console and report it...
      x.printStackTrace();
      desktop.getTfTitle().setText("(internal error)");
      desktop.getTfFirstName().setText("(internal error)");
      desktop.getTfLastName().setText("(internal error)");
      setAccounts(new Account[0]);
   }
}
```

16. Save and close BankDesktopController.java.

Archived

The code for the ITSO Bank J2EE application client is now complete. Now we just need to register the BankDesktopController class as the main class for the application client.

16.4.6 Register the BankDesktopController class as the Main class

The BankDesktopController class, which was created in the previous section, contains the logic for the J2EE application client. We need to register that this is the main class for the application client, such that J2EE application client containers know how to launch the application.

To register the BankDesktopController class as the main class, do the following:

1. In the Project Explorer, expand **Application Client Projects** → **BankAppClient** → **appClientModule** → **META-INF**.
2. Right-click **MANIFEST.MF** and select **Open With** → **JAR Dependency Editor**.
3. When the JAR Dependency editor appears, click **Browse** next to the Main-Class entry field.
4. When the Type Selection dialog appears, enter `BankDesktopController` in the Select a class using field, ensure that **itso.bank.client.control.BankDesktopController** is selected in the Qualifier list, and click **OK**.
5. Save the file and close the JAR Dependency Editor.

   The error regarding the missing main class should disappear from the Problems view when the file is saved.

16.5 Test the J2EE application client

Now that the code has been updated, we can test the J2EE application client as follows:

1. Ensure that the server is started.
   a. In the J2EE perspective, switch to the **Servers** view.
   b. Check that the status shown for WebSphere Application Server v6.0 is `Started`. If not, right-click the server and select **Start**.
2. Ensure that the BankEJBEAR enterprise application is deployed on the server.
   a. In the J2EE perspective, switch to the Servers view.

Archived

b. Right-click **WebSphere Application Server v6.0** and select **Add and remove projects**.

c. When the Add and Remove Projects dialog appears, the BankEJBEAR project should appear in the Configured projects list, as shown in Figure 16-9 on page 947. If it does not, select **BankEJBEAR** and click **Add**.

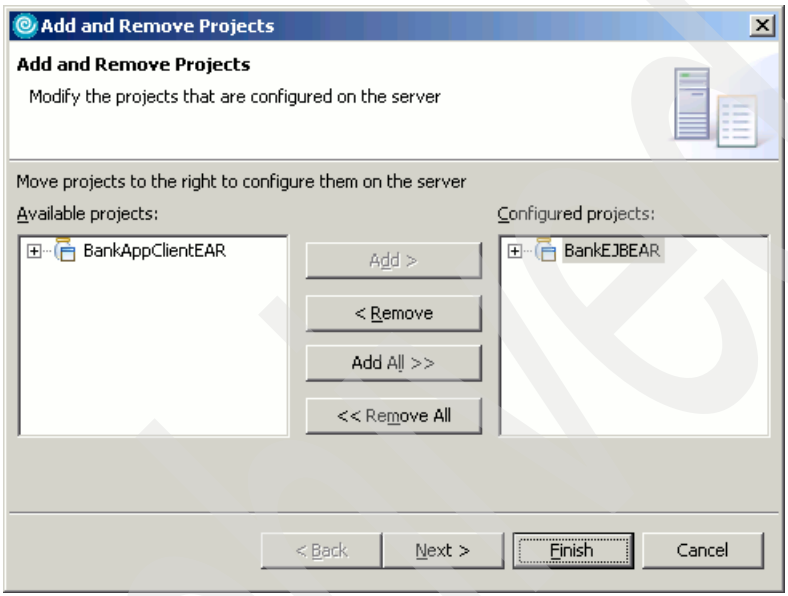


Figure 16-9 Add and Remove Projects dialog showing BankEJBEAR as deployed

d. Click **Finish**.

If you had to add the BankEJBEAR project to the list, Rational Application Developer will now deploy that project to the server. This may take a while.

> **Note:** We found that sometimes, the server would restart in Debug mode after adding or removing projects for the server. If this happens, for you, right-click the server in the Servers view and select **Restart** → **Start**.

3. In the Project Explorer, expand **Application Client Projects**, right-click **BankAppClient** and select **Run** → **Run...** from the menu.
4. When the Run dialog appears, select the **WebSphere v6.0 Application Client** in the Configurations list and click **New**.

The right-hand side of the dialog will change to allow you to set up the new Run configuration.

5. Enter `BankAppClient` in the Name field, as shown in Figure 16-10 on page 948, click **Apply** and then **Run**.

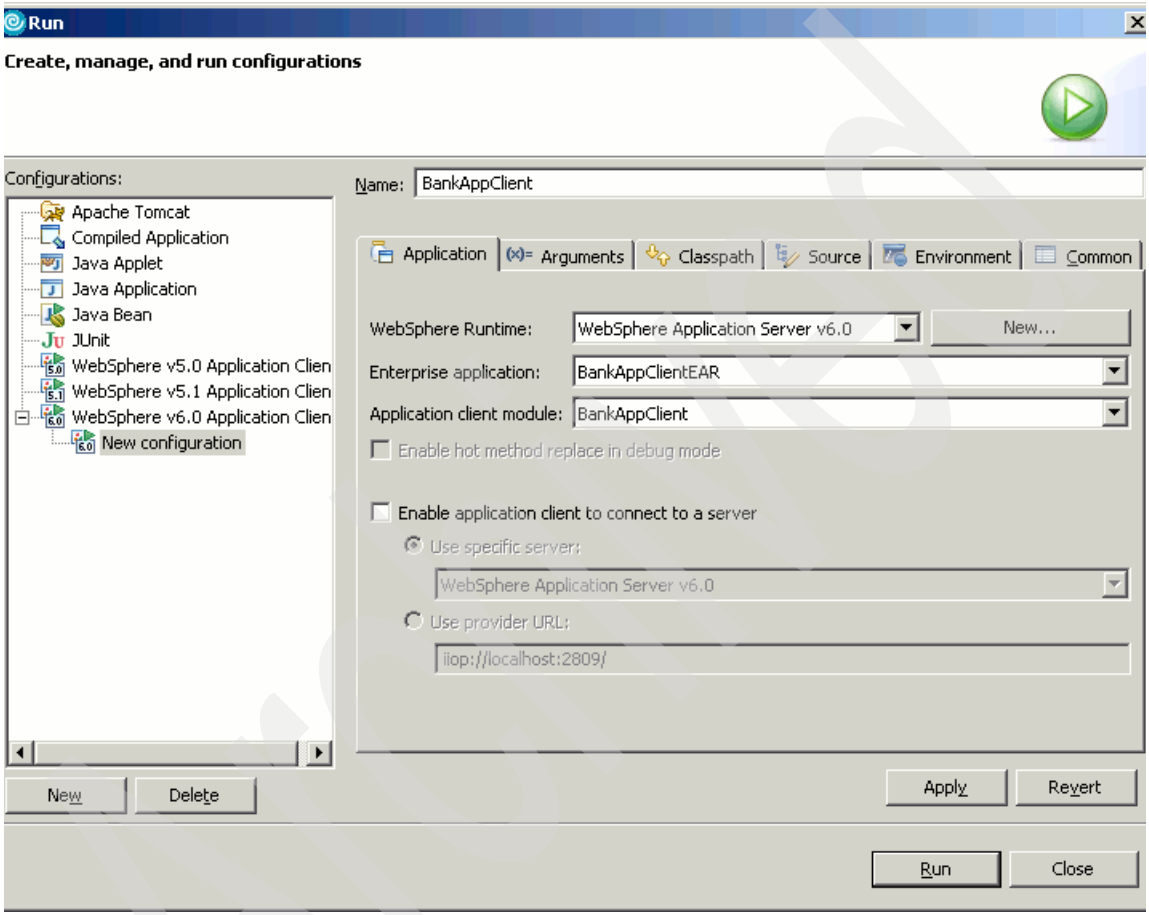


Figure 16-10 Create a new run configuration for the ITSO Bank application client

6. When the Bank Desktop window appears, enter `111-11-1111` in the Search SSN field and click **Search**. The results will be displayed as shown in Figure 16-11 on page 949. Try other SSN values, such as 222-22-2222 or 999-99-9999.

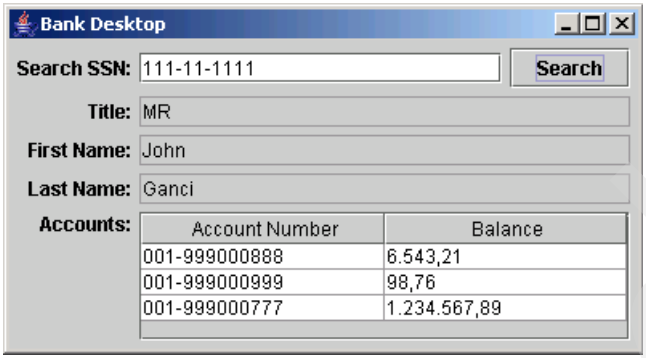


Figure 16-11 Showing the details for customer 111-11-1111

7. Enter an invalid value for the Search SSN and observe the output in the GUI.
8. When you have finished testing the J2EE application client, close the window.

We have successfully built and tested a J2EE application client.

16.6 Package the application client project

In order to run the application client outside Rational Application Developer, we need to package the application.

Note: Although the J2EE specification names the JAR format as the principle means for distributing J2EE application clients, the WebSphere Application Server application client container, Application Client for WebSphere Application Server, expects an enterprise application archive (EAR) file.

To package the application client for deployment, do the following:

1. In the Project Explorer view of the J2EE perspective, expand **Enterprise Applications**, right-click **BankAppClientEAR**, and select **Export** → **EAR file** from the menu.
2. When the EAR Export dialog appears, enter the name of the EAR file (for example, `c:\deployment\BankAppClient.ear`) in the Destination field, and click **Finish**.

Tip: If you check the **Export source files** and **Include project build paths and meta-data files** check boxes, you will be able to later import the EAR file to Rational Application Developer.

The exported EAR file can now be deployed to a client node and executed using the Application Client for WebSphere Application Server.

Archived

17

Develop Web Services applications

This chapter introduces the concepts of a service-oriented architecture (SOA) and explains how such an architecture can be realized using the Java 2 Platform Enterprise Edition (J2EE) Web Services implementation.

We will explore the features provided by Rational Application Developer for Web Services development and look at two common examples: Create Web Services based on a JavaBean, and on an Enterprise JavaBean. We will also demonstrate how Rational Application Developer can help with testing Web Services and developing Web Services client applications.

The chapter is organized into the following sections:

- Introduction to Web Services
- Web Services tools in Application Developer
- Preparing for the samples
- Create a Web Service from a JavaBean
- Create a Web Service from an EJB
- Web Services security
- Publish a Web Service using UDDI

Note: For more detailed information refer to *WebSphere Application Server V6: Web Services Development and Deployment*, SG24-6461.

© Copyright IBM Corp. 2005. All rights reserved.

17.1 Introduction to Web Services

This section introduces architecture and concepts of the service-oriented architecture (SOA) and Web Services.

17.1.1 Service-oriented architecture (SOA)

In a service-oriented architecture, applications are made up from loosely coupled software services, which interact to provide all the functionality needed by the application. Each service is generally designed to be very self-contained and stateless to simplify the communication that takes place between them.

There are three main roles involved in a service-oriented architecture:

- Service provider
- Service broker
- Service requester

The interactions between these roles are shown in Figure 17-1.

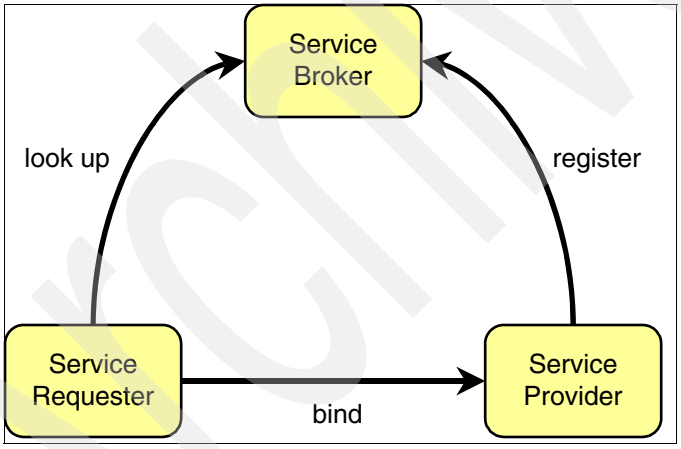


Figure 17-1 Service-oriented architecture

Service provider

The *service provider* creates a service and may publish its interface and access information to a *service broker*.

A service provider must decide which services to expose and how to expose them. There is often a trade-off between security and interoperability; the service provider must make technology decisions based on this trade-off. If the service provider is using a service broker, decisions must be made on how to categorize

Archived

the service, and the service must be registered with the service broker using agreed-upon protocols.

Service broker

The *service broker,* also known as the *service registry*, is responsible for making the service interface and implementation access information available to any potential service requester.

The service broker will provide mechanisms for registering and finding services. A particular broker might be public (for example, available on the Internet) or private—only available to a limited audience (for example, on an intranet). The type and format of the information stored by a broker and the access mechanisms used will be implementation-dependent.

Service requester

The *service requester*, also know as a *service client*, discovers services and then uses them as part of its operation.

A service requester uses services provided by service providers. Using an agreed-upon protocol, the requester can find the required information about services using a broker (or this information can be obtained in some other way). Once the service requester has the necessary details of the service, it can bind or connect to the service and invoke operations on it. The binding is usually static, but the possibility of dynamically discovering the service details from a service broker and configuring the client accordingly makes dynamic binding possible.

17.1.2 Web Services as an SOA implementation

Web Services provides a technology foundation for implementing a service-oriented architecture. A major focus during the development of this technology is to make the functional building blocks accessible over standard Internet protocols which are independent of platforms and programming languages to ensure that very high levels of interoperability are possible.

Web Services are self-contained software services that can be accessed using simple protocols over a network. They can also be described using standard mechanisms, and these descriptions can be published and located using standard registries. Web Services can perform a wide variety of tasks, ranging from simple request-reply to full business process interactions.

Using tools like Rational Application Developer, existing resources can be exposed as Web Services very easily.

The core technologies used for Web Services are as follows:

- XML
- SOAP
- WSDL
- UDDI

XML

Extensible Markup Language (XML) is the markup language that underlies Web Services. XML is a generic language that can be used to describe any kind of content in a structured way, separated from its presentation to a specific device. All elements of Web Services use XML extensively, including XML namespaces and XML schemas.

The specification for XML is available at:

http://www.w3.org/XML/

SOAP

Simple Object Access Protocol (SOAP) is a network, transport, and programming language neutral protocol that allows a client to call a remote service. The message format is XML. SOAP is used for all communication between the service requester and the service provider. The format of the individual SOAP messages depends on the specific details of the service being used.

The specification for SOAP is available at:

http://www.w3.org/TR/soap/

WSDL

Web Services Description Language (WSDL) is an XML-based interface and implementation description language. The service provider uses a WSDL document in order to specify:

- The operations a Web Service provides
- The parameters and data types of these operations
- The service access information

WSDL is one way to make service interface and implementation information available in a UDDI registry. A server can use a WSDL document to deploy a Web Service. A service requester can use a WSDL document to work out how to access a Web Service (or a tool can be used for this purpose).

The specification for WSDL is available at:

http://www.w3.org/TR/wsdl/

Archive

UDDI

Universal Description, Discovery and Integration (UDDI) is both a client-side API and a SOAP-based server implementation that can be used to store and retrieve information on service providers and Web Services.

The specification for UDDI is available at:

http://www.uddi.org/

Figure 17-2 shows a how the Web Services technologies are used to implement an SOA.

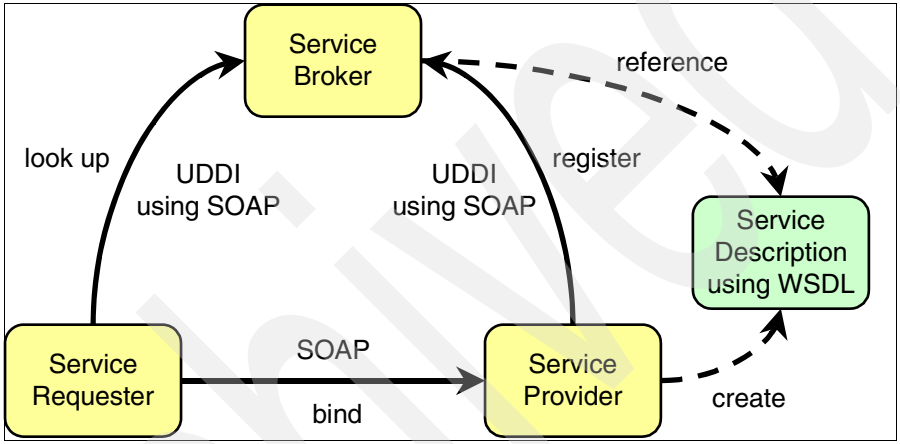


Figure 17-2 Web Services implementation of an SOA

17.1.3 Related Web Services standards

The basic technologies of XML, SOAP, WSDL, and UDDI are fundamental to Web Services, but many other standards have been developed to help with developing and using them.

An excellent resource for information on standards related to Web Services can be found at:

http://www.ibm.com/developerworks/views/webservices/standards.jsp

Web Services in J2EE V1.4

One of the main changes in moving from J2EE V1.3 to V1.4 is the incorporation of Web Services into the Platform standard. J2EE V1.4 provides support for Web

Archived

Services clients and also allows Web Services to be published. The main technologies in J2EE V1.4 that provide this support are as follows:

- Java API for XML-based Remote Procedure Calls (JAX-RPC): JAX-RPC provides an API for Web Services clients to invoke services using SOAP over HTTP. It also defines standard mappings between Java classes and XML types.
- SOAP with Attachments API for Java (SAAJ): Allows SOAP messages to be manipulated from within Java code. The API includes classes to represent such concepts as SOAP envelopes (the basic packaging mechanism within SOAP), SOAP faults (the SOAP equivalent of Java exceptions), SOAP connections, and attachments to SOAP messages.
- Web Services for J2EE: This specification deals with the deployment of Web Service clients and Web Services themselves. Under this specification, Web Services can be implemented using JavaBeans or stateless session EJBs.
- Java API for XML Registries (JAXR): This API deals with accessing XML registry servers, such as servers providing UDDI functionality.

The specifications for Web Services support in J2EE V1.4 are available at:

`http://java.sun.com/j2ee/`

Web Services interoperability

In an effort to improve the interoperability of Web Services, the Web Services Interoperability Organization (known as WS-I) was formed. WS-I produces a specification known as the *WS-I Basic Profile*, which describes the technology choices that maximize interoperability between Web Services and clients running on different platforms, using different runtime systems and written in different languages.

The WS-I Basic Profile is available at:

`http://ws-i.org/deliverables/workinggroup.aspx?wg=basicprofile`

Web Services security

Although not all runtimes support security for Web Services, a body of standards is evolving that describes how Web Services can be secured. The technical basis for these standards is known as *WS-Security*, which provides the basic encryption and digital signature technologies. In addition, several other specifications now use WS-Security for defining trust models, creating secure channels between Web Services and their clients, and ensuring that clients are authorized to use Web Services.

The specification for WS-Security is managed by OASIS:

`http://www.oasis-open.org/`

Web Services workflow

Business Process Execution Language for Web Services (BPEL4WS) provides a language for the specification of business processes and business interactions protocols, extending the basic Web Services model to include business transaction support.

The specification for BPEL4WS is available at:

http://www.ibm.com/developerworks/webservices/library/ws-bpel/

Web Services Inspection Language

Web Services Inspection Language (WS-IL) can be used as an alternative to registering Web Services using UDDI. With WS-IL, a site can be inspected for Web Services, and all the necessary information about the available Web Services can be obtained from this inspection.

The WS-IL specification is available at:

http://www.ibm.com/developerworks/webservices/library/ws-wsilspec.html

17.2 Web Services tools in Application Developer

Rational Application Developer provides tools to create Web Services from existing Java and other resources or from WSDL files, as well as tools for Web Services client development and for testing Web Services.

17.2.1 Creating a Web Service from existing resources

Application Developer provides wizards for exposing a variety of resources as Web Services. The following resources can be used to build a Web Service:

- JavaBean: The Web Service wizard assists you in creating a new Web Service from a simple Java class, configures it for deployment, and deploys the Web Service to a server. The server can be the WebSphere Application Server V6.0 Test Environment included with Rational Application Developer or another application server.
- EJB: The Web Service wizard assists you in creating a new Web Service from a stateless session EJB, configures it for deployment, and deploys the Web Service to a server.
- DADX: Document access definition extension (DADX) is an XML document format that specifies how to create a Web Service using a set of operations that are defined by DAD documents and SQL statements. A DADX Web Service enables you to expose DB2 XML Extender or regular SQL statements as a Web Service. The DADX file defines the operations available

Archived

to the DADX run-time environment and the input and output parameters for the SQL operation.

- URL: The Web Service wizard assists you in creating a new Web Service that directly accesses a *servlet* running on a server.
- ISD: An ISD file is a legacy Web Service deployment descriptor. It provides information to the SOAP runtime about the service that should be made available to clients (for example, URI, methods, implementation classes, serializers, and deserializers). When using a Web Services runtime based on Apache SOAP, ISD files are concatenated into the SOAP deployment descriptor (dds.xml). This mechanism has been replaced in more recent Web Services runtimes, such as Apache Axis and J2EE Web Services runtimes.

17.2.2 Creating a skeleton Web Service

Rational Application Developer provides the functionality to create Web Services from a description in a WSDL (or WSIL) file:

- JavaBean from WSDL: The Web Service wizard assists you in creating a skeleton JavaBean from an existing WSDL document. The skeleton bean contains a set of methods that correspond to the operations described in the WSDL document. When the bean is created, each method has a trivial implementation that you replace by editing the bean.
- Enterprise JavaBean from WSDL: The Web Services tools support the generation of a skeleton EJB from an existing WSDL file. Apart from the type of component produced, the process is similar to that for JavaBeans.

17.2.3 Client development

To assist in development of Web Service clients, Rational Application Developer provides these features:

- Java client proxy from WSDL: The Web Service client wizard assists you in generating a proxy JavaBean. This proxy can be used within a client application to greatly simplify the client programming required to access a Web Service.
- Sample Web application from WSDL: Rational Application Developer can generate a sample Web application, which includes the proxy classes described above, and sample JSPs that use the proxy classes.

Artifact

17.2.4 Testing tools for Web Services

To allow developers to test Web Services, Rational Application Developer provides a range of features:

- WebSphere Application Server V6.0 Test Environment: The V6.0 server is included with Rational Application Developer as a test server and can be used to host Web Services. It provides a range of Web Services runtimes, including an implementation of the J2EE specification standards.
- Sample Web application: The Web application mentioned above can be used to test Web Services and the generated proxy it uses.
- Web Services Explorer: This is a simple test environment that can be used to test any Web Service, based only on the WSDL file for the service. The service can be running on a local test server or anywhere else on the network.
- Universal Test Client: The Universal Test Client (UTC) is a very powerful and flexible test application that is normally used for testing EJBs. Its flexibility makes it possible to test ordinary Java classes, so it can be used to test the generated proxy classes created to simplify client development.
- TCP/IP Monitor: The TCP/IP Monitor works like a proxy server, passing TCP/IP requests on to another server and directing the returned responses back to the originating client. In the process of doing this, it records the TCP/IP messages that are exchanged, and can display these in a special view within Rational Application Developer.

17.3 Preparing for the samples

This section describes the steps required for preparing the environment for the Web Services application samples.

This section includes the following tasks:

- Import the sample code.
- Enable the Web Services Development capability.
- Set up the sample back-end database.
- Add Cloudscape JDBC driver (JAR) to the project.
- Define a server to test the application.
- Test the application.

17.3.1 Import the sample code

To prepare for this sample, we will import some sample code. This is a simple Web application that includes Java classes and an Enterprise JavaBean.

1. Switch to the J2EE perspective Project Explorer view.
2. Right-click **Enterprise Applications**, and select **Import...** → **EAR file**.
3. When the Enterprise Application Import dialog appears, click the **Browse** button next to the EAR file, navigate to and select the **BankWebServiceEAR.ear** from the c:\6449code\webservices folder, and click **Open**.

   Note: For information on downloading and unpacking the redbook sample code, refer to Appendix B, “Additional material” on page 1395.

4. Ensure that the Import EAR project is checked and click **Next >**.
5. Click **Select All** in the part of the next page dealing with Utility JARs and Web libraries, and then click **Finish**.

17.3.2 Enable the Web Services Development capability

By default, the Web Services Development capability is not enabled in IBM Rational Application Developer V6.0.

To enable the Web Services Development capability, do the following:

1. From the Workbench, select **Window** → **Preferences**.
2. Select **Workbench** → **Capabilities**.
3. Expand **Web Service Developer**.
4. Check **Web Services Development** and **Component Test for Web Services** (Core Database Development is already checked by default), as seen in Figure 17-3 on page 961, and then click **OK**.

Archived

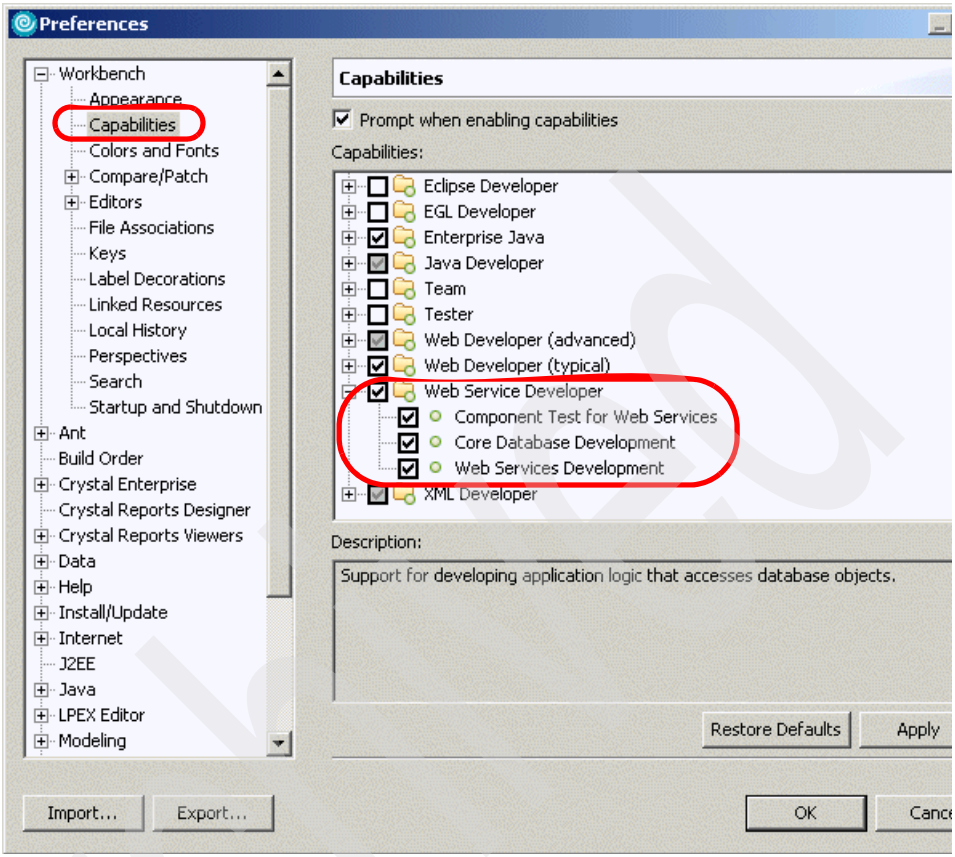


Figure 17-3 Enable Web Services Development capability

17.3.3 Set up the sample back-end database

This section provides instructions for deploying the BANK sample database and populating the database with sample data. For simplicity we will use the built-in Cloudscape database.

To create the database, create the connection, and create and populate the tables for the BANK sample from within Rational Application Developer, do the following:

1. Create the database and the connection to the Cloudscape BANK database from within Rational Application Developer.

   For details refer to “Create a database connection” on page 347.

2. Create the BANK database tables from within Rational Application Developer.

For details refer to “Create database tables via Rational Application Developer” on page 350.

3. Populate the BANK database tables with sample data from within Rational Application Developer.

   For details refer to “Populate the tables within Rational Application Developer” on page 352.

17.3.4 Add Cloudscape JDBC driver (JAR) to the project

To add the Cloudscape JDBC driver (JAR) to the BankWebServiceUtility project, do the following:

1. From the J2EE perspective, expand **Other Projects**.
2. Select **BankWebServiceUtility**, right-click, and select **Properties**.
3. Select **Java Build Path**.
4. Select the **Libraries** tab at the top of the dialog and click **Add Variable...**.
5. A further dialog appears, allowing you to select from a list of predefined variables. By default, there is no variable defined for the JAR file we need, so we will have to create one.
6. Click **Configure Variables...** and in the resulting dialog click **New...**.
7. Enter `CLOUDSCAPE_DRIVER_JAR` in the Name field and click **File...**.
8. Find the appropriate JAR file, which is in <rad_home>\runtimes\base_v6\cloudscape\lib and is called db2j.jar.
9. Click **Open**, **OK**, and **OK** and you will be back at the New Variable Classpath Entry dialog.
10. Select the **CLOUDSCAPE_DRIVER_JAR** variable you just created and click **OK**.
11. Modify the sample code URL to point to the correct back-end database.
    a. From the J2EE perspective expand **Other Projects** → **BankWebServiceUtility** → **src** → **itso.bank.model.entity.java**.
    b. Open the **DatabaseManager.java** file.
    c. Modify the URL string for the database in the getConnection method to reflect the actual location of our database. For example:

       ```
       "jdbc:db2j:C:\\databases\\BANK"
       ```

    d. Save and close the file.

Archived

17.3.5 Define a server to test the application

Next, we need to define the server that the project will be added to to run and test the application. You can add the project to an existing server or create a new server.

Add project to an existing server

If you already have a server defined for testing purposes, do the following.

Note: We chose this option for our example, since we already had a test server configured.

1. Right-click the server (for example, WebSphere Application Server v6.0), and select **Add and remove projects**.
2. When the Add and Remove Projects dialog appears, select **BankWebServiceEAR** under Available projects, and click **Add >**.
3. The BankWebServiceEAR should now appear under the Configured projects column. Click **Finish**.
4. Verify that the server can start and stop successfully.

Create a new server and add the project

To create a new server to run the application, do the following:

1. From the J2EE perspective, select the **Servers** view.
2. Right-click in the Servers view and select **New** → **Server**.
3. When the Define a New Server dialog appears, accept the default Host name as *localhost* and ensure that **WebSphere v6.0 Server** is selected under Select the server type. Click **Next >**.
4. When the WebSphere Server Settings dialog appears, accept the defaults and click **Next >**.
5. When the Add and Remove Projects dialog appears, select **BankWebServiceEAR** under Available projects, and click **Add >**.
6. The BankWebServiceEAR should now appear under the Configured projects column. Click **Finish**.

17.3.6 Test the application

To start and test the application, do the following:

1. Ensure that CView or another application is not connected to the sample BANK database.

Archived

2. Expand **Dynamic Web Projects** → **BankWebServiceWeb** → **WebContent**.
3. Right-click **search.html** and select **Run** → **Run on Server...**.
4. Select the **Choose an existing server** radio button and the desired server to run the application and then click **Finish**.
5. When search.html opens in a Web browser, enter an appropriate value in the Social Security field, such as 111-11-1111, and click **Search**.

   If everything is working correctly, you should see the details of the customer and one of the customer's accounts, which have been read from the database.
6. The stateless session EJB, CustomerFacade, can be tested using the Universal Test Client (UTC). See 15.5, "Testing EJB with the Universal Test Client" on page 915, for more information on using the UTC.

We now have some resources in preparation for the Web Services sample, including a Java class in the BankWebServicesWeb project and an EJB in the BankWebServicesEJB project. We will use these as a base for developing and testing the Web Services samples.

17.4 Create a Web Service from a JavaBean

As explained above, Web Services can be created in several different ways. In this first example, we will create a Web Service from an existing Java class.

The imported application contains a Java class called SimpleBankBean, which has various methods for querying the database.

This section is organized into the following tasks:

- Create a Web Service using the Web Service wizard.
- Resources generated by the Web Services wizard.
- Test the Web Service using the Web Services Explorer.
- Generate and test the client proxy.
- Monitor the Web Service using the TCP/IP Monitor.

17.4.1 Create a Web Service using the Web Service wizard

To create a Web Service using the Web Service wizard, do the following:

1. Ensure that the Web Services Development capability has been enabled.

   For details refer to "Enable the Web Services Development capability" on page 960.

2. Ensure the test server (WebSphere Application Server v6.0) is started. This is needed by the wizard to create a service endpoint interface.
3. From the J2EE perspective, select the **Dynamic Web Projects** folder.
4. Select **File** → **New** → **Other...**.
5. When the Select a Wizard dialog appears, select **Web Service** and then click **Next**.

> **Note:** If you have not previously enabled the Web Services Development capability within Rational Application Developer, you will see the dialog with the message `This action requires the "Web Services Development". Enable the required capability?` Click **OK**.

6. When the Web Service dialog appears, we selected the following options (as seen in Figure 17-4 on page 966), and then clicked **Next**:
   - Web service type: Select **Java bean Web Service**.
   - Check **Start Web Service in Web project** (default).
   - Check **Create folders when necessary** (default).

Archived

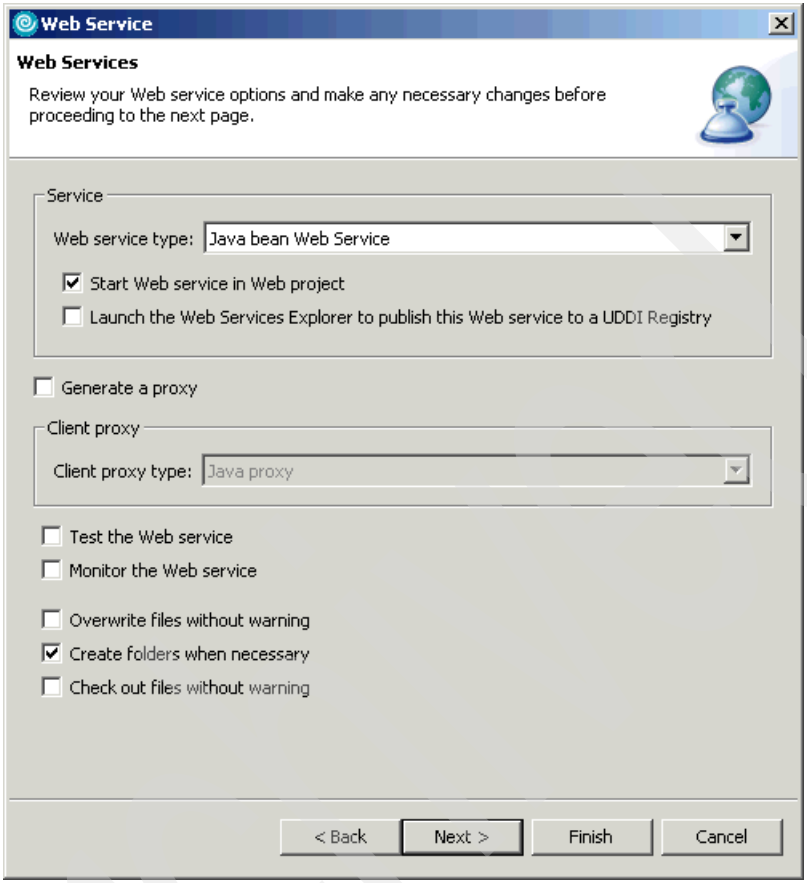


Figure 17-4 New Web Service wizard

7. When the Object Selection Page appears, click **Browse Classes...**, type `SimpleBankBean`, and click **OK**.
8. The Bean name should be itso.bank.model.simple.SimpleBankBean. Click **Next >**.
9. When the Service Deployment Configuration dialog appears, we accepted the values displayed in Figure 17-5 on page 967 (based on our previous settings), and then clicked **Next.**

   If these are not the value shown, you will need to click **Edit...** and set them as appropriate or choose them from the drop-down lists.

Tip: We found that if the server was not started, we received a null pointer exception from the wizard when attempting to create the service endpoint interface in the next step. This is the reason we started the server prior to running the wizard.

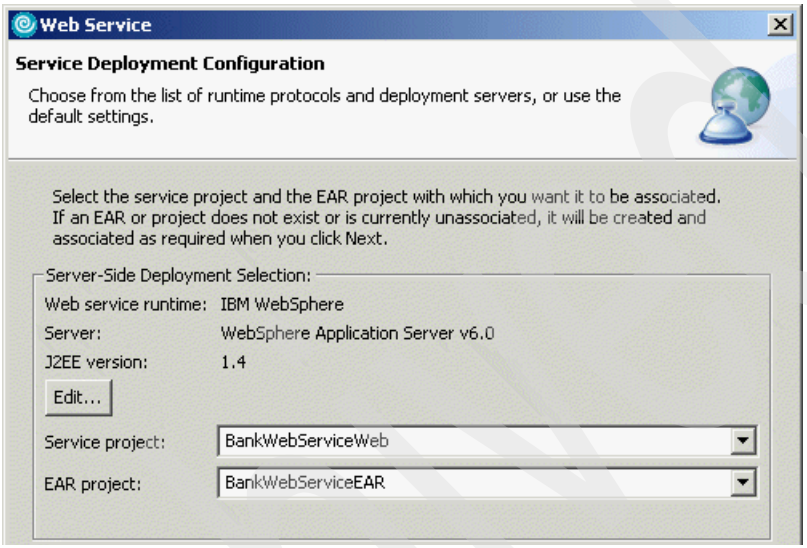


Figure 17-5 New Web Service wizard - Deployment settings

10. When the Service Endpoint Interface Selection dialog appears, accept the default ("Use an existing service endpoint interface" check box unchecked). This will cause the wizard to create a new service end point. Click **Next**.
11. When the Web Service Java Bean Identity dialog appears, we entered the following, as seen in Figure 17-6 on page 968:
    - WSDL File: `SimpleBankBean.wsdl` (default)
    - Methods:
      - Check **getNumAccounts(String)**.
      - Check **getAccountId(String,int)**.
      - Check **getCustomerFullName(String)**.
      - Check **getAccountBalance(String)**.
    - Style and use: select **Document/ Literal**.
    - Accept the default values for the remaining settings.

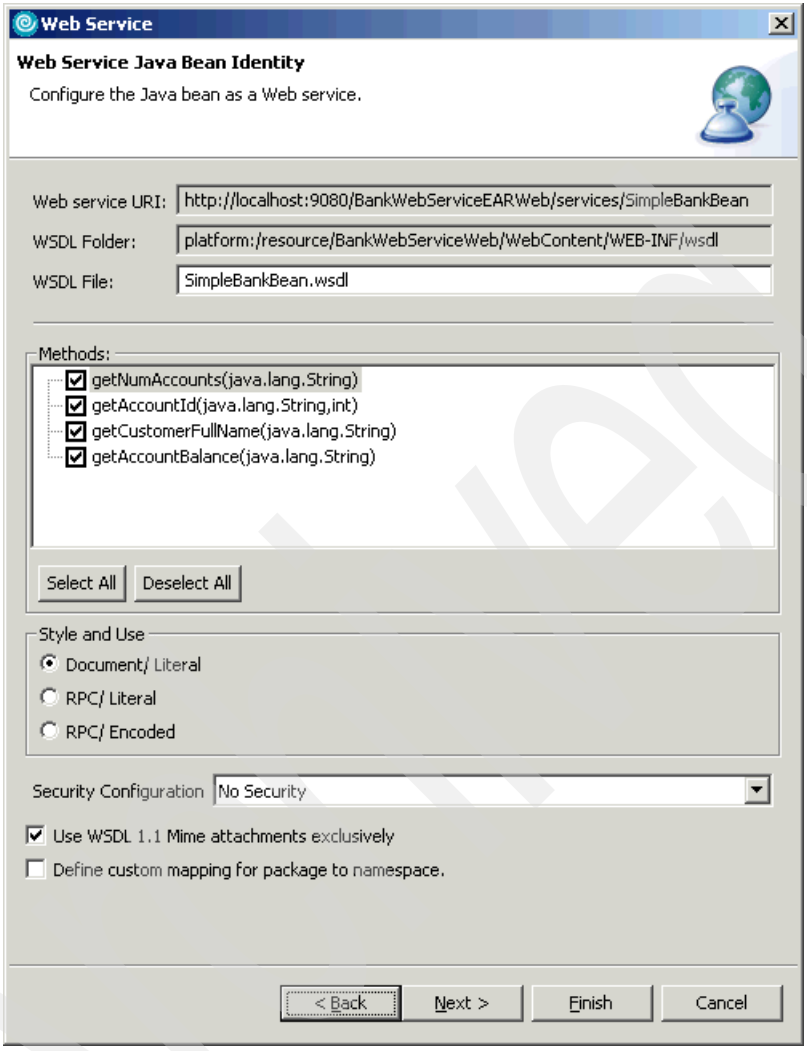


Figure 17-6 New Web Service wizard - Configure the Web Service

12. Click **Next >**.
13. Click **Finish**.

17.4.2 Resources generated by the Web Services wizard

The wizard generates a lot of files based on the choices made. Since the original Java classes are located in the BankWebServiceWeb project, all of the generated code is also located in the BankWebServiceWeb project. The

generated files are visible in two different views, as seen in Figure 17-7 and Figure 17-8 on page 970.

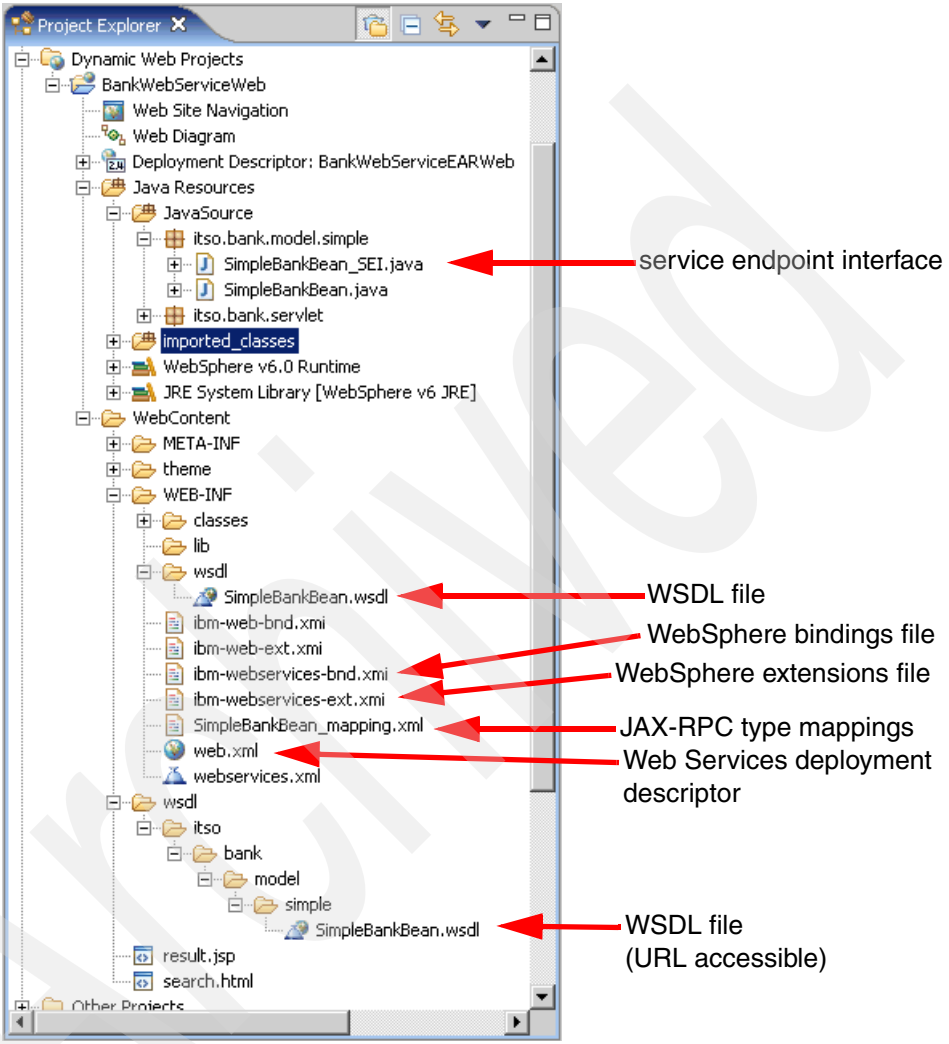


Figure 17-7 Web Services generated resources - Web project view

- The service endpoint interface is the Java interface that is implemented by the Web Service. This will include a subset of the public methods on the class that has been exposed as a Web Service.
- The WSDL file appears in two places:
  - WEB-INF folder: This copy is used by the server for deployment purposes, but is not accessible to external clients through HTTP (the Servlet

Archived

specification states that resources contained within the WEB-INF folder are not visible externally).

  - wsdl folder: This copy is accessible to external clients and can therefore be used by a client to obtain all the necessary information about the Web Service.

- The WebSphere bindings file is used to map local names to global names (for example, to map EJB references to real names used to register EJBs in JNDI).
- The WebSphere extensions file stores information relating to WebSphere extensions to the J2EE specification. This acts as an extension to the information contained within the deployment descriptor.
- The JAX-RPC type mapping file contains information on the relationships between types used in Java code and their equivalents in XML.
- The Web Services deployment descriptor contains information used by the server to deploy the Web Services in the project. The format of this file is defined by the Web Services for J2EE specification.

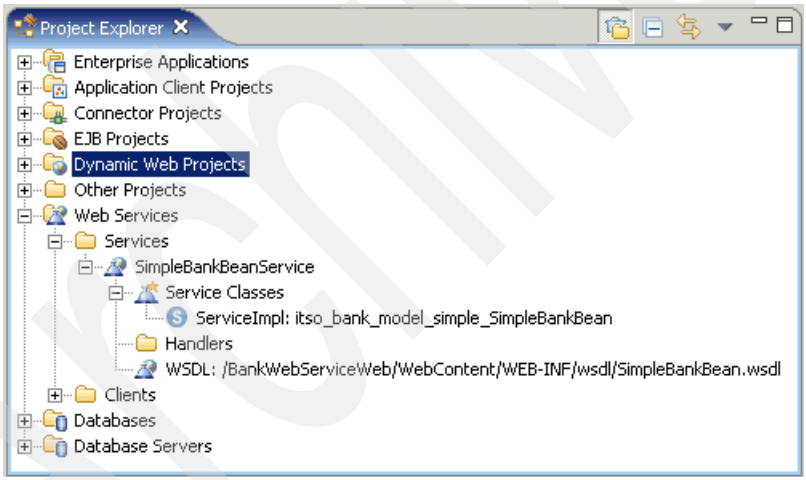


Figure 17-8 Web Services generated files - Web Services view

This view only shows information about Web Services defined in projects within the workspace.

- The Service Classes section shows how the Web Service is registered in a Web project as a Servlet (this can also be seen in the Web project's deployment descriptor).
- The WSDL section shows the location of the internally visible copy of the WSDL file for the Web Service.

- The Handlers section (empty in this case) lists the handlers that are configured for this Web Service. These are similar in concept to Servlet filters.

17.4.3 Test the Web Service using the Web Services Explorer

Rational Application Developer includes a versatile tool for working with Web Services, called the Web Services Explorer. It works with WSDL files and automatically generates the appropriate interface for the Web Services being tested.

1. Ensure that you have closed other application connections to the sample BANK database (exit CView or disconnection from within Application Developer).
2. Open the J2EE perspective Project Explorer view.
3. Expand **Web Services** → **Services** → **SimpleBankBeanService**.

   Note: Alternatively, expand **Dynamic Web Projects** → **BankWebServiceWeb** → **WebContent** → **WEB-INF** → **wsdl**.

4. Right-click **SimpleBankBean.wsdl** from the SimpleBankBeanService folder, and select **Test with Web Services Explorer**, as seen in Figure 17-9 on page 972.

   Note: Alternatively, right-click **SimpleBankBean.wsdl**, and select **Web Services** → **Test with Web Services Explorer**.

Archived

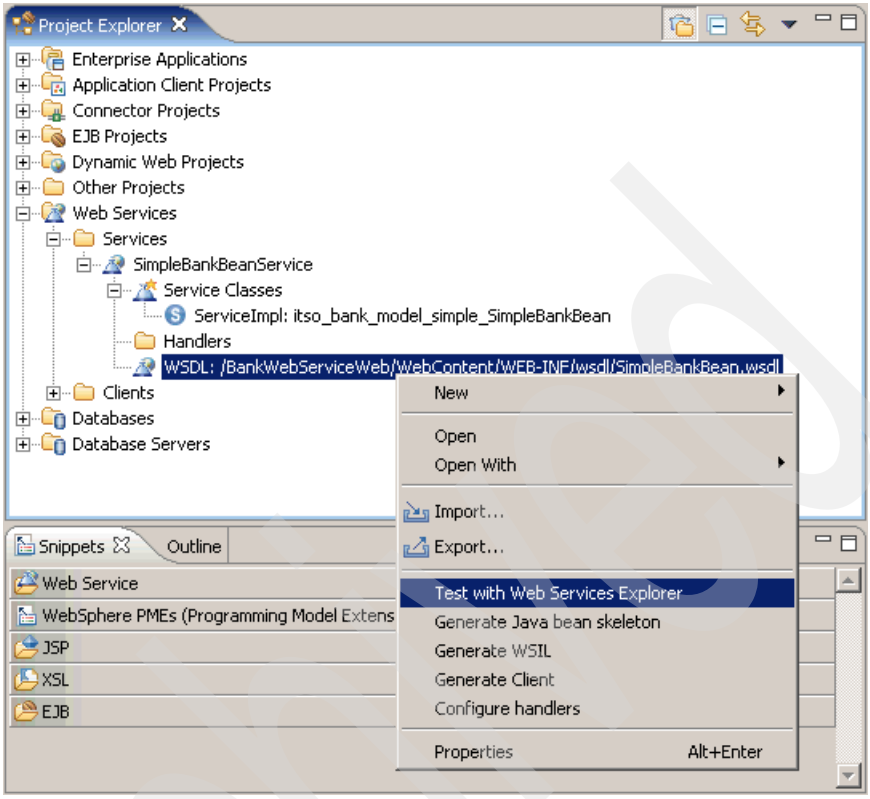


Figure 17-9 Launching the Web Services Explorer

5. When the Web Services Explorer edit dialog appears, expand **SimpleBankBeanSoapBinding** to see the methods available on the Web Service.
6. Select one of the methods (for example, getCustomerFullName).

   The Actions pane on the right displays a simple interface allowing you to enter values for the method parameters
7. Enter a value for the customerId, such as 111-11-1111, and click **Go**.

   The results of your Web Service invocation should appear in the Status pane at the bottom right of the Web Services Explorer.
8. Test the other methods in a similar way.

17.4.4 Generate and test the client proxy

We can also test out Web Service using a proxy class, which is generated by Rational Application Developer.

Note: The proxy is generated using only the WSDL file, so in fact a proxy can be generated for any Web Service for which a WSDL file is available. The Web Service can be running anywhere on the Internet or on an intranet.

The intention of this feature of Rational Application Developer is to simplify client development by encapsulating the lookup and invocation code in a simple Java class.

For more information on using this feature refer to the *WebSphere Application Server V6: Web Services Development and Deployment*, SG24-6461, redbook.

To generate a client and test the client proxy, do the following:

1. Open the J2EE perspective Project Explorer view.
2. Expand **Web Services** → **Services** → **SimpleBankBeanService**.

   Note: Alternatively, expand **Dynamic Web Projects** → **BankWebServiceWeb** → **WebContent** → **WEB-INF** → **wsdl**.

3. Right-click **SimpleBankBean.wsdl**, and select **Generate Client**.

   Note: Alternatively, right-click **SimpleBankBean.wsdl**, and select **Web Services** → **Generate Client**.

4. When the Web Services Client dialog appears, we entered the following (as seen in Figure 17-10 on page 974), and then clicked **Next**:
   - Check **Test the Web Service**.
   - Check **Monitor the Web Service**.
   - Uncheck **Monitor the Web Service** (we will come to this shortly).

   In addition to creating the proxy, this will allow us to generate a sample application based on JSPs, which we can use to test the proxy.

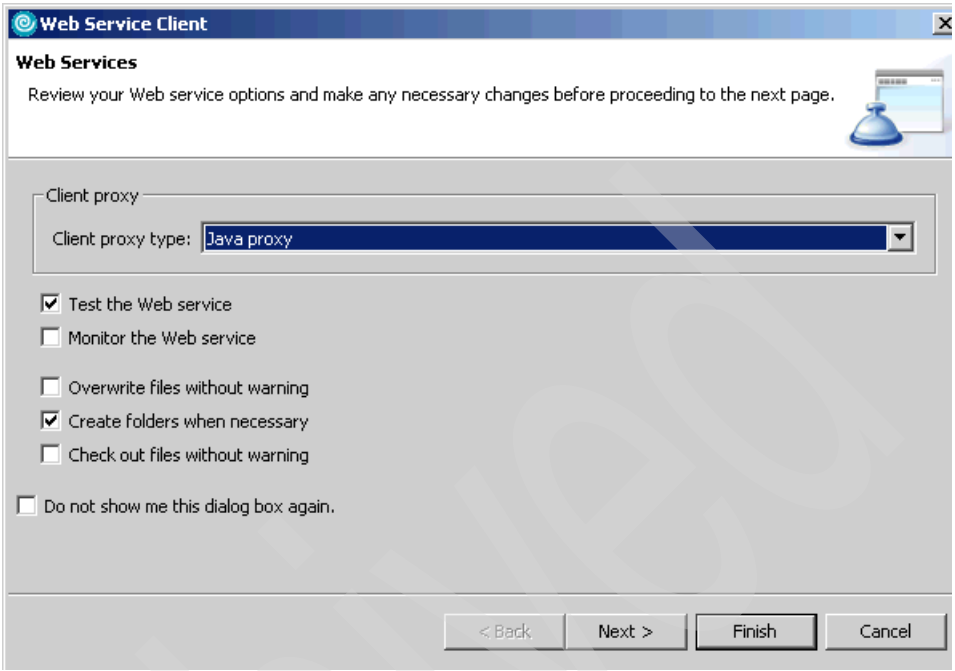


Figure 17-10 Generate Web Service client proxy

5. When the Web Service Selection Page dialog appears, we accepted the default (correct WSDL file selected) and clicked **Next >**.
6. When the Client Environment Configuration dialog appears, we entered the following and then clicked **Next**:
   - Web Service runtime: `IBM WebSphere`
   - Server: `WebSphere Application Server v6.0`
   - J2EE version: `1.4`
   - Client type: Select **Web** (default). The other possible options include EJB, Application Client (J2EE), and Java (standalone application).
   - Client project: `BankWSClient` (This will be created for us.)
   - EAR project: Select **BankWebServiceEAR**.
7. When the Web Service Proxy Page dialog appears, we accepted the default (No Security) and clicked **Next**.
8. You may be prompted with `Cannot create the file "web.xml" relative to the path "/BankWSClient/WebContent/WEB-INF" because automatic file overwriting has not been enabled. Do you want to enable it for this file?` Click **Yes**.

9. When the Web Service Client Test dialog appears, we entered the following and then clicked Finish:
   - Check **Test the generated proxy**.
   - Test facility: Select **Web Service sample JSPs** (default). The test facility options include Web Service sample JSPs, Web Services Explorer, and Universal Test Client.
   - Folder: `sampleSimpleBankBeanProxy` (default)

     You can specify a different folder for the generated application if you wish.
   - Methods: Leave all methods checked.
   - Check **Run test on server**.

   The sample application should start and be displayed in a Web browser.
10. Select the **getCustomerFullName** method, entera valid value in the customerId field such as 111-11-1111, and then click **Invoke**.

    The results should be displayed in the result pane, as seen in Figure 17-11.

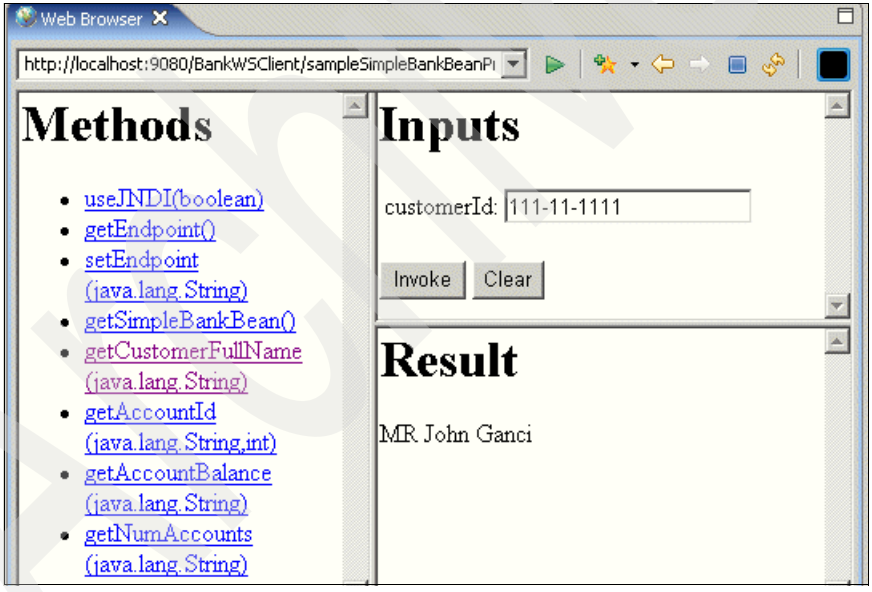


Figure 17-11 Sample JSP results

The code that uses the generated proxy is in Result.jsp and can be viewed in the Source view of the Page Designer editor, although it is not easy to read.

17.4.5 Monitor the Web Service using the TCP/IP Monitor

When developing Web Services, it is often useful to be able to observe the SOAP messages that are passed in both directions between the client and the Web Service. Rational Application Developer provides a tool to allow you to do this, known as the *TCP/IP Monitor*.

To monitor a Web Service using the Rational Application Developer TCP/IP Monitor, do the following:

1. Ensure that you have closed other application connections to the sample BANK database (exit CView or disconnection from within Application Developer).
2. Ensure the test server is started.
3. Create a server to act as the TCP/IP Monitor.
   a. From the Workbench, select **Window** → **Preferences**.
   b. Select **Internet** → **TCP/IP Monitor**.
   c. Ensure that **Show the TCP/IP Monitor view when there is activity** is checked.
   d. Under the TCP/IP Monitors lists, click Add.
   e. When the new Monitor dialog appears, we entered the following, as seen in Figure 17-12 on page 977:
      - Local Monitoring port: `9081`

        Specify a unique port number on your local machine.
      - Host name: `localhost`

        Specify the hostname or IP address of the machine where the server is running.
      - Port: `9080`

        Specify the port number of the remote server.
      - Type: Select **HTTP** (or TCP/IP).
   f. Select the newly added hostname to monitor and then click **Start**.
   g. Click **OK** to save preference settings.

Archived

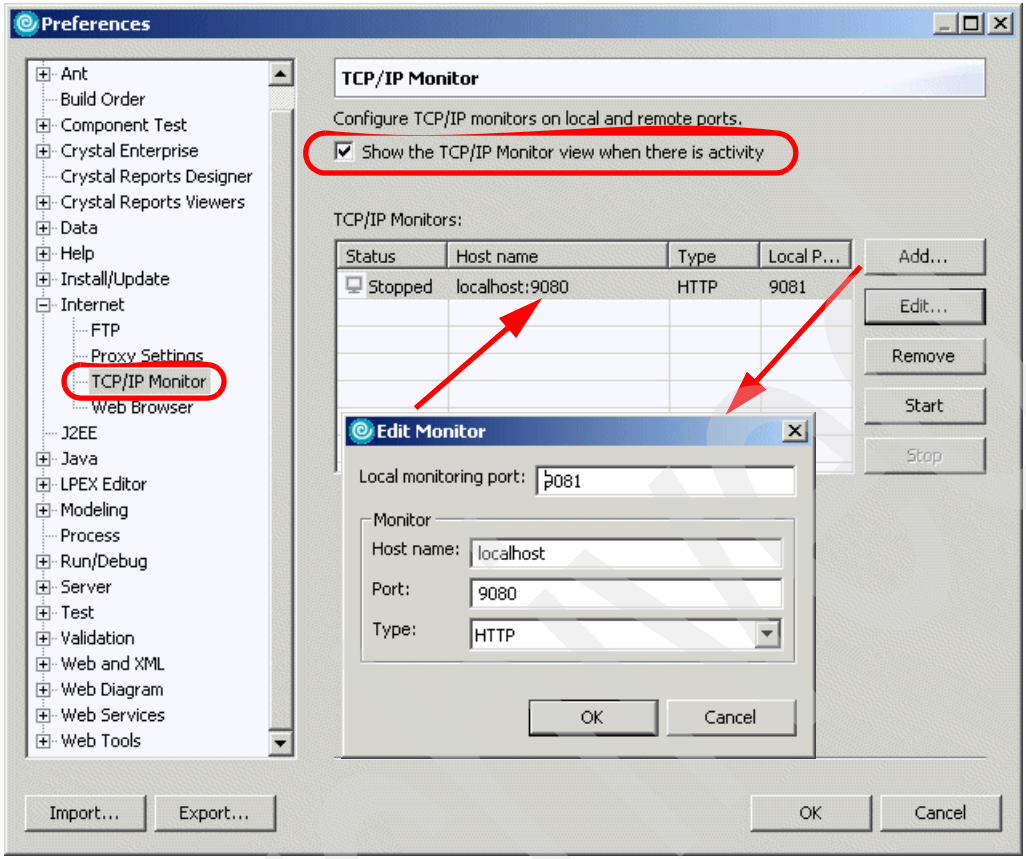


Figure 17-12 TCP/IP Monitor preferences

4. Open the J2EE perspective Project Explorer view.
5. Expand **Web Services** → **Services** → **SimpleBankBeanService**.

> **Note:** Alternatively, expand **Dynamic Web Projects** → **BankWebServiceWeb** → **WebContent** → **WEB-INF** → **wsdl**.

6. Right-click **SimpleBankBean.wsdl**, and select **Test with Web Services Explorer**.

> **Note:** Alternatively, right-click **SimpleBankBean.wsdl**, and select **Web Services** → **Test with Web Services Explorer**.

7. From the Web Services Explorer, select **SimpleBankBeanSoapBinding** in the Navigator pane.
8. Scroll down to the Endpoints section in the Actions pane and click **Add**.
9. A new editable endpoint address will appear, pre-filled with the original address. Change the port number from 9080 to 9081, so the new address will be as follows:

   ```
   http://localhost:9081/BankWebServiceEARWeb/services/SimpleBankBean
   ```

10. Check the new endpoint, and click **Go**.

    You should see a message `Endpoints were successfully updated`.

11. Select the method you wish to test. For example, select **getCustomerFullName**, and from the Endpoints list select the endpoint with 9081 as the port number.
12. Enter a suitable value for the customerId such as 555-55-5555 and click **Go**.

    The TCP/IP Monitor view will automatically open, as seen in Figure 17-13 on page 979.

    As long as port 9081 is used for the endpoint instead of 9080, all requests and responses will be routed through the TCP/IP monitor and will appear in the TCP/IP Monitor view.

    The TCP/IP Monitor view shows all the intercepted requests in the top pane, and when a request is selected, the messages passed in each direction are shown in the bottom panes (request in the left pane, response in the right). This can be a very useful tool in debugging Web Services and clients.

Archive

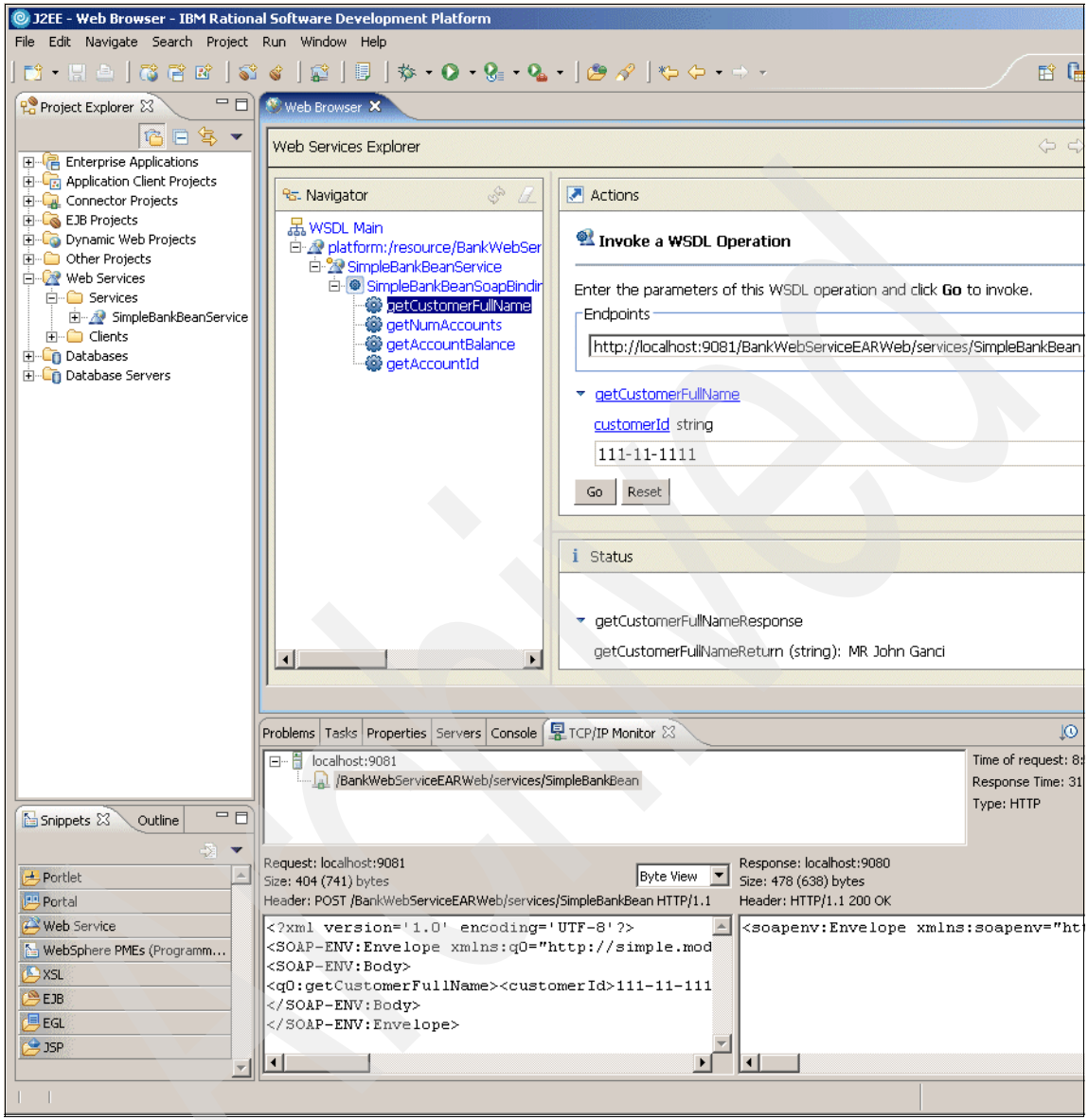


Figure 17-13 TCP/IP Monitor view

17.5 Create a Web Service from an EJB

The process of creating a Web Service from an EJB session bean is very similar to the process for a JavaBean.

1. Expand **EJB Projects** → **BankWebServiceEJB** → **Deployment Descriptor** → **Session Beans**.
2. Right-click **SimpleBankFacade**, and select **Web Services** → **Create Web Service.**
3. The **Web Service type** should be **EJB Web Service**. Leave all the other check boxes at their default values.
4. Click **Next >** and select the appropriate session EJB if it is not already selected.
5. Click **Next >** and **Next >** (these pages are the same as before).
6. On the page entitled Web Service EJB Configuration, you need to specify which Web project will contain the routing Servlet to forward requests to the EJB. We will use the same project we used for the other Web Service, so select **BankWebServiceWeb** from the list.
7. It is possible to invoke message-driven EJBs using SOAP over JMS as opposed to SOAP over HTTP (this is a WebSphere extension). We are using a stateless session EJB, so we need to leave the radio button set to SOAP over HTTP.
8. Click **Next >**.
9. On this page we can select which methods we want to make available to clients. Leave all the methods checked as before and click **Next >**.
10. Click **Finish** on the Web Service Publication page.

The mechanisms for testing Web Services based on stateless session EJBs are exactly the same as for services based on JavaBeans (or any other Web Service). Try testing your Web Service using the Web Services Explorer or the sample JSP application as described above. You will find the WSDL file in the EJB project under ejbModule META-INF wsdl.

17.6 Web Services security

Rational Application Developer allows you to create Web Services that communicate securely with clients. This can be selected in the wizard on the Web Service Java Bean Identity page, as shown in Figure 17-14 on page 982.

The options available are as follows:

- No security
- XML Digital Signature
- XML Encryption
- XML Digital Signature and XML Encryption

Although standards are evolving for Web Services security, they are not yet universally implemented, so the use of these security mechanisms is forbidden by the WS-I Basic Profile. Consequently, enabling any of these security options will produce the warning dialog shown in Figure 17-14 on page 982, stating `You have made a choice that may result in a Web Service that does not comply with WS-I Simple SOAP Basic Profile`. You can choose to ignore this message and continue to use secure Web Services, but this decision will impact the interoperability of your service with Web Services clients running on different technologies.

Secure Web Services can still be tested using the client proxy that is generated by Rational Application Developer, since the proxy includes the necessary code to communicate securely with the Web Service. The Web Services Explorer and the TCP/IP Monitor cannot be used effectively in this case, however.

Archive

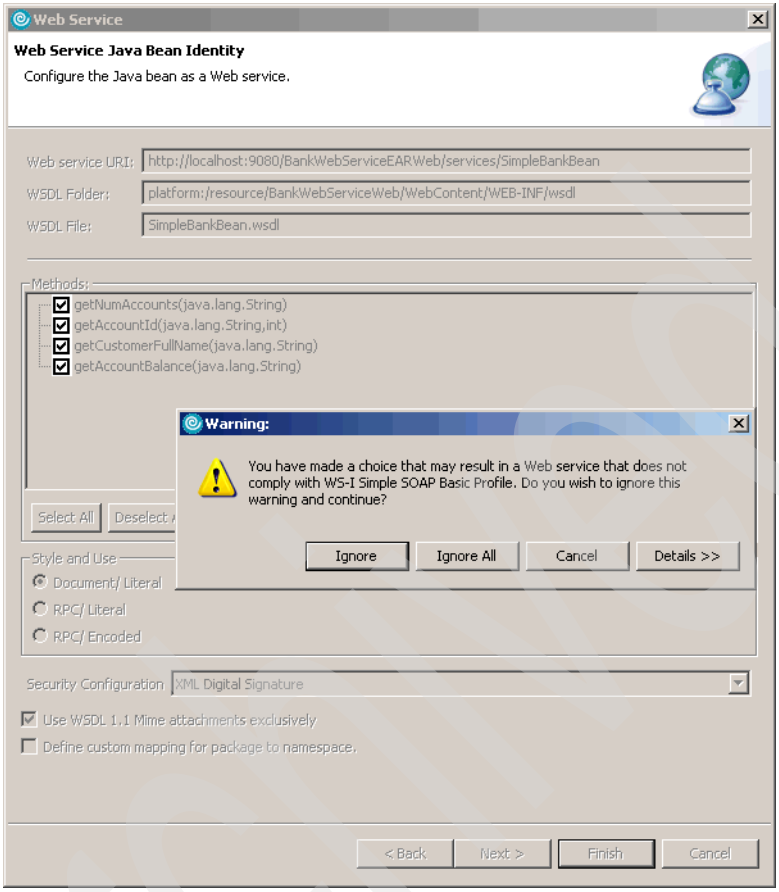


Figure 17-14 Web Services security

17.7 Publish a Web Service using UDDI

Rational Application Developer includes a test UDDI registry, which runs in the integrated WebSphere server and uses Cloudscape or DB2 as the back-end database. To create the test registry:

1. Select **File** → **New** → **Other...** → **Web Services** → **Unit Test UDDI** and click **Next >**.
2. Leave the first page of the wizard with the default values (Private UDDI Registry for WAS v6.0 with Cloudscape) and click **Next >**.
3. The second page of the wizard allows you to specify which server to use for the registry. Leave the default values and click **Finish**.

The wizard imports an EAR file, which includes Web and EJB components and sets up the necessary database tables and data sources in the server; this takes some time. The Web Services Explorer is opened on completion with the UDDI page open. This can be used to register and search for businesses and services using the test registry. The Web Services Explorer can also be used with other registries, such as the Public Business Registries available at locations such as http://uddi.ibm.com/.

Archived

Archived

18

Develop portal applications

This chapter describes the portal development tools and test environment for IBM WebSphere Portal that are now included with IBM Rational Application Developer V6.0. We will also highlight how the portal tools in Rational Application Developer can be used to develop a portal and associated portlet applications for WebSphere Portal. Lastly, we have included a development scenario to demonstrate how to use the new integrated portal tooling and test environment to develop a portal, customize the portal, and develop two portlets.

The chapter is organized into the following topics:

- Introduction to portals
- Developing applications for WebSphere Portal
- Portal development scenario

Note: For more detailed information on IBM WebSphere Portal V5.0 and V5.1 application development, refer to the following:

- *IBM WebSphere Portal V5 A Guide for Portlet Application Development*, SG24-6076
- *IBM WebSphere Portal V5.1 Portlet Application Development*, SG24-6681

© Copyright IBM Corp. 2005. All rights reserved.

18.1 Introduction to portals

As J2EE technology has evolved, much emphasis has been placed on the challenges of building enterprise applications and bringing those applications to the Web. At the core of the challenges currently being faced by Web developers is the integration of disparate user content into a seamless Web application and well-designed user interface. Portal technology provides a framework to build such applications for the Web.

Because of the increasing popularity of portal technologies, the tooling and frameworks used to support the building of new portals has evolved. The main job of a portal is to aggregate content and functionality. Portal servers provide:

- A server to aggregate content
- A scalable infrastructure
- A framework to build portal components and extensions

Additionally, many portals offer personalization and customization features. Personalization enables the portal to deliver user-specific information targeting a user based on their unique information. Customization allows the user to organize the look and feel of the portal to suit their individual needs and preferences.

Portals are the next-generation desktop, delivering e-business applications over the Web to many types of client devices from PCs to PDAs. Portals provide site users with a single point of access to multiple types of information and applications. Regardless of where the information resides or what format it is in, a portal aggregates all of the information in a way that is relevant to the user.

The goal of implementing an Enterprise portal is to enable a working environment that integrates people, their work, personal activities, and supporting processes and technology.

18.1.1 Portal concepts and definitions

Before beginning development for portals, you should become familiar with some common definitions and descriptions of portal-related terminology.

Portal page

A portal page is a single Web page that can be used to display content aggregated from multiple sources. The content that appears on a portal page is displayed by an arrangement of one or more portlets. For example, a World Stock Market portal page might contain two portlets that display stock tickers for popular stock exchanges and a third portlet that displays the current exchange rates for world currencies.

Portlet

A portlet is an individual application that displays content on a portal page. To a user, a portlet is a single window or panel on the portal page that provides information or Web application functionality. To a developer, portlets are Java-based plugable modules that can access content from a source such as another Web site, an XML feed, or a database, and display this content to the user as part of the portal page.

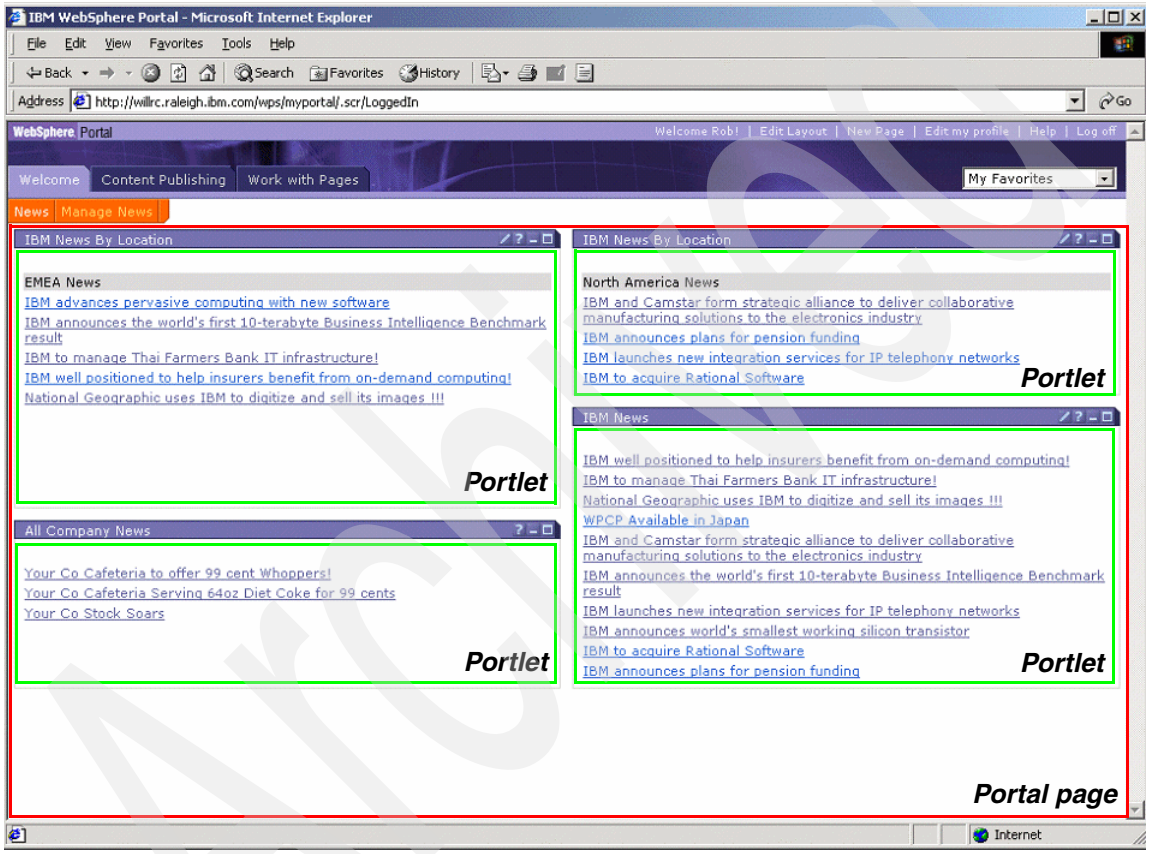


Figure 18-1 Portal page and portlets

Portlet application

Portlet applications are collections of related portlets and resources that are packaged together. Portlets within the same portlet application can exchange and share data and act as a unit. All portlets packaged together share the same context, which contains all resources such as images, properties files, and classes.

Archive

Portlet states

Portlet states determine how individual portlets look when a user accesses them on the portal page. These states are very similar to minimize, restore, and maximize window states of applications run on any popular operating system just in a Web-based environment.

The state of the portlet is stored in the PortletWindow.State object and can be queried for changing the way a portlet looks or behaves based on its current state. The IBM portlet API defines three possible states for a portlet:

- Normal: The portlet is displayed in its initial state, as defined when it was installed.
- Minimized: Only the portlet title bar is visible on the portal page.
- Maximized: The portlet fills the entire body of the portal page, hiding all other portlets.

Portlet modes

Portlet modes allow the portlet to display a different *face* depending on how it is being used. This allows different content to be displayed within the same portlet, depending on its mode. Modes are most commonly used to allow users and administrators to configure portlets or to offer help to the users. There are four modes in the IBM Portlet API:

- View: Initial face of the portlet when created. The portlet normally functions in this mode.
- Edit: This mode allows the user to configure the portlet for their personal use (for example, specifying a city for a localized weather forecast).
- Help: If the portlet supports the help mode, this mode displays a help page to the user.
- Configure: If provided, this mode displays a face that allows the portal administrator to configure the portlet for a group of users or a single user.

Portlet events

Some portlets only display static content in independent windows. To allow users to interact with portlets and to allow portlets to interact with each other, portlet events are used. Portlet events contain information to which a portlet might need to respond. For example, when a user clicks a link or button, this generates an *action* event. To receive notification of a given event, the portlet must also have the appropriate *event listener* implemented within the portlet class. There are three commonly used types of portlet events:

- Action events: Generated when an HTTP request is received by the portlet that is associated with an action, such as when a user clicks a link.

Archived

- Message events: Generated when one portlet within a portlet application sends a message to another portlet.
- Window events: Generated when the user changes the state of the portlet window.

18.1.2 IBM WebSphere Portal

IBM WebSphere Portal provides an extensible framework that allows the end user to interact with enterprise applications, people, content, and processes. They can personalize and organize their own view of the portal, manage their own profiles, and publish and share documents. WebSphere Portal provides additional services such as Single Sign-On (SSO), security, credential vault, directory services, document management, Web content management, personalization, search, collaboration, search and taxonomy, support for mobile devices, accessibility support, internationalization, e-learning, integration to applications, and site analytics. Clients can further extend the portal solution to provide host integration and e-commerce.

WebSphere Portal allows you to plug in new features or extensions using portlets. In the same way that a servlet is an application within a Web server, a portlet is an application within WebSphere Portal. Developing portlets is the most important task when providing a portal that functions as the user's interface to information and tasks.

Portlets are an encapsulation of content and functionality. They are reusable components that combine Web-based content, application functionality, and access to resources. Portlets are assembled into portal pages that, in turn, make up a portal implementation.

Portal solutions such as IBM WebSphere Portal are proven to shorten the development time. Pre-built adapters and connectors are available so that customers can leverage on the company's existing investment by integrating with the existing legacy systems without re-inventing the wheel.

18.1.3 IBM Rational Application Developer

IBM Rational Application Developer provides development tools for applications destined to WebSphere Portal. Bundled with IBM Rational Application Developer V6.0 are a number of portal tools that allow you to create, test, debug, and deploy portal and portlet applications. These tools are described in more detail in 18.2, "Developing applications for WebSphere Portal" on page 992.

Portal tools

Unlike WebSphere Studio Application Developer where the tools where installed as a separate toolkit, Portal Tools can now be installed as a feature when installing IBM Rational Application Developer V6.0. For this reason, there is no longer a separate portal toolkit or separate installation procedure.

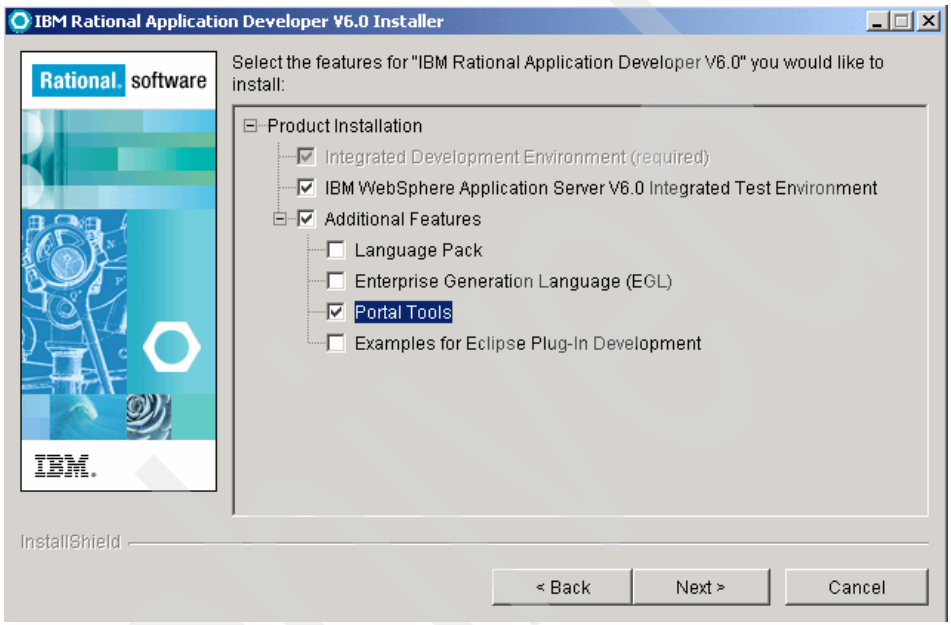


Figure 18-2 Portal tools installation

Portal test environments

As part of this tool set, Rational Application Developer provides an integrated test environment to run and test your portal and portlet projects from within the Rational Application Developer Workbench.

At the time IBM Rational Application Developer V6.0 was released, the WebSphere Portal V5.0.2.2 Test Environment was included as an installed component of the Rational Application Developer installer known as the Launchpad (see Figure 18-3 on page 991).

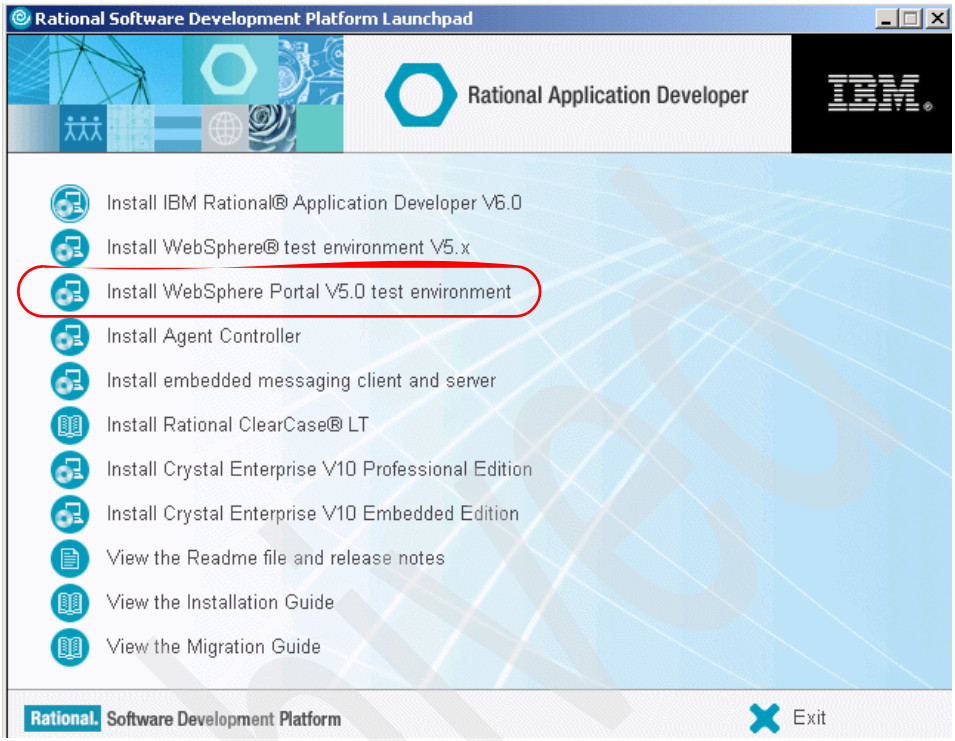


Figure 18-3 Product installation launchpad

CDs to install the WebSphere Portal V5.1 Test Environment are also included with the IBM Rational Application Developer V6.0 distribution. To install the WebSphere Portal V5.1 Test Environment, you must run the WebSphere Portal V5.1 setup program and select **Test Environment** as the setup type, as seen in Figure 18-4 on page 992. Follow the instructions in the InfoCenter to configure this test environment so that it works from within Rational Application Developer. The WebSphere Portal V5.0.2.2 and V5.1 Test Environments can co-exist within the same product installation.

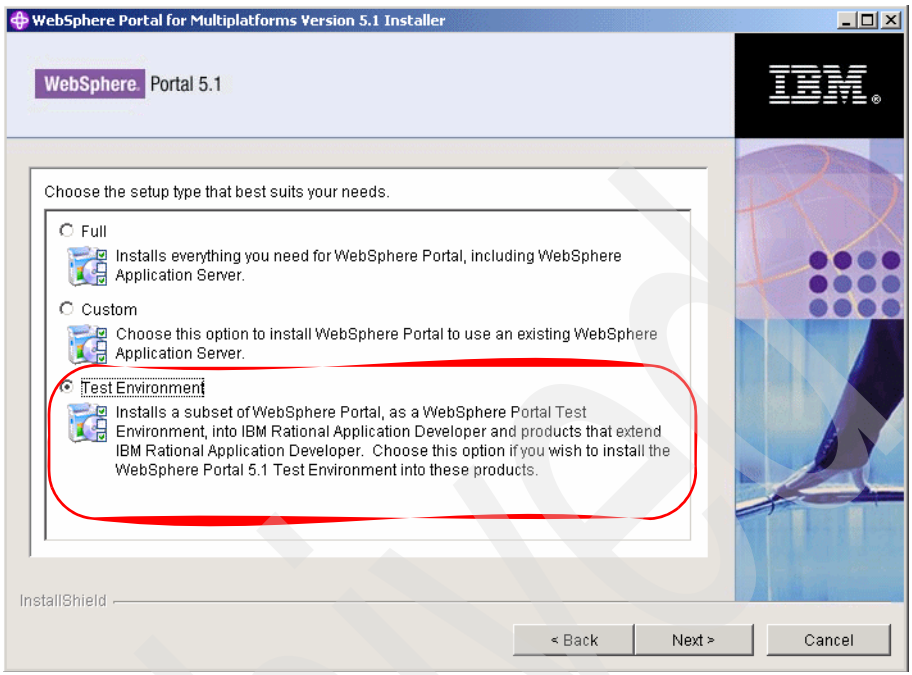


Figure 18-4 Installing the WebSphere Portal V5.1 Test Environment

18.2 Developing applications for WebSphere Portal

Rational Application Developer includes many tools to help you quickly develop portals and individual portlet applications. In this section, we cover some basic portlet development strategies and provide an overview of the tools included with IBM Rational Application Developer V6.0 to aid with development of WebSphere Portal.

18.2.1 Portal samples and tutorials

Rational Application Developer also comes with several samples and tutorials to aid you with the development of WebSphere Portal. The Samples Gallery provides sample portlet applications to illustrate portlet development.

To access portlet samples, click **Help** → **Samples Gallery**. Then expand **Technology samples** and **Portlet**. Here you can select a Basic Portlet, Faces Portlet, or Struts Portlet Framework to view sample portlet application projects that you can then modify and build upon for your own purposes.

The Tutorials Gallery provides detailed tutorials to illustrate portlet development. These are accessible by selecting **Help** → **Tutorials Gallery**. Then expand **Do and Learn**. You can select **Create a portal application** or **Examine the differences between portlet APIs** to view the content of these tutorials.

18.2.2 Development strategy

A portlet application consists of Java classes, JSP files, and other resources such as deployment descriptors and image files. Before beginning development, several decisions must be made regarding the development strategy and technologies that will be used to develop a portlet application.

Choosing an API - IBM or JSR 168

WebSphere Portal supports portlet development using the IBM portlet API and the JSR 168 portlet API standard. The portal tools included with IBM Rational Application Developer V6.0 support both APIs.

The IBM portlet API was initially supported in WebSphere Portal V4.x and will continue to be supported by WebSphere Portal. It is important to note that the IBM portlet API extends the servlet API. More information about the IBM portlet API can be found at:

`http://www.ibm.com/developerworks/websphere/zones/portal/portlet/5.0api/WPS`

JSR 168 is a Java specification from the Java Community Process Program that addresses the requirements of content aggregation, personalization, presentation, and security for portlets running in a portal environment. It was finalized in October of 2003. Portlets conforming to JSR 168 are more portable and reusable because they can be deployed to any JSR 168 compliant portal. Rational Application Developer supports Faces portlet development based on the JSR 168 specification.

Unlike the IBM portlet API, the JSR 168 API does not extend the servlet API. It does, however, share many of the same characteristics such as servlet specification, session management, and request dispatching. The JSR 168 API as implemented in WebSphere Portal V5.0.2.2 does not support Click-to-Action cooperative portlets or portlet messaging. However, in WebSphere Portal V5.1, the JSR 168 container has been enhanced with Property Broker support, which can act as a messaging broker for either portlet messaging or wired (automatic cooperating) portlets. At the time of writing, the support for Click-to-Action (user-initiated cooperating portlets) was still under development.

For new portlets, consider using JSR 168 when the functionality it provides is sufficient for the portlet's needs or when the portlet is expected to be published as a Web Service for Remote Portlets (WSRP) service.

IBM will further support JSR 168 in follow-on versions to make the JSR 168 portlet API as robust as the current IBM counterpart, and offer tooling to support JSR 168 development. IBM is committed to the wider adoption of open standards in WebSphere Portal.

More information can be found on JSR 168 at:

`http://www.jcp.org/en/jsr/detail?id=168`

Choosing markup languages

WebSphere Portal supports mobile devices by generating pages in any markup language. Three markup languages are officially supported in Rational Application Developer:

- HyperText Markup Language (HTML) is used for Web browsers on desktop computers. All portlet applications support HTML at a minimum.
- Wireless Markup Language (WML) is used for WAP devices that are typically Web-enabled mobile telephones.
- compact Hyper Text Markup Language (cHTML) is used for mobile devices in the NTT DoCoMo i-mode network. For more information on the i-mode network, visit the following Web site:

  `http://www.nttdocomo.co.jp/english/`

Adding emulator support for other markup languages

To run a portlet application that supports WML or cHTML, you must use an emulator provided by the device vendor. To add device emulator support to Rational Application Developer, do the following:

1. Select **Window** → **Preferences**.
2. Expand **Internet** and select **Web Browser**.
3. Click the **Add** button to locate the device emulator appropriate for the device that you wish to test and debug.

Enabling transcoding for development in other markup languages

Transcoding is the process by which WebSphere Portal makes portal content displayable on mobile devices. By default, it is *not enabled* in the WebSphere Portal Test Environment. Therefore, you need to make some configuration changes before you can test and debug applications on mobile device emulators. You will need to remove the comments from lines beginning with `#Disable Transcoding` from three files.

The PortletFilterService.properties and PortalFilterService.properties files are all located by default in the following directory:

`<rad_home>\runtimes\portal_v50\shared\app\config\services`

The services.properties file is located by default in the following directory:

```
<rad_home>\runtimes\portal_v50\shared\app\config
```

Choosing other technologies

Struts technology and JavaServer Faces technology can also be incorporated into a portlet development strategy.

Struts

Struts-based application development can be applied to portlets, similar to the way that Struts development is implemented in Web applications. The Struts Portal Framework (SPF) was developed to merge these two technologies. SPF support in Rational Application Developer simplifies the process of writing Struts portlet applications and eliminates the need to manage many of the underlying requirements of portlet applications. In addition, multiple wizards are present to help you create Struts portlet-related artifacts. These are the same wizards used in Struts development. These wizards include: Action Class, Action Mapping, ActionForm, Form-Bean Mapping, Struts Configuration, Struts Module, Struts Exception, and Web diagram. Refer to the Rational Application Developer Struts documentation for usage details.

In WebSphere Portal V5.0.2.2, Struts is only supported using the IBM portlet API. Struts is fully supported in both the IBM and JSR 168 APIs in WebSphere Portal V5.1; however, there is no tooling support in Rational Application Developer for this configuration.

More information on Struts can be found at:

http://struts.apache.org/

JavaServer Faces (JSF)

JavaServer Faces is a GUI framework for developing J2EE Web applications (JSR 127). It includes reusable user interface components, input validation, state management, server-side event handling, page lifecycle management, accessibility, and internationalization. Faces-based application development can be applied to portlets, similar to the way that Faces development is implemented in Web applications. Similar to Struts, there are many wizards to help you with Faces development. Both WebSphere Portal V5.0.2.2 and V5.1 support JavaServer Faces.

There are certain limitations to Faces portlet development in the current release. Service Data Objects (SDO), formerly referred to as WebSphere Data Objects (WDO), are limited to prototyping purposes only. Applications that rely on SDOs should be limited in a production environment. File upload and binary download are not supported for Faces components. Finally, document root-relative URLs are not supported for images.

Refer to the Rational Application Developer Faces documentation in the InfoCenter for usage details. Alternatively you can refer to the following Web site:

http://www.jcp.org/en/jsr/detail?id=127

Beginning development

After making these decisions, you can now begin development using the portal tools included with Rational Application Developer.

18.2.3 Portal tools for developing portals

A portal is essentially a J2EE Web application. It provides a framework where developers can associate many portlets and portlet applications via one or more portal pages.

Rational Application Developer includes several new portal site creation tools that enable you to visually customize portal page layout, themes, skins, and navigation.

Portal Import wizard

One way to create a new portal project is to import an existing portal site from a WebSphere Portal V5.0 server into Rational Application Developer. Importing is also useful for updating the configuration of a project that already exists in IBM Rational Application Developer.

The portal site configuration on WebSphere Portal server contains the following resources: The global settings, the resource definitions, the portal content tree, and the page layout. Importing these resources from WebSphere Portal server to Rational Application Developer overwrites duplicate resources within the existing portal project. Non-duplicate resources from the server configuration are copied into the existing portal project. Likewise, resources that are unique to the portal project are not affected by the import.

Rational Application Developer uses the XML configuration interface to import a server configuration, and optionally retrieves files under the websphere_installation_directory/installedApps/node/wps.ear directory. These files include the JSP, CSS, and image files for themes and skins. When creating a new portal project, retrieving files is mandatory. To retrieve files, Rational Application Developer must have access to this directory, as specified when you define a new server for this project.

You can access the Portal Import Wizard by selecting **File** → **Import**, then selecting **Portal**. You will need to specify the server and options for importing the project into Rational Application Developer.

Follow the instructions in the Help Topics on Developing Portal Applications to ensure that the configuration in the development environment accurately reflects that of the staging or runtime environment. If you do not do this, you may experience compilation errors after the product is imported or unexpected portal behaviors.

Portal Project wizard

The New Portal Project wizard will guide you through the process of creating a portal project within Rational Application Developer.

During this process, you are able to:

- Specify a project name.
- Specify the default server.
- Choose a default theme (optional).
- Choose a default skin for the theme (optional).

Important: You should not name your project *wps* or anything that resembles this string, in order to avoid internal naming conflicts.

The project that you create with this wizard will not have any portlet definitions, labels, or pages. The themes and skins that are available in this wizard are the same as if you had imported a portal site from a WebSphere Portal server.

You can access this wizard by clicking **File** → **New** → **Project** and then selecting **Portal Project** from the list. Figure 18-5 on page 998 displays the options to specify when creating a Portal Project after clicking the **Show Advanced** button.

Archived

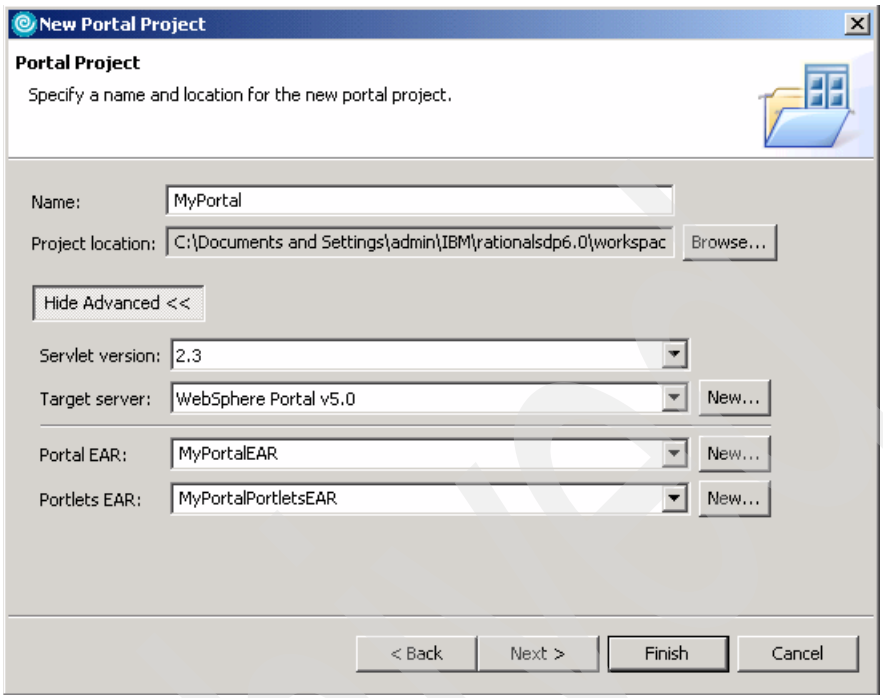


Figure 18-5 Portal Project wizard

Portal Designer

Rational Application Developer allows for editing both the graphic design and portlet layout within your portal. Portal Designer is the Workbench interface that you see upon opening a portal project.

When using Portal Designer, the portal page layout can be altered. The layout refers to the number of content areas within a page and the number of portlets within those content areas. Page content includes rows, columns, URLs, and portlets.

Once the project is published to the Portal server, Portal administrators can use the administration portlets to give site users permission to edit the page layout.

In terms of portal layout and appearance, you can think of Portal Designer as a What-You-See-Is-What-You-Get (WYSIWYG) editor. It will render graphic interface items such as themes, skins, and page layouts.

Portal Designer will also render the initial pages of JSF and Struts portlets within your portal pages, but not anything else with regard to portlet content.

Portal Designer provides the capability to alter the layout of the content within a portal page with respect to navigation (the hierarchy of your labels, pages, and URLs) and content (the arrangement of portlets via rows and columns on the portal pages).

Portal Configuration is the name of the layout source file that resides in the root of the portal project folder (see Figure 18-6). To open Portal Designer, double-click the file in the Project Explorer.

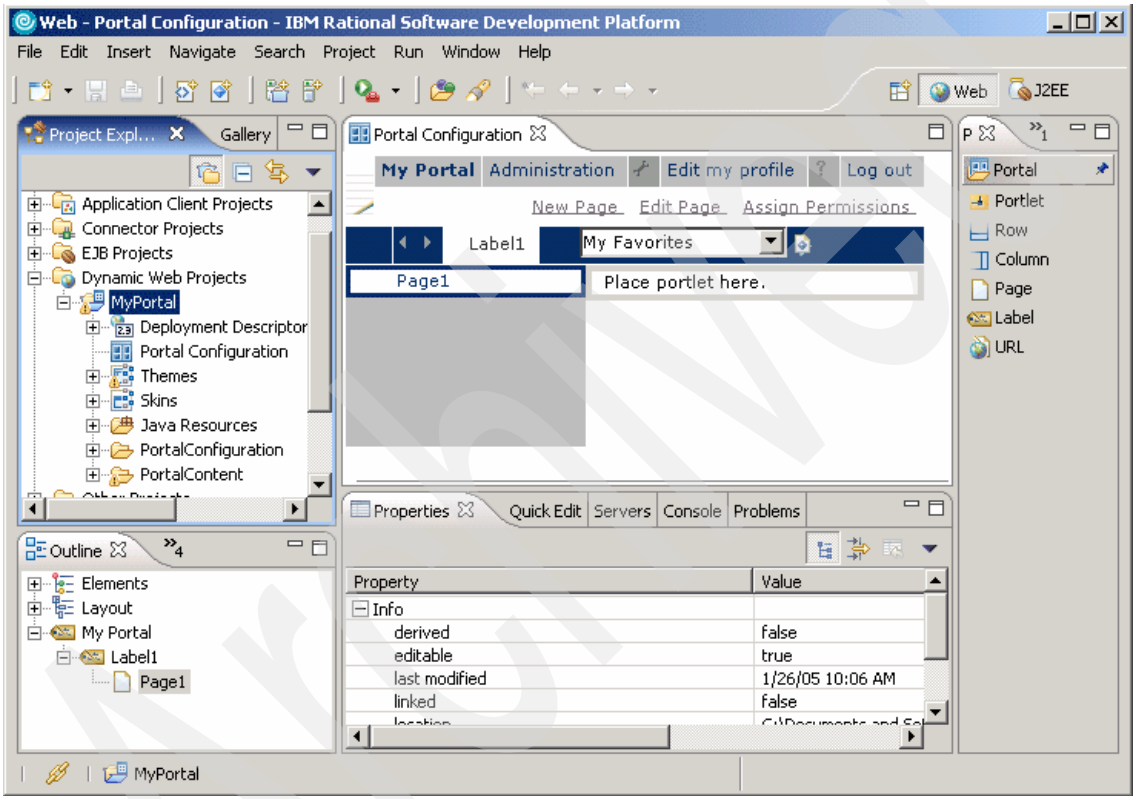


Figure 18-6 Portal Designer

Skin and theme design and editing

A skin is the border around each portlet within a portal page. Unlike themes, which apply to the overall look and feel of the portal, skins are limited to the look and feel of each portlet that you insert into your portal application.

The IBM Rational Application Developer installation includes pre-built themes and skins to use with portal projects. There are also wizards to create new themes and skins. Changing themes and skins was previously done through

portal administration. In addition to these wizards for creating new skins and themes, there are tools that can be used to change or edit these.

Once created, skins and themes will be displayed in the Project Explorer view. Double-click a skin or theme to manually edit it.

New Skin wizard

In addition to using the pre-made skins that came with the installation, you can use the New Skin wizard to create customized skins for your project. Right-click the portal project in the Project Explorer view and select **New** → **Skin**.

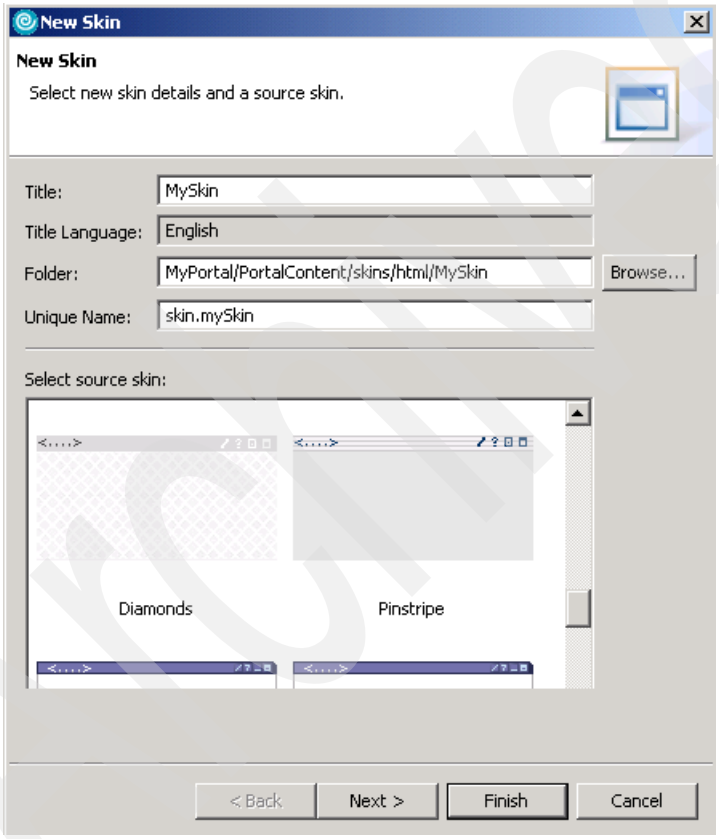


Figure 18-7 New Skin wizard

New Theme wizard

Themes provide the overall look and feel of your portal application. In addition to using the pre-existing themes, you can use the New Theme wizard to create customized themes for your project. Right-click the portal project in the Project Explorer view and select **New** → **Theme** (see Figure 18-8 on page 1001).

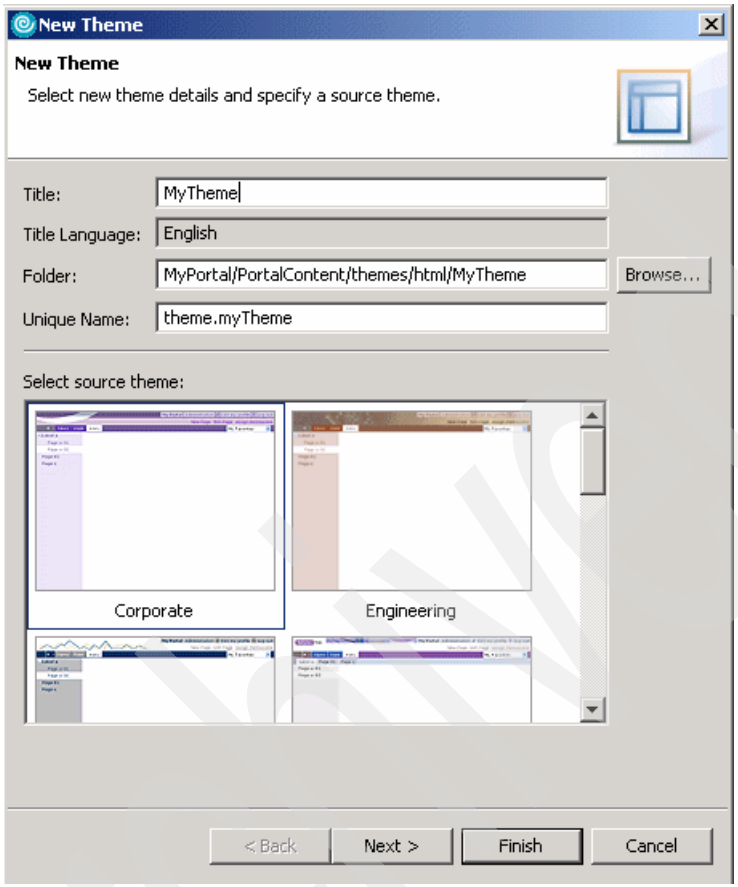


Figure 18-8 New Theme wizard

Deploying portal projects

From Rational Application Developer, you can choose to publish a portal project to a WebSphere Portal server either manually (export) or automatically (deploy).

There are two models of publishing your portal project to WebSphere Portal:

- *Export*: This method is recommended for publishing to a staging or production server. You need to manually copy the packaged portal project to the portal server. Since exporting does not require FTP or copy access to the portal server, there is very little chance of interruption during publishing.

  To export, select **File** → **Export**, then select **Portal Project Deploy Set**. You will need to specify the Portal EAR file and specify a target Portal server. The wizard examines the Portal server so that it can generate specific deploy

information. It then generates a set of files for manually deploying the portal project to the Portal server. These files include:

- WPS.ear
- XmlAccess for deployment portal configuration (contained in the EAR)
- Readme file with instructions for deploying to a server
- WAR files for each portlet project used in the portal project
- XMLAccess script for deploying portlets

The wizard also created a file named DeployInstructions.txt, which is a set of instructions that will guide you through the process of manually deploying your exported project to the server.

Note: Do not attempt to manually deploy the exported files to a portal server other than the one you specify. The export operation contains information from this portal server, and it will not work with other servers.

- *Deploy*: This method automatically publishes a configuration from a portal project to a Portal Server. The deploy method is recommended for publishing to a test, integration, or staging server.

  If you are also transferring the theme and skin files during the deployment, you must also have FTP or copy access to the portal server.

  To deploy a portal, right-click the portal project and select **Deploy Portal**. From here, a wizard will guide you through the deployment process. This will include specifying the portal server to where you are deploying.

  Once you start the deploy process, do not interrupt it. Errors in a project or an unfinished deploy may cause a portal server to become inoperable. As such, you should not deploy directly to a production server. Before running deploy, it is recommended that you back up or image your server.

  Note: If a transfer interruption (for example, network failure) occurs during deployment, there is a slight chance that the portal server will become inoperable.

  Since the portal project does not have any access control information, use administration portlets in the published Portal site to set appropriate access control.

18.2.4 Portal tools for developing portlets

Whether beginning or continuing development of individual portlets and portlet projects, Rational Application Developer has tools that can make this process easier.

Project Interchange files

If you are not using a software configuration management (SCM) system, such as ClearCase or CVS, and you want to share portlet projects with team members or develop a portlet application among multiple computers, you can use the Project Interchange feature.

There are other ways that you can share projects and files including manually copying the project's workspace and importing via WAR files. The recommended method of accomplishing project portability is using the Project Interchange feature. When you export using Project Interchange, the entire project structure is maintained, including metadata files. You can also export several unrelated projects or include required projects for an entire application. Projects exported using this feature can be easily imported into another workspace with a single action.

The Project Interchange mechanism exports projects as they exist in the Workbench, including the project property that specifies the target server runtime for the project. If a user imports the exported project and does not have the same target server runtime installed, the project will not compile. This can be corrected by modifying the target server for the project.

Important: It is important that the IBM Rational Application Developer V6 install path is common for all team members sharing code to avoid absolute library path problems found in projects when importing.

Exporting a Project Interchange file

To export to a Project Interchange file, follow these steps:

1. Right-click the project that you want to export, and select **Export**.
2. Select **Project Interchange**, and click **Next**.
3. Select the projects that you want to export. You have the following options for selection:
   - Click **Select All** to select all projects in the window.
   - Click **Deselect All** to clear all the check boxes.
   - Click **Select Referenced** to automatically select projects that are referenced by any of the currently selected projects.
4. In the To zip file field, enter the full path, including the ZIP file name, where you want to export the selected projects.
5. Click **Finish**.

Import a Project Interchange file

To import a Project Interchange file, do the following:

1. Click **File** → **Import**.
2. Select **Project Interchange** and click **Next**.
3. In the From ZIP file field, click **Browse** to navigate to the ZIP file that contains the shared projects. The Import wizard lists the projects that are in the ZIP file.
4. Select the projects that you want to import. You have the following options for selection:
   - Click **Select All** to select all projects in the window.
   - Click **Deselect All** to clear all the check boxes.
   - Click **Select Referenced** to automatically select projects that are referenced by any of the currently selected projects.
5. Click **Finish**.

Import WAR files

An alternate method of transferring a portlet project to another computer is via a WAR file. WAR files package all pieces of a portlet project into a single file. They are most commonly used to manually deploy portlet projects to Portal Servers.

For development purposes, WAR files can be used to move portlet projects from one computer to another. WAR files are not optimized for this purpose, and moving projects in this way may result in lost meta data or lost time due to any reconfigurations that may be required upon import.

For portlet projects completed with a version of the Portal Toolkit prior to V5.0.2.2, importing by WAR file is the only supported migration path.

To import a project by WAR file, follow these steps:

1. Select **File** → **Import** and select **WAR File**. Then click **Next**.
2. Locate the WAR file to import by using the **Browse** button.
3. The wizard assumes you want to create a new Web project with the same name as the WAR file. Accepting these defaults will create a new project with the same servlet version as specified by the WAR file and in the same location. To override these settings, click **New** and specify the new settings in the Dynamic Web Project wizard.
4. To import a WAR file into an existing Web project, select the project from the Web project drop-down list. If this method is used, the option to overwrite existing resources without warning can be selected.

Archived

5. Click **Finish** to populate the Web project.

Note: When a portlet project is exported to a WAR file, the source files must be included. This procedure is detailed in “Export WAR Files” on page 1020.

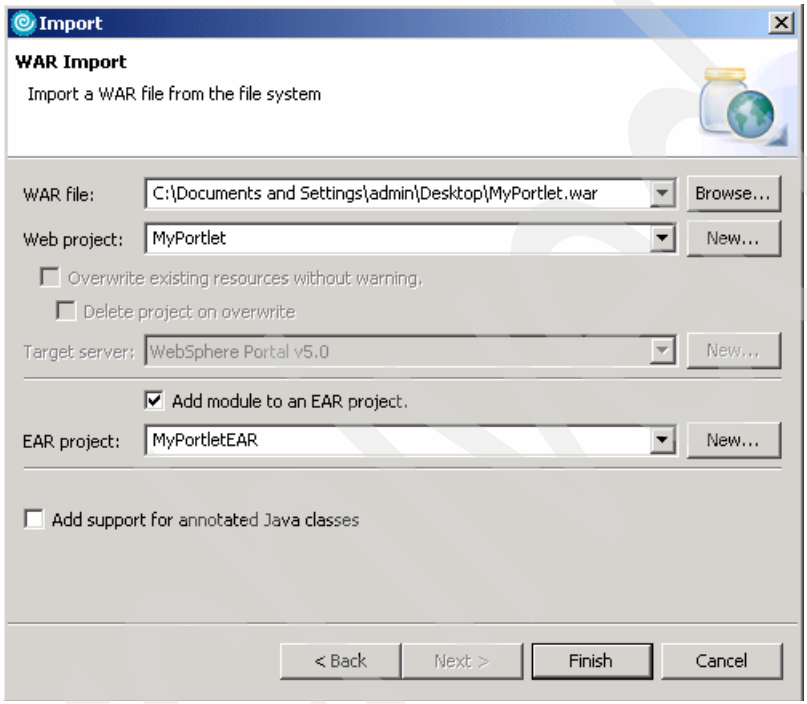


Figure 18-9 WAR Import screen

Portlet Project wizard

Portlet projects are used for developing portlet applications in Rational Application Developer. To create a portlet application, first create a portlet project using the Portlet Project wizard.

The Portlet Project wizard is a very powerful tool that automatically assembles a framework for a portlet project containing all the resources that are necessary for testing, debugging, or deploying a portlet.

To use the Portlet Project wizard, do the following:

1. Select **File** → **New** → **Project**.
2. Select **Portlet Project** and click **Next**.

3. On the first screen in the wizard, enter a project name. You can also specify an alternate project location by clicking the **Browse** button.
4. If you do not want to create the initial portlet definitions in the project, clear the **Create a portlet** check box. Typically, a portlet does not need to be created when importing a WAR file.
5. Click the **Show Advanced** button to see more options (see Figure 18-10 on page 1007).

   The advanced options allow changes to be made to the project's J2EE settings and runtime server environment. The Servlet version specifies the version of Servlet and JSP specifications to be included in your portlet.

   Tip: Use the 2.2 Servlet version if importing a WebSphere Portal V4.x project WAR file. Note that features such as Servlet filters and life cycle event listeners are not supported if this level is chosen.

   Choosing a Servlet version also determines the choice of target servers that appear in the drop-down list. When choosing a server, do not accidentally select any WebSphere Application Server options.

   a. Deselect the **Add module to an EAR project** option only if you do not intend to deploy the portlet. Name an EAR project according to the name of the enterprise application project (EAR) that the portlet project should be associated with during deployment. All portlet applications associated with a single EAR project will run on a single session on a WebSphere Portal Server. You should use the same EAR project for portlet projects that are related.

      The context root is used as the top-level directory in the portlet project when the portlet is deployed. It must not be the same as ones used by other projects.

   b. Ensure that the **Add support for annotated Java classes** check box is selected if using model annotations to generate code in portlet projects.

   c. Click **Next** to continue with the Portlet Project wizard or **Finish** to generate a portlet project based on the defaults associated with a Basic IBM API portlet project.

Archive

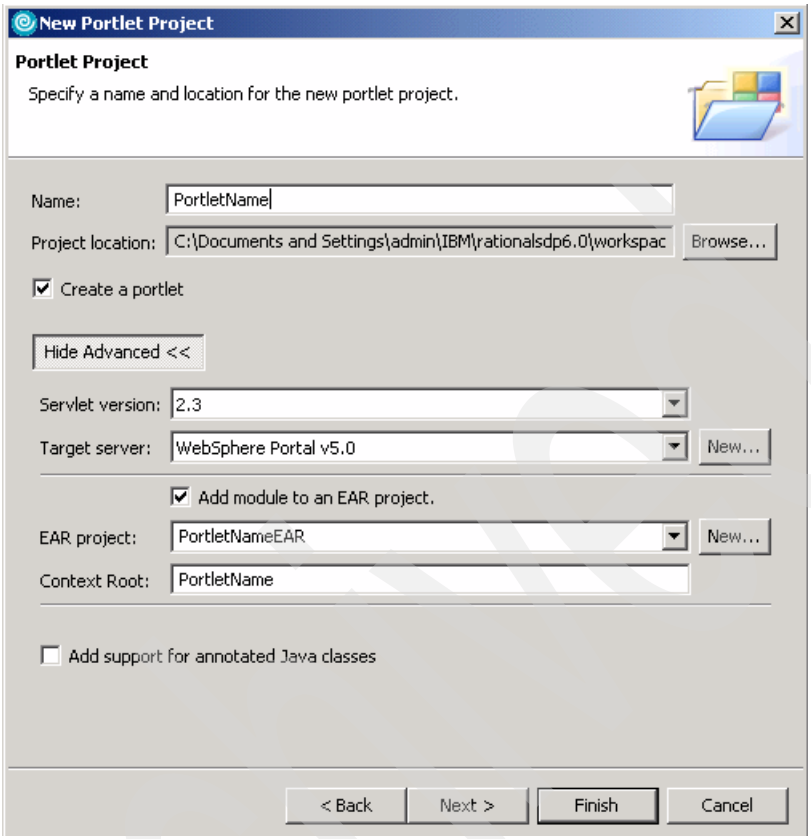


Figure 18-10 Portlet Project wizard

d. On the following screen, select a portlet type that is appropriate for the portlet project. There are four types of portlets (see Figure 18-11 on page 1008):

- Empty portlet: Creates a portlet application that extends the PortletAdapter class with minimum code included. This option is selected if importing a project from a WAR file or when customizing empty portlet projects from scratch.
- Basic portlet: Creates a basic portlet application that extends the PortletAdapter class comprised of a complete concrete portlet and concrete portlet application. It contains a portlet class, sample JSP files that are used when rendering the portlet, and a sample Java bean.
- Faces portlet: Creates a Faces portlet application based on Java Faces technology.

- Struts portlet: Creates a Struts portlet application based on Java Struts technology.

e. When finished selecting options on this screen, click **Next**.

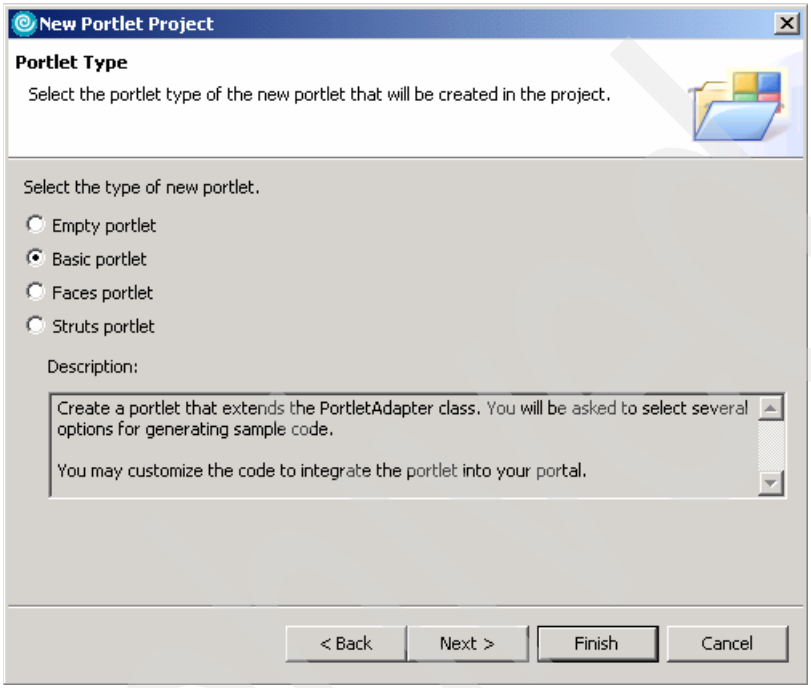


Figure 18-11 Portlet type screen

When creating a Faces portlet, you will be presented with the following screen.

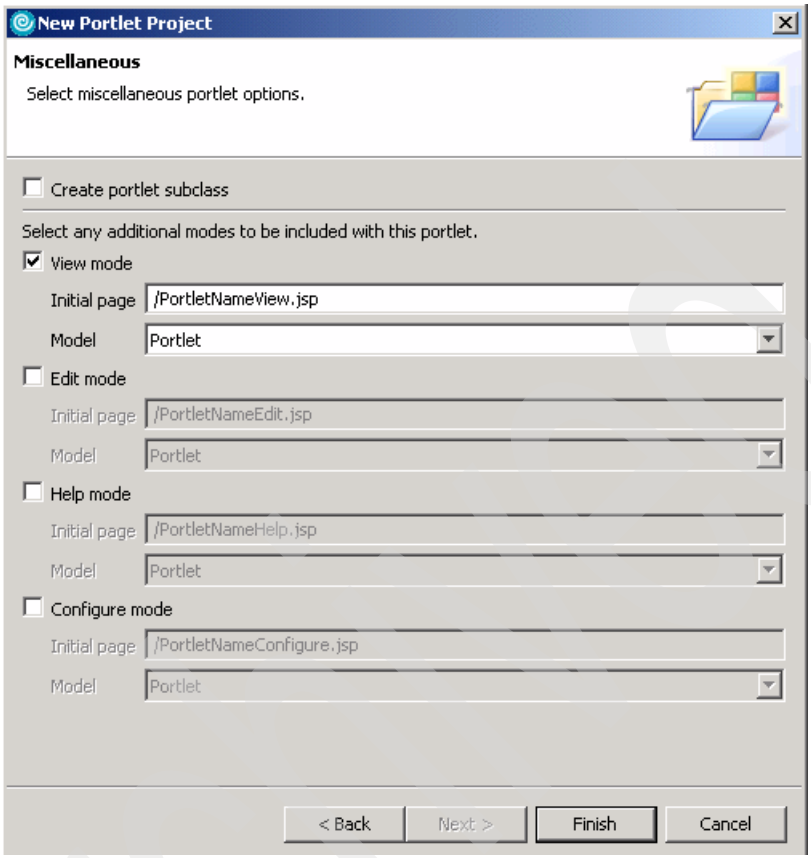


Figure 18-12 Faces portlet miscellaneous screen

When creating a Struts portlet, you will be presented with the following screen.

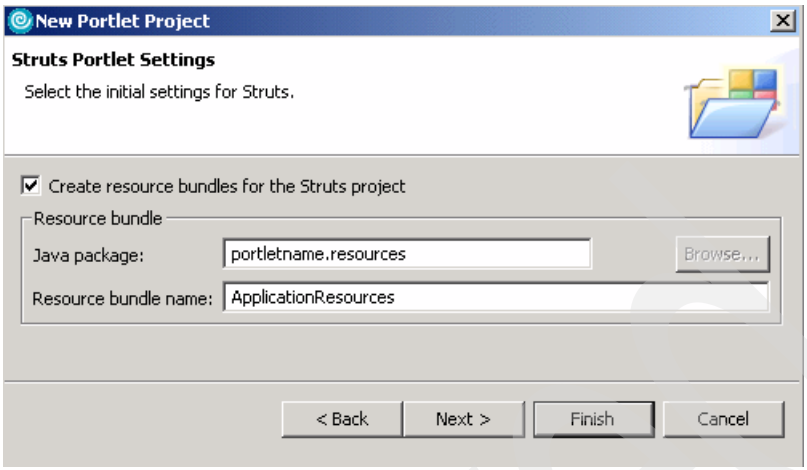


Figure 18-13 Struts Portlet Settings screen

When creating a Basic portlet, you will be presented with the following screen. It allows you to select features that provide additional functionality in the portlet application. Select features as necessary. Deselect the Web Diagram check box if you are creating a Basic or Empty portlet. Select **JSP Tag Libraries** to include the functionality of this technology in the portlet project.

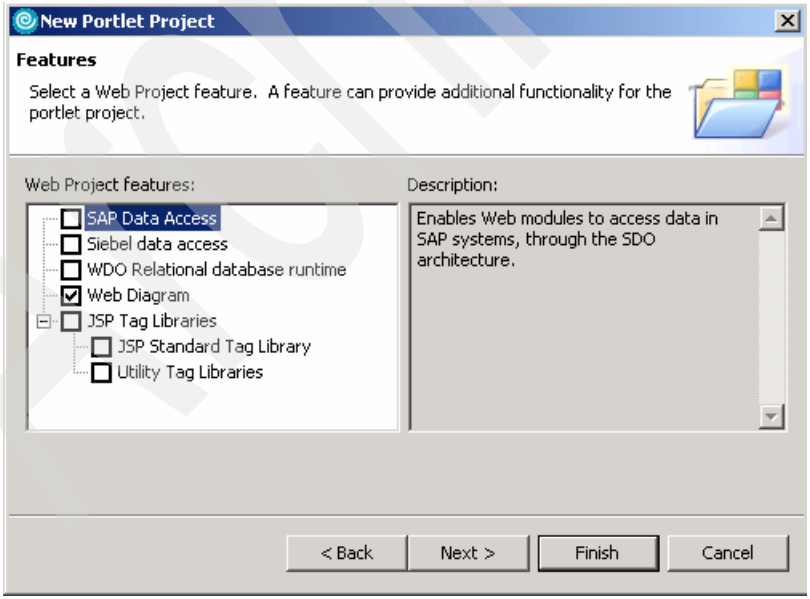


Figure 18-14 Portlet Features screen

On the Portlet Settings screen, update or add any general portlet settings. The application name is the name of the portlet application as used to manage it by the portal administrator. To update this name after generating your portlet project, use the deployment descriptor editor to modify portlet.xml. Modify the Display name of each concrete portlet application.

The portlet name is the name of the portlet. It is also used by the portal administrator. It also can be updated by using the deployment descriptor editor and modifying the Display name of each concrete portlet.

The default locale specifies a default locale to use if the client locale cannot be determined. You can add supported locale using the deployment descriptor editor and adding a locale to each concrete portlet.

The portlet title appears in the portlet title bar. To update this in the future, you can use the deployment descriptor editor to modify the title of each concrete portlet.

Change code generation options can be used to change the package and class prefixes.

Click **Next** to continue. If creating an empty portlet, the Miscellaneous screen (as seen in Figure 18-18 on page 1015) will be shown. If creating a Basic portlet, the Event Handling screen will be shown.

Archived

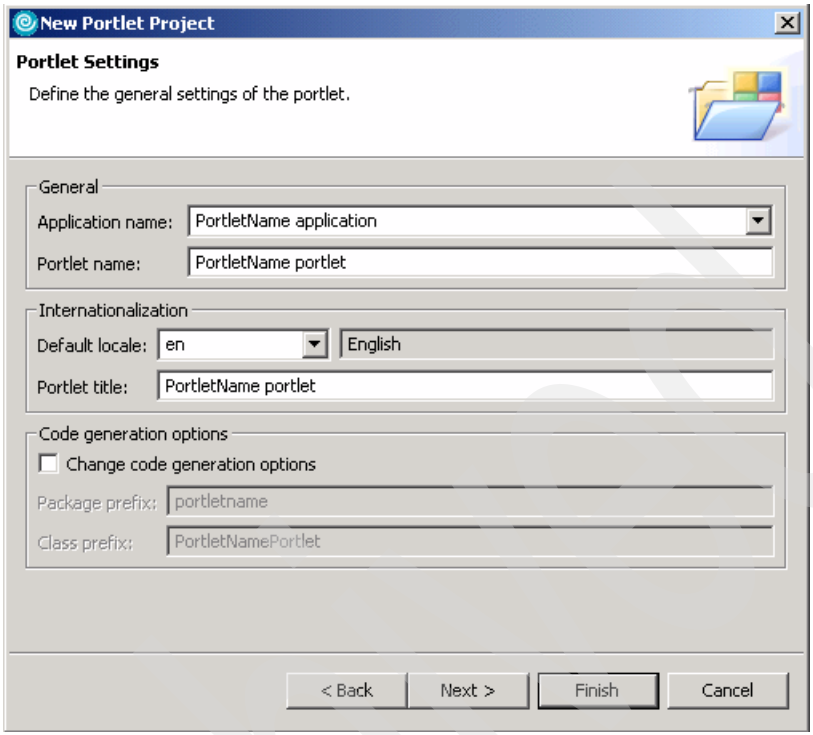


Figure 18-15 Portlet settings screen

On this screen, you have the ability to optionally add event handling capabilities to the portlet application. An action event is sent when an HTTP request is received that is associated with a portlet action. The Add action listener option implements the ActionListener interface to handle action events. The Add form sample option generates code to demonstrate action event handling with a simple form example.

Cooperative portlets provide a model for declaring, publishing, and sharing information with each other using the WebSphere Portal property broker. Cooperative portlets are only available when the Servlet level is 2.3. Cooperative portlets can run on WebSphere Portal V5.x servers. Create a portlet application that extends the PortletAdapter class. “Enable cooperative target” adds a sample WSDL file so that the Click-to-Action target can receive input properties. If you select this option with the Add form sample option in the Action Event handling section of this screen, the generated portlet project will be enabled as a Click-to-Action receiver. It is also possible to create an action handler and form and customize the WSDL file as required. The “Add Click-to-Action sender portlet sample” option adds a simple Click-to-Action sender portlet that is useful to test receiver function and provides sample code. The “Enable cooperative

source” option adds the Click-to-Action tag library directive for JSP files of the Click-to-Action source portlet.

Message events can be sent from one portlet to others if the recipient portlets are placed on the same portal page as the sending portlet. To get a Java class that implements the MessageListener interface, select the **Add message listener** option. The “Add message sender portlet sample” option generates a sample message sender portlet.

To add a function showing events received by listeners in view mode select **Add event log viewer**. To select this option, you need to add at least one of the event listeners. The option to add edit panel allows you to change the default maximum event count while in edit mode.

Click **Next** to continue with the Portlet Project wizard.

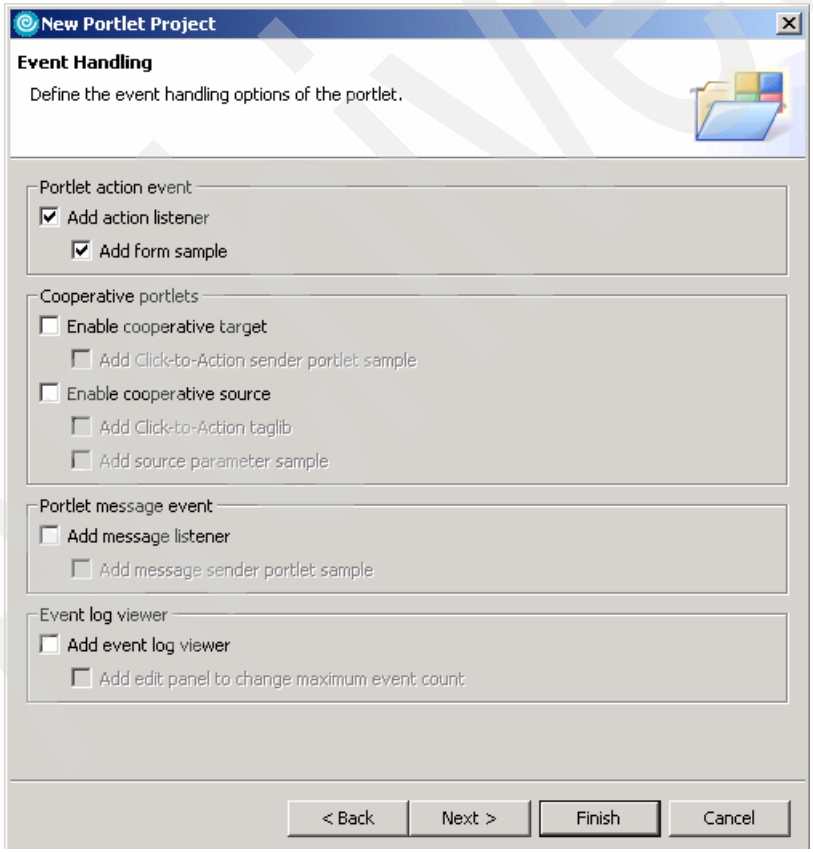


Figure 18-16 Event handling screen

Use the Single Sign-On screen to add sample code to support credential vault handling, which is used to safely store credentials that are used in portlet authentication. Portlets written to extract users' credentials from the credential vault can hide the login challenge from the user. A portlet private credential vault clot stores user credentials that are not shared among portlets. A shared credential vault slot shares user credentials among all of a user's portlets. The administrative credential vault slot allows each user to store their confidential information for accessing administrator-defined resources such as Lotus Notes® databases. A system credential vault slot stores system credentials where the actual confidential information is shared among all users and portlets.

The slot name defines the name of the credential vault slot to store and retrieve the credentials. The Show password option allows a password to be displayed on the screen while in View mode.

Click **Next** to continue with the Portlet Project wizard.

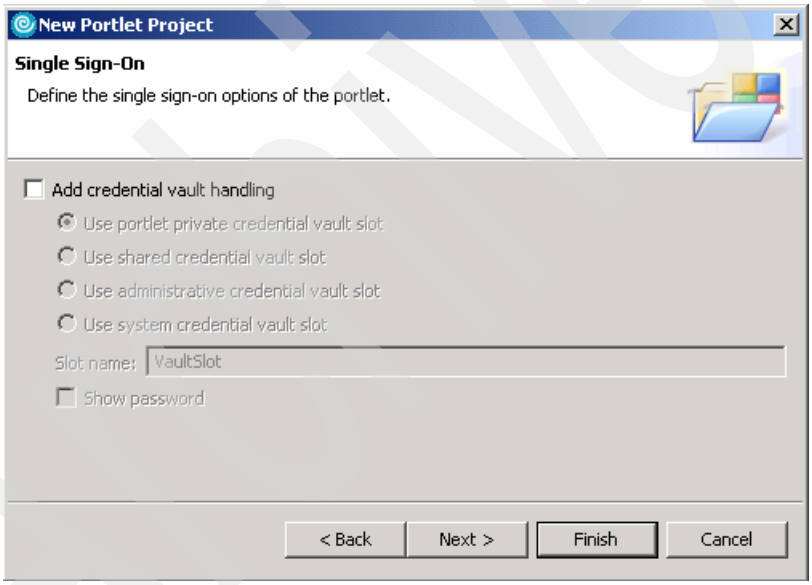


Figure 18-17 Single Sign-On screen

The Miscellaneous screen allows other supported markup languages and portlet modes to be selected. For more information on markup languages, see "Choosing markup languages" on page 994. For more information on modes, see "Portlet modes" on page 988.

Click **Finish** to generate the new portlet project. You may be presented with an option to switch to the Web perspective to work on this project. Click **Yes** if the Confirm Perspective Switch is shown.

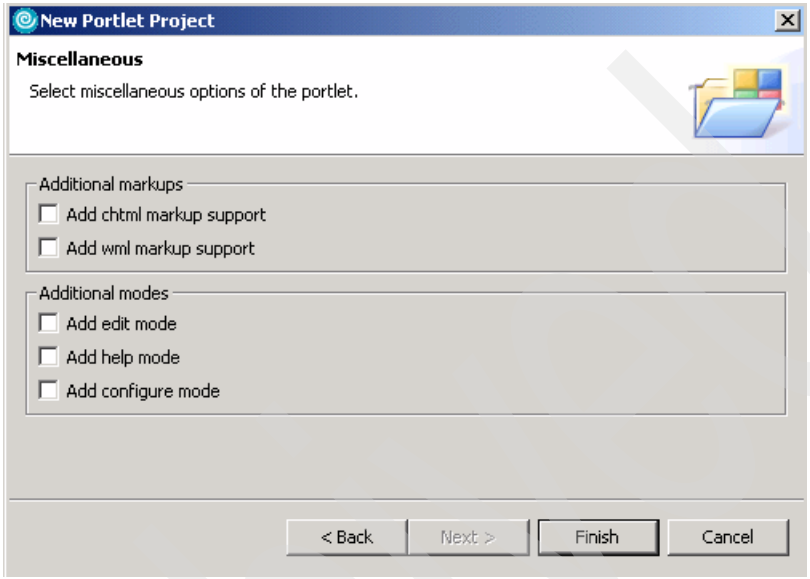


Figure 18-18 Supported markups and modes screen

Web perspective

The Web perspective combines views and editors that assist you with Web application development. This perspective is used to edit the project resources, such as HTML and JSP files, and deployment descriptors that make up the portlet project.

More information on the Web perspective can be found in 4.2.14, “Web perspective” on page 162.

Page Designer

Page Designer is an editor for HTML, XHTML, JSP, and Faces JSP files. It provides three representations of each file: Design, Source, and Preview. Each of these provides a different way to work with a file while it is being edited. You can switch between these by clicking the tabs at the bottom of the editor (see Figure 18-19 on page 1016):

- Design: The Design page provides a visual environment to create and work with a file while viewing its elements on the page.
- Source: The Source page enables you to view and directly work with a file's source code.

- Preview: The preview page shows you how the current page is likely to look when viewed in an external Web browser. Previewing dynamic content requires running the portlet or portal page on a local or remote test server.

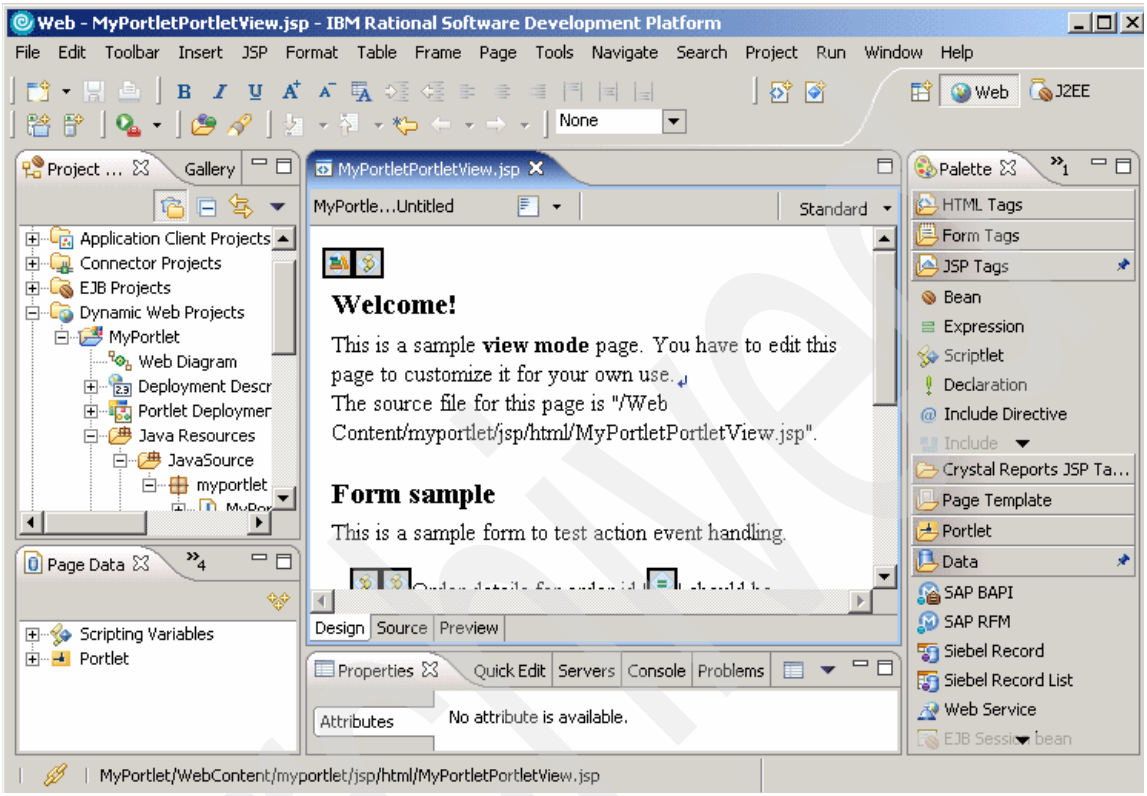


Figure 18-19 Page Designer showing the Design page

18.2.5 Portal tools for testing and debugging portlets

Once development is underway, you will need to test and debug your applications. Rational Application Developer provides many ways for you to do this. When defining a remote (server attach) server for testing, debugging, or profiling a portlet project, you must create and configure the server.

Portal server configuration

Portlet tools provide an additional type of server configuration, called the portal server configuration, which contains the server configuration information needed to publish your portlet application on a WebSphere Portal machine. After it is published, your target portlet will appear on the test page and the debug page of your WebSphere Portal. Source-level debugging is also supported.

Remote server test

When developing portlet projects, you have the option of testing and debugging on a remote server or in a local test environment (as described in the next section). To test portlets on a remote WebSphere Portal Server, you will use this feature.

Before testing portlets with a server attach server, you may need to configure the remote server. See the section titled *Preparing WebSphere Portal for remote testing and debugging* in the product help for more information. The configuration steps detailed in this section are required when performing any of the following tasks.

- Testing or debugging with multiple users to the same remote server
- Testing or debugging a JSR 168 portlet on WebSphere Portal 5.0.2
- Debugging to a remote Server Attach server
- Testing or debugging to a remote server behind a firewall
- Testing or debugging to a remote server running Linux

Important: If multiple users are testing portlets to the same portlet server, ensure that the UIDs of the portlets are unique. Otherwise, when the portlet is installed on the portlet server, it may replace the original portlet using that UID.

- For the IBM portlet API, modify the UID using the portlet deployment descriptor editor.
- For the JSR 168 portlet API, the UID is constructed using the ID attribute of the portlet-app element.
- If the ID attribute is not specified, the UID is generated automatically using the login user ID and project name.

To use the remote server testing feature, do the following:

1. Right-click your portlet project and select **Run** → **Run on Server**.
2. To use an existing server, select **Choose an existing server** and choose a **WebSphere Portal Server Attach** server from the list.
3. To define a new external test server, you will need to use the New Server wizard to configure it. See the section titled *Configuring remote servers for testing portlets* in the product help.
4. Click **Finish**.

   After the server starts and the portal is deployed, the Web browser opens to the URL of the portal application on the external server.

Note: An XML Exception occurs and the server attach fails to start if the project name, the filename, the file directory structure, or the user ID for the WebSphere Portal login name are excessively long.

To correct this, shorten the length of the filename, the file directory structure, or the user ID for WebSphere Portal login at the WebSphere Portal Server Attach server configuration.

WebSphere Portal Test Environment

Rational Application Developer includes the WebSphere Portal Test Environment to locally test and debug portlet applications.

The WebSphere Portal Universal Test Environment allows you to locally test and debug portlets developed with the portal tools from within the Rational Application Developer Workbench. This is similar to running a Java servlet webapp in the WebSphere (Application Server) Test Environment.

The test environment is a WebSphere Portal runtime built on top of the WebSphere Test Environment. By default, the test environment uses Cloudscape as the portal configuration database. This can be configured to use DB2 UDB or Oracle.

When using the WebSphere Portal Test Environment, the server is running against the resources that are in the workspace. It supports adding, changing, or removing a resource from the portlet project without needing to be restarted or republished for these changes to be reflected.

To run your project in the WebSphere Portal Test Environment, right-click the portlet project and select **Run** → **Run on Server**. The Server Selection dialog is displayed. You may either choose to run the application on an existing server or manually define a new server.

To define a new local test server, perform the following steps:

1. Choose the option to **Manually define a server**.
2. Select **WebSphere Portal v5.0 Test Environment** from the list of server types.

   Note: You must have installed the WebSphere Portal V5.0 Test Environment when installing IBM Rational Application Developer for this option to be available.

3. Click **Next**.

4. On the WebSphere Server Configuration Settings page, select one of the following values:
   - Select **Use default port numbers** and set the HTTP port number to use the default HTTP port (9081).
   - Select **Use consecutive port numbers** and set the first port number to use port numbers other than the default numbers used by WebSphere Application Server.

     This setting causes the test environment to use sequential port numbers, starting with the number you specify. You must specify a port number that begins a range of port numbers that are not being used by another application. This option allows you to have an external portal server or WebSphere Application Server running on your system, and allows the test environment to use different port numbers. You can also configure the test environment server's HTTP port numbers by editing the server configuration, as explained below.
5. Click **Finish**.

   Additional options for local servers can be viewed and changed by double-clicking the server in the Servers view. This opens the server configuration editor. You can change any of the settings that were defined previously. In addition, the Portal tab has several additional settings that can be changed to suit your individual configuration.

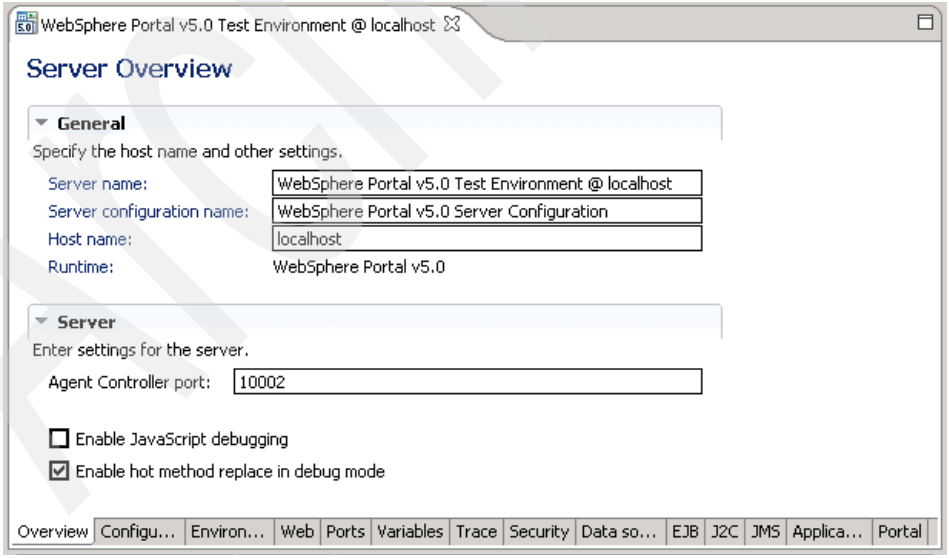


Figure 18-20 Server configuration editor

When you test a portlet on the local test environment server, the default theme and skin are used.

The test environment does not support features that rely on WebSphere Enterprise Edition (personalization and asynchronous rendering of portlets) or LDAP. Transcoding is also not enabled by default. It must be enabled to use a WML device emulator when developing portlets for mobile phones and other devices. See “Choosing markup languages” on page 994 for instructions on how to do this.

When testing or debugging, you may experience the following limitations:

- Help mode does not function correctly in the test environment while using the internal browser. Using an external browser corrects this issue.
- Single sign-on using LDAP is not supported when using the local test environment. LDAP is supported when testing portlets by remotely attaching to another WebSphere Portal server.
- You cannot create new portal users while in debug mode. Use the normal mode to create users.
- Portlet modifications are not previewed correctly when the portlet has been cached by the browser. Logging out and back in to the portal server corrects this.
- If using Linux, you may not be able to start the test environment server without the appropriate user permissions. Users need full permissions on <STUDIO_HOME>/runtimes/portal_v50/cloudscape/wps50.

18.2.6 Portal tools for deploying and managing portlets

When development is complete, these tools will help you to load your completed portlet project onto a WebSphere Portal Server.

Export WAR Files

Exporting a portlet project to a WAR file allows you to install it on a WebSphere Portal server. To export a WAR file for a portlet project, do the following:

1. Right-click the portlet project and select **Export** → **WAR** file. The Export wizard opens.
2. On the WAR Export page, select a destination directory for the WAR file. Enter a name for the WAR file or accept the default.
3. Select the **Export Source files** check box to include source files in the WAR file. When deploying to a WebSphere Portal Server, you do not need to include the source files. If you were exporting a WAR file to continue development on another machine, you would want to select this option.

Archived

4. Select the **Overwrite existing file** option to replace an existing WAR file with the same name.
5. Click **Finish**.

Install the WAR file on the WebSphere Portal server by using the WebSphere Portal administrative tools.

Remote Server Deploy

Remote Server Deploy is a function that allows portlets developed for a WebSphere Portal V5.0 Server to be deployed in an automated fashion.

This functionality is not available for WebSphere Portal V5.1 servers. To deploy portlets to a WebSphere Portal V5.1 server, you must export portlet projects to WAR files, and then install them to WebSphere Portal V5.1 using the WebSphere Portal administration interface. The process of exporting WAR files is described in “Export WAR Files” on page 1020.

To deploy a portlet project to a WebSphere Portal server, follow the steps below.

1. Right-click the portlet project and select **Deploy Portlet**. The Deploy Portlet wizard opens.
2. Select an existing server from the list or create a new one. Then click **Next**.
3. On the Portlets page, define these options for Portlet Overwriting:
   a. Select **Automatically overwrite portlets to replace existing portlets without warning**.
   b. Select the **Update** or the **Remove & Deploy** option.
      - Use the **Update** option to install the portlet, but preserve any customization data that was added in the configure or edit modes.
      - Use the **Remove & Deploy** option to remove and reinstall the portlet. During the removal process, all customization data is also removed, and portlets are removed from any pages where they were already placed. The install process only installs portlets, but does not restore customization data nor place portlets on pages. Use this option if you want to clean up portlet settings, or your portlet is not compatible with the old version.
4. Click **Finish**. Do not interrupt the deployment process.

Archived

Note: An XML Exception occurs and the server attach fails to start if the project name, the filename, the file directory structure, or the user ID for the WebSphere Portal login name are excessively long. To correct this, shorten the length of the filename, the file directory structure, or the user ID for WebSphere Portal login at the WebSphere Portal Server Attach server configuration.

Portal administration

Administrative portlets can be enabled in the server configuration by using the Portal Server Configuration editor described in “Portal server configuration” on page 1016.

You can use the administrative portlets to configure advanced options when running portal and portlet projects.

There are several limitations to using the administrative portlets. You cannot install portlets using the administration portlets. In addition, any changes that are made are reset to the default values the next time the test environment is started.

It is recommended to only use this option when necessary. It affects the performance of the test environment.

To debug portlets in a particular layout, use the test and debug options of a portal project, not the administration portlets in the test environment.

18.2.7 Enterprise Application Integration Portal Tools

Rational Application Developer also includes some tools to help you with Enterprise Application Integration with SAP and Siebel.

Service Data Objects and Tools

Service Data Objects (SDO), the JSR 235 standard, is a new model for representing data, accessing persistent data, and passing data between tiers. It provides a single, consistent interface to any data source.

The JSF tools for SDO in IBM Rational Application Developer provide minimal or zero coding for building dynamic data-bound JSPs.

IBM has included SDO mediators for applications, including SAP and Siebel, that are supported on WebSphere Portal V5.1 servers.

SDO mediators are added to portlets through drag-and-drop from the Palette view and the Page Data view.

Archived

Business Process Portlet Development Tools

The portal tools in IBM Rational Application Developer V6.0 also include support for Business Process Execution Language (BPEL)-based business process portlet development. These portlets are supported on WebSphere Portal V5.1 servers.

To use these tools, process designers develop business processes by using the BPEL editor on WebSphere Studio Application Developer - Integration Edition V5.1.1 and test them in the WebSphere Test Environment.

You can then import the resultant business processes as JAR files to develop and compile task processing portlets using the remote server attach function for testing and debugging portlets.

18.2.8 Coexistence and migration of tools and applications

When installing multiple versions of IBM development software and working with portal and portlet projects developed with different versions of development software, there are some important issues to consider.

WebSphere Studio and Rational Application Developer

WebSphere Studio Application Developer and IBM Rational Application Developer can coexist with regards to the Portal Toolkit 5.0.x on WebSphere Studio 5.x.

Portlet Projects (Portal Toolkit 5.0.2.2 and later)

Portlet projects completed using the Portal Toolkit V5.0.2.2 will be migrated automatically to Rational Application Developer V6.0 Portal Tools by either migrating the Portal Toolkit workspace or importing the project using the Project Interchange feature.

During migration of Portal Toolkit V5.0.2.2 projects, some additional changes take place:

- The target server is set to WebSphere Portal V5.0, if no target server is set to the project.
- The portlet build path is corrected.
- A portlet project nature is added.

Portlet projects (Portal Toolkit earlier than V5.0.2.2)

If migrating portlet projects from earlier versions of Portal Toolkit (prior to V5.0.2.2), the best practice is to export your portlet projects to WAR files and

then import the WAR files into new portlet projects within IBM Rational Application Developer V6.0.

Manually migrate your portlet projects by following these directions:

1. Export the existing project to a WAR file, and include its source files.
   a. Right-click the project and select **Export**.
   b. Select **WAR file** and **Export source files** and click **Finish**.
2. Import the portlet WAR file into a new portlet project:
   a. In the Portal Tools for Rational Application Developer V6.0, create a new empty portlet project.
      i. Select **File** → **New** → **Project** → **Portal** → **Portlet Project** or **Portlet Project (JSR 168)**.
      ii. Deselect **Create a portlet**.
      iii. Click **Show Advanced**.
      iv. If you are importing a WebSphere Portal V4.2 portlet, select **2.2** as the servlet version.
      v. Select **WebSphere Portal v5.0** as the target server, and click **Finish**.
   b. Import the WAR file to this new empty portlet project.
      i. Select **File** → **Import**.
      ii. Select **WAR file** and specify the WAR file from the portlet project that you exported.
      iii. Select the newly created empty portlet project.
      iv. Select **Overwrite existing resources without warning**.
      v. Do not select Delete project on overwrite.
      vi. Delete the TLD file.

      It is recommended that you delete the portlet TLD file from the project if it exists. Otherwise, you will get a warning message when you rebuild the project. Leaving it may cause a problem when the portlet project is deployed to WebSphere Portal and the TLD file of the portlet is different from the file in the server.

Note: If you are migrating a WebSphere Portal V4.2 portlet, you will need to migrate this migrated portlet project to WebSphere Portal V5.x. Backward compatibility of portlet projects is not supported.

Archived

18.3 Portal development scenario

To gain an understanding of the portal development process, this scenario demonstrates how the portal tools can be used to create a portal site.

The portal development scenario is organized into the following tasks:

- Prepare for the sample.
- Add and modify a portal page.
- Create and modify two portlets.
- Add portlets to a portal page.
- Run the project in the test environment.

Note: The sample code described in this chapter can be completed by following along in the procedures documented. Alternatively, you can import the sample Portal code provided in the c:\6449code\portal\Portal.zip Project Interchange file. For details refer to Appendix B, “Additional material” on page 1395.

When importing the Project Interchange file, we found some errors when using the IBM WebSphere Portal V5.0.2 Test Environment due to issues related files outside of the scope of the Project Interchange packaging (specifically, themes and related JSPs).

18.3.1 Prepare for the sample

Prior to working on the portal development scenario, ensure that you have prepared the environment by installing the Portal Tools and WebSphere Portal Test Environment (V5.0.2 or V5.1).

Install the Portal Tools

For details on installing the Portal Tools as a component of the Rational Application Developer installation, refer to “Rational Application Developer installation” on page 1372.

Install WebSphere Portal Test Environment

For the scenario in this chapter, you can install either the V5.0.2 or V5.1 WebSphere Portal Test Environments (sample applies to both).

For more information on installing the WebSphere Portal Test Environments refer to the following:

- IBM WebSphere Portal V5.0.2 Test Environment

  Refer to “WebSphere Portal V5.0 Test Environment installation” on page 1376

- IBM WebSphere Portal V5.1 Test Environment

  Refer to “WebSphere Portal V5.1 Test Environment installation” on page 1377.

Install the Rational Application Developer V6 Interim Fix

We recommend that you install the latest Rational Application Developer interim fixes. For details refer to “Rational Application Developer Product Updater - Interim Fix 0004” on page 1380.

Start Rational Application Developer

To begin, start the IBM Rational Application Developer Workbench. By default, click **Start** → **Programs** → **IBM Rational** → **IBM Rational Application Developer V6.0** → **Rational Application Developer**.

Once Developer is open, you will begin using the Portal Tools to develop a portal site, as instructed below.

18.3.2 Create a portal project

To create a portal project, do the following:

1. Select **File** → **New** → **Project**.
2. When the New Project dialog appears, select **Portal Project** and then click **Next**.
3. If prompted with the dialog displayed in Figure 18-21, click **OK** to enable portal development capabilities.

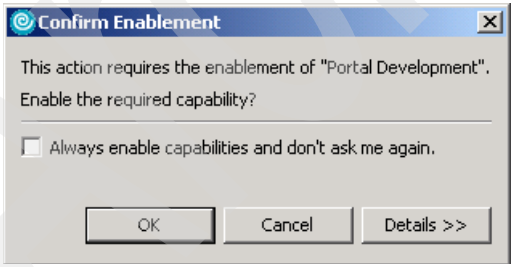


Figure 18-21 Enable portal development

4. When the Portal Project dialog appears, enter `MyPortal` in the Name field, click **Show Advanced** to see more options (we accepted the defaults), and then click **Next**.
5. When the Select Theme dialog appears, select the desired theme. For example, we select the **Corporate** theme and then clicked **Next**.

Archived

6. When the Select Skin dialog appears, select the desired skin. For example, we selected the **Outline** skin and then clicked **Finish**.

   This will generate the framework for the portal site.

7. If prompted to change to the Web perspective as seen in Figure 18-22, click **Yes**.

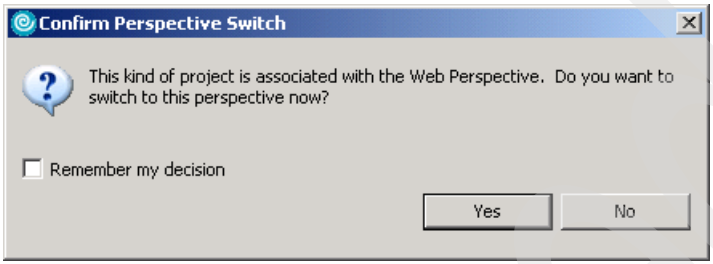


Figure 18-22 Confirm perspective switch

18.3.3 Add and modify a portal page

This section describes how to add and modify a portal page for a portal site.

To add a new portal page, do the following:

1. Drag-and-drop the **Page** button from the Palette and place it in the same column as the existing Page1 was created (see Figure 18-23 on page 1028).
2. Click **New Page**.
3. Select the **Title** tab from the Properties view at the bottom of the window.
4. Change the page names.

   Change the names of the pages *Page1* and *New Page* to `Top Page` and `Bottom page`, respectively.

Archive

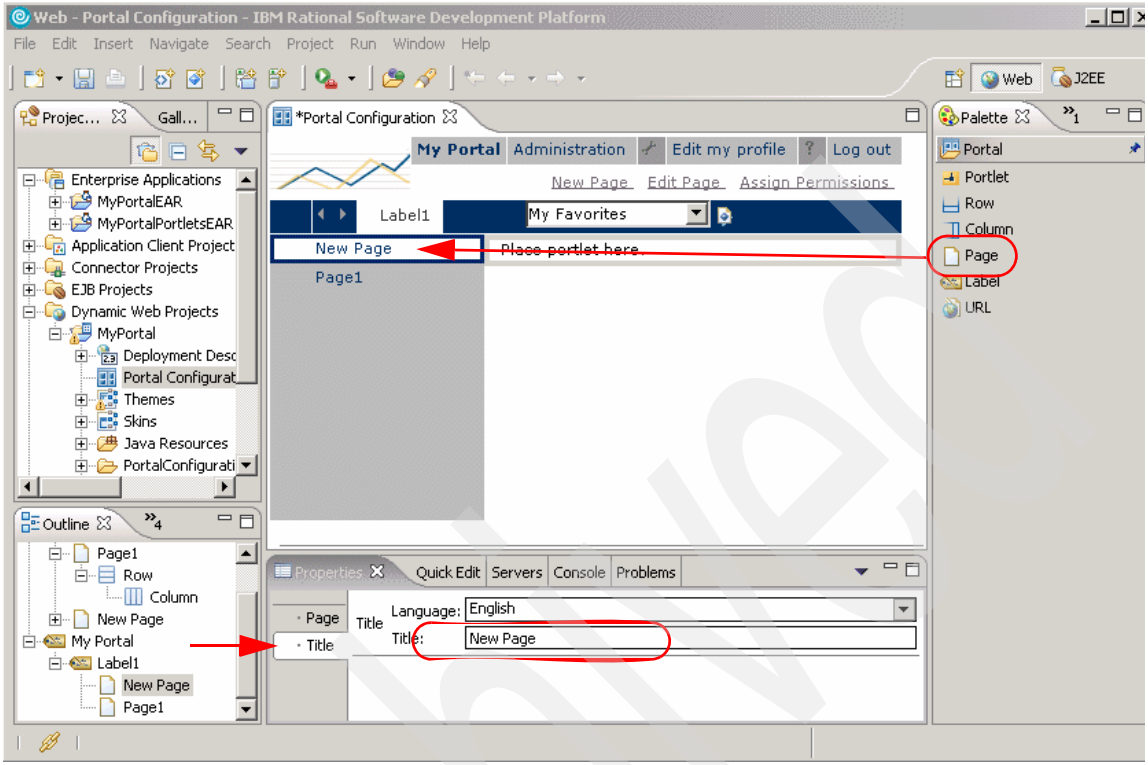


Figure 18-23 Insert a new page and modify title

5. Add a label to the page (see Figure 18-24 on page 1029).
   a. Drag and drop the **Label** button from the Palette view, and place it to the right of the existing Label1.
   b. Drag and drop the **Page** button from the Palette view onto the New Label to add a new page on which to place portlets.

Archive

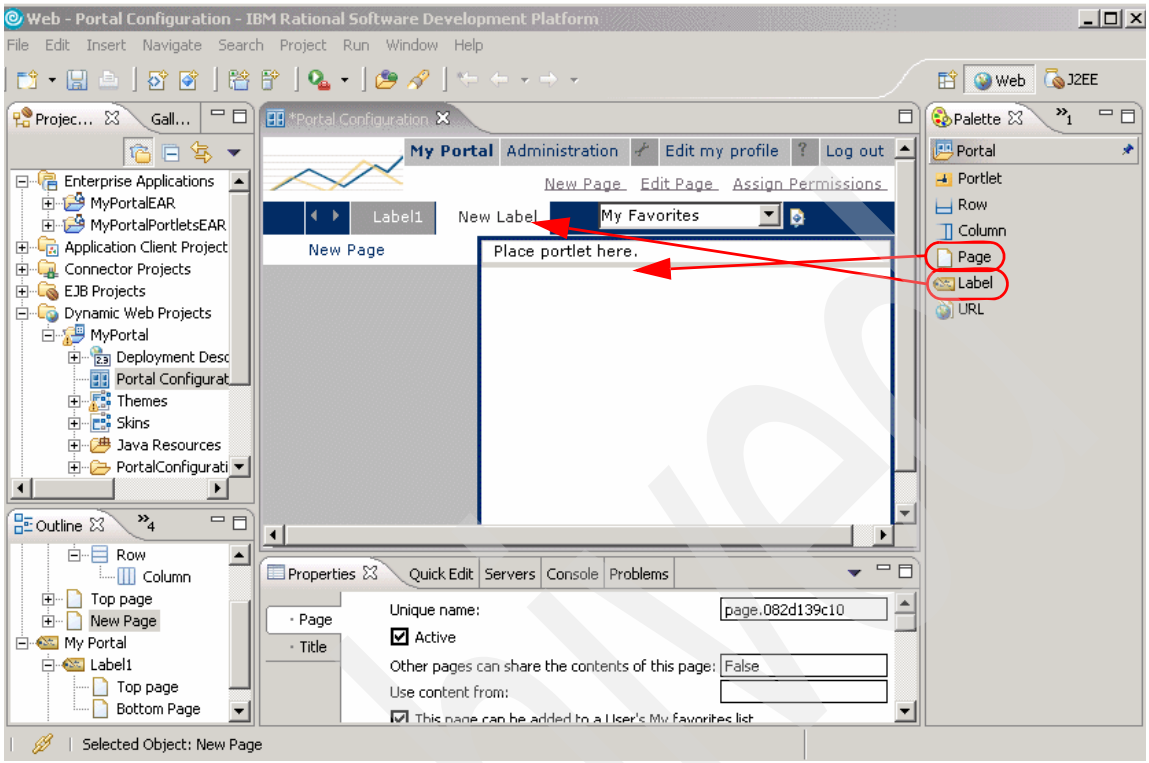


Figure 18-24 Add a new label and page

6. Change the label names (see Figure 18-25).

   In the same way that the titles of pages are modified, change the names of the two labels on the portal site. Name the right label `Right Label` and the left label `Left Label`.

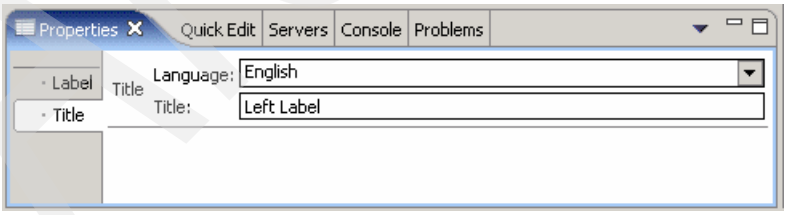


Figure 18-25 Changing label titles

7. Click **File** → **Save All** to save all the changes you have made.
8. By adding labels and pages, you are able to alter the navigational structure of the portal. You can also view an outline view of this structure by looking at the

Outline view, which appears in the lower left corner of the Workbench (see Figure 18-26).

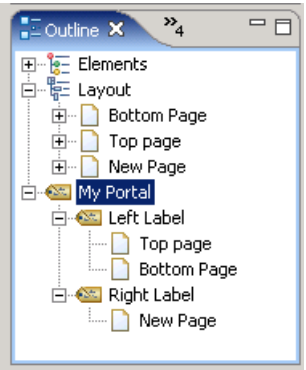


Figure 18-26 Outline view

18.3.4 Create and modify two portlets

Now that the portal site and its navigational structure have been defined, we can add content. Content is added to portals by placing portlets on each of the pages. We will create two portlet projects for our example in this section.

Create the first portlet

To create the first portlet, do the following:

1. Click **File** → **New** → **Project**.
2. When the New Project dialog appears, select **Portlet Project** and click **Next**.
3. Enter `Basic Portlet` in the Name field and click **Next**.
4. When the Portlet Type dialog appears, select the **Basic portlet** type and click **Next**.
5. When the Features dialog appears, uncheck **Web Diagram** and click **Next**.
6. When the Portlet Settings dialog appears, we accepted the default portlet settings and click **Next**.
7. When the Event Handling dialog appears, do the following and then click **Next**:
   - Uncheck **Add form sample**.
   - Uncheck **Add action listener**.
8. When the Single Sign-on dialog appears, accept the default values for credential vault handling and click **Next**.

Archived

9. When the Miscellaneous dialog appears, check **Add edit mode** on the miscellaneous settings page, and click **Finish** to generate your portlet code.

   The portlet's view mode JSP is now displayed in the Workbench to be edited.

10. Expand **Dynamic Web Projects** → **MyPortal** in the Project Explorer view (see Figure 18-27).

    Under this directory are all the resources associated with the portlet including the supporting JSP files, Java classes, and the portlet's deployment descriptor. You can double-click any resource to edit it in its default editor.

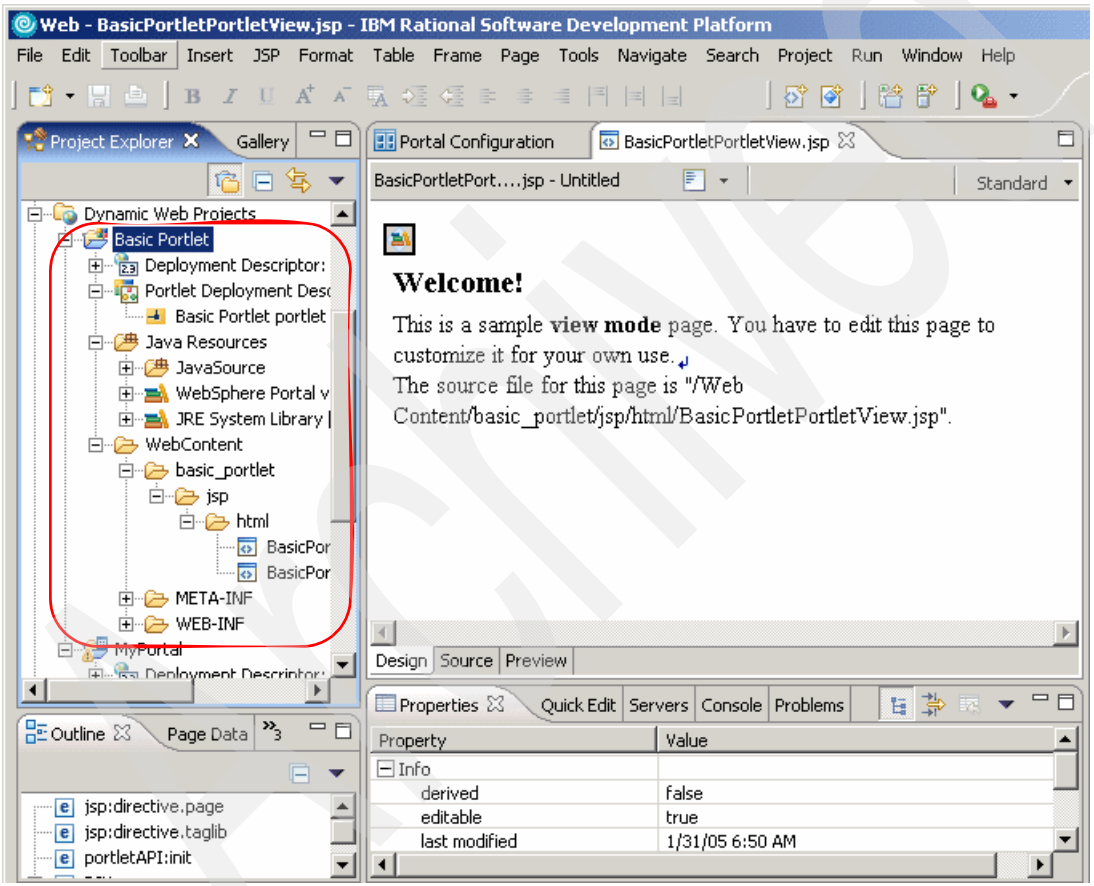


Figure 18-27 Basic Portlet project in the Workbench

Create the second portlet

Now, create a second portlet for the portal site. This portlet will process a form and display the results.

1. Select **File** → **New** → **Project**.

2. When the New Project dialog appears, select **Portlet Project**, and click **Next**.
3. Enter `Form Portlet` in the Name field and click **Next**.
4. When the Portlet Type dialog appears, select **Basic portlet** and click **Next**.
5. When the Features dialog appears, uncheck **Web Diagram**, check the **JSP Tag Libraries**, and then click **Next**.
6. When the Portlet Settings dialog appears, we accepted the default portlet settings, and clicked **Next**.
7. When the Event Handling dialog appears, accept the defaults on the event handling screen. The **Add action listener** and **Add form sample** options should be selected. Click **Next**.
8. When the Single Sign-on dialog appears, accept the default values for credential vault handling and click **Next**.
9. When the Miscellaneous dialog appears, accept the default and click **Finish** to generate your portlet code.
10. In the FormPortletPortletView.jsp file that is displayed on your screen, delete Welcome!
11. Figure 18-28 on page 1033 displays a sample view mode page. You have to edit this page to customize it for your own use.

    The source file for this page is as follows, leaving only the form sample to be displayed on this page:

    `/WebContent/form_portlet/jsp/html/FormPortletPortletView.jsp`

Archived

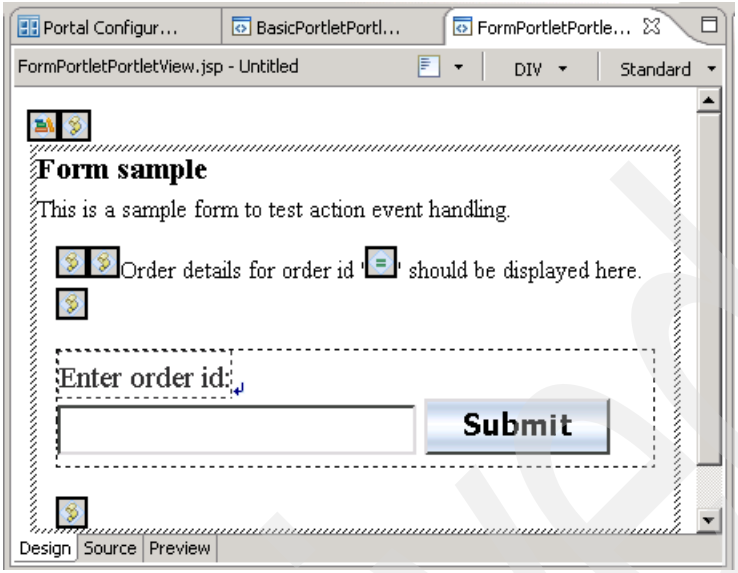


Figure 18-28 Modified second portlet

12. Click **File** → **Save All** to save all the changes made to the portlet projects.

18.3.5 Add portlets to a portal page

Now return to the Portal Configuration editor used in 18.3.3, “Add and modify a portal page” on page 1027, to add portlets to a portal page.

1. Expand **Dynamic Web Projects** → **MyPortal**.
2. Double-click **Portal Configuration** to open in the editor.
3. Add portlets to the Left Label of the Top Page.
   a. Select the **Left Label** of the Top Page.
   b. Drag and drop the **Column** button from the Palette view into the area of the Top Page that says **Place portlet here**.

      By doing this, the layout of the page is changed to accommodate two portlets side-by-side, as seen in Figure 18-29 on page 1034.

Archive

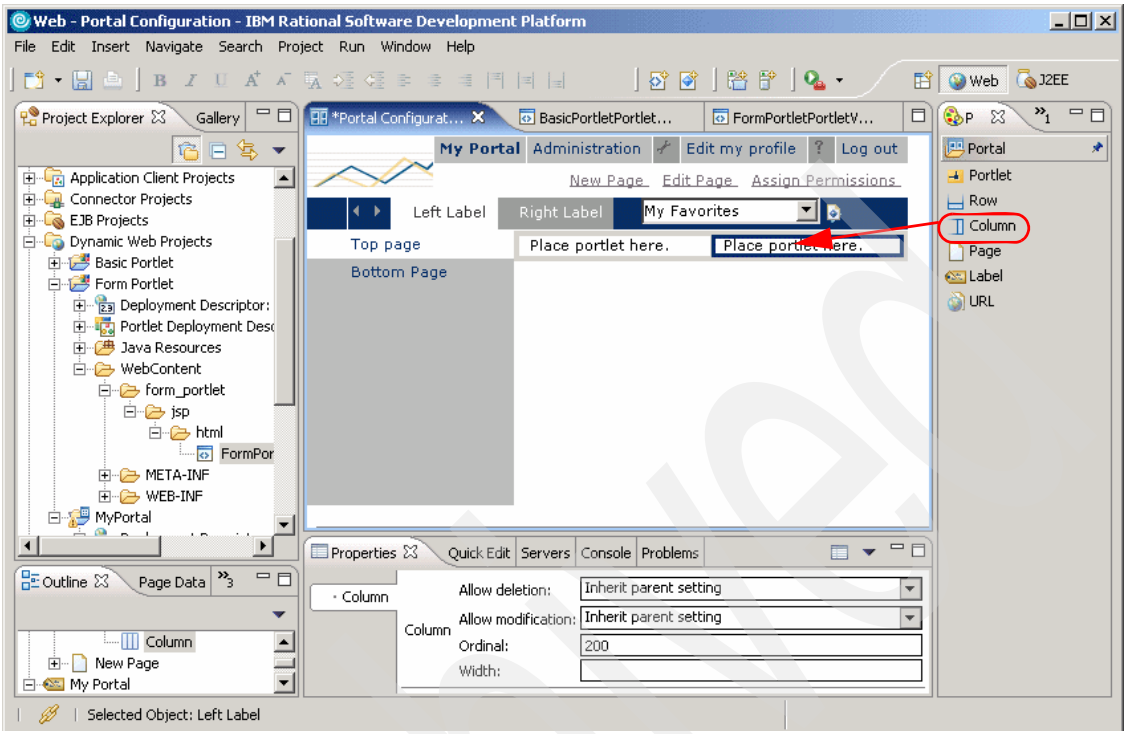


Figure 18-29 Adding a column

c. Right-click the left column and click **Insert Portlet** → **As Child**.

d. Select the **Basic Portlet portlet**, as seen in Figure 18-30 on page 1035, and click **OK**.

Archive

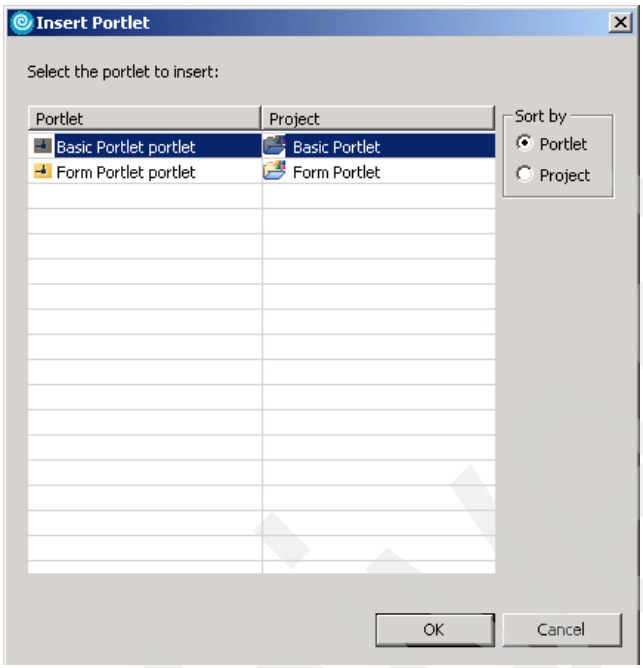


Figure 18-30 Select portlet to insert

4. Add portlets to the Left Label of the Bottom Page.
   a. Select the **Left Label** of the Bottom Page.
   b. Drag and drop the **Column** button from the Palette view into the area of the Bottom Page that says **Place portlet here**.
   c. Right-click the right column and click **Insert Portlet** → **As Child**.
   d. Select the **Form Portlet portlet** and click **OK**.
5. Perform the same action to insert the *Basic Portlet portlet* to the Left Label of the Bottom Page (see Figure 18-31 on page 1036).

Archive

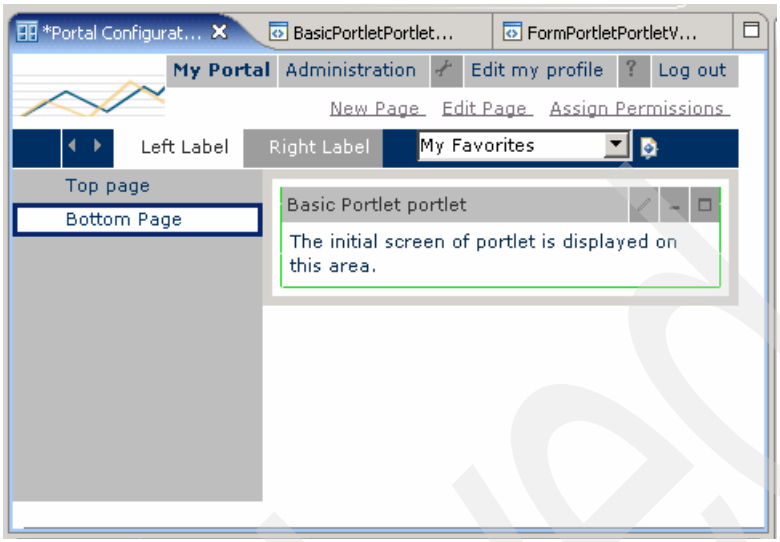


Figure 18-31 Basic portlet on the bottom page of the left label

6. Now click the **Right Label**.
7. Insert the Form Portlet onto this page (see Figure 18-32).

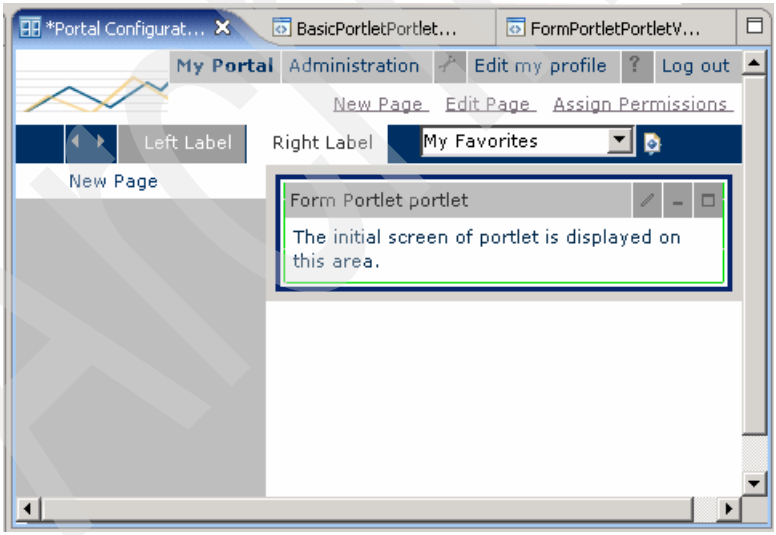


Figure 18-32 Inserting the form portlet into the right label new page

8. Click **File** → **Save All** to save all the changes made to your portal site.

18.3.6 Run the project in the test environment

Now you can run and test the project in the WebSphere Portal Test Environment. This section assumes that you have not previously defined a WebSphere Portal Test Environment server and will configure a server for you as part of the procedure.

1. Open the Web perspective.
2. Expand **Dynamic Web Projects**.
3. Right-click **MyPortal**, and select **Run** → **Run on Server**, as seen in Figure 18-33.

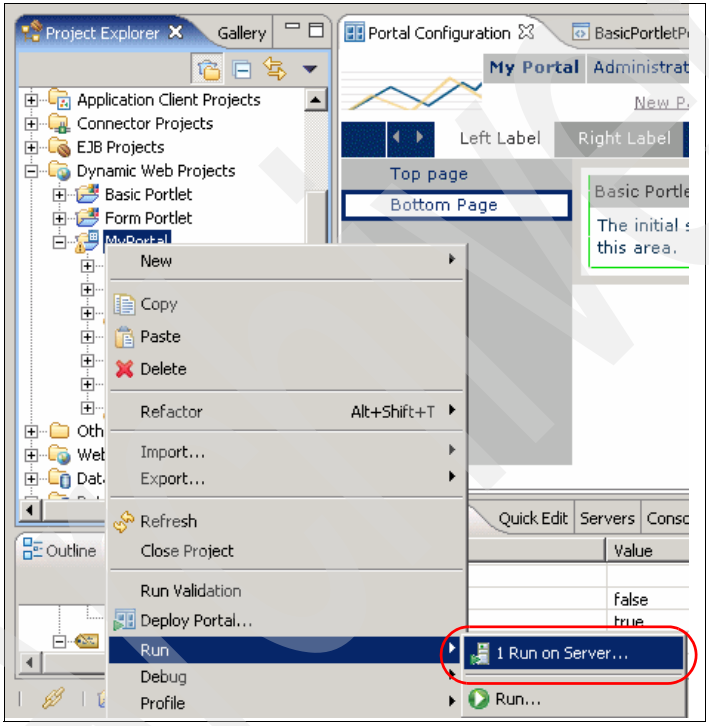


Figure 18-33 Run on Server

4. When the Define a New Server dialog appears, select **Manually define a server**, and select the desired WebSphere Portal Test Environment (V5.0 or V5.1). For example, we selected **WebSphere Portal v5.0 Test Environment** and clicked **Next**.
5. When the WebSphere Server Configuration Settings dialog appears, we accepted the default port (9081) and clicked **Next**.

6. When the Add and Remove Projects dialog appears, select each of the following projects and click **Add**:
   - Form PortletEAR
   - Basic PortletEAR
   - MyPortalEAR
7. When done adding the projects to the Configured projects column, you should have four projects (MyPortalEAR, Form PortletEAR, Basic PortletEAR, MyPortalPortletsEAR) associated with your MyPortal project so that they can run on the server. Click **Finish**.
8. Click **OK** if you receive the Repair Server Configuration dialog window (see Figure 18-34). This indicates that your portlets will be added to the server so that they run in your portal project.

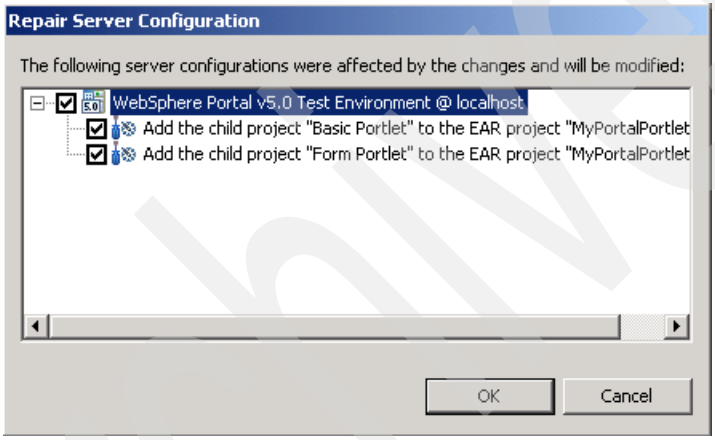


Figure 18-34 Repair Server Configuration dialog

9. The server will now start, and your portal site will load in the Web browser, as seen in Figure 18-35 on page 1039.

   Test the portal site.

   a. Navigate the portal site using the labels and page links.
   b. Enter edit mode on the Basic Portlet by clicking **Edit Page**.
   c. Submit a value using the form.

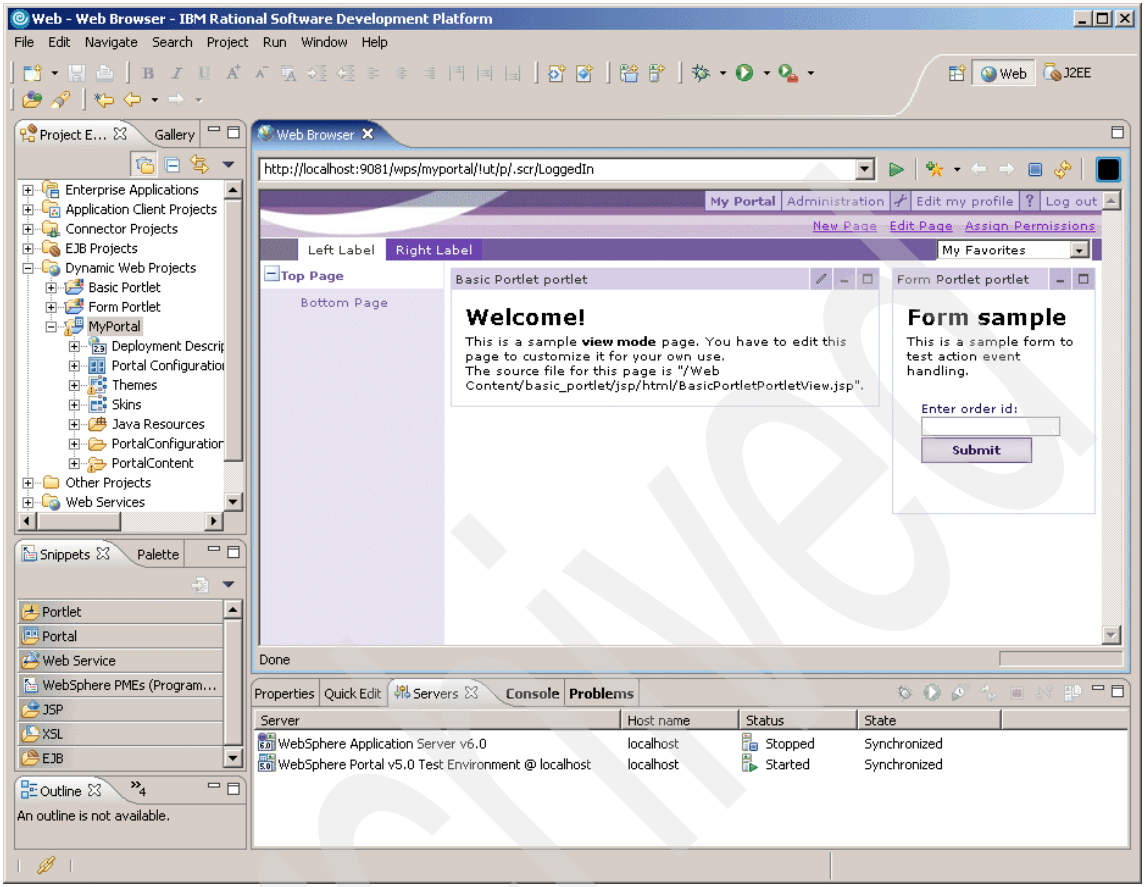


Figure 18-35 My Portal project in Web browser within IBM Rational Application Developer

Archive

Archived

Part 3

Test and debug applications

Archive

© Copyright IBM Corp. 2005. All rights reserved.

Archived

19

Servers and server configuration

Rational Application Developer provides support for testing, debugging, profiling, and deploying Enterprise applications to local and remote test environments.

To run an Enterprise application or Web application in Application Developer, the application must be published (deployed) to the server. This is achieved by installing the EAR project for the application into an application server. The server can then be started and the application can be tested in a Web browser, or by using the Universal Test Client (UTC) for EJBs and Web Services.

This chapter describes the features and concepts of server configuration, as well as demonstrates how to configure a server to test applications.

The chapter is organized into the following sections:

- Introduction to server configuration
- Configure a WebSphere V6 Test Environment
- Add a project to a server
- Remove a project from a server
- Publish application changes
- Configure application and server resources

© Copyright IBM Corp. 2005. All rights reserved.

19.1 Introduction to server configuration

IBM Rational Application Developer V6.0 includes integrated test environments for IBM WebSphere Application Server V5.0/V5.1/V6.0 and IBM WebSphere Portal V5.0.2/V5.1, as well as support for many third-party servers obtained separately. In Version 6 the server configuration is the same for IBM WebSphere Application Server V6.0 (base), Express, and Network Deployment Editions. One of the many great features of V6 is the ability to simultaneously run multiple server configurations and test environments on the same development node where Rational Application Developer is installed.

In previous releases, there was a separate server configuration for WebSphere Application Server and only the base and Express servers were supported. In IBM Rational Application Developer V6.0, the new architecture supports deploying and testing on IBM WebSphere Application Server V6.0 (base), Express, and Network Deployment Editions. Also, test environment configuration for the WebSphere Application Server V6.0 Test Environment is done from the WebSphere Administrative Console.

All communication with all WebSphere V6 servers occurs through JMX calls over SOAP (start, stop, install applications, set status) using port 8880 by default.

When using Rational Application Developer it is very common for a developer to have multiple test environments or server configurations, which are made up of workspaces, projects, and preferences, and supporting test environments (local or remote).

Some of the key test environment configuration items include:

- Multiple Workspaces with different projects, preferences, and other configuration settings defined
- Multiple Rational Application Developer Test Environment servers configured
- When using WebSphere Application Server V6.0 test environments, multiple profiles, each potentially representing a different server configuration

For example, a developer may want to have a separate server configuration for WebSphere Application Server V6.0 with a unique set of projects and preferences in a workspace and server configuration pointing to a newly created and customized WebSphere Application Server V6.0 profile. On the same system, the developer can create a separate portal server configuration with unique portal workspace projects and preferences, as well as a WebSphere Portal V5.1 Test Environment. This chapter describes how to create, configure, and run both a WebSphere Application Server V6.0 and WebSphere Portal V6.1 Test Environments on the same development system.

Archive

19.1.1 Supported test server environments

IBM Rational Web Developer V6.0 and IBM Rational Application Developer V6.0 support a wide range of test server environments for running, testing, and debugging application code.

In IBM Rational Application Developer V6.0, the integration with the IBM WebSphere Application Server V6.0 for deployment, testing, and administration is the same for IBM WebSphere Application Server V6.0 (Test Environment, separate install, and the Network Deployment edition). In previous versions of WebSphere Studio, the configuration of the test environment was different than a separately installed WebSphere Application Server.

We have categorized the test server environments as follows:

- Integrated test servers

  Integrated test servers refers to the test servers included with the Rational Developer edition (see Table 19-1).

- Test servers available separately

  Test servers available separately refers to the test servers that are supported by the Rational Developer edition, but available separately from the Rational Developer products (see Table 19-2 on page 1046).

Table 19-1 Integrated test servers

Install option	Integrated test server	Web Developer	Application Developer
IBM WebSphere Application Server V6.0			
	IBM WebSphere Application Server V6.0	X	X
IBM WebSphere Application Server V5.x			
	IBM WebSphere Application Server V5.1	X	X
	IBM WebSphere Application Server Express V5.1	X	X
	IBM WebSphere Application Server V5.0.2	X	X
	IBM WebSphere Application Server Express V5.0.2	X	X
IBM WebSphere Portal			
	IBM WebSphere Portal V5.0.2.2 **Note**: Available on Windows (not Linux)	na	X
	IBM WebSphere Portal V5.1	na	X

Table 19-2 Additional supported test servers available separately

Integrated Test Server	Web Developer	Application Developer
Tomcat V5.0	X	X
WebLogic V6.1	X	X
WebLogic V7.1	X	X
WebLogic V8.1	X	X

19.1.2 Local vs. remote test environments

When configuring a test environment, the server can be either a local integrated server or a remote server. Once the server itself is installed and configured, the server definition within Rational Application Developer is very similar for local and remote servers.

In either case, local or remote, you will need to specify the SOAP connector port from the Rational Application Developer WebSphere V6.0 server configuration and the WebSphere Application Server V6.0 Profile.

19.1.3 Commands to manage test servers

Once the server is setup, there are a few key commands used to manage the test servers.

- Debug: Only available for local test servers
- Start: Only available for local test servers
- Profile: Only available for local test servers
- Restart: Available on all active WebSphere v6.0 Servers
- Restart: Can restart in different modes (normal, debug, and profile)
- Stop

19.2 Configure a WebSphere V6 Test Environment

This section describes how to create and configure a WebSphere Application Server V6 Test Environment within IBM Rational Application Developer V6.0. In this example, we create a new WebSphere V6 profile, a new Rational Application Developer server configuration, and a workspace.

Archived

19.2.1 Understanding WebSphere Application Server V6.0 profiles

New with IBM WebSphere Application Server V6.0 is the concept of profiles. The WebSphere Application Server installation process simply lays down a set of core product files required for the runtime processes. After installation you will need to create one or more profiles that define the runtime to have a functional system. The core product files are shared among the runtime components defined by these profiles.

Note: In V5, the **`wsinstance`** command was used to create multiple runtime configurations using the same installation. With V6, profiles allow you to do this.

With WebSphere Application Server and WebSphere Application Server Express Editions you can only have standalone application servers, as shown in Figure 19-1. Each application server is defined within a single cell and node. The administration console is hosted within the application server and can only connect to that application server. No central management of multiple application servers are possible. An application server profile defines this environment.

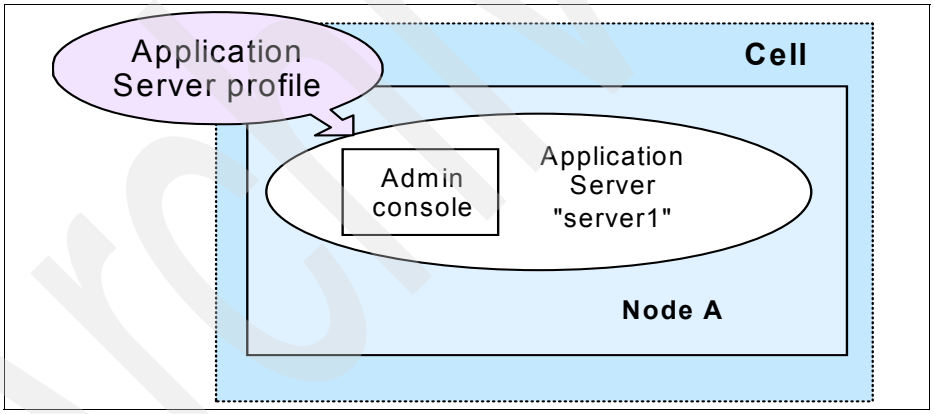


Figure 19-1 System management topology - Standalone server (Base and Express)

You can also create standalone application servers with the Network Deployment package, though you would most likely do so with the intent of federating that server into a cell for central management at some point.

With the Network Deployment package, you have the option of defining multiple application servers with central management capabilities. For more information on profiles for the IBM WebSphere Application Server V6.0 Network Deployment Edition, refer to the *WebSphere Application Server V6 Systems Management and Configuration*, SG24-6451.

Types of profiles

There are three types of profiles for defining the runtime:

- Application server profile

  Note: The application server profile is used by Rational Application Developer WebSphere Application Server V6.0 Test Environment.

- Deployment manager profile
- Custom profile

Application server profile

The application server profile defines a single standalone application server. Using this profile will give you an application server that can run standalone (unmanaged) with the following characteristics:

- The profile will consist of one cell, one node, and one server. The cell and node are not relevant in terms of administration, but you will see them when you administer the server through the administrative console (scopes).
- The name of the application server is “server1”.
- The application samples are automatically installed on the server.
- The server has a dedicated administrative console.

The primary use for this type of profile would be:

- To build a server in a Base or Express installation (including a test environment within Rational Application Developer).
- To build a standalone server in a Network Deployment installation that is not managed by the deployment manager (for example, to build a test machine).
- Or to build a server in a distributed server environment to be federated and managed by the deployment manager. If you are new to WebSphere Application Server and want a quick way of getting an application server complete with samples, this is a good option. When you federate this node, the default cell will become obsolete and the node will be added to the deployment manager cell. The server name will remain as server1 and the administrative console will be removed from the application server.

Deployment manager profile

The deployment manager profile defines a deployment manager in a Network Deployment installation. Although you could conceivably have the Network Deployment package and run only standalone servers, this would bypass the primary advantages of Network Deployment, which are workload management, failover, and central administration.

Archived

In a Network Deployment environment, you should create one deployment manager profile. This will give you:

- A cell for the administrative domain
- A node for the deployment manager
- A deployment manager with an administrative console.
- No application servers

Once you have the deployment manager, you can:

- Federate nodes built either from existing application server profiles or custom profiles.
- Create new application servers and clusters on the nodes from the administrative console.

Custom profile

A custom profile is an empty node, intended for federation to a deployment manager. This type of profile is used when you are building a distributed server environment. The way you would use this is:

- Create a deployment manager profile.
- Create one custom profile on each node on which you will run application servers.
- Federate each custom profile, either during the custom profile creation process or later using the **addNode** command, to the deployment manager.
- Create new application servers and clusters on the nodes from the administrative console.

Directory structure and default profiles

Within Rational Application Developer the integrated test environments are located in the <rad_home>/runtimes directory, where <rad_home> is the installation path, such as:

```
C:/Program Files/IBM/Rational/SDP/6.0
```

The IBM WebSphere Application Server V6.0 Test Environment is located in the following directory:

```
<rad_home>/runtimes/base_v6
```

We will refer to the root of each profile directory as <profile_home>:

```
<rad_home>/runtimes/base_v6/profiles/<profile_name>
```

Archived

The default profile is determined by the following:

- The profile was defined as the default profile when you created it. The last profile specified as the default will take precedence. You can also use the **wasprofile** command to specify the default profile.
- Or, if you have not specified the default profile, it will be the first profile you create.

When a profile is created, the profile is created from a template and copied to its unique <profile_home>. In addition, an entry is made to the profileRegistry.xml found at:

```
<rad_home>\runtimes\base_v6\properties\profileRegistry.xml
```

Example 19-1 lists the contents of a sample profileRegistry.xml. In this example, two profiles exist (default, AppSrv01). Notice the profile named *default* is marked as isDefault="true".

Example 19-1 Sample profileRegistry.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<profiles>
    <profile isDefault="true" name="default" path="C:\Program
Files\IBM\Rational\SDP\6.0\runtimes\base_v6\profiles\default" template="C:\Program
Files\IBM\Rational\SDP\6.0\runtimes\base_v6\profileTemplates\default"/>
    <profile isDefault="false" name="AppSrv01" path="C:\Program
Files\IBM\Rational\SDP\6.0\runtimes\base_v6\profiles\AppSrv01" template="C:\Program
Files\IBM\Rational\SDP\6.0\runtimes\base_v6\profileTemplates\default"/>
</profiles>
```

19.2.2 WebSphere Application Server V6 installation

The IBM WebSphere Application Server V6.0 Integrated Test Environment is an installation option from the main Rational Application Developer Installer.

For details on how to install the WebSphere Application Server V6.0 Test Environment, refer to "IBM Rational Application Developer V6 installation" on page 1372 (see Figure A-2 on page 1375 for install component selection).

Prior to the IBM WebSphere Application Server V6.0 Test Environment installation, the runtimes directory will look as follows:

```
<rad_home>\runtimes\base_v6_stub
```

After the IBM WebSphere Application Server V6.0 Test Environment is installed the directory will look as follows (no more stub):

```
<rad_home>\runtimes\base_v6
```

19.2.3 WebSphere Application Server V6 profile creation

In 19.2.1, “Understanding WebSphere Application Server V6.0 profiles” on page 1047, we reviewed the concepts for WebSphere V6.0 profiles. Profiles can be created using the **wasprofile** command line tool or the WebSphere Profile Creator wizard (pctWindows.exe), which is an interface to the wasprofile tool. For the purposes of development, we chose to use the WebSphere Profile wizard in the following example.

Create new profile using the WebSphere Profile wizard

To create a new WebSphere Application Server V6.0 profile using the WebSphere Profile Creator wizard (application server profile), do the following:

1. Start the WebSphere Profile Creation wizard.
   a. Navigate to the following directory:

      ```
      <rad_home>\runtimes\base_v6\bin\ProfileCreator
      ```

   b. Run **pctWindows.exe** to launch the WebSphere Profile Creation wizard.

   Note: Alternatively, if you have installed IBM Rational Application Developer Fix level V6.0.0.1, then you can do the following:

   1. Select **Window** → **Preferences**.
   2. Expand **Server** → **WebSphere**.
   3. Click **Create Profile**.

4. When the Welcome dialog appears, click **Next**.
5. When the Profile name dialog appears, we entered the following (as seen in Figure 19-2 on page 1052), and then clicked **Next**:
   – Profile name: `AppSrv01`
   – Uncheck **Make this profile the default**.

Archived

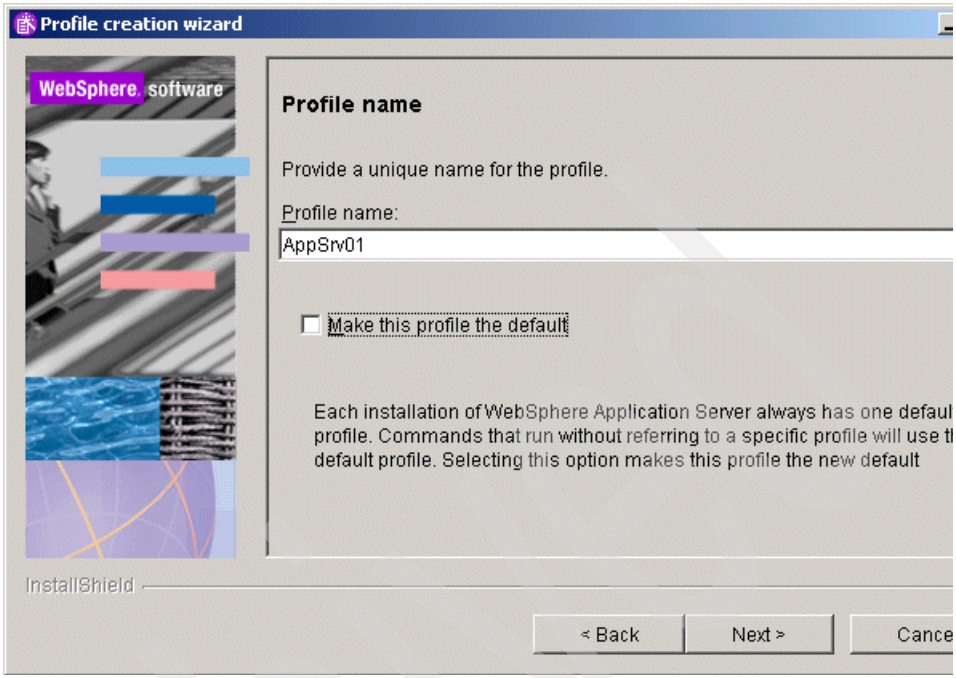


Figure 19-2 WebSphere Profile Creator wizard - Profile name

6. When the Profile directory dialog appeared, we accepted the following default directory and then clicked **Next**:

   `<rad_home>\runtimes\base_v6\profiles\AppSrv01`

   Where <rad_home> is the Rational Application Developer installation path.

7. When the Node and host names dialog appeared, we accepted the following defaults and then clicked **Next**:

   – Node name: `rad6win1Node02`

     In this case, rad6win1 is the host name of our computer, and Node01 was already used to create the default server, so the wizard used the next available node (Node02).

   – Host name: `rad6win1.itso.ral.ibm.com`

     This is the fully qualified host name of our computer.

8. When the Port value assignment dialog appeared, we accepted the generated port values. In this case, notice in Figure 19-3 on page 1053 that each of the port values is increased by one from the original default since a default profile existed before creating the new profile (AppSrv01).

   Take note of the port values and click **Next**.

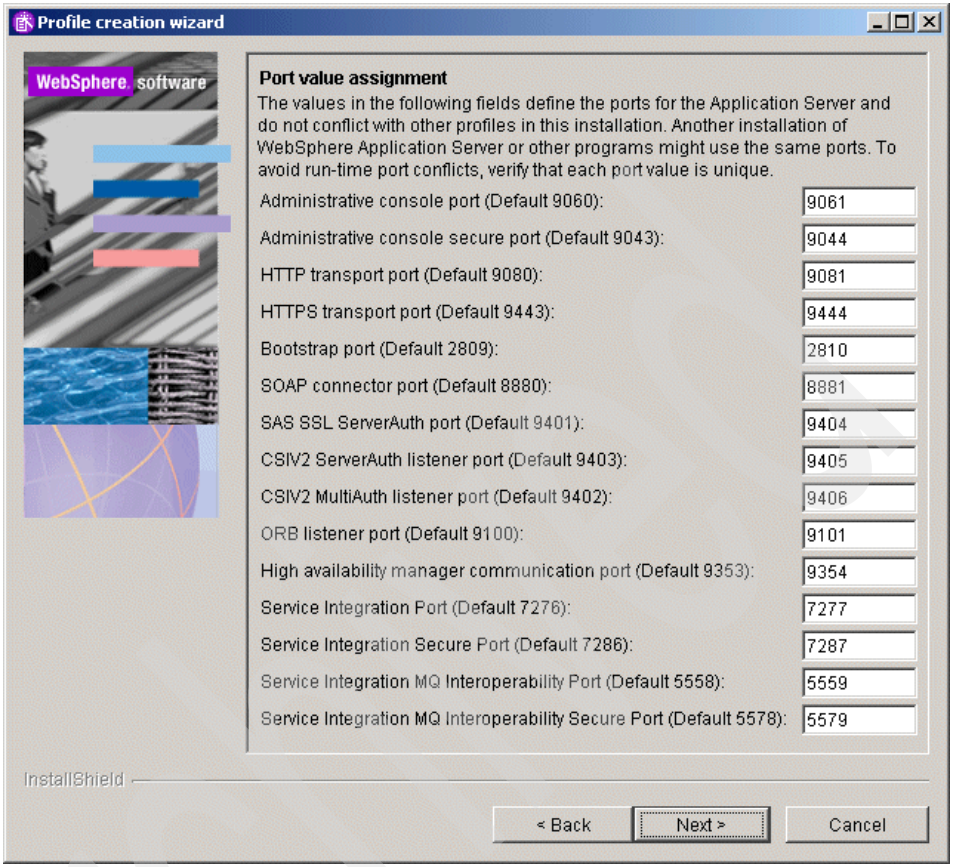


Figure 19-3 WebSphere Profile Creator - Port value assignments

Note: If you want to have multiple WebSphere Profiles but only intend to run them one at a time, you may consider using the default port values for each of them. Of course, this would preclude running them simultaneously without port conflicts.

9. When the Windows service definition dialog appears, we unchecked Run the Application Server process as a Windows service, as seen in Figure 19-4 on page 1054, and then clicked **Next**.

Note: We will be starting and stopping the server from Rational Application Developer. Remember, the Rational Application Developer server configuration is a pointer to the server defined in the WebSphere Profile.

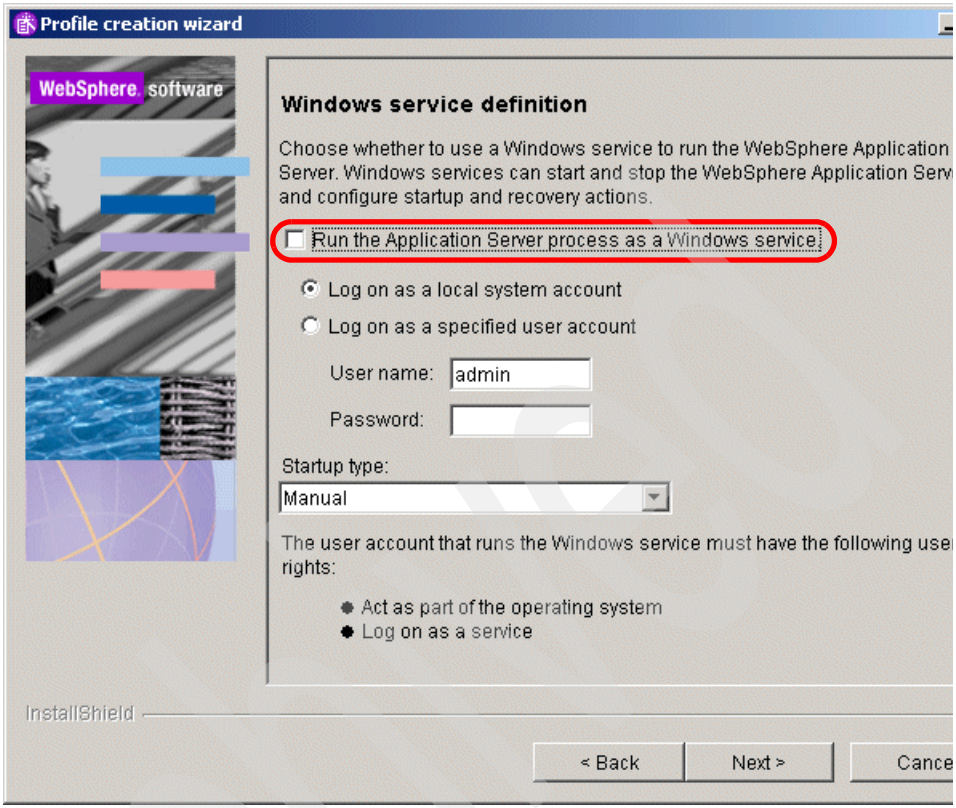


Figure 19-4 WebSphere Profile Creator - Windows service definition

10. When the Profile summary dialog appears, review the profile settings and then click **Next** to begin creating the new profile.

    The WebSphere Profile creation process takes approximately 10 minutes to complete.

11. When the Profile creation complete dialog appears, you should see a message at the top of the dialog: *The Profile creation wizard created the profile successfully.*

    By default, **Launch the First steps console** is checked. Click **Finish**.

Verify the new WebSphere Profile

After creating the WebSphere Profile, we recommend that you verify it was created properly and familiarize yourself with how to use it.

1. View the directory structure and find the new profile.

   `<rad_home>/runtimes/base_v6/profiles/<profile_name>`

Where *<profile_name>* is the name of the WebSphere Profile.

This is where you will find, among other things, the config directory containing the application server configuration files, the bin directory (for entering commands), and the logs directory where information is recorded.

Note: For simplicity, we will refer the entire path for the profile as *<profile_home>*.

2. Start the server.

   If you ran the installation verification, the server should already be started. You can check using the following commands:

   ```
   cd <profile_home>\bin
   serverStatus -all
   ```

   If the server status is not started, then start it from the First Steps menu or with the following commands:

   ```
   cd <profile_home>\bin
   startServer server1
   ```

3. Verify the server startup and installation.

   You can do this directly from the First Steps menu. This process will start the application server and verify the proper operation of the Web and EJB containers. Messages are displayed on the First Steps window and logged in the following places:

   ```
   <profile_home>/logs/server1/startServer.log
   <profile_home>/logs/server1/SystemOut.log
   ```

4. Open the WebSphere Administrative Console either by selecting the option in the First Steps window, or by accessing its URL from a Web browser:

   ```
   http://<appserver_host>:<admin_console_port>/ibm/console
   ```

   For example:

   ```
   http://localhost:9061/ibm/console/
   ```

   The administrative console port was selected during the profile creation wizard (see Figure 19-3 on page 1053).

   Click the **Log in** button. Since security is not active at this time, you do not have to enter a user name. If you choose to enter a name, it can be any name. If you enter a name it will be used to track changes you made to the configuration.

Archived

5. Display the configuration from the console. You should be able to see the following items from the administrative console:

   a. Application servers: Select **Servers** → **Application servers**. You should see `server1`, as seen in Figure 19-5 on page 1056. To see the configuration of this server, click the name in the list.

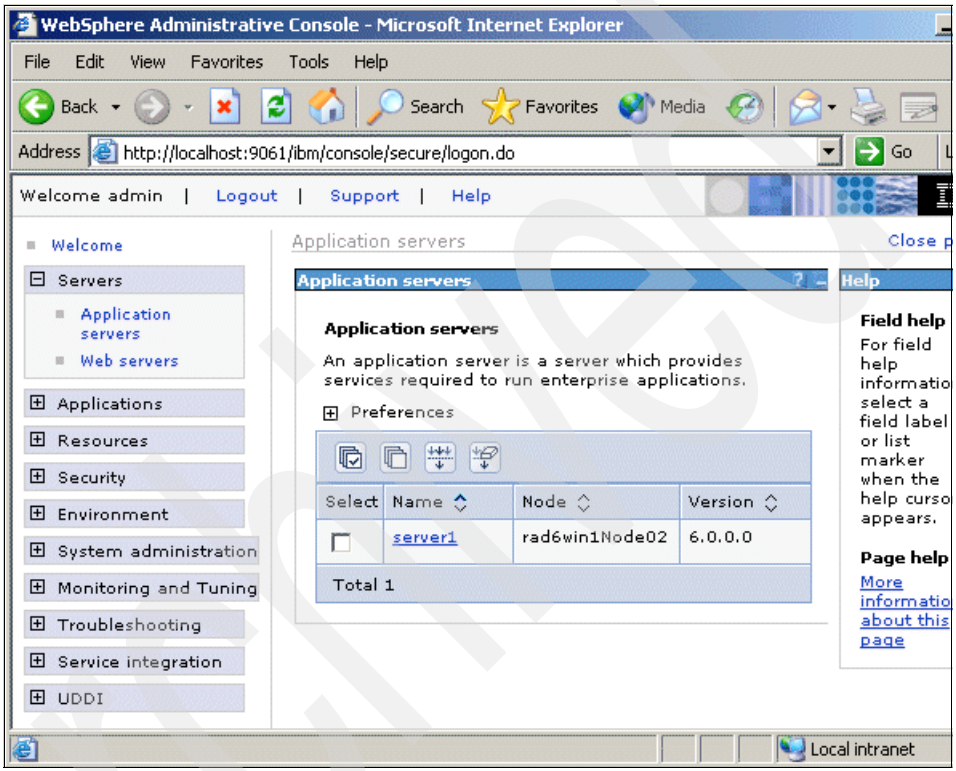


Figure 19-5 Application server defined by the application server profile

   b. Enterprise applications: Select **Applications** → **Enterprise Applications**. You should see a list of applications installed on server1.

> **Note:** Although you cannot display the cell and node from the administrative console, they do exist. You will see this later as you begin to configure resources and choose a scope, and also in the *<profile_home>* /config directory structure.

   c. Click **Logout** and close the browser.

6. Stop the application server. You can do this from the First Steps menu, or better yet, use the **stopServer** command:

```
cd <profile_home>\bin
stopServer server1
```

Tip: Delete a WebSphere Profile.

To delete a WebSphere Profile, do the following:

1. Stop the server that the profile is associated with.
2. Delete the WebSphere Profile.
   - Delete using the **wasprofile** command:

     ```
     wasprofile -delete -profileName AppSrv01
     ```

     Where AppSrv01 is the WebSphere Profile to delete.

     Or do the following
   - Manually delete the profile.
     i. Remove the profile entry from the profileRegistry.xml.

     ```
     <rad_home>\runtimes\base_v6\properties\profileRegistry.xml
     ```

     ii. Delete the <profile_name> directory:

     ```
     <rad_home>\runtimes\base_v6\profiles\<profile_name>
     ```

     iii. Delete the <profile_name>.bat from the following directory:

     ```
     <rad_home>\runtimes\base_v6\properties\fsdb\<profile_name>.bat
     ```

19.2.4 Define a new server in Rational Application Developer

Once you have defined the WebSphere Profile or chosen to use the default profile, you can create a server in Rational Application Developer. The server points to the server defined within the WebSphere Profile you configured.

With the new server configuration architecture, there are a few considerations to be aware of:

- The profile *default* is created when the WebSphere Application Server V6.0 Integrated Test Environment feature is selected during Rational Application Developer installation. A Rational Application Developer WebSphere Application Server V6.0 Test Environment is configured to use the default profile.
- The Rational Application Developer server configuration is essentially a pointer to the WebSphere Profile.

- You must manually stop the WebSphere Application Server Test Environment within Rational Application Developer before closing the Rational Application Developer, otherwise the server (server1 of the WebSphere Profile) will continue to run as a standalone WebSphere Application Server.

  Tip: If you have installed IBM Rational Application Developer Fix level V6.0.0.1, then you can do the following to stop the server when the Workbench is closed:

  1. From the Server view, double-click the server to open the Server Overview page settings.
  2. At the bottom of the page, there is a check box to **Terminate server on Workbench shutdown**.
  3. Save the settings.

To create a server in Rational Application Developer, do the following:

1. Open the J2EE perspective.
2. Select the **Servers** view.
3. Right-click in the Servers view, and select **New** → **Server**.
4. When the Define a New Server dialog appears, we did the following (as seen in Figure 19-6 on page 1059), and then clicked **Next**:
   - Host name: `localhost` (default)
   - Select the server type: Select **WebSphere v6.0 Server**.

Archive

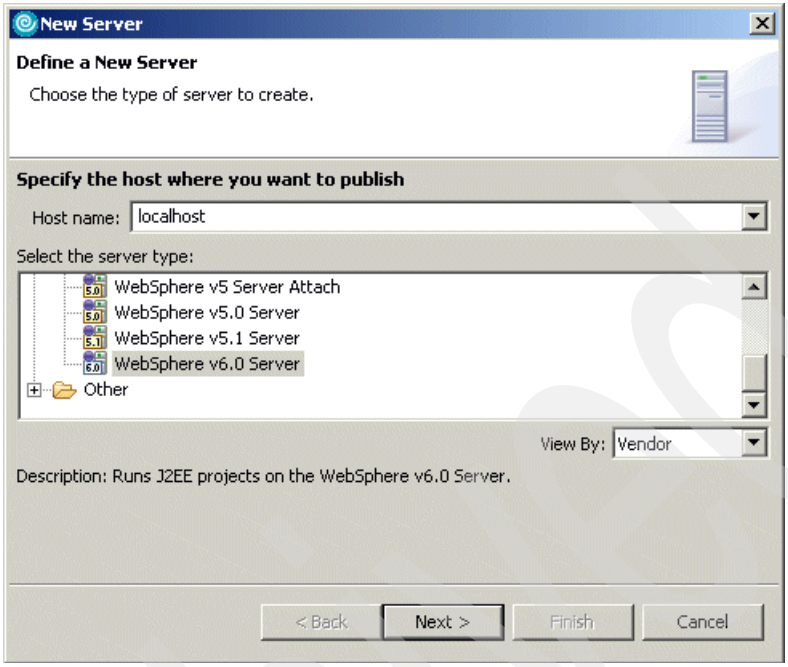


Figure 19-6 Define a New Server

5. When the WebSphere Server Settings dialog appears, we did the following (as seen in Figure 19-7 on page 1060), and then clicked **Next**:
   - WebSphere Profile name: Select **AppSrv01**.

     In our example, AppSrv01 is the WebSphere Profile we created. The *default* WebSphere Profile is the default value.
   - Server admin port number (SOAP connector port): 8881

     Note: The port will need to match the SOAP port defined when creating the WebSphere Profile.
   - Server name: server1
   - Check **Run server with resources within the workspace**.
   - Server type: Select Base or Express server.
   - We left the remaining fields as the default settings.

Archive

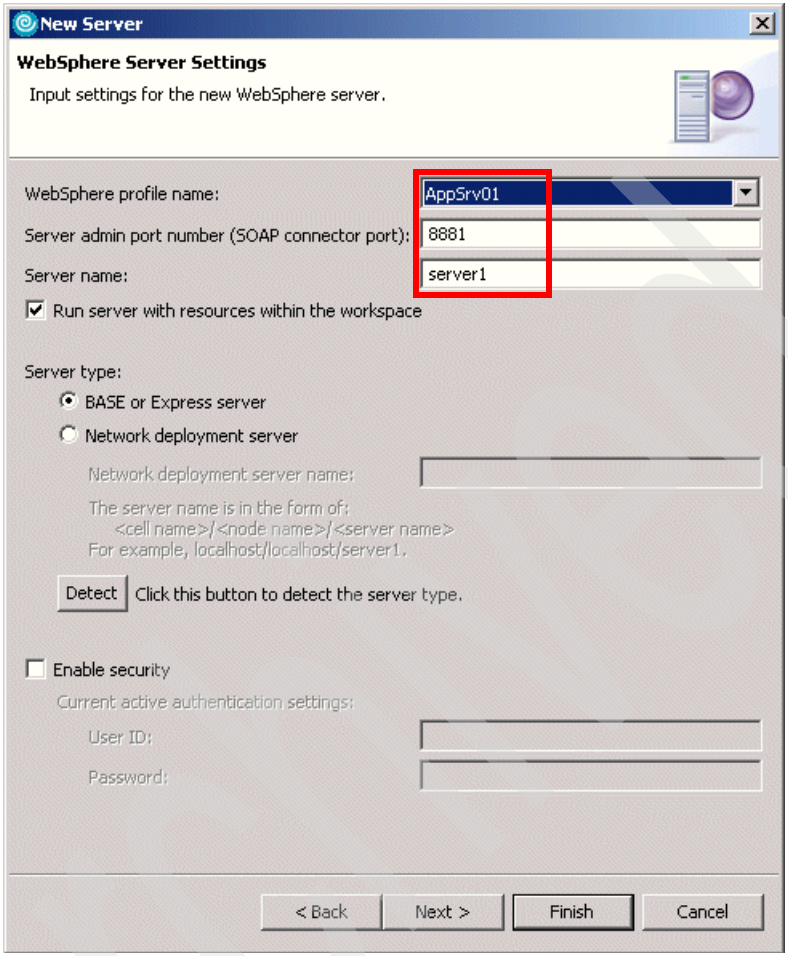


Figure 19-7 Define a New Server - WebSphere Server Settings

6. When the Add and Remove Projects dialog appears, we did not select a project at this time, and clicked **Finish**.

> **Note:** We address the topic of adding projects to a server in 19.3, “Add a project to a server” on page 1064.

The server will be created and should be displayed in the Servers view. In our example, the server WebSphere v6.0 @ localhost was created.

19.2.5 Verify the server

After you have completed defining the server within Rational Application Developer, we recommend that you perform some basic verification steps to ensure the server is configured properly.

1. Open the J2EE perspective.
2. Select the **Servers** view.
3. Double-click the server you created (for example, WebSphere v6.0 @ localhost) in the Servers view.
4. Review properties settings for the server.

   When the WebSphere v6.0 @ localhost server properties dialog opens, verify that the proper WebSphere Profile name is selected (for example, AppSrv01). Review the other settings. When done, save the file if changes were made and then close it.
5. Select the server (for example, WebSphere v6.0 @ localhost) in the Servers view, right-click, and select **Start**.

   In this example, the WebSphere v6.0 @ localhost defined in Rational Application Developer is configured to use server1 of the AppSrv01 WebSphere Profile.
6. Review the server startup output in the console. Also check the startServer.log and SystemOut.log for errors. The server logs can be found in the following directory:

   ```
   <profile_home>/logs/server1/startServer.log
   <profile_home>/logs/server1/SystemOut.log
   ```
7. Open the WebSphere Administrative Console either by selecting the option in the First Steps window, or by accessing its URL from a Web browser:

   ```
   http://<appserver_host>:<admin_console_port>/ibm/console
   ```

   For example:

   ```
   http://localhost:9061/ibm/console/
   ```

   The WebSphere Administrative Console port was selected during the WebSphere Profile Creation wizard (see Figure 19-3 on page 1053).
8. Click the **Log in** button. Since security is not active at this time, you do not have to enter a user name. If you choose to enter a name, it can be any name. If you enter a name it will be used to track changes you made to the configuration.
9. After accessing several pages of the WebSphere Administrative Console to verify it is working properly, click **Logout** and close the browser.

Archived

10. Verify the server stops properly by doing the following:

    a. From the J2EE perspective Server view, select the server (for example, WebSphere v6.0 @ localhost), right-click, and select **Stop**.

       The server status should change to `Stopped`.

    b. Verify that the server1 application server of the AppSrv01 WebSphere Profile has really stopped by entering the following command to check the server status:

       ```
       cd <profile_home>/bin
       serverStatus -all
       ```

       The server status output should show that the server has been stopped. If not, stop the server by entering the following command:

       ```
       stopServer server1
       ```

19.2.6 Customize a server in Rational Application Developer

Once the server has been created in Rational Application Developer, it is very easy to customize the settings.

1. Open the J2EE perspective.
2. Double-click the server you wish to customize in the Servers view.

There are a couple of key settings to point out for the server configuration, as seen in Figure 19-8 on page 1063.

- Server
  - WebSphere Profile name: Select the desired WebSphere Profile from the drop-down list.
  - Server admin port number (SOAP connector port): The SOAP connector port was defined for the WebSphere Profile at the time the profile was created. If you have more than one profile, the default behavior of the WebSphere Profile wizard is to increment this port number by 1. For example, the SOAP connector port for the *default* profile is 8880, and the AppSrv01 profile is 8881.
- Publishing

  Modify the publishing settings.

  - Select one of the following:
    - **Run server with resources within the workspace**
    - **Run server with resources on a Server**

Archived

– Enable automatic publishing: The default is for this to be enabled. This can be very time consuming, so you may consider turning this off and publishing as needed. Alternatively, change the interval of publishing.

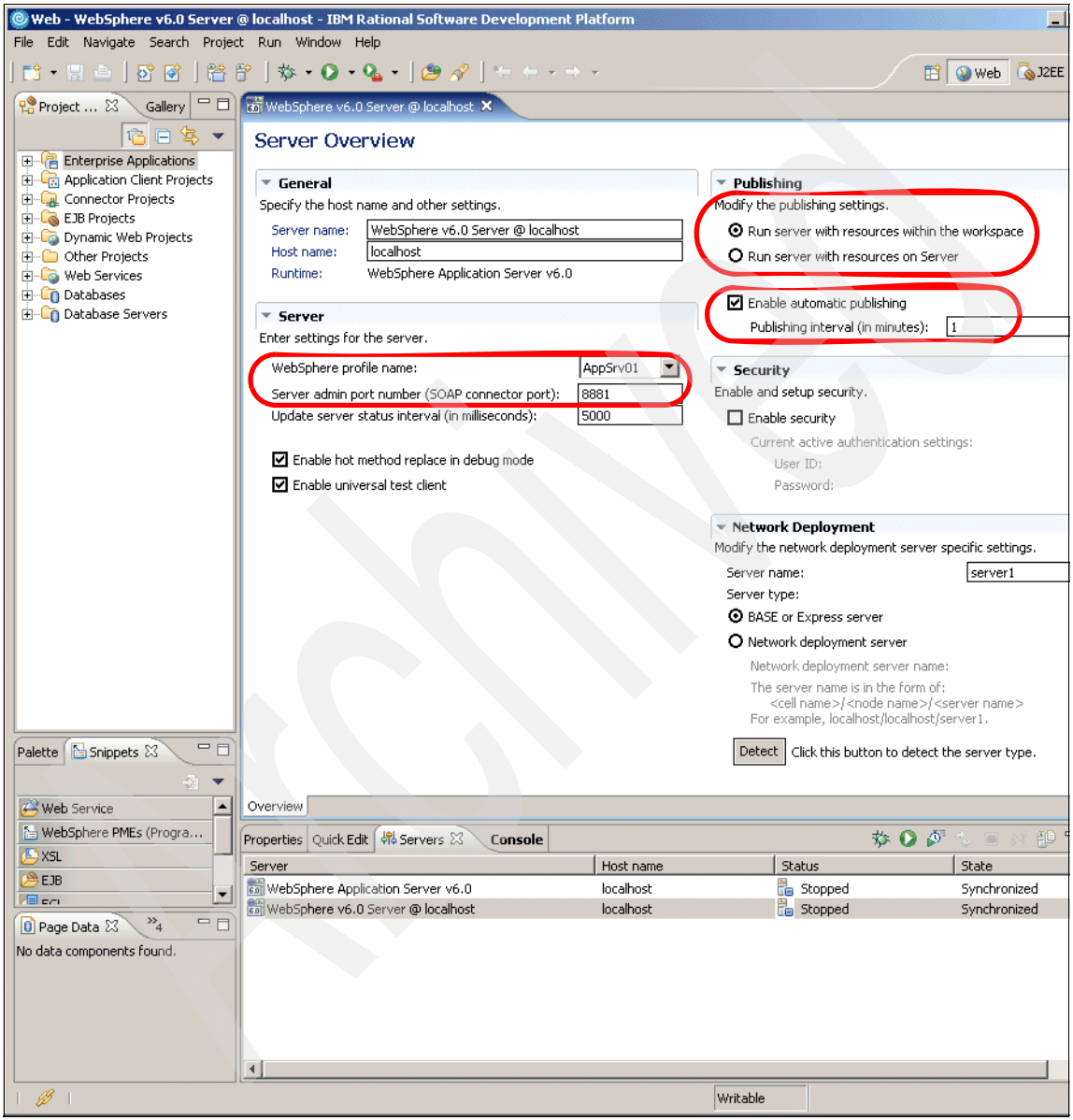


Figure 19-8 Customize server settings

3. After making changes save and close the file.

19.3 Add a project to a server

Once the server is configured, it can be further configured with server resources and be used to run applications by adding projects to it.

19.3.1 Considerations for adding a project to a server

When applications are added to a local test server or a separate installation, the actual binary files can be located in different places. The server definition in Rational Application Developer provides the selection of “Run server with resources within the workspace” or “Run server with resources on Server,” as seen in Figure 19-9. This configuration page can be accessed by double-clicking the server in the Servers view.

The publishing setting applies to all applications added from a workspace. When a project or application is removed via the Add/Remove Projects from the server context menu or from the WebSphere Administrative Console, the project still remains within the workspace.

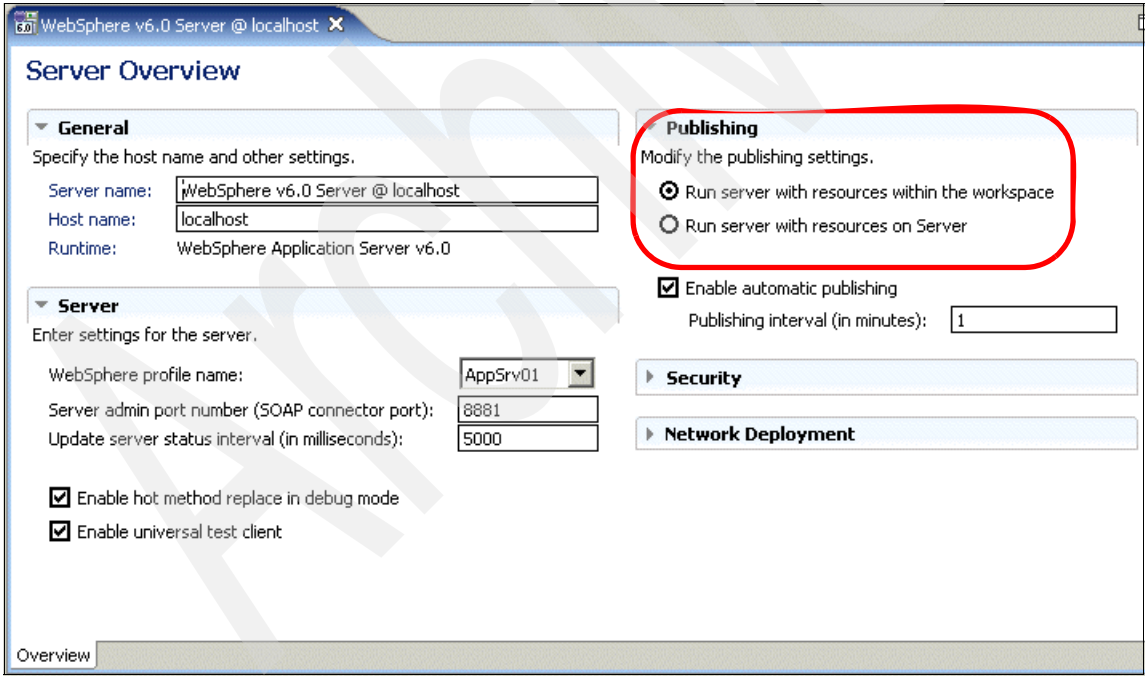


Figure 19-9 Application binary file location

19.3.2 Add a project to a server

This section describes how to add a Web application project to a server. We will use the simple JSP and Servlet Web application sample provided with Rational Application Developer.

Import the JSP and Servlet sample

To import the JSP and Servlet sample as a basis to demonstrate debug capabilities within Rational Application Developer, do the following:

1. From the Workbench select **Help** → **Samples Gallery**.
2. When the Samples Gallery window appears, select **Technology Samples** → **Web** → **JSP and Servlet**.
3. Scroll down the page and click **Import the Sample**.
4. When the Create a Sample Web Project window appears, accept the defaults and then click **Finish**.

Add the Web project to the server

To add a Web project to the server, do the following:

1. From the J2EE Perspective, select the **Servers** view.
2. Right-click the desired server to add the project, and select **Add and remove projects**.

   For example, we right-clicked the WebSphere v6.0 Server @ localhost we defined in “Define a new server in Rational Application Developer” on page 1057.
3. When the Add and Remove Projects dialog appears, we selected the **JSPandServletExampleEAR** and then clicked **Add**.
4. When the project appears in the Configured projects column, as seen in Figure 19-10 on page 1066, click **Finish**.

Archived

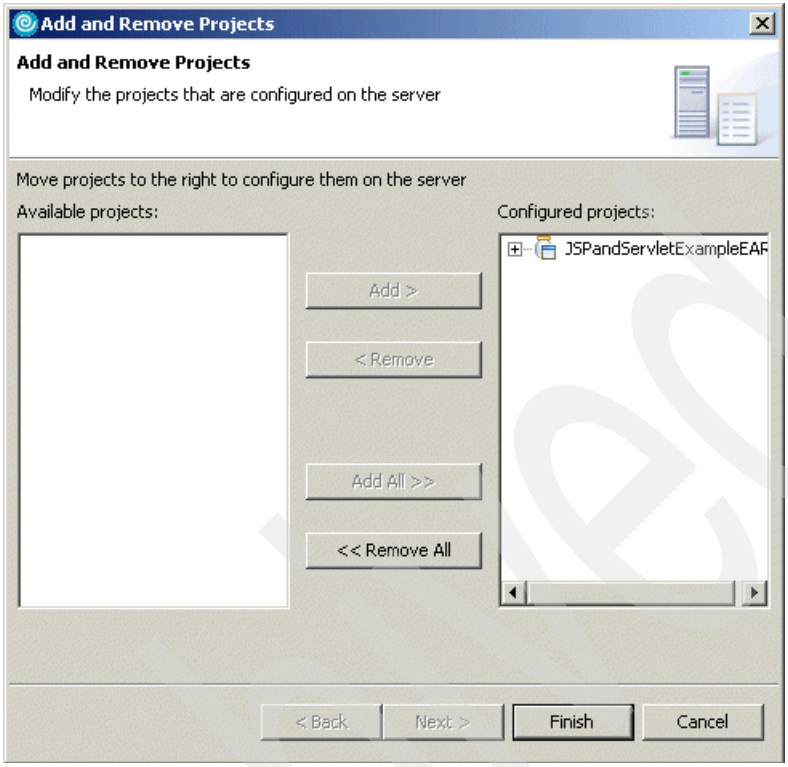


Figure 19-10 Add and Remove Projects

19.4 Remove a project from a server

As discussed previously, the Rational Application Developer server configuration is essentially a pointer to a server defined in the WebSphere Profile it is configured with. In this section we describe two scenarios for removing published projects from the server.

19.4.1 Remove a project via Rational Application Developer

In most cases, you can remove the project from the test server within Rational Application Developer as follows:

1. Open the Servers view.
2. Right-click the server where the application is published, and select **Add and remove projects** from the context menu.

3. When the Add and Remove Projects dialog appears, select the project in the Configured projects list, click the **< Remove** button, and then click **Finish**.

This operation will uninstall the application from the server defined for the WebSphere Profile your Rational Application Developer test server is configured with.

19.4.2 Remove a project via WebSphere Administrative Console

We found it necessary in some cases to uninstall the application from the WebSphere Administrative Console. For example, if you have published a project in Rational Application Developer to your test server, it will have been deployed to the server defined in the WebSphere Profile. If you then delete your Rational Application Developer server configuration or switch workspaces without first removing the project from the server, you will have a broken association between the Rational Application Developer server and the server defined in the WebSphere Profile.

Note: We have listed a couple of possible resolutions to this issue:

- Create a new WebSphere Profile for each new Rational Application Developer workspace (server configuration). This approach will require more diskspace for each WebSphere Profile and server configuration, and if run simultaneously will require additional memory.

  For details refer to 19.2, “Configure a WebSphere V6 Test Environment” on page 1046.

- Manually uninstall the application from the WebSphere Administrative Console.

  The following procedure explains how to uninstall a deployed application using the WebSphere Administrative Console.

To address issues like the scenarios described, uninstall the enterprise application from the WebSphere Administrative Console as follows:

1. Start the test server in Rational Application Developer by selecting the server, right-clicking, and selecting **Start**.
2. Start the WebSphere Administrative Console either by right-clicking the server and selecting **Run administrative console**, or by accessing its URL from a Web browser:

   `http://<appserver_host>:<admin_console_port>/ibm/console`

   For example:

   `http://localhost:9060/ibm/console/`

3. Click the **Log in** button.

   WebSphere security is not enabled, thus a user ID and password are not required.
4. Click **Applications** → **Enterprise Applications**.
5. Check the desired application to uninstall.
6. Click the **Uninstall** button.
7. When prompted, click **OK**.
8. Save your changes.

19.5 Publish application changes

Depending on the type of application change, new artifacts may be published automatically.

1. The source code and configuration data are created/modified and saved.
2. The Builder runs, and then starts the AppInstaller.
3. The AppInstaller then determines the type of application change (for example, delta) and then publishes the change to the server.

Table 19-10 on page 1066 lists the actions of different components in Rational Application Developer and automatic publish actions.

Table 19-3 Automatic publish of changes

Action in Rational Application Developer	Automatic publish action	Result on server
Web module added to EAR	Module update sent to server	Module added to EAR on server and module started
Web module removed from EAR	Module update sent to server	Module removed, EAR remains started
EJB module added to EAR	Full application update sent to server	EJB module added and EAR restarted
Enhanced EAR information added/changed in EAR	Full application update sent to server	EAR replaced and restarted
JSP added/changed/removed to Web module	Single file update sent to server	JSP added to Web module
JSP removed from Web module	Single file update sent to server	JSP removed and Web module remains started

Archived

Action in Rational Application Developer	Automatic publish action	Result on server
Servlet added/changed/removed to Web module or web.xml (IBM Extensions and Bindings) changed	Module update sent to server	Web module added to EAR and Web module restarted
EJB added/changed/removed or ejb-jar.xml (IBM Extensions and Bindings) changed	Full application update sent to server	EAR replaced and restarted

Considerations with automatic publish:

- If applications are set to run from the workspace, changes will be picked up and take affect without publishing.
- Test server restarts are minimized with Automatic Publish enabled:
  - Enabled defined resources (data sources at server level and above, JMS resources)
  - Debugging applications
  - Profiling servers
  - Enabling security
  - Changing JVM values on application server
- Automatic publish may slow system and development/test if making a large number of changes.
- Automatic publish requires server to be started, which requires more system resources.

19.6 Configure application and server resources

This section is organized into the following options for configuring application and server resources.

- Configure application resources.
- Configure server resources.
- Configure messaging resources.
- Configure security.

19.6.1 Configure application resources

In IBM WebSphere Application Server V6, application-related properties and data sources can be defined within an enhanced EAR file to simplify application

Archived

deployment (see Figure 19-3 on page 1053). The properties are used by the application after being deployed. When the extended EAR is deployed, the data source is registered with WebSphere Application Server V6.0 once the target application server is restarted. To provide greater flexibility, variables can be defined for substitution with server values when the application is deployed.

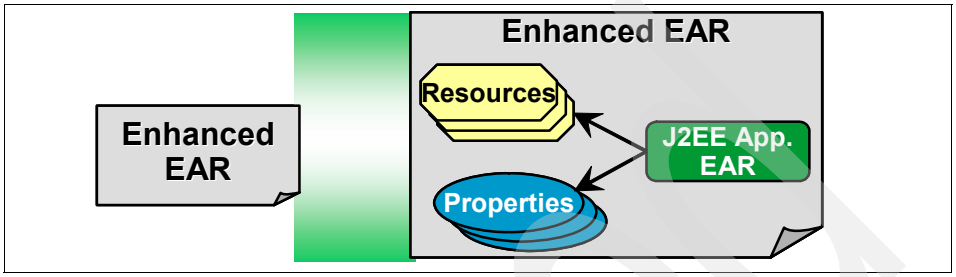


Figure 19-11 Enhanced EAR

Enhanced EAR tooling

The enhanced EAR tooling is provided from the Deployment tab of the Application Deployment Descriptor Editor, as seen in Figure 19-12 on page 1071. Deployment information is saved under the application ../META-INF/ibmconfig directory.

The following resource types can be added to the enhanced EAR:

- Virtual Hosts
- JAAS Authorization entries
- Shared Library
- Application class loader settings
- JDBC resources

Archive

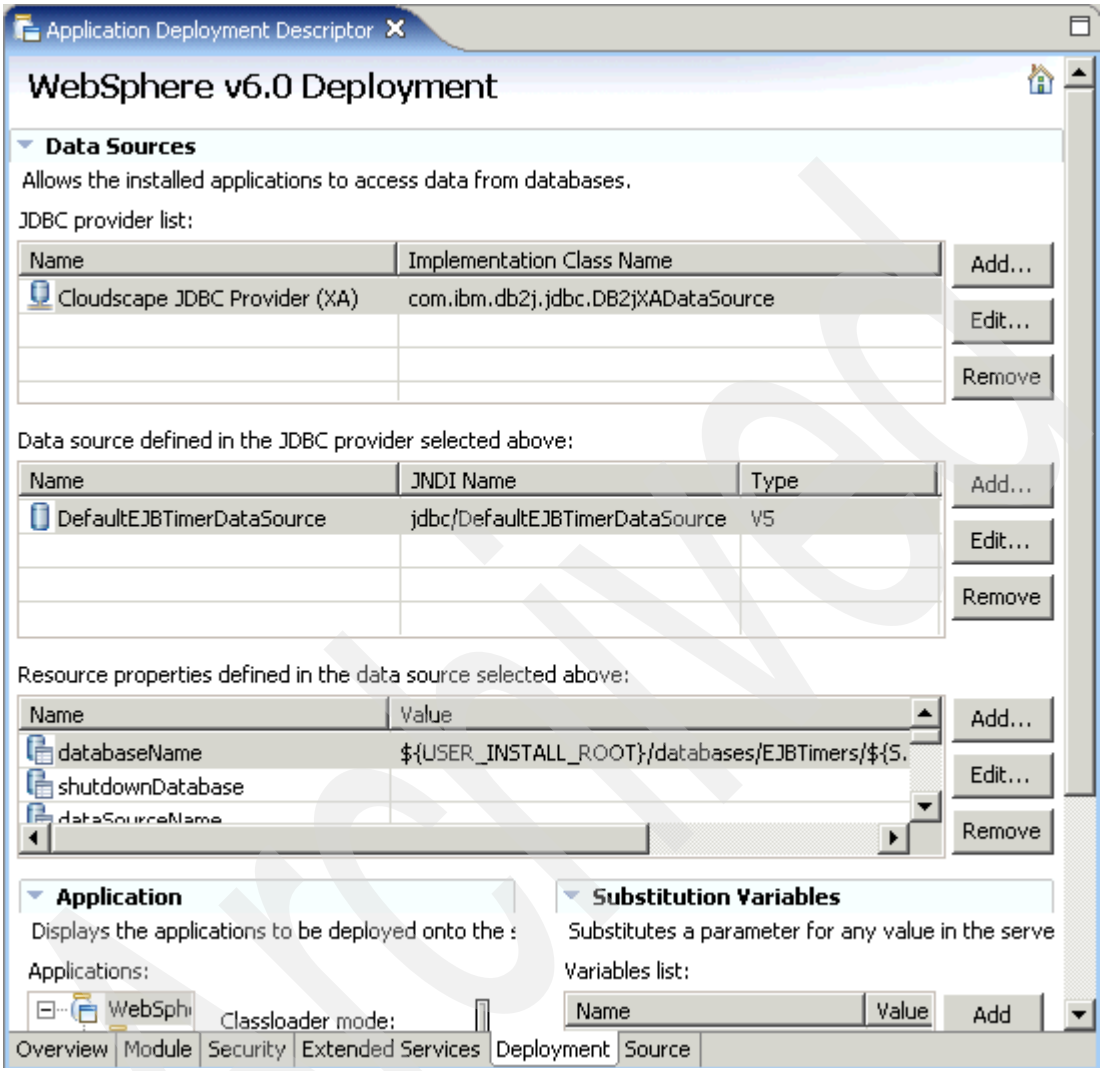


Figure 19-12 Application Deployment Descriptor Editor

Create a data source for EJB access

The data sources that support the entity beans must be specified before the application can be started. There are several ways to do it, but the easiest is to use the Enhanced EAR Editor.

Note: The Enhanced EAR Editor is used to edit several WebSphere Application Server V6 specific configurations, like data sources, classloader policies, substitution variables, shared libraries, virtual hosts, and authentication settings. It lets you configure these settings with little effort and publish them every time you publish the application.

The upside of the tool is that it makes the testing process simpler and easily repeatable, because the configurations it makes are saved to files that are usually shared at the team's repository. Thus, even though it will not let you configure every possible runtime setting, it is a good tool for development purposes because it eases the process of configuring the most common ones.

The downside is that the configurations the tool makes will be attached to the EAR, and will not be visible from WebSphere Application Server's administrative console. The console is only able to edit settings that belong to the cluster, node, and server contexts. When you change a configuration using the Enhanced EAR Editor, this change is made at the application context. The deployer can still make changes to the EAR file using the Application Server Toolkit (AST), but it is a separate tool. Furthermore, in most cases these settings are dependent on the node the application server is installed in anyway, so it makes little sense to configure them at the application context for deployment purposes.

The following sample demonstrates how to create a data source for EJB access for the EJB project created in Chapter 15, "Develop Web applications using EJBs" on page 827.

1. On the Project Explorer view, double-click the BankEJBEAR enterprise application's deployment descriptor.
2. Select the **Deployment** page as shown in Figure 19-13 on page 1073.

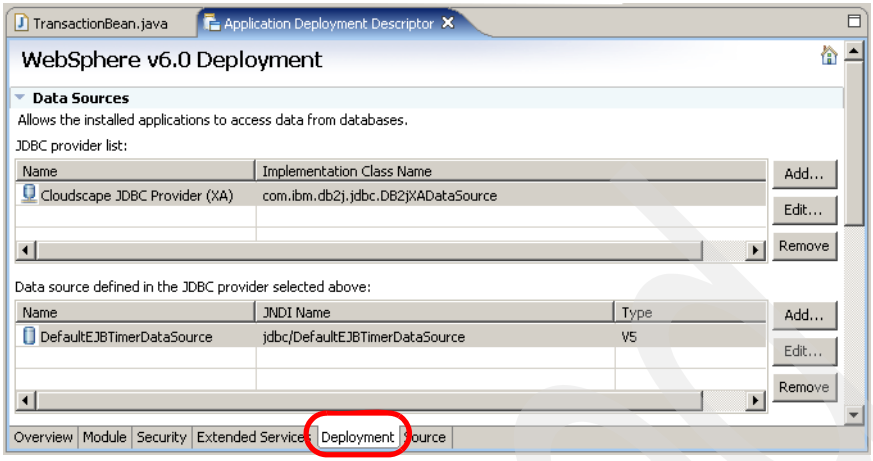


Figure 19-13 Enhanced EAR Editor

3. Scroll down the page until you find the Authentication section. It allows you to define a login configuration used by JAAS.
4. Click **Add** to include a new configuration (Figure 19-14).

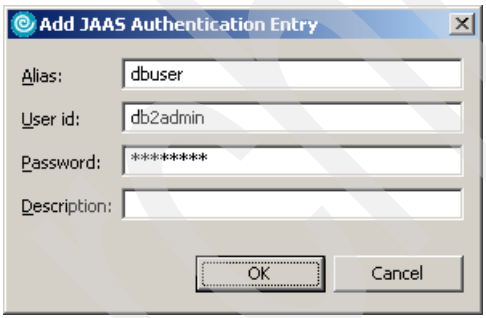


Figure 19-14 JAAS Authentication Entry

5. Enter `dbuser` as the entry's alias, and the appropriate user ID and password for your configuration. Click **OK** to complete the configuration.
6. Back at the Enhanced EAR editor, scroll back up to the Data Sources section. By default, the Cloudscape JDBC provider and Timer service datasource are defined. Since we are using DB2 in this example, we will need to add a DB2 JDBC provider by clicking the **Add** button right next to the provider list.

   The dialog depicted in Figure 19-15 on page 1074 is displayed.

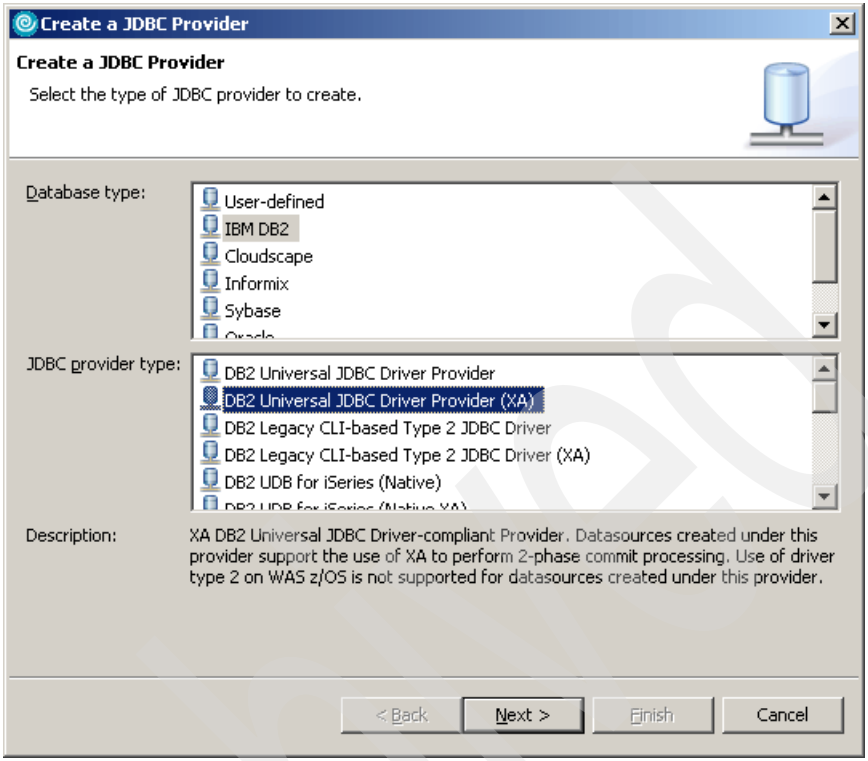


Figure 19-15 Creating a JDBC provider (page 1)

7. Select **IBM DB2** as the database type. Then select **DB2 Universal JDBC Driver Provider (XA)** as the provider type.

> **Note:** Note that for our development purposes, the DB2 Universal JDBC Driver Provider (non XA) would work fine, because we will not need XA (two-phase commit) capabilities.

8. Click **Next** to proceed to the second page (see Figure 19-16 on page 1075).

Archived

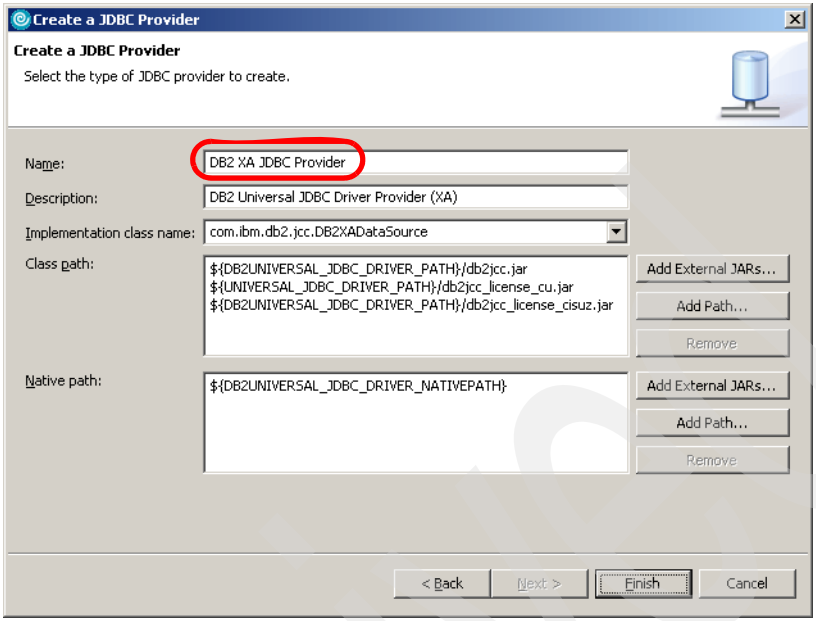


Figure 19-16 Creating a JDBC provider (page 2)

9. In this page, you only need to name the provider. We chose to call it DB2 XA JDBC Provider. The rest of the settings are good, so click **Finish**.
10. Go back to the Enhanced EAR Editor. With the new DB2 provider selected, click the **Add** button next to the defined data sources list (see Figure 19-17 on page 1076).

Archive

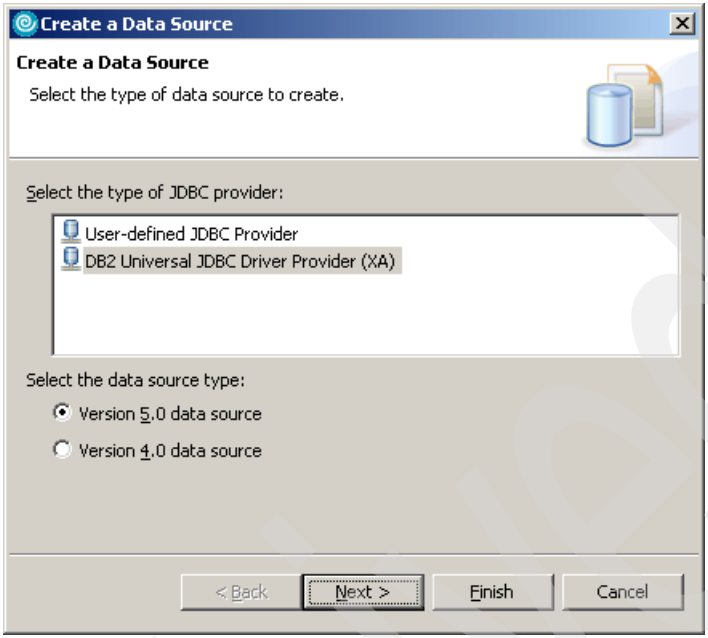


Figure 19-17 Create a Data Source (page 1)

11. Select **DB2 Universal JDBC Driver Provider (XA)** from the JDBC provider type list, and select the **Version 5.0 data source** radio button. Click **Next** to continue to the next page (see Figure 19-18 on page 1077).

Archived

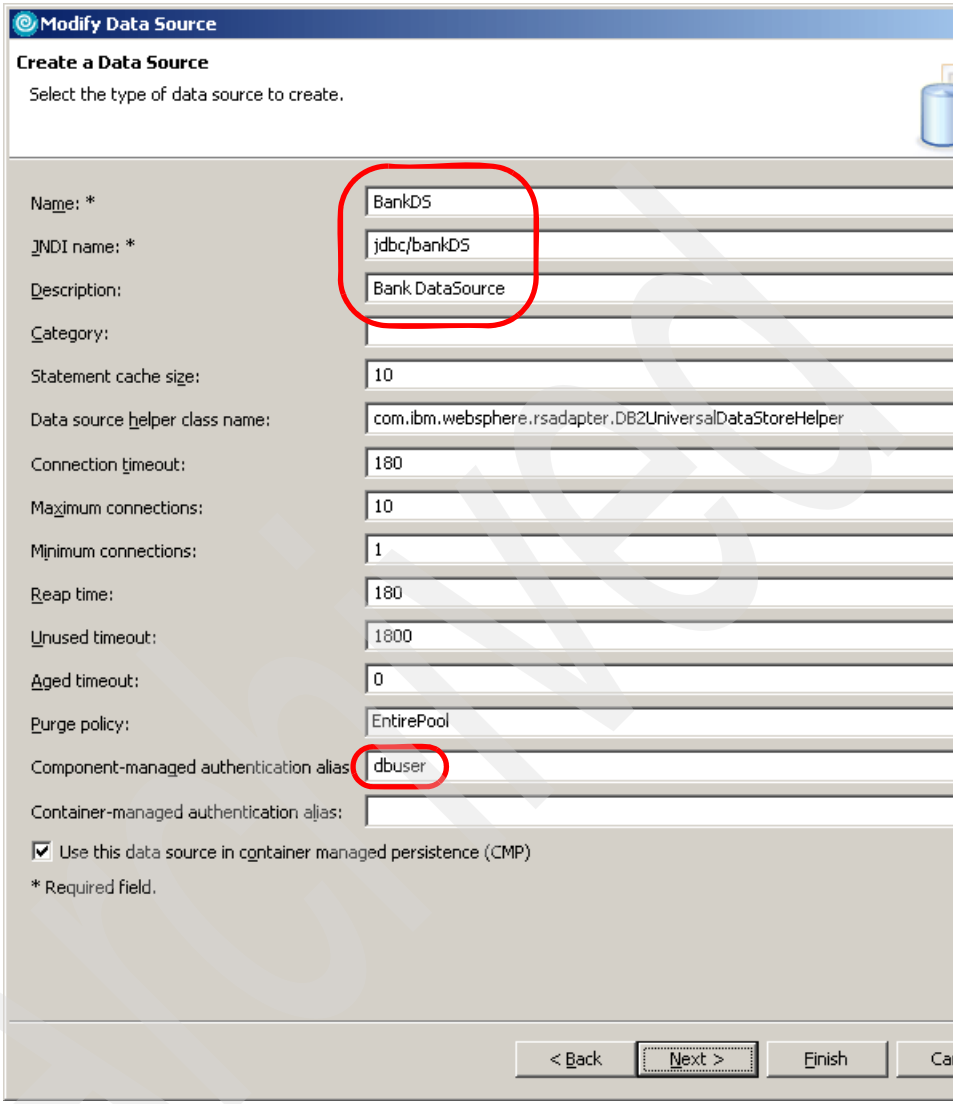


Figure 19-18 Create a Data Source (page 2)

12. Name the data source `BankDS`, give it the `jdbc/bankDS` JNDI name, fill out a description if you want, and select the **dbuser alias** from the "Component-managed authentication alias" drop-down box. Click **Next** to proceed to next wizard's last page, shown in Figure 19-19 on page 1078.

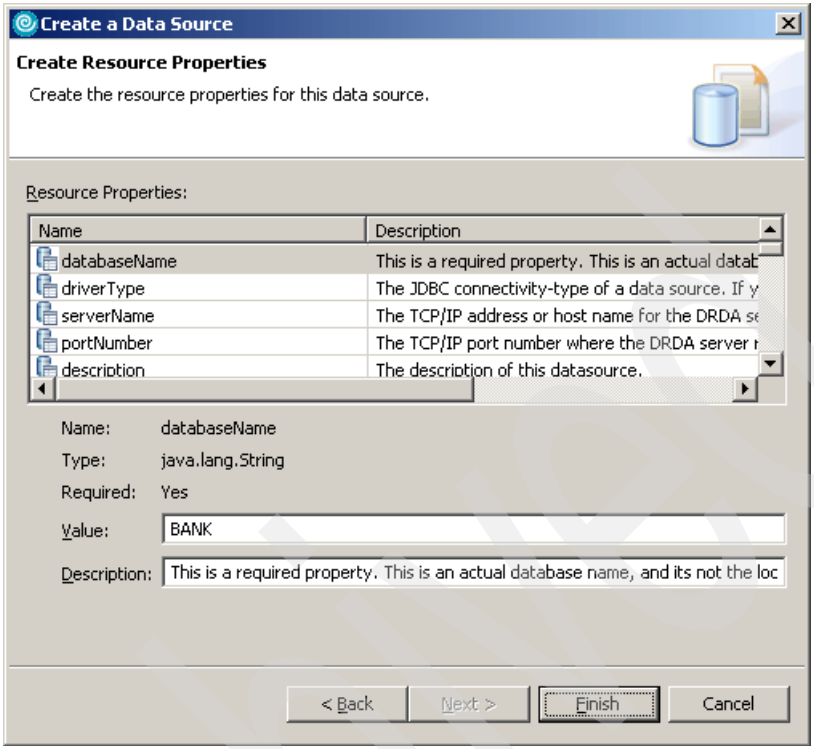


Figure 19-19 Create a Data Source (page 3)

13. Finally, just set the databaseName variable value to BANK, and click **Finish** to conclude the wizard. Save the deployment descriptor.

19.6.2 Configure server resources

Within IBM Rational Application Developer V6.0, the WebSphere Administrative Console is the primary interface for configuring WebSphere Application Server V6.0 test servers (local and remote). The Server editor replaces the Server Configuration editor from Version 5.x. The Server editor only contains details that point to the Rational Application Developer test server.

There are a couple of methods for accessing the WebSphere Administrative Server; however, in any case the WebSphere Application Server V6.0 test server must be started.

Once the WebSphere Application Server V6.0 test server is started, you can right-click the server and select **Run administrative server**. Alternatively, once the server is started you can enter one of the following URLs in a Web browser:

```
http://localhost:9060/ibm/console
http://localhost:9060/admin
```

Where 9060 is the port defined for the WebSphere Administrative Console for the application server in the WebSphere Profile.

19.6.3 Configure messaging resources

All Messaging Resources are set up through the WebSphere Administrative Console. This includes the Service Integration Bus, Bus Member, Messaging Engine, Bus Destinations, and JMS Destinations.

Note: For details on how to configure messaging refer to the WebSphere Application Server online information found in the InfoCenter. Also, in this book our sample EJB application includes messaging. For details on configuring messaging for this sample, refer to Chapter 23, “Deploy enterprise applications” on page 1189.

For 2.1 message-driven beans, they must bind to ActivationSpec JNDI name, Authentication Alias, and Destination JNDI name. The Listenerport is only for 2.0 Message-driven beans.

For EJB or Web messaging client, the message references must bind to messaging resources (similar to V5).

19.6.4 Configure security

Security is enabled from the WebSphere Administrative Console for the test environment. User ID and password fields are used to authenticate to a running server to find status, publish applications, and stop or restart servers. A user ID and password are not used for the user ID and password the server runs under.

Note: For more information on configuring WebSphere security, refer to *WebSphere Application Server V6 Security*, SG24-6316.

19.7 TCP/IP Monitor

The TCP/IP Monitor is a simple server that monitors all the requests and the responses between a Web browser and an application server. By default, when

the TCP/IP Monitor is started, it listens for requests on port 9081, and then it forwards these requests to the application server on port 9080. For responses from the application server, the TCP/IP Monitor forwards them back.

For an example of using the TCP/IP Monitor refer to 17.4.5, “Monitor the Web Service using the TCP/IP Monitor” on page 976.

Archived

20

JUnit and component testing

The IBM Rational Application Developer V6.0 test framework is built upon the open source Eclipse Hyades test framework, and includes JUnit, which can be used for automated component testing. Rational Application Developer also includes profiling capabilities for memory, performance, and other execution time code analysis. We explore profiling in more detail in Chapter 24, “Profile applications” on page 1237.

In this chapter we introduce application testing concepts, and provide an overview on the Hyades and JUnit, as well as the features of Rational Application Developer for testing. In addition, we include working examples to demonstrate how to create, run, and automate component tests using JUnit, as well as demonstrate how to test Web applications.

The chapter is organized into the following sections:

- Introduction to application testing
- JUnit testing
- Automated component testing
- Web application testing

© Copyright IBM Corp. 2005. All rights reserved.

20.1 Introduction to application testing

Although the focus of this chapter is on component testing, we have included an introduction to testing concepts such as test phases and environments to put into context where component testing fits within the development cycle. Next we provide an overview on the Hyades and JUnit testing frameworks. The remainder of the chapter provides a working example of using the features of Hyades and JUnit within Rational Application Developer.

20.1.1 Test concepts

Within a typical development project, there are various types of testing performed within different phases of the development cycle. Project needs based on size, complexity, risks, and costs determine the levels of testing to be used. The focus of this chapter is on component testing.

Test phases

We have outlined the key test phases and categories as follows:

- Unit test: Unit tests are informal tests that are generally executed by the developers of the application code. They are often quite low-level in nature, and test the behavior of individual software components such as individual Java classes, servlets, or EJBs.

  Because unit tests are usually written and performed by the application developer, they tend to be white-box in nature—that is to say, they are written using knowledge about the implementation details and test-specific code paths. This is not to say all unit tests have to be written this way; one common practice is to write the unit tests for a component based on the component specification before developing the component itself. Both approaches are valid, and you may want to make use of both when defining your own unit testing policy.

- Component test: Component tests are used to verify particular components of the code before they are integrated into the production code base. Component tests can be performed on a developer's machine. Within the context of Rational Application Developer, a developer configures a test environment and supporting testing tools such as JUnit. Using the test environment, you can test customized code including Java beans, Enterprise JavaBeans, and JavaServer Pages without needing to deploy this code to a runtime system (WebSphere Application Server, WebSphere Portal).

- Build Verification Test (BVT): Members of the development team check their source code into the source control tool, and mark the components as part of a build level. The build team is responsible for building the application in a controlled environment based on the source code available in the source

Archived

control system repository. The build team extracts the source code from the source control system, executes scripts to compile the source code (link if needed), packages the application, and tests the application build.

The test run on the application of the build produced is called a Build Verification Test (BVT). BVT is a predefined and documented test procedure to ensure that basic elements of the application are working properly before accepting the build and making it available to the test team for Function Verification Test (FVT) and/or System Verification Test (SVT).

- Function Verification Test (FVT): These tests are used to verify individual functions of an application. For example, you may verify if the taxes are being calculated properly within a banking application.

  Note: Within the Rational product family, the IBM Rational Function Tester is an ideal choice for this type of testing.

- System Verification Test (SVT): System tests are used to test a group of functions. A dedicated test environment should be used with the same system and application software as the target production environment. To get the best results from your tests, you need to find the most similar environment and involve as many components as possible, and verify that all functions are working properly in an integrated environment.

  Note: Within the Rational product family, the IBM Rational Manual Tester is an ideal choice for this type of testing.

- Performance test: Performance tests simulate the volume of traffic that you expect to have for the application(s) and ensure that it will support this stress and determine if the system performance is acceptable.

  Note: Within the Rational product family, the IBM Rational Performance Tester is an ideal choice for this type of testing.

- Customer Acceptance Test: This is a level of testing in which all aspects of an application or system are thoroughly and systematically tested to demonstrate that it meets business and non-functional requirements. The scope of a particular acceptance test is defined in the acceptance test plan.

Archive

Test environments

When sizing a project, it is important to consider the system requirements needed for your test environments. We have listed some common test environments that are used.

- Component test environment: This is often the development system and the focus of this chapter. In larger projects, we recommend that development teams have a dedicated test environment to be used as a sandbox to integrate team members' components before putting the code into the application build.
- Build verification test environment: This test environment is used to test the application produced from a controlled build. For example, a controlled build should have source control, build scripts, and packaging scripts for the application. The Build Verification Team will run a subset of tests, often known as regression tests, to verify basic functionality of the system that is representative to a wider scale of testing.
- System test environment: This test environment is used for FVT and SVT to verify the functionality of the application and integrate it with other components. There may be many test environments with teams of people focused on different aspects of the system.
- Staging environment: This staging environment is critical for all sizes of organizations. Prior to deploying the application to production, the staging environment is used to simulate the production environment. This environment can be used to perform customer acceptance tests.
- Production environment: This is the live runtime environment that customers will use to access your e-commerce Web site. In some cases, customer acceptance testing may be performed on the production environment. Ultimately, the customers will test the application. You will need a process to track customer problems and implement fixes to the application within this environment.

Calibration

By definition calibration is a set of gradations that show positions or values. When testing, it is important to establish a base line for such things as performance and functionality for regression testing. For example, when regression testing, you need to provide a set of tests that have been exercised on previous builds of the application, before you test the new build. This is also very important when setting entrance and exit criteria.

Archived

Test case execution and recording results

Sometimes the easiest way to know what broke the functionality of a component within the application is to know when the test case last worked. Recording the successes and failures of test cases for a designated application build is essential to having an accountable test organization and a quality application.

20.1.2 Benefits of unit and component testing

It may seem straightforward to many people as to why we test our code. Unfortunately, there are many people who do not understand the value of testing. Simply, we test our code and applications to find defects in the code, and to verify that changes we have made to existing code do not break that code. In this section, we highlight the key benefits of unit and component testing.

Perhaps it is more useful to look at the question from the opposite perspective, that is to say, why do developers *not* perform unit tests? In general, the simple answer is because it is too hard or because nobody forces them to. Writing an effective set of unit tests for a component is not a trivial undertaking. Given the pressure to deliver that many developers find themselves subjected to, the temptation to postpone the creation and execution of unit tests in favor of delivering code fixes or new functionality is often overwhelming.

In practice, this usually turns out to be a false economy, since developers very rarely deliver bug-free code, and the discovery of code defects and the costs associated with fixing them are simply pushed further out into the development cycle, which is inefficient. The best time to fix a code defect is immediately after the code has been written, while it is still fresh in the developer's mind.

Furthermore, a defect discovered during a formal testing cycle must be written up, prioritized, and tracked. All of these activities incur cost, and may mean that a fix is deferred indefinitely, or at least until it becomes critical.

Based on our experience, we believe that encouraging and supporting the development and regular execution of unit test cases ultimately leads to significant improvements in productivity and overall code quality. The creation of unit test cases does not have to be a burden. If done properly, developers can find the intellectual challenge quite stimulating and ultimately satisfying. The thought process involved in creating a test can also highlight shortcomings in a design, which may not otherwise have been identified when the main focus is on implementation.

We recommend that you take the time to define a unit testing strategy for your own development projects. A simple set of guidelines, and a framework that makes it easy to develop and execute tests, pays for itself surprisingly quickly.

Once you have decided to implement a unit testing strategy for your project, the first hurdles to overcome are the factors that dissuade developers from creating and running unit tests in the first place. A testing framework can help by making it easier to:

- Write tests
- Run tests
- Rerun a test after a change

Tests are easier to write, because a lot of the infrastructure code that you require to support every test is already available. A testing framework also provides a facility that makes it easier to run and re-run tests, perhaps via a GUI. The more often a developer runs tests, the quicker problems can be located and fixed, because the difference between the code that last passed a unit test, and the code that fails the test, is smaller.

Testing frameworks also provide other benefits:

- Consistency: Every developer is using the same framework, all of your unit tests work in the same way, can be managed in the same way, and report results in the same format.
- Maintenance: A framework has already been developed and is already in use in a number of projects, and you spend less time maintaining your testing code.
- Ramp-up time: If you select a popular testing framework, you may find that new developers coming into your team are already familiar with the tools and concepts involved.
- Automation: A framework may offer the ability to run tests unattended, perhaps as part of a daily or nightly build.

Automatic builds: A common practice in many development environments is the use of daily builds. These automatic builds are usually initiated in the early hours of the morning by a scheduling tool.

20.1.3 Eclipse Hyades

The Eclipse Hyades Test framework provides integrated testing, tracing, and monitoring framework. Within the scope of Rational Application Developer, this includes three types of testing:

- JUnit testing
- Manual testing
- Web browser-based application testing

Although each of these areas of testing has its own unique set of tasks and concepts, two sets of topics are common to all three types of testing:

- Creation and use of data pools
- Creating a test deployment

The primary purpose of the Eclipse Hyades project is to provide a common framework for test tools, so that Eclipse-based test tools can easily communicate with one another and work together. As such, the primary users of Hyades are Eclipse test tool developers. In addition, Hyades has an important secondary audience—testers of HTTP-based applications.

20.2 JUnit testing

This section provides JUnit fundamentals as well as a working example of how to create and run a JUnit test within Rational Application Developer.

20.2.1 JUnit fundamentals

A unit test is a collection of tests designed to verify the behavior of a single unit with in a class. JUnit tests your class by scenario, and you have to create a testing scenario that uses the following elements:

- Instantiate an object.
- Invoke methods.
- Verify assertions.

Note: An assertion is a statement that allows you to test the validity of any assumptions made in your code.

Example 20-1 lists a simple test case to verify the result count of a database query.

Example 20-1 Sample JUnit test method

```
//Test method
public void testGetAccount(){
      //instantiate
      Banking banking = new Banking();
      //invoke a method
      Account account = banking.getAccount("104-4001");
      //verify an assertion
      assertEquals(account.getAccountId(),"104-4001");
}
```

Archived

In JUnit, each test is implemented as a Java method that should be declared as *public void* and take no parameters. This method is then invoked from a test runner defined in a different package. If the test method name begins with `test...`, the test runner finds it automatically and runs it. This way, if you have a large number of test cases, there is no need to explicitly define all the test methods to the test runner.

TestCase class

The core class in the JUnit test framework is junit.framework.TestCase, of which all of the JUnit test cases inherit (see Example 20-2).

Example 20-2 Sample to highlight junit.framework.TestCase

```
import junit.framework.TestCase;

public class ITSOBankTest extends TestCase {

   /**
    * Constructor for ITSOBankTest.
    * @param arg0
    */
   public ITSOBankTest(String arg0) {
      super(arg0);
   }
```

As a best practice, your test case should have a constructor with a single string parameter. This is used as a test case name to display in the log and reports. All the reports will have the name of the test, which can make more sense than the entire java package and class name when seeing the report.

TestRunner class

Tests are executed using a test runner. To run this test in the text mode, use TestRunner, as seen in Example 20-3.

Example 20-3 Sample to highlight TestRunner

```
public static void main (String[] args) {
   junit.textui.TestRunner.run (ITSOBankTest);
}
```

TestSuite class

Test cases can be organized into test suites, managed by the junit.framework.TestSuite class. JUnit provides tools that allow every test in a suite to be run in turn and to report on the results.

Archived

TestSuite can extract the tests to be run automatically. To do so, you pass the class of your TestCase class to the TestSuite constructor, as seen in Example 20-4.

Example 20-4 Sample to highlight TestSuite class - TestSuite constructor

```
TestSuite suite = new TestSuite(ITSOBankTest.class);
```

This constructor creates a suite containing all methods starting with test, and that takes no arguments.

Alternatively, you can add new test cases using the addTest method of the TestSuite class, as seen in Example 20-5.

Example 20-5 Sample to highlight TestSuite class - addTest method

```
TestSuite suite = new TestSuite();
suite.addTest(new ITSOBankTest("testBankingConnection"));
suite.addTest(new ITSOBankTest("testBanking"));
```

20.2.2 Prepare for the sample

We use the Java Bank sample application created in Chapter 7, “Develop Java applications” on page 221, for the JUnit test working example.

Import the c:\6449code\java\BankJava.zip Project Interchange file into Rational Application Developer.

After importing the Java Bank sample, verify that it runs properly in the Java perspective (BankClient). For more information refer to 7.2.12, “Run the Java Bank application” on page 286.

Note: The completed ITSOBankTest.java source used for the JUnit test case is included in the itso.bank.test package of the BankJava project.

20.2.3 Create the JUnit test case

Rational Application Developer contains wizards to help you build JUnit test cases and test suites. We will use this wizard to create the ITSOBankTest test class to test the AllTests JavaBean, which is a facade of a banking application that allows you to get information about your account, and withdraw and deposit funds.

Archived

Create the itso.bank.test.junit package

The completed version of the ITSOBankTest.java JUnit test case is included in the itso.bank.test package of the BankJava project. For the purposes of illustrating how to create a JUnit test case, we will create a new package named itso.bank.test.junit.

To create the itso.bank.test.junit package, which we will use for JUnit test cases, do the following:

1. Select the **BankJava** project, right-click, and select **New** → **Package**.
2. When the Java Package dialog appears, enter `itso.bank.test.junit` in the Name field and then click **Finish**.

Create a JUnit test case

To create a test case for the processTransaction method of the AllTests facade, do the following:

1. Open the Java perspective Package Explorer view.
2. Expand **BankJava** → **src** → **itso.bank.model.facade**.
3. Select **ITSOBank.java**, right-click, and select **New** → **JUnit Test Case**.
4. To use JUnit in Rational Application Developer, the JUnit packages need to be added to the build path of the Java project. This is automatically detected, as seen Figure 20-1. Click **Yes**.

> **Note:** In our example, since we imported the BankJava project with the existing JUnit test code, the JUnit packages have already been added to the Java build path for the BankJava project.

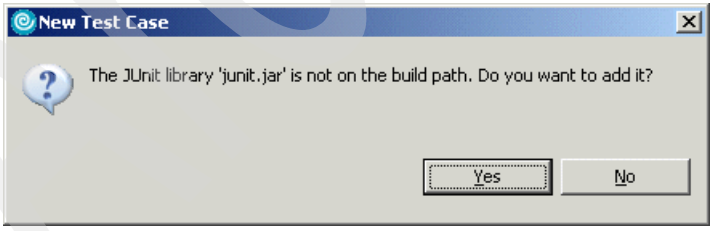


Figure 20-1 Adding JUnit library to the build class path

Archived

Tip: Sometimes you may need to add the JUnit library manually to your project. In that case, right-click the project and select **Properties**. Select **Java Build Path** and switch to the **Libraries** tab. Add a `JUNIT` variable that points to the junit.jar file, which can be found in the following directory:

```
<RAD_HOME>\eclipse\plugins\org.junit_3.8.1\junit.jar
```

5. When the JUnit Test Case dialog appears, we entered the following (as seen in Figure 20-2 on page 1092), and then clicked **Next**:
   - Source Folder: `BankJava/src`
   - Package: `itso.bank.test.junit`

     This is the package we created in "Create the itso.bank.test.junit package" on page 1090. It is a best practice to use a separate package for test cases.
   - Name: `ITSOBankTest`

     Test is concatenated to the original source file name *ITSOBank*.
   - Superclass: `junit.framework.TestCase`

     This is the default since we are creating a JUnit test case.
   - Check **public static void main(String[] args)**.
   - Check **Add TestRunner statement for:** and select **text ui**.

     This creates a main method in the test case, and adds a line of code that executes the TestRunner to run the test methods and output the results.
   - Check **setUp()**.

     This creates a stub for this method in the generated file.
   - We accepted the default values for the remaining fields.

Archived

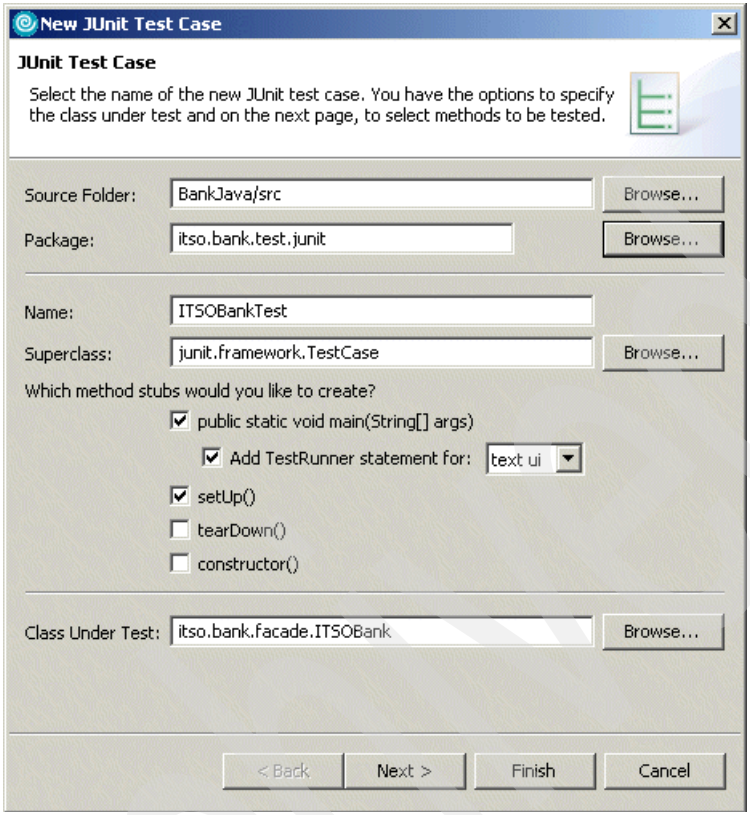


Figure 20-2 Create a JUnit test case

- *main* is the method that is executed to start the JUnit test case as a Java application. This is not required, as within Rational Application Developer, we can run Java classes as JUnit test cases.
- The “Add TestRunner statement for” check box has three options: text ui, swing ui, and awt ui. These options add a single line of code to the main method to run the test case and output the results in three different user interfaces. Text is plain text, while swing and awt are graphical outputs.
- *setUp* is a method that is executed before the tests.
- *tearDown* is a method that is executed after the tests.
- *constructor* is constructor stub for the class.
- *Class Under Test:* This is the Java class that this new test case is testing.

Note: A *stub* is a skeleton method, generated so that you can add the body of the method yourself.

6. When the Test Methods dialog appears, we checked the **processTransaction** method, as seen in Figure 20-3, and then clicked **Finish**.

> **Tip:** Try to think of the stub methods for a few scenarios to test in your test case. You can add as many methods to the generated file as you would like, and the naming conventions are up to you. This page of the wizard gets you started.

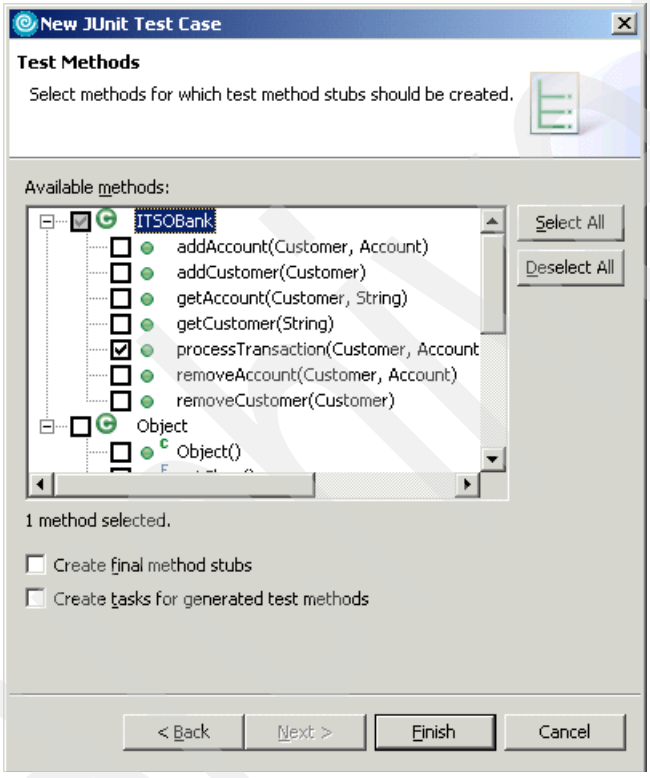


Figure 20-3 JUnit - Select test methods

The wizard will generate the ITSOBankTest.java and open the file in an editor. This file can be used to test the ITSOBank class. All that remains to do is to write the testing code.

Complete the setUp and tearDown methods

Typically, you run several tests in one test case. To make sure there are no side effects between test runs, the JUnit framework provides the setUp and tearDown

methods. Every time the test case is run, setUp is called at the start and tearDown at the end of the run.

1. Expand **BankJava** → **src** → **itso.bank.test.junit**.
2. Double-click **ITSOBankTest.java** to open the file in the Java editor.
3. Add the private object code highlighted in Example 20-6 to ITSOBankTest.java.

   This private object will be instantiated before starting the test and is available for use in the test methods.

Example 20-6 Add private object to ITSOBankTest.java

```
public class ITSOBankTest extends TestCase {

   Bank bank;
   Customer customer1;
   Account account11, account12;
```

4. Add the code highlighted in Example 20-7 to the setUp method of ITSOBankTest.java.

   The code is used to create an instance of the Bank facade and instantiate customer and account.

Example 20-7 Add code to setUp method

```
protected void setUp() throws Exception {
   super.setUp();

   bank = new ITSOBank();
      try {
         customer1 = new Customer("111-11-1111","John","Ganci");
         bank.addCustomer(customer1);
         System.out.println ( "Successfully Added customer1. "+customer1);
         account11 = new Account("11",new BigDecimal(10000.00D));
         bank.addAccount(customer1,account11);
         account12 = new Account("12",new BigDecimal(11234.23));
         bank.addAccount(customer1,account12);
         System.out.println("Successfully Added 2 Accounts to Customer1... ");
         System.out.println(customer1);

      }catch(InvalidCustomerException e){
         e.printStackTrace();
      }
}
```

5. Resolve imports.

Archived

As code is added, you will need to update the imports accordingly. Sometimes this is done automatically by Rational Application Developer, sometimes you can simply press Ctrl+Shift+O and add the imports, and other times you may need to manually add the imports.

6. Save the changes to the ITSOBankTest.java file by clicking **File** → **Save**.

In our example, there is no tearDown method (not needed). This method could be used to clean up tasks such as disconnecting from a database.

Complete the test methods

When the ITSOBankTest.java was generated, the test methods were added in stub form. This section describes the steps to complete the test methods.

1. Complete the testProcessTransaction method by adding the code highlighted in Example 20-8.

   This test method adds a new customer and account, then deposits funds into that account. Then it retrieves the account balance again and verifies that the new balance is the sum of the original balance and the deposited amount.

Example 20-8 Add code to the testProcessTransaction method

```
public void testProcessTransaction() {
   try {
      BigDecimal balanceBefore = new BigDecimal( account11.getBalance().toString() );
      BigDecimal debitAmount = new BigDecimal(2399.99D);

      bank.processTransaction( customer1, account11, debitAmount,TransactionType.DEBIT);

      assertEquals(account11.getBalance(), balanceBefore.subtract(debitAmount));

   }catch(InvalidCustomerException e){
      e.printStackTrace();
      fail("InvalidCustomerException. Message: " + e.getMessage());
   }catch(InvalidAccountException e){
      e.printStackTrace();
      fail("InvalidAccountException. Message: " + e.getMessage());
   }catch(InvalidTransactionException e){
      e.printStackTrace();
      fail("InvalidTransactionException. Message: " + e.getMessage());
   }
}
```

2. Add a new test method called testInvalidProcessTransaction, as seen in Example 20-9 on page 1096.

   This method verifies that if you try withdrawing more than the account holds, you will receive the proper InvalidTransactionException exception.

Example 20-9 Add new testInvalidProcessTransaction method

```
public void testInvalidProcessTransaction(){
   try {
      BigDecimal balanceBefore = new BigDecimal(account11.getBalance().toString());
      BigDecimal debitAmount = new BigDecimal(12399.99D);

      bank.processTransaction(customer1, account11, debitAmount,TransactionType.DEBIT);

      fail("Transaction should not be processed. Negative balance");

   }catch(InvalidCustomerException e){
      e.printStackTrace();
      fail("InvalidCustomerException. Message: " + e.getMessage());
   }catch(InvalidAccountException e){
      e.printStackTrace();
      fail("InvalidAccountException. Message: " + e.getMessage());
   }catch(InvalidTransactionException e){
      assertTrue(true);
   }
}
```

3. Save the changes to the ITSOBankTest.java file by clicking **File** → **Save**.

   This test case is now ready to run and can already be executed from the context menu (**Run** → **JUnit Test**).

Before continuing on to run the JUnit test, we need to review the assert and fail methods that we used in our test. In addition, we will also introduce the concept of a *test suite*.

JUnit-supplied assert and fail methods

The assertEquals, assertTrue, and fail methods are provided by the JUnit framework.

JUnit provides a number of methods that can be used to assert conditions and fail a test if the condition is not met. These methods are inherited from the class junit.framework.Assert (see Table 20-1).

Table 20-1 JUnit assert methods

Method name	Description
assertEquals	Assert that two objects or primitives are equal. Compares objects using equals, and compares primitives using ==.
assertNotNull	Assert that an object is not null.
assertNull	Assert that an object is null.

Archived

Method name	Description
assertSame	Assert that two objects refer to the same object. Compares using ==.
assertTrue	Assert that a boolean condition is true.
fail	Fails the test.

All of these methods include an optional String parameter that allows the writer of a test to provide a brief explanation of why the test failed. This message is reported along with the failure when the test is executed. Example 20-10 includes an assertEquals sample message that will fail for the working example.

Example 20-10 Sample assertEquals message

```
assertEquals(account11.getBalance().toString(), balanceBefore.add(debitAmount).toString());
```

Create a TestSuite

A TestSuite is used to run one or more test cases at once. Rational Application Developer contains a simple wizard to create a test suite as follows:

1. Expand **BankJava** → **src**
2. Right-click **itso.bank.test.junit**, and select **New** → **Other** → **Java** → **JUnit** → **JUnit Test Suite**. Click **Next**.

   Note: You may need to check **Show All Wizards** if not accessed previously.

3. When the JUnit Test Suite dialog appears, we entered the following, as seen in Figure 20-4 on page 1098, and then clicked **Finish:**
   - Source Folder: `BankJava/src`
   - Package: `itso.bank.test.junit`
   - Test suite: `AllTests`

     By default, the test suite is called AllTests. If you had multiple test classes, you could include them in one suite. We currently have a single test class only, but you can add to the suite later.
   - Check **ITSOBankTest**.
   - Check **public static void main(String[] args)**.
   - Check **Add TestRunner statement for:** and select **text ui**.

Archived

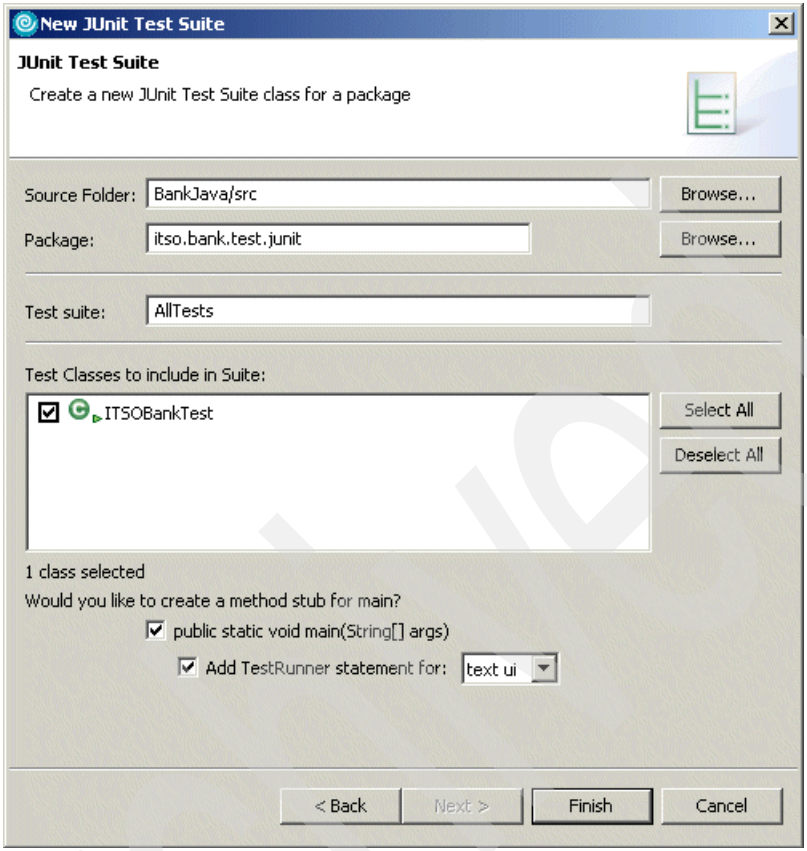


Figure 20-4 Sample JUnit Test Suite settings

The generated AllTests.java source opens, and here we can add more test classes later by using the suite.addTestSuite() method.

This code uses the text-based Test Runner tool in the JUnit framework, which runs the tests and reports the results.

4. No changes are required this time. Close the AllTests.java file.

In our example we only have a single test, and thus test suite is not required. However, as you add more and more test cases, a test suite quickly becomes a more practical way to manage your unit testing.

20.2.4 Run the JUnit test case

This section includes a couple of scenarios for running the JUnit test case. First, we examine the JUnit view and console output if the JUnit test was run after

completing the test methods (before “JUnit-supplied assert and fail methods” on page 1096). Second, we add asserts to have the test create a failure (not error).

Run JUnit test case

Now that the JUnit test case has been created (no assert failure), it can be run as follows:

1. Expand **BankJava** → **src** → **itso.bank.test.junit**.
2. Right-click **ITSOBankTest.java**, and select **Run** → **JUnit Test**.

   For our working example, you should see a JUnit view like Figure 20-5 with the results of the test run (no failures, no errors).

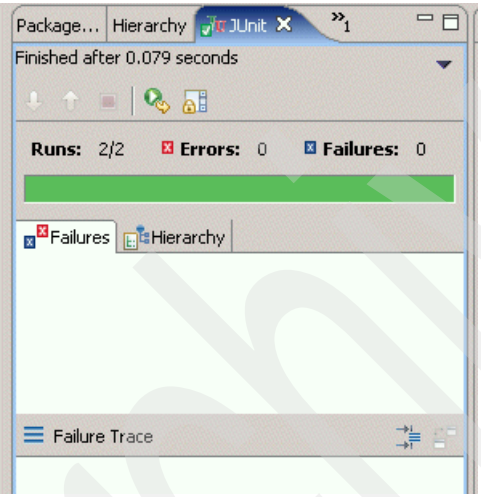


Figure 20-5 JUnit view - No assert condition (no failures, no errors)

Tip: To run just one test case, select the test case class and then **Run** → **JUnit Test**. To run all the test cases, select the test suite class (by default AllTests) and then the same action.

Note the Rerun Last Test button on the JUnit view menu bar ( ).

Modify and run the JUnit test case with assert failures

In the previous example we tested only for success. A test is considered to be *successful* if the test method returns normally. A test *fails* if one of the methods from the Assert class signals a failure. An *error* indicates that an unexpected exception was raised by the test method, or the setUp or tearDown method was invoked before or after it.

Archived

The JUnit view is more interesting when an error or failure occurs. This section describes how to modify both methods in ITSOBankTest to include assert test failures.

1. Expand **BankJava** → **src** → **itso.bank.test.junit**.
2. Double-click **ITSOBankTest.java** to open the file in the Java editor.
3. Modify the testProcessTransaction method by changing the transaction type from DEBIT to CREDIT, as seen in Example 20-11, so the balance will not match, and thus get an assert failure.

Example 20-11 Modified testProcessTransaction method - CREDIT instead of DEBIT

```
bank.processTransaction(customer1,account11,debitAmount,TransactionType.CREDIT);
```

4. Modify the testInvalidProcessTransaction methods so that the transaction is attempted on an account that is unassigned to customer, as listed in Example 20-12.

Example 20-12 Modified testInvalidProcessTransaction method - Unassigned customer

```
bank.processTransaction(new Customer("374-594-3994", "a", "b"),
account12,debitAmount,TransactionType.DEBIT);
```

5. Run the modified JUnit ITSOBankTest.java test case.

   Select **ITSOBankTest.java**, right-click, and select **Run** → **JUnit Test**.

   Figure 20-6 shows the JUnit view when the test case is run as a JUnit test again. This time, failures are displayed as well as failure trace information for each failure.

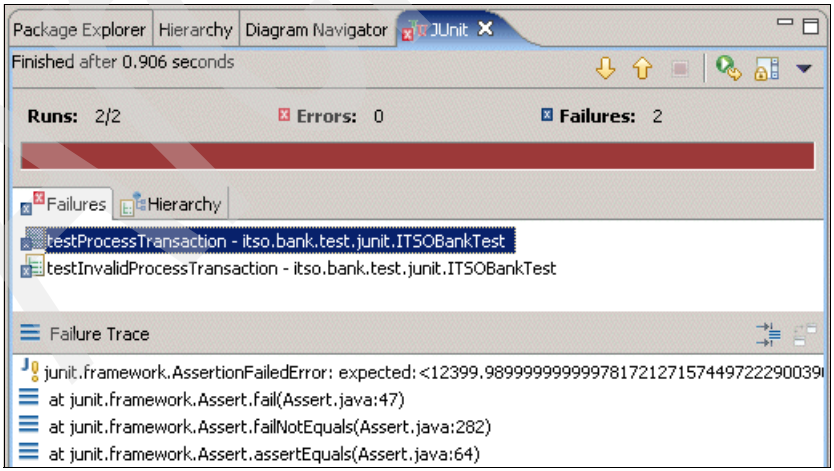


Figure 20-6 Junit view with failures

6. Select the **testProcessTransaction** method from the failures list.
7. This will update the Failure Trace window to show the stack trace of the failure. This makes it easy for you to track where the failure occurred. Double-clicking the entry in the Failure Trace list takes you to the specified line in the specified Java source file.

   Alternatively, the ITSOBankTest class can be run as a Java application by selecting **Run** → **Java Application**, which executes the main method and uses the TestRunner from the JUnit framework to run and output the test results in the console.
8. Example 20-13 displays the Console output from the test case.

   In this example, there was one success and one failure. The failure occurred when running testProcessTransaction.

Example 20-13 Console output

```
.F.Debit: Could not process Transaction. Reason: Negative/Zero Debit Amount.
Amount: $12399.99

Time: 0.02
There was 1 failure:
1) testProcessTransaction(itso.junit.bank.model.facade.java.ITSOBankTest)
      junit.framework.AssertionFailedError:
      expected:<12399.98999999999978172127157449722290039062 5> but was:
         <7600.01000000000021827872842550277709960937 5>
...
FAILURES!!!
Tests run: 2,  Failures: 1,  Errors: 0
```

   Each dot (.) in the first line of the output represents the start of a test. We have two tests in our test case, so there are two dots. An “F” indicates a failure, so one test failed. There is no special symbol printed for a passed test. Once all the tests have completed, the test runner shows how long they took to run and provides a summary of the results.
9. Once the error is corrected, the output should look like the Console view displayed in Example 20-14.

Example 20-14 New console output

```
..Debit: Could not process Transaction. Reason: Negative/Zero Debit Amount.
Amount: $12399.99

Time: 0.02

OK (2 tests)
```

Test the Web applications

You can also create test cases that run against one of the Web projects, BankBasicWeb or BankStrutsWeb. However, when testing anything that runs inside a servlet container, a testing framework like *Cactus* could make the testing much easier.

Note: Cactus is an open source sub-project in the Apache Software Foundation's Jakarta Project. It is a simple framework for unit testing server-side Java code such as servlets, EJBs, Tag Libs, Filters, etc.

The objective of Cactus is to lower the cost of writing tests for server-side code. Cactus supports so-called white box testing of server-side code. It extends and uses JUnit.

20.3 Automated component testing

The automated component testing features in Rational Application Developer allows you to create, edit, deploy, and run automated tests for Java components, EJBs, and Web Services. These features comply with the UML Testing Profile standard and they use the JUnit testing framework.

All the tests that you create with Rational Application Developer are extensions of JUnit tests. The automated component testing features extend JUnit with the following families of primitives:

- Initialization points (IP): Initialize variables or attributes of a component-under-test (CUT).
- Validation actions (VA): Verify the validity of a variable.
- Timing constraints (TC): Measure the duration of method calls.

A major difference between validation actions and the original JUnit assert methods is that with validation actions, failed assertions do not stop the execution of the entire JUnit test suite.

20.3.1 Prepare for the sample

This section outlines the tasks to complete the preparation for the automating component testing sample.

IBM Rational Agent Controller installation

The IBM Rational Agent Controller must be installed and running as a prerequisite to automated component testing.

For information on installing the IBM Rational Agent Controller included with Rational Application Developer, refer to “IBM Rational Agent Controller V6 installation” on page 1382.

Import the sample application

We use the sample Java Bank application for the automated component testing sample. If you have not already done so, import the c:\6449code\java\BankJava.zip Project Interchange file. This is the same project that was used to implement the JUnit tests in 20.2, “JUnit testing” on page 1087.

20.3.2 Create a test project

To test your components, you must first create a test project. The test project is linked to one or several development projects that contain the components you want to test. Development projects can include Java development projects, Enterprise Application projects, or Dynamic Web Projects. The components targeted for each test are known as the component-under-test (CUT).

To create a new component test project, do the following:

1. Select **File** → **New Project**.
2. When the New Project dialog appears, select **Component Test** → **Component Test Project** and then click **Next**.
3. Enter `BankComponentTest` in the Name field and then click **Next**.
4. When the Define the scope of the component test project dialog appears, check **BankJava** and then click **Finish**.
5. When the Confirm Perspective Switch dialog appears, click **Yes**.

20.3.3 Create a Java component test

To create a Java Component test, do the following:

1. From the Test perspective Test Navigator view, select the **BankComponentTest** project.
2. Select **File** → **New** → **Other**.
3. Select **Component Test** → **Java** → **Java Component Test**, and then click **Next**.
4. When the Select a test project dialog appears, select **BankComponentTest** and click **Next**.
5. If you completed the JUnit example found in 20.2, “JUnit testing” on page 1087, a Metrics Analysis pop-up dialog appears notifying you that the

ITSOBankTest class is not a valid class for further testing because it extends TestCase. This makes sense and we do not need to test this class. Click **OK**.

6. When the Select components under test dialog appears, do the following (as seen in Figure 20-7 on page 1105) and then click **Next**:

   – Check **Customer**.
   – Check **Account**.
   – Check **ITSOBank**.

   Figure 20-7 on page 1105 displays the components with highlighted values or high numerical values considered high-priority test candidates.

   A static analysis was performed on the Java source files associated with the BankJava project. These files were selected during the creation of the test and help to define the scope of the test. The list of files in the test project can be updated later by modifying the *Test Scope* properties of the project.

Archived

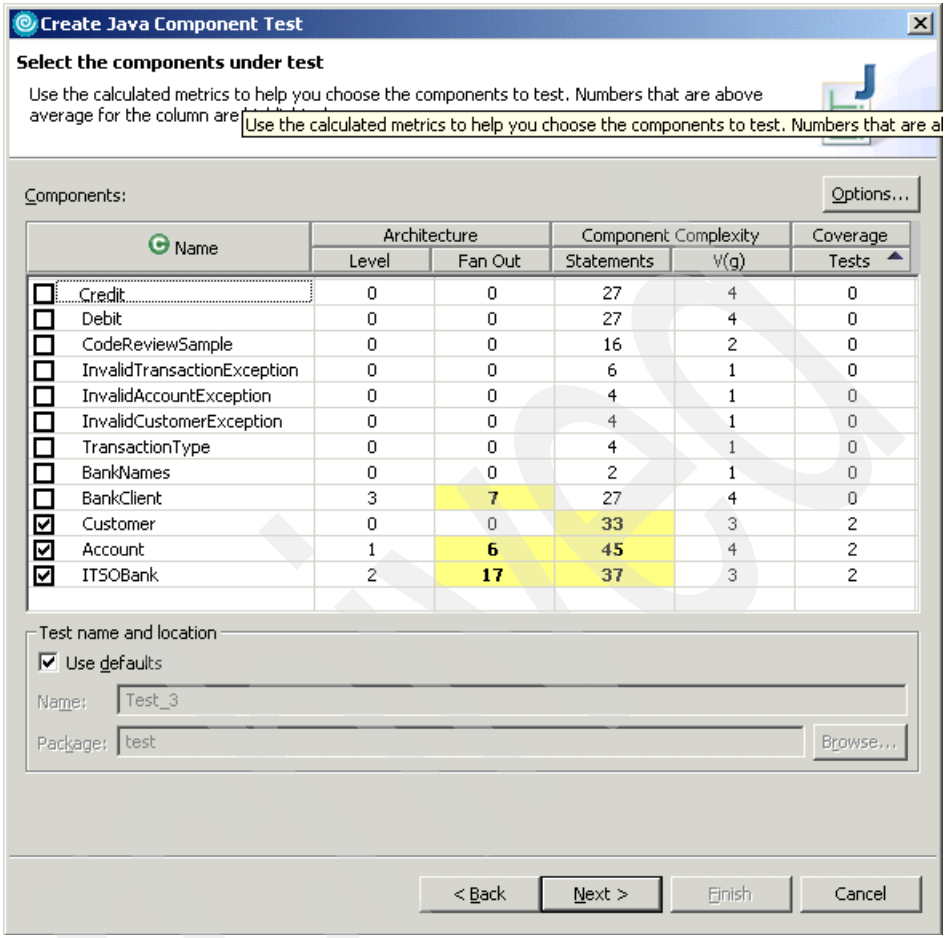


Figure 20-7 Select the components under test

7. When the Select a test pattern dialog appears, select **Scenario-based pattern** and then click **Next**.
8. When the Define a test scenario dialog appears, do the following, as seen in Figure 20-8 on page 1106:
   a. Add an instance of each class-under-test by double-clicking each constructor from the list on the left side**.** For our example, double-click each of the following: **ITSOBank**, **Customer**, and **Account**.
   b. On the right side of the dialog, double-click a particular method to be included in the test scenario. Double-click **addAccount**, **processTransaction**, and **addCustomer**.
   c. When you are finished building the scenario, click **Finish**.

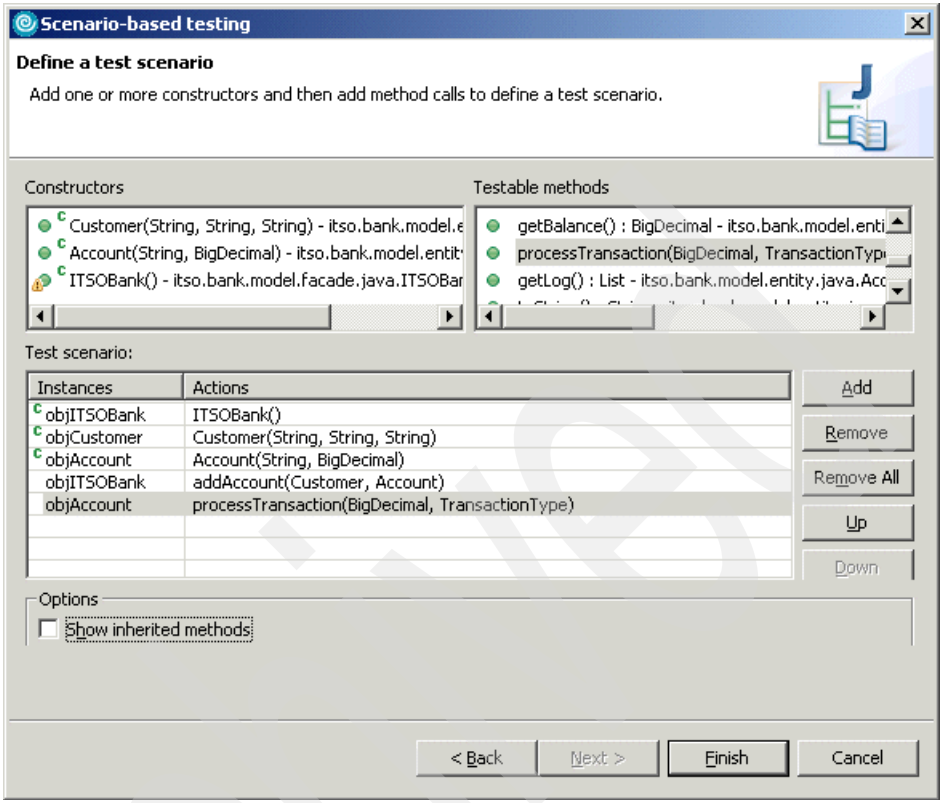


Figure 20-8 Define the test scenario

We have now created a scenario-based test, and a single test case has been created within the test suite. In the test behavior code, the test case is implemented as a single JUnit test method.

The Test Overview dialog appears, as seen in Figure 20-9 on page 1107. From this page, you can edit the name of test, add a description of the test, and open test behavior code in the Java editor.

Click **/BankComponentTest/Behavior/test/Test.java** next to Behavior, to view and start editing the test code.

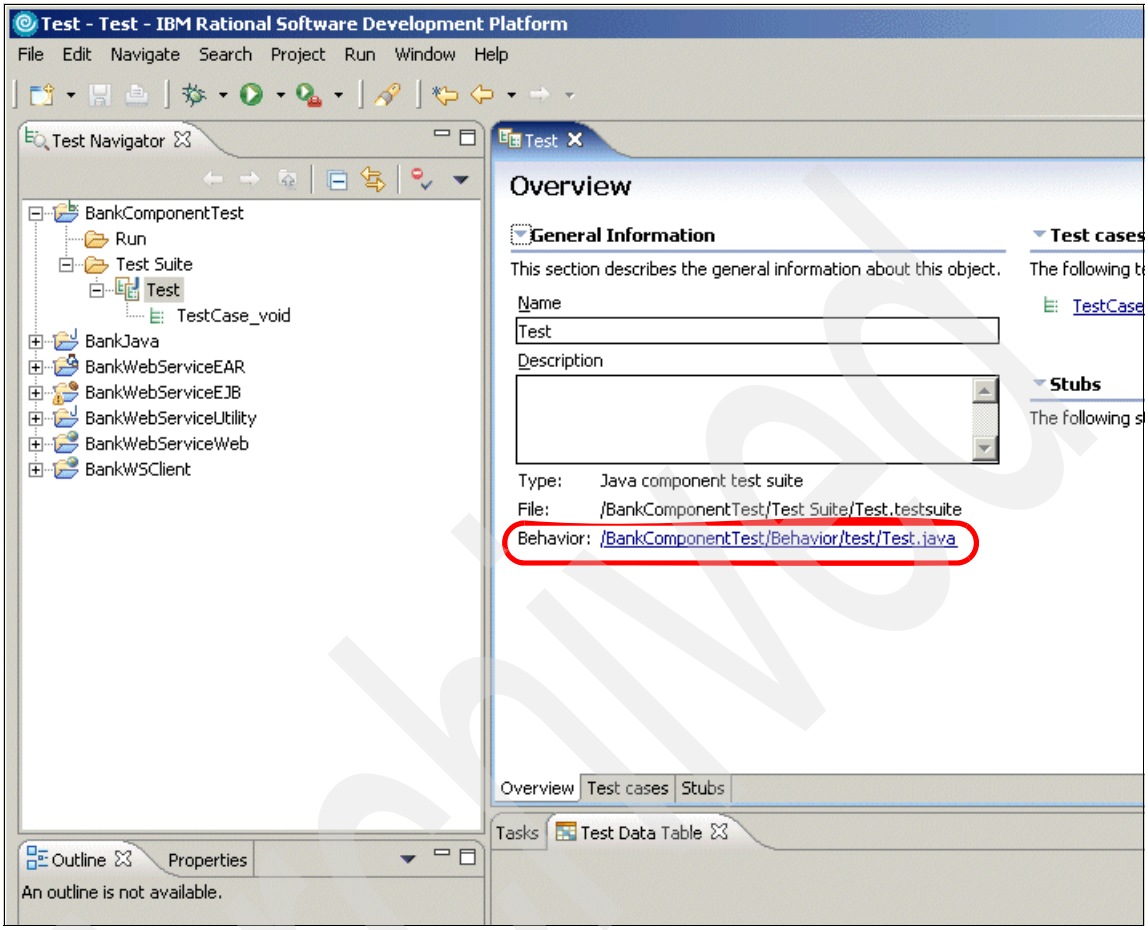


Figure 20-9 Component test for processing a transaction

20.3.4 Complete the component test code

To complete the component test, do the following:

1. The Java test file should be opened in the editor. If not opened, do the following:
   a. From the Test Perspective Test Navigator view, expand **Component Test** → **Test Suite**.
   b. Double-click **Test** to open in the editor.
2. Click **/BankComponentTest/Behavior/test/Test.java** next to Behavior, to view and start editing the test code.

3. Clean up the import statements.

   When the Test.java file is open, you will notice a minor problem with the imports. To fix the import issue, do the following:

   a. Click **import >** to expand the import statements.

   b. Click the twistie next to the import line and then delete the last unnecessary import statement, as seen in Figure 20-10.

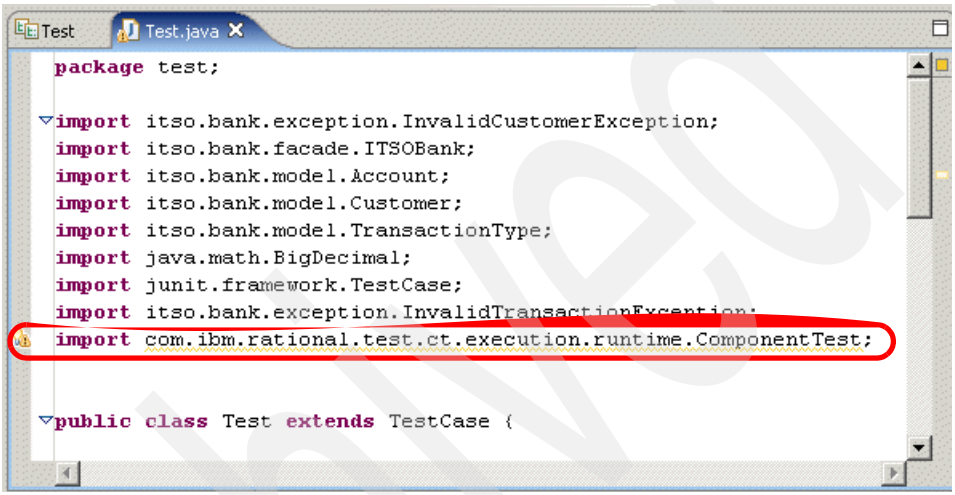


Figure 20-10 Clean up the import statements

4. Modify the code to implement the desired tests. For example:

   – Use test data tables to define test data specific to your test.
   – Use the Java editor to edit the test behavior code.
   – Create stubs for classes that the code you are testing interacts with.

   a. To view a test data table, click any test method in the editor.

   b. Maximize the Test Data Table view.

      We can now see the outline of the test. On the right-hand side there is the Test Data column, which is further divided into the In data column and Expected test result column. Note that some of the cells are plain white and the others are shaded. Our next task is to fill in the empty white cells.

   c. Fill in the test input data in the In column, as shown in Figure 20-11 on page 1109.

      You can define the data ranges for the id and balance fields by right-clicking the field and selecting **Define Set** from the context menu.

Note: Some of the entries will not be present until after modifying the code in the next step, at which time we enter the remaining *In* values.

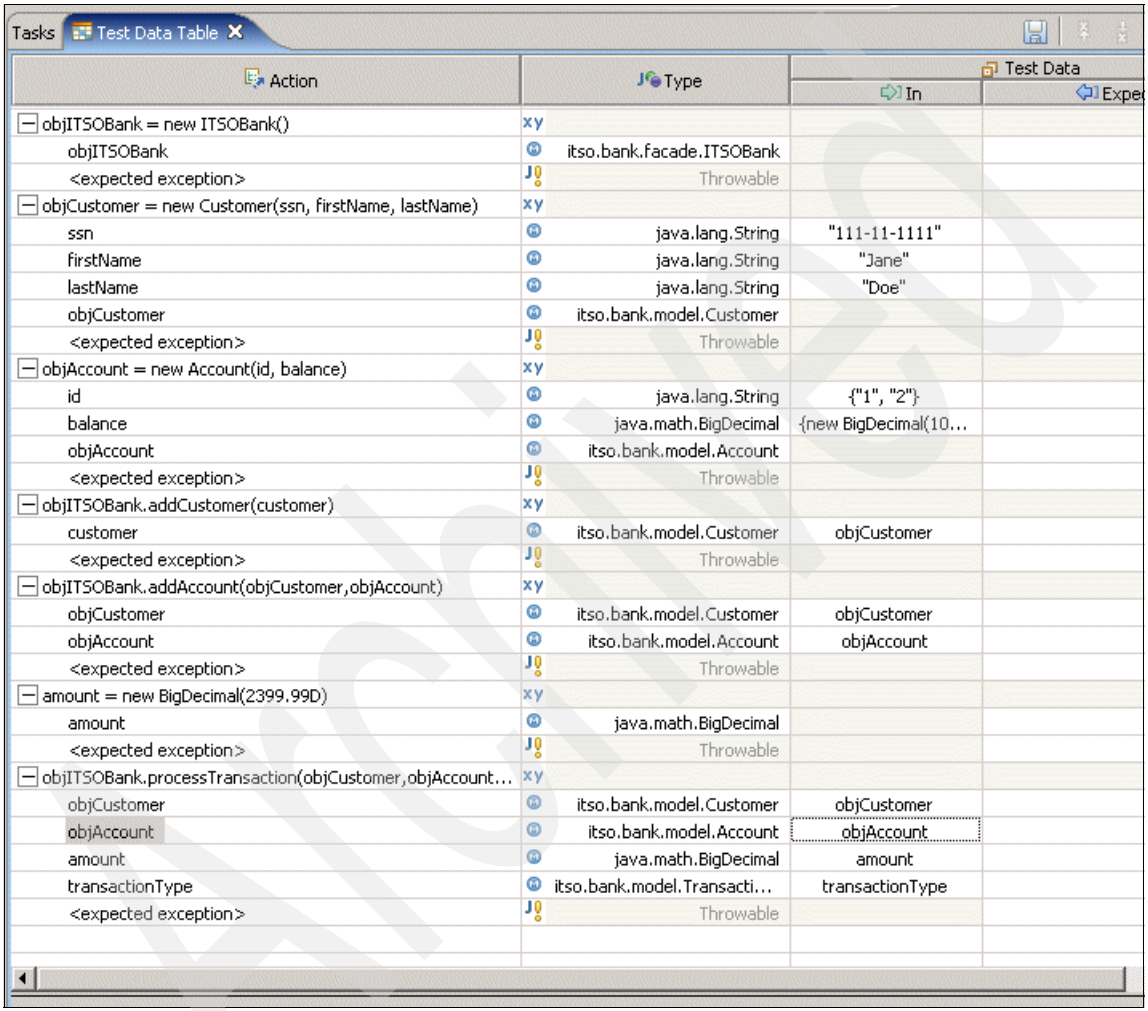


Figure 20-11 Editing the test data table

d. Save your changes by clicking the diskette icon on the toolbar, or by pressing Ctrl+S. There is a synchronization link between the table and the code, but not everything in the code gets displayed in the test table.

Note that you need to enclose the strings and define the data types. Note also that the local *amount* variable gets assigned in the Action part of the test table and that the *type* variable does not appear.

5. Example 20-15 displays the test behavior code for Test.java. The local variables should be assigned based on the actions of the previous steps.

Example 20-15 Modified test behavior code for Test.java

```
public class Test extends TestCase {

   public void test_void() throws InvalidCustomerException,
            InvalidAccountException, InvalidTransactionException {
   ITSOBank objITSOBank = null;
   Customer objCustomer = null;
   Account objAccount = null;
   objITSOBank = new ITSOBank();
   String ssn = "";
   String firstName = "";
   String lastName = "";
   objCustomer = new Customer(ssn,firstName,lastName);
   String id = "";
   BigDecimal balance = null;
   objAccount = new Account(id, balance);
   Customer customer = null;
   objITSOBank.addCustomer(customer);
   objITSOBank.addAccount(objCustomer,objAccount);
   BigDecimal amount = null;
   amount = new BigDecimal(2399.99D);
   TransactionType transactionType = null;
   transactionType = TransactionType.DEBIT;
   objITSOBank.processTransaction(objCustomer,objAccount,
      amount,transactionType);
   }
}
```

6. Now that the code has been modified, If you have not done so already, go back to the Test Data Table and enter the In values listed in Figure 20-11 on page 1109.
7. Save the modifications to the Test.java and close the file.

20.3.5 Run the component test

We can now run the test and view the results. A test runs with the behavior code and uses the additional input data you have supplied for it in a test data table.

As we noticed, you can also supply sets or ranges of values in the test data table. In that case, running a single test results in the running of many individual tests;

for example, in our case where we supplied two values for two arguments, running the test results in four individual tests.

1. From the Test Navigator, expand **BankComponentTest**.
2. Right-click **Test Suite**, and select **Run** → **Component Test**.
3. After the test is completed, expand the **BankComponentTest** → **Run** folder until you find the individual tests, as seen in Figure 20-12.

   We can see that two tests passed and two tests had some kind of failure.

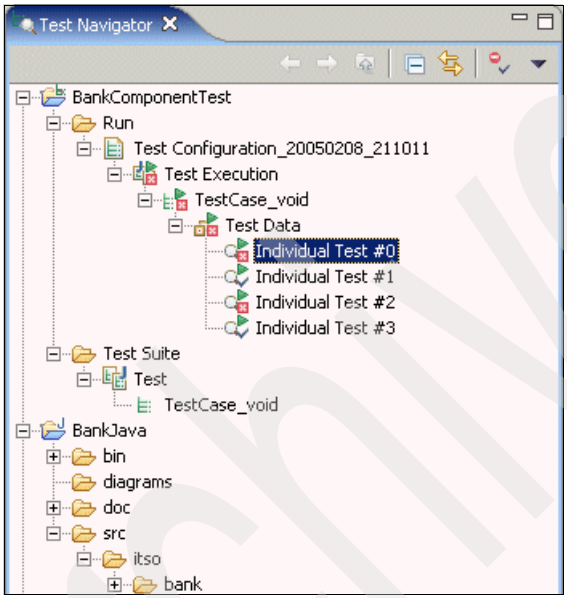


Figure 20-12 Selecting test result

4. Click an individual test to display the test results in the Test Data Comparator.

   The Test Data Comparator is quite similar to the Test Data Table. There are now three columns for test data:

   – Input data (including derived input)
   – Expected output
   – Actual result

   The actual results column appears in green when the actual result matches the expected result, and in red when there are discrepancies.

   As you can see in Figure 20-13 on page 1112, we have discrepancies, as can be expected, because the first balance was too low, and because the second account was not assigned to a customer.

Archived

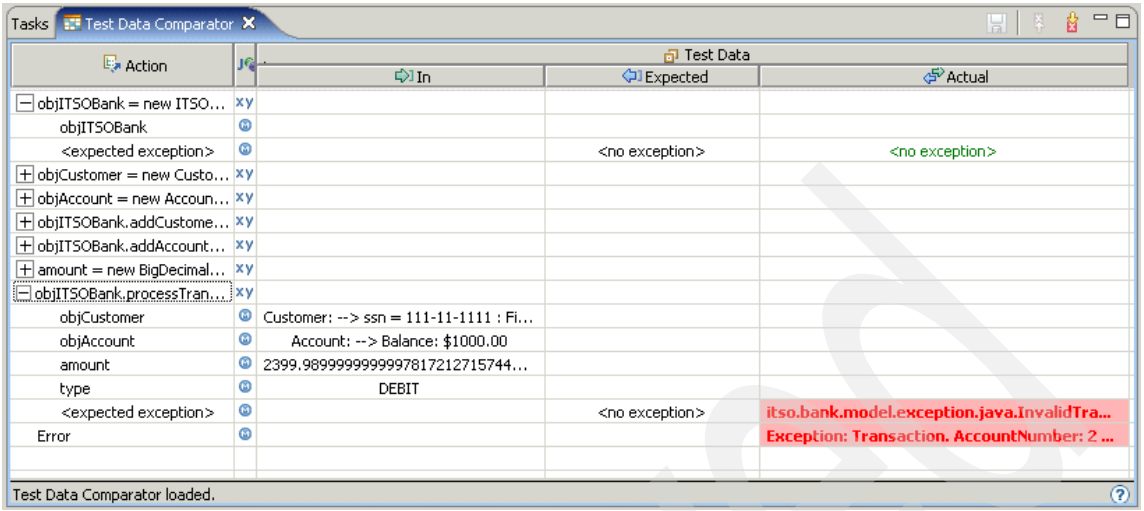


Figure 20-13 Test Data Comparator

5. Try to achieve a 100 percent successful pass by modifying the input in the Test Data Table and Java editor and running the component test again.

Note: For additional information on automated component testing, we recommend that you refer to the Rational Application Developer online help and the tutorial, which can be accessed by clicking **Help** → **Tutorial Gallery** → **Do and Learn** → **Test Java components**.

20.4 Web application testing

In addition to providing a common framework for test tools and support for JUnit test generation, Hyades includes features allowing you to test Web-based applications.

The Hyades framework has an important secondary audience—testers of HTTP-based applications. You can perform the following Web-testing tasks with Hyades, without modifying the framework:

- Recording a test: The test creation wizard starts the Hyades proxy recorder, which records your interactions with a browser-based application. When you stop recording, the wizard starts a test generator, which creates a test from the recorded session.
- Editing a test: You can inspect and modify a test prior to compiling and running it.

- Generating an executable test: Follow this procedure to generate an executable test. Before a test can be run, the test's Java source code must be generated and compiled. This process is called code generation.
- Running a test.
- Analyzing test results: At the conclusion of a test run you see an execution history, including a test verdict, and you can request two graphical reports showing a page response time and a page hit analysis.

20.4.1 Preparing for the sample

As a prerequisite to the Web application testing sample, you will need to have the WebSphere Application Server V6.0 Test Environment or runtime server installed and running. We will run a simple test using the WebSphere Administration Console.

1. Open the J2EE perspective.
2. Click the **Servers** view.
3. Select **WebSphere Application Server v6.0**, right-click, and select **Start**.

20.4.2 Create a Java project

To create a Java project, do the following:

1. Open the Java perspective.
2. Create a new Java project to hold the test case behavior and other test elements.
   a. Select **File** → **New** → **Project**.
   b. Select **Java Project** and then click **Next**.
3. Enter `ITSO HTTP Test` as the project name and click **Finish**.
4. Create a source folder under the project.
   a. Right-click **ITSO HTTP Test** and select **New** → **Source folder**.
   b. Enter `src` as the folder name and click **Finish**.

20.4.3 Create (record) a test

We use the WebSphere Administration Console to demonstrate how to record a HTTP recording for a Web application (could be any application).

Archived

We now create a simple HTTP test case in Rational Application Developer, as follows:

1. Open the Test perspective Test Navigator view.
2. Right-click **ITSO HTTP Test** and select **New** → **Test Artifact**.
3. When the New Test Artifact dialog appears, select **Test** → **Recording** → **HTTP Proxy Recorder**, and then click **Next**.
4. Select the **ITSO HTTP Test** container, enter `adminconsole` in the Recording file name field, and then click **Finish**.

   A progress dialog box opens while your browser starts. Your browser settings are updated and a local proxy is enabled. If you are using a browser other than Microsoft Internet Explorer, see the online help for detailed instructions on how to configure the proxy. Recording has now started.
5. Start the WebSphere Administrative Console by entering the following URL in a Web browser:

   ```
   http://localhost:9060/admin
   ```

   Or:

   ```
   http://localhost:9060/ibm/console
   ```
6. Log in to the console (security is not enabled, so any user ID will work) and access some pages. A small set is sufficient.
7. When done, close the Web browser to stop recording.

   Alternatively, stop the recording by clicking the **Stop** button on the right side of the Recorder Control viewer, as seen in Figure 20-14.

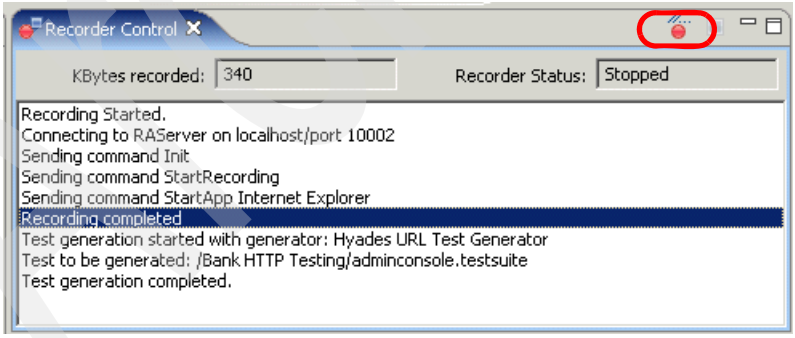


Figure 20-14 Recorder Control view

Notice the message `Recording completed` in the Recorder Control view seen in Figure 20-14 after closing the browser.

20.4.4 Edit the test

The Hyades URL test suite now appears under the HTTP Test project. We can inspect and modify it before compiling and running it. The test is not Java code yet, but we can check the requests and behavior and modify them.

1. Click the **Behavior** tab of the Hyades URL Test Suite.
2. Change the behavior of the test. For example, you may want to adjust the number of iterations or think times for some of the requests, as seen in Figure 20-15.
3. Save and close the file.

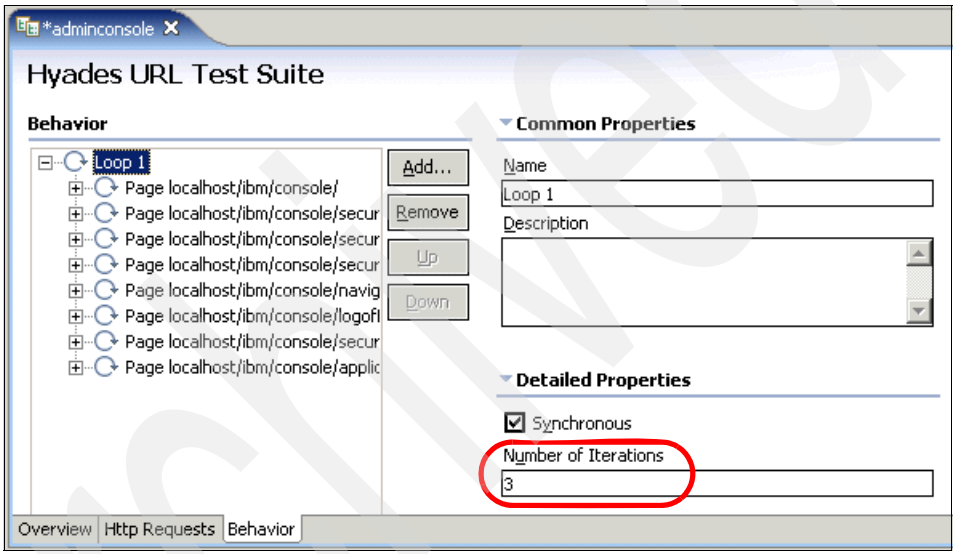


Figure 20-15 Edit number of iterations

20.4.5 Generate an executable test

Before a test can be run, the test's Java source code must be generated and compiled. This process is called *code generation*.

The compiled code is stored in the *src* folder in the same Java project as the test. Once the test's Java source code is generated, the Eclipse IDE automatically compiles the source code into an executable test.

1. In the Test Navigator, right-click **adminconsole** and select **Generate**.
2. When the JUnit Test Definition Code Generation dialog appears, we entered the following and then clicked **Finish**:
   - Java Project: `ITSO HTTP Test`

– Source Folder: `src`

The code is now generated and compiled.

3. To examine the code, switch to the Java perspective and you will find Adminconsole.java JUnit source in the src/test package.

20.4.6 Create a deployment definition

Before a test can be run, it must be deployed. That requires creating a *deployment* definition for the test.

A test deployment definition typically consists of one or more pairs of test *artifacts* and *locations*. Test artifacts are test suites and data pools. A location identifies the computer where you run the test suite. Hyades reads the pairing and deploys the test artifacts on the computer specified.

You can also create a test deployment that specifies only a location and does not specify test artifacts. Such a deployment is convenient when you want to run a specific test suite on a specific computer. Since we have only one test suite in our test, we need to define one deployment definition containing one location.

1. Open the Test perspective Test Navigator view.
2. Right-click **ITSO HTTP Test**, and select **New** → **Test Artifact**.
3. When the New Test Artifact dialog appears, select **Test** → **Test Elements** → **Deployment** and click **Next**.
4. When the New Deployment dialog appears, select **ITSO HTTP Test** as the folder, and enter `deployment` in the File name field, and then click **Next**.
5. When the New Deployment dialog appears, click **Next**. We have only one test suite and it will be automatically included in the deployment.
6. When the New Deployment - define the locations dialog appears, click **Add**.
   a. In the Add Location Association dialog, select **Create a new resource** and click **Next**.
   b. In the New Location dialog, select **ITSO HTTP Test** as the folder and enter `location` as the filename.
   c. Click **Finish**.
7. In the New Deployment - define the locations dialog, make sure the location entry appears in the list of locations, and then click **Finish**.

   You should now have the structure in Test Navigator seen in Figure 20-16 on page 1117.

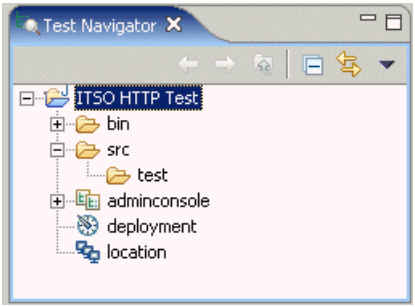


Figure 20-16 Testing artifacts in Test Navigator

20.4.7 Run the test

The test code has now been generated and deployment has been defined. To run the test, do the following:

1. Open the Test perspective Test Navigator.
2. Right-click **adminconsole**, and select **Run** → **Run**.
3. When the Create, manage, and run configurations dialog appears, select **Hyades URL Test** and then click **New**.
4. A new test configuration, initially named New_Configuration is created. Change the name to `ITSO URL Test`.
5. In Select Test to run, navigate to the **ITSO HTTP Test** project and click **adminconsole**.

   The deployment definition should now appear under the deployments selection, as seen in Figure 20-17 on page 1118.

Archived

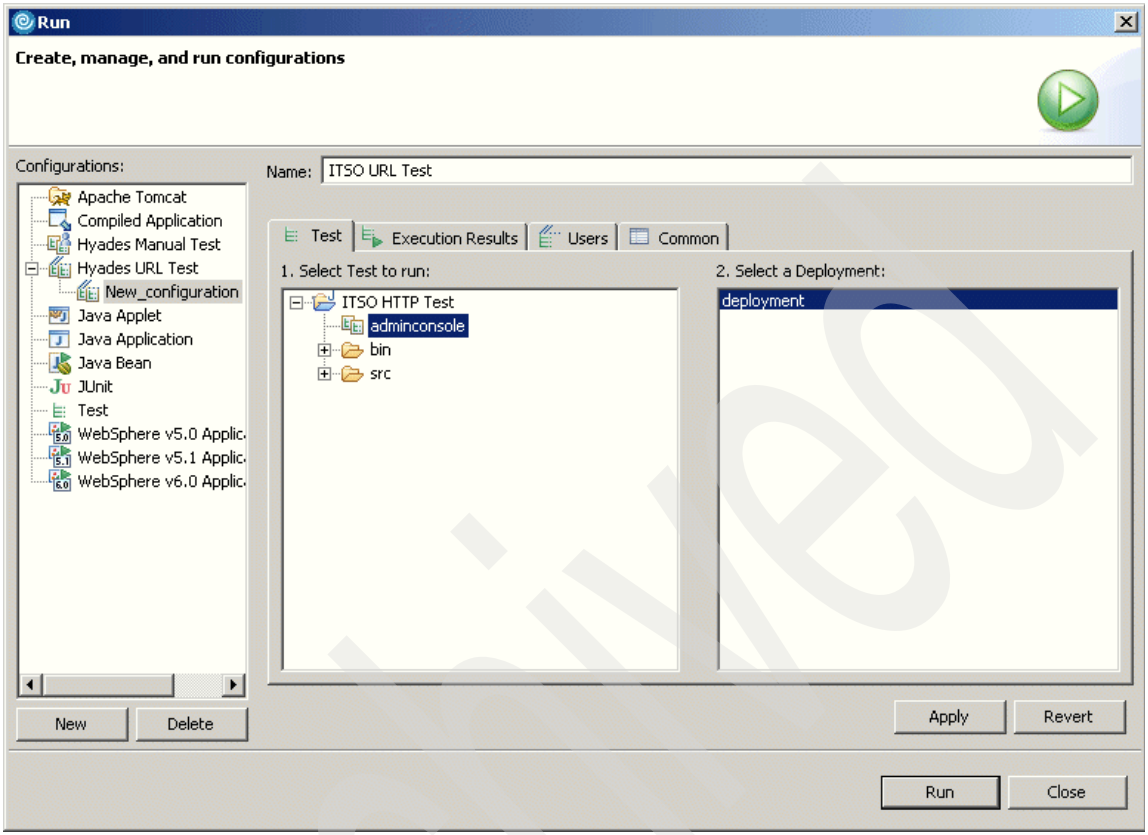


Figure 20-17 Creating the run configuration

6. Select the **Users** tab.

   You may set the number of users to emulate in the run. If you are running everything locally, do not use too high a number.

7. When done click **Apply**.

8. Select the test case to run, such as **adminconsole**, and then click **Run**.

20.4.8 Analyze the test results

When the test run is finished, the execution result appears in the Test Navigator. If you execute the test multiple times, a running sequence number is appended to the result, as seen in Figure 20-18 on page 1119.

Archive

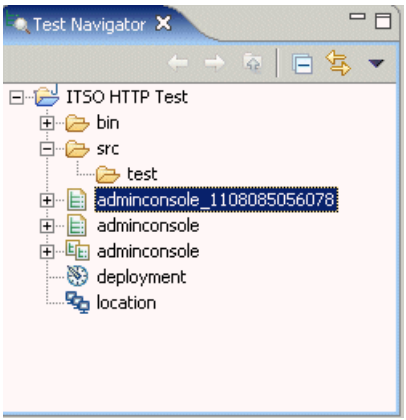


Figure 20-18 Test execution results

1. In order to view the Page Response Time and the Page Hit Rate reports, you need to install Adobe's Scalable Vector Graphics (SVG) browser plug-in. You can get this free viewer from the Adobe Web site at:

   `http://www.adobe.com/svg/viewer/install/main.html`

2. Double-click the execution, and the execution summary is displayed.

   The execution summary gives the test's verdict, the time recording started, and the time recording stopped. The verdict may be one of the following:

   – fail: One or more requests returned a code of 400 or greater, or the server could not be reached during playback.

   – pass: No request returned a code of 400 or greater.

   – inconclusive: The test did not run to completion.

   – error: The test itself contains an error.

   For tests that fail, the Events tab shows you the overall verdict and allows you to drill down to the requests in each page that returned a fail code.

   Two analysis reports are available.

   – Page Response Time report: Bar graph showing the seconds required to process each page in the test and the average response time for all pages.

   – Page Hit Rate report: Bar graph showing the hits per second to each page and the total hit rate for all pages.

3. To generate the reports from the execution result, right-click the **adminconsole** test suite (the last adminconsole entry with the icon) and select **Report** from the context menu.

Archived

4. When the New report dialog appears, select **HTTP Page Response Time** and then click **Next**.
5. When the New Report - Report dialog appears, select **ITSO HTTP Test** and enter `adminconsole page response time` as the file name, and click **Finish**.
6. If you have multiple test results, the HTTP Report Generator - select result for report dialog appears next. Select a result you want to base the report on and click **Finish**.

   A report like the one in Figure 20-19 should appear.

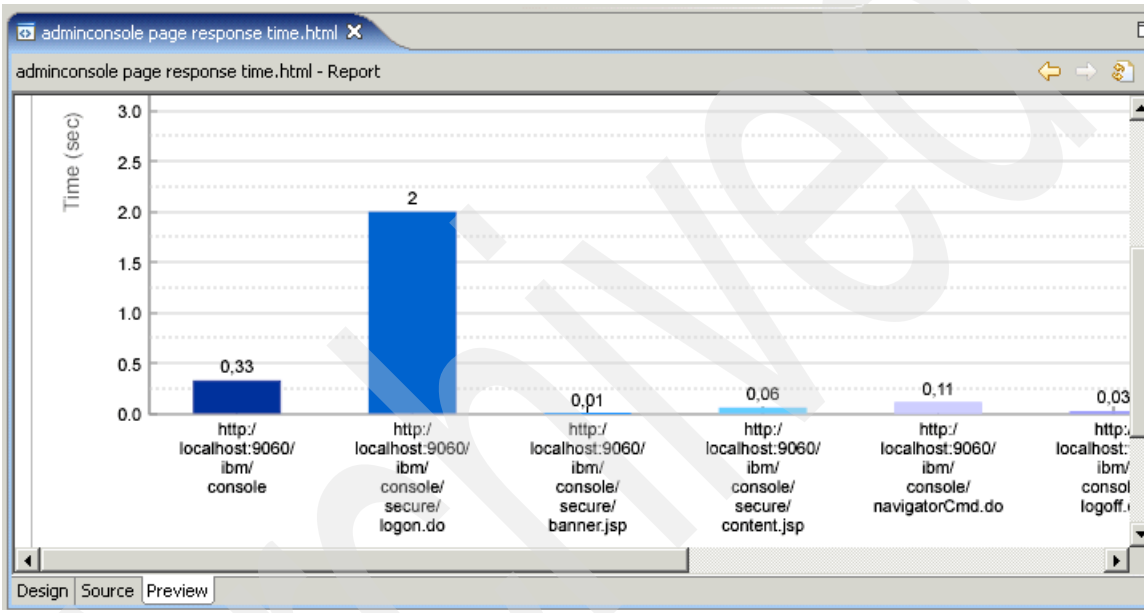


Figure 20-19 HTTP Page Response Time

The exact numbers displayed at the top of the bars do not appear unless you click one of the bars.

If you need to access the report later, you can find it under the Package Explorer view: `adminconsole page response time.html`.

Archived

21

Debug local and remote applications

The debug tooling included with IBM Rational Application Developer V6.0 can be used to debug a wide range of applications (languages and environments) either in a local integrated test environment or on remote servers such as WebSphere Application Server or WebSphere Portal.

In this chapter, we highlight the new and enhanced debug tooling features included with Rational Application Developer, as well as provide examples for using the debug tooling. In the first example, we demonstrate how to use the debugger within the Workbench and integrated WebSphere Application Server V6.0 Test Environment with Web application. Second, we describe how to debug a Web application on a remote WebSphere Application Server.

This chapter describes the following topics:

- Introduction to the debug tooling
- Prepare for the sample
- Debug a Web application on a local server
- Debug a Web application on a remote server

© Copyright IBM Corp. 2005. All rights reserved.

21.1 Introduction to the debug tooling

This section provides an overview of the following new and enhanced debug tooling features included in IBM Rational Application Developer V6.0. The debug tooling can be used on local or remote test environments.

- Summary of new Version 6 features
- Supported languages and environments
- General functionality
- Drop-to-frame
- View Management
- XSLT debugger

Note: Most of the features outlined in this section include screen shots to display the menu options and dialogs. If you want to see these features on a live Rational Application Developer system, we suggest that you jump ahead to 21.3, “Debug a Web application on a local server” on page 1132. By completing the setup for this section, you will have a Web application imported to test these new and enhanced features.

21.1.1 Summary of new Version 6 features

The main areas of enhancements in Version 6 are as follows:

- Debug many different languages and environments, including mixed languages.
- User interface enhancements to make debugging the application easier.
- XSLT debugger now utilizes Eclipse Debug framework. This provides a common look and feel for the debuggers in Rational Application Developer.
- Debug option available from context menu.

  Figure 21-1 on page 1123 displays the Debug configuration options.
- Debug from any perspective.

  When you run an application using Debug on Server, you will be prompted as to whether you want to switch to the Debug perspective. You can choose to debug in the current perspective (Web, Java, etc.) as well as the Debug perspective.
- New debug predefined configuration options.

  For a list of new predefined configurations see Figure 21-1 on page 1123. You may choose to use an existing configuration or define your own settings for your application.

Archived

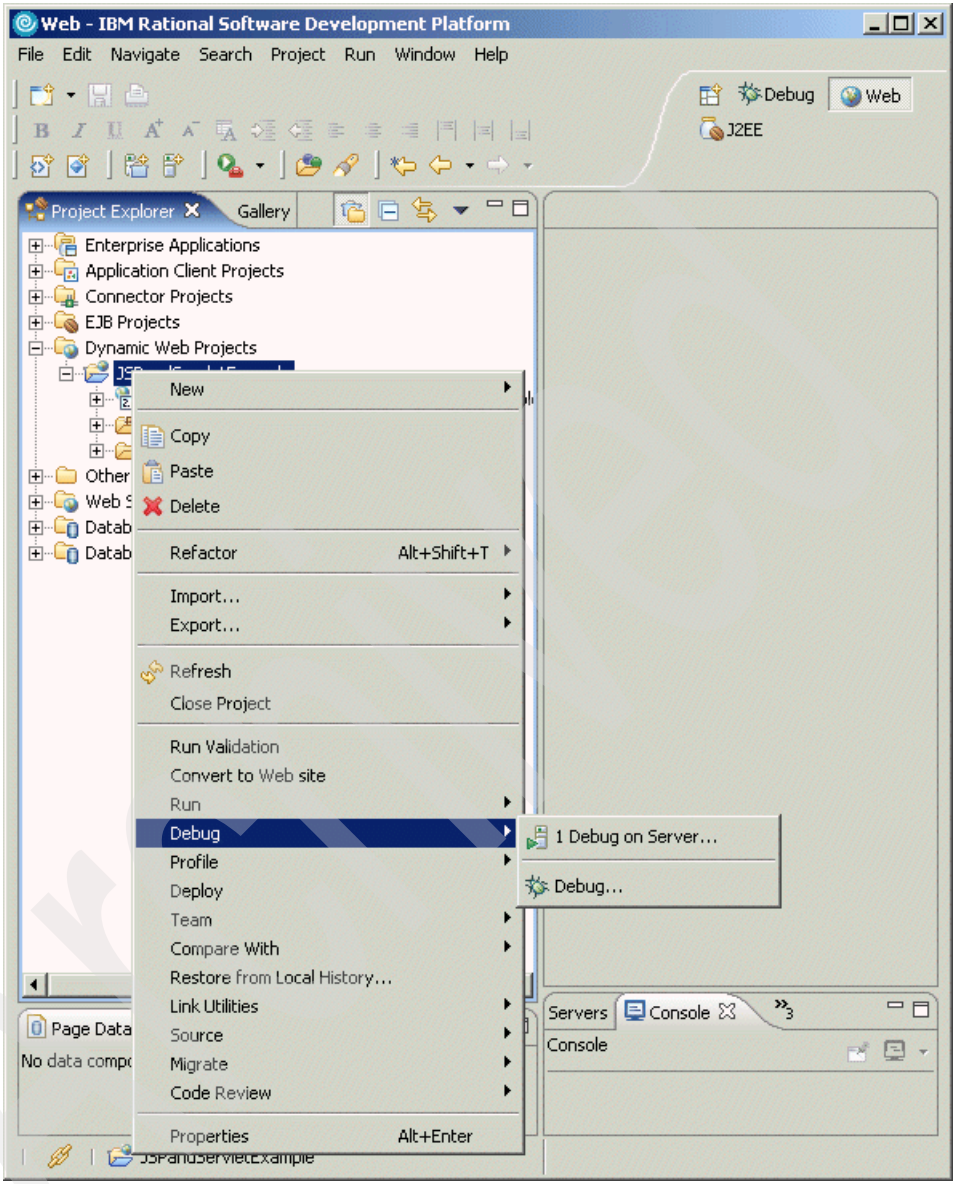


Figure 21-1 Debug - Debug on Server

21.1.2 Supported languages and environments

IBM Rational Application Developer V6.0 includes support for debugging many different languages and environments, including mixed languages:

- Java
- Compiled Languages
- Active Script (client-side JavaScript or VisualBasic script)
- SQL Stored Procedures
- EGL
- XSL Transformations (XSLT)
- SQLJ
- Mixed language (new to V6)
- WebSphere Application Server (servlets, JSPs, EJBs, Web Services)
- WebSphere Portal (portlets)

21.1.3 General functionality

In this section, we explore some features that provide general functionality throughout the debugging tools.

Breakpoint enable/disable

Breakpoints can be enabled and disabled. To enable a breakpoint in the code, double-click in the grey area of the left frame for the line of code you for which to enable the breakpoint.

Alternatively, the breakpoints can be enabled and disabled from the Breakpoints view (as seen in Figure 21-2) once they have been created. If the breakpoint is unchecked in the Breakpoints view, it will be skipped during execution.

To disable or enable all breakpoints, click the Breakpoint icon in the tool bar highlighted in the Debug perspective, as seen in Figure 21-2. If disabled, the next execution will skip the breakpoint.

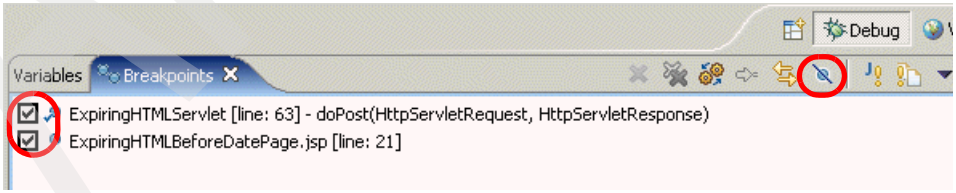


Figure 21-2 Breakpoint view - Enable/disable breakpoints

Archived

Step-by-step disable default

By default, step-by-step debugging is disabled in the Workbench preferences. To enable step-by-step debug, do the following:

1. Select **Window** → **Preferences**.
2. Expand **Run/Debug** → **Java and Mixed Language Debug**.
3. Check **Enable step-by-step debug mode by default** (as seen in Figure 21-3), and then click **OK**.

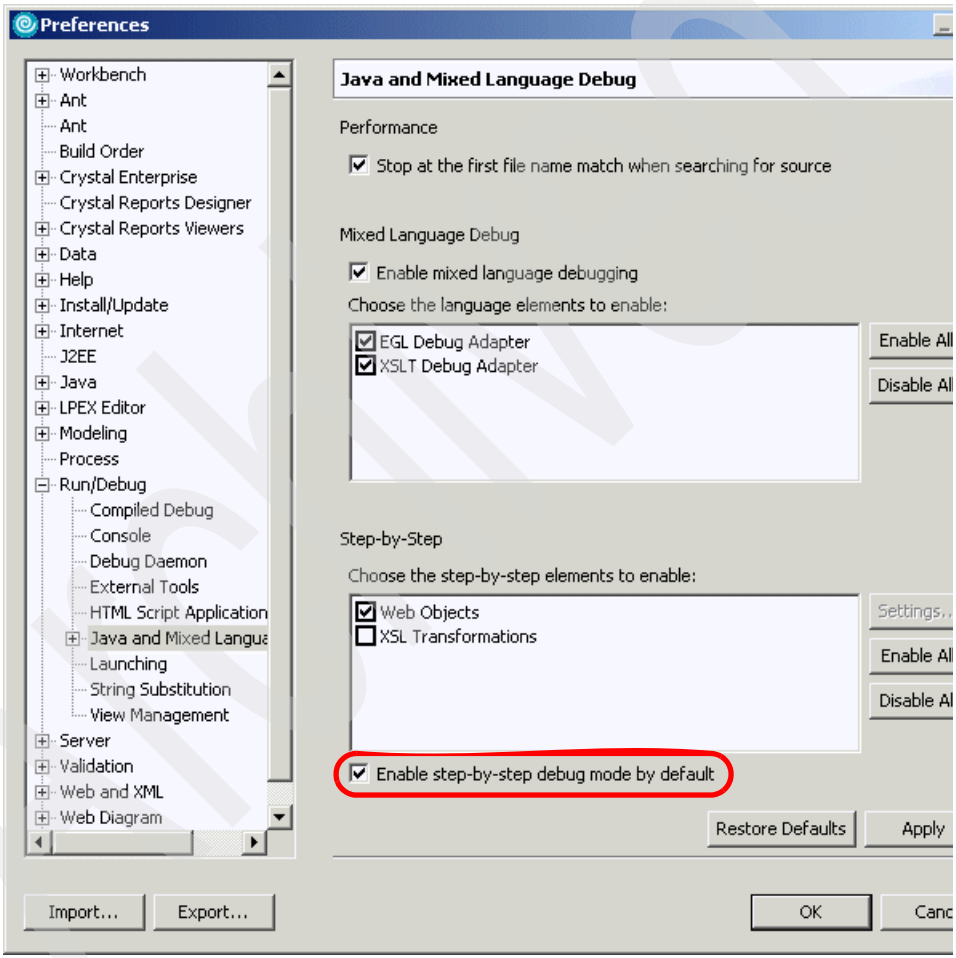


Figure 21-3 Enable step-by-step debug

Once the step-by-step debug feature is enabled, it can be toggled on and off by clicking the Step-By-Step Mode icon ( ) in the Debug view.

Enable/disable Step Filter/Step Debug in Debug view

Within the Debug view, there is a new Step Filter/Step Debug icon ( ). This feature allows step functions such as step into, step over, etc. to be used. By default this feature is disabled in the Workbench preferences.

To enable the Step Filter/Step Debug feature in the Debug view, do the following:

1. Select **Window** → **Preferences**.
2. Expand **Run/Debug** → **Java and Mixed Language Debug** → **Step Filters**.
3. Click **Add Filter**.
4. Enter the new filter and click **OK**.

Once the Step Filter/Step Debug feature is enabled, it can be toggled on and off by clicking the Step Filter icon ( ) in the Debug view.

21.1.4 Drop-to-frame

The Drop-to-frame feature allows you to back up execution of the application. This feature is available when debugging Java applications and Web applications running on WebSphere Application Server. This feature is useful when you need to test a range of values. With this feature multiple input values can be entered without having to rerun the application.

When running an application in the Debug perspective, you will see the stack frame in the Debug view, as seen in Figure 21-4. Drop-to-frame allows you to back up your application's execution to previous points in the call stack by selecting the desired frame and then clicking the Drop-To-Frame icon ( ).

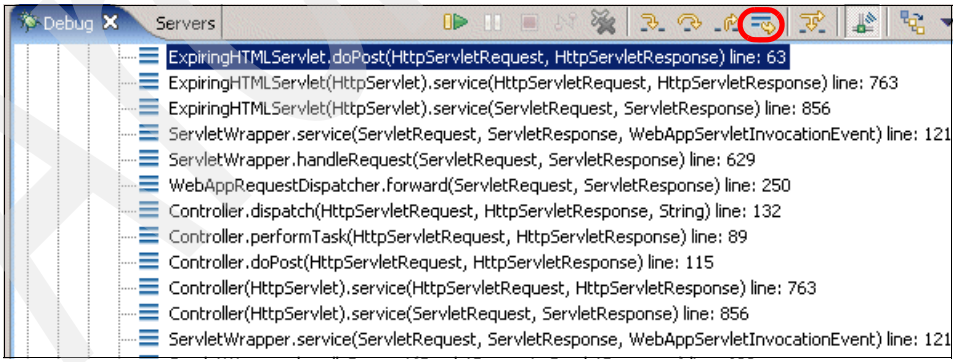


Figure 21-4 Debug view - Drop-to-frame feature

Archived

21.1.5 View Management

The View Management feature allows debug-related views to be opened in non-debug perspectives. This helps reduce user interface clutter by only opening the views necessary for debugging for the launched process. After the process has terminated the opened debug-related views are closed.

The View Management feature can be configured through either the Debug view or the Workbench preferences. The Debug perspective participates in View Management by default. Additional perspectives can be configured to participate. For example:

- Debugging the Java process opens Debug, Breakpoints, Variables, and Expressions views.
- Debugging the compiled language process opens Debug, Breakpoints, Variables, Registers, Memory Rendering, Monitors, and Modules views.

To configure the View Management from the Workbench preferences, do the following:

1. Select **Window** → **Preferences**.
2. Expand **Run/Debug** → **View Management**.
3. From the View Management tab, check the desired perspective(s) to be enabled, as seen in Figure 21-5 on page1124. This feature is used to determine which perspective(s) and supporting views will be displayed when the Debug view is opened and closed. Click **OK** when done.

Archived

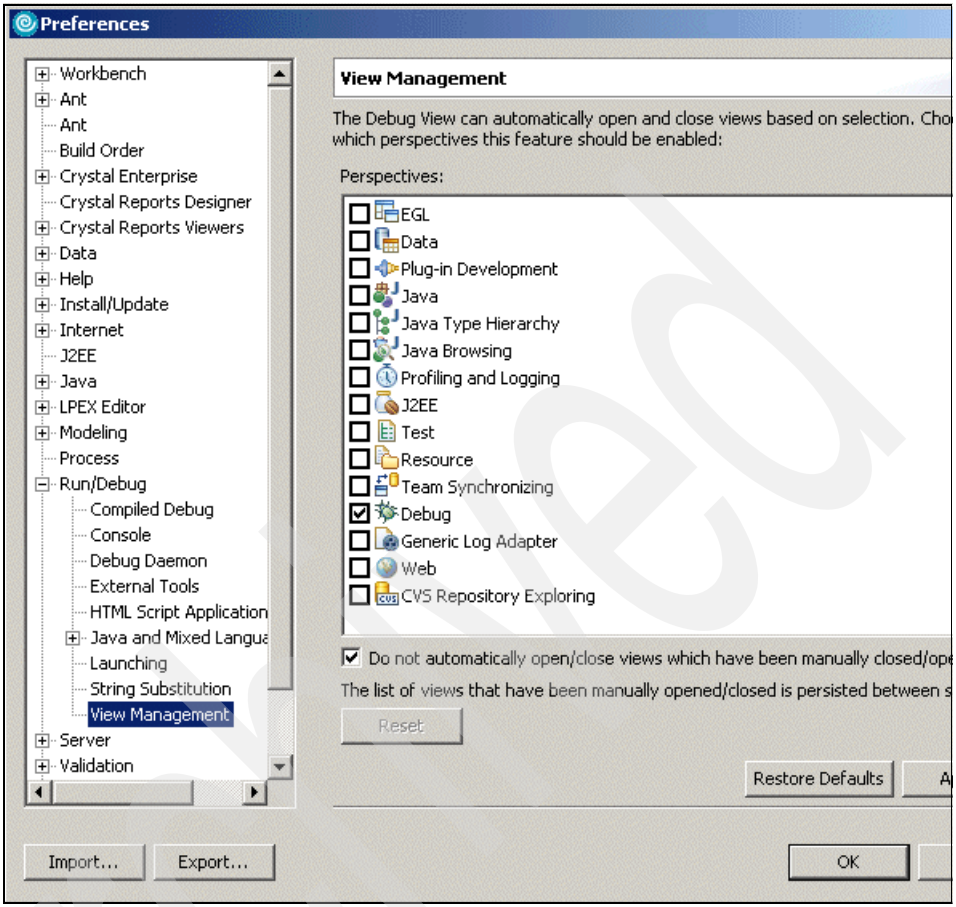


Figure 21-5 View Management

21.1.6 XSLT debugger

XSL Transformations (XSLT) is a language for transforming XML documents into other XML documents. XML Path Language (XPath) is used in matching parts of the source XML document with one or more predefined templates in the transformation script. For example, XSLT can be used to transform an XML document into an HTML document.

Archived

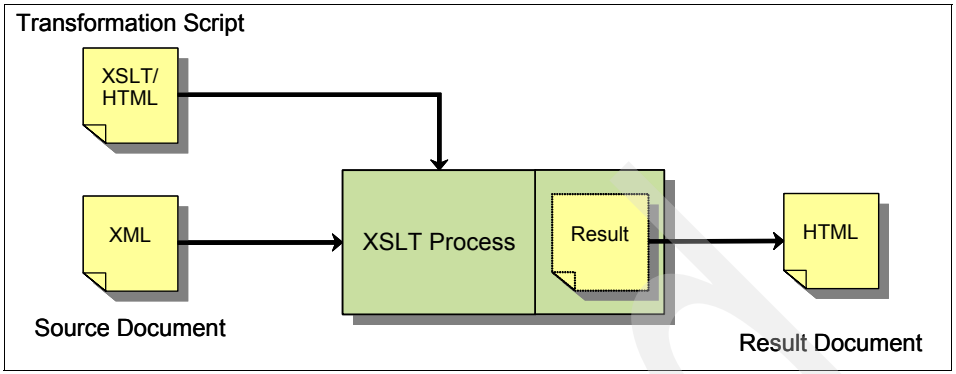


Figure 21-6 Example XSLT transform XML to HTML

In IBM Rational Application Developer V6, the XSLT debugger is implemented using an Eclipse Debug framework. Some of the new capabilities include a similar look and feel to other debuggers, the ability to set breakpoints in code, single step through code, and a XSLT Context view.

Transformations can be debugged even when the source is in DOM or SAX format. The XSLT debugger supports XSL Transformations (XSLT) Version 1.0:

`http://www.w3.org/TR/xslt`

We have listed some of the other key features of the new XSLT debugger:

- Launch configurations
  - Standalone XSL transformations
    - Apply XSL directly to a single XML file (able to step into Java extensions called from the transformation).
    - Allows user to verify transformations outside of a complex environment.
  - Mixed language transformations
    - Use when debugging Java to XSLT or XSLT involving Java extensions.
    - Allows user to debug transformations in real-world scenarios.
- New preferences
  - Node-by-node stepping

    Enabling allows the user to step between nodes that appear on the same line (disabled by default).
  - Built-in template rules filter
    - Enabling causes the debugger to step over built-in template rules (enabled by default).

    - Built-in rules are still displayed on call stack.
  - Step-by-step debugging for XSLT

    Enabling allows step-by-step functionality to additionally work with XSLT debugging (disabled by default).

- Debug activities
  - Iterate through XSLT using standard debugging operations.
  - Add/remove breakpoints (set in both the XSL source and the XML input).
  - Set watches on XPath expressions.
  - View XSLT processor execution using XSLT Context view.
  - View XSLT output using XSL Transformation Output view.
- Setting breakpoints
  - XSL files can be set anywhere between opening and closing template tags.
  - XML files must be set on the line containing the closing “>” of either the opening tag or the closing tag of an element.
  - Breakpoints set in files generated by the transformation will not be persisted between debug sessions.
- XSLT view
  - Visualizes XSLT processor’s execution
  - Enables users to debug XPath expressions
- XSL Transformation Output view
  - View output from transformation in either a text or Web browser viewer.
  - Supports Xalan redirect extension.
- Troubleshooting
  - Verify launch configuration problems by checking the configuration’s properties (Edit Launch Configuration properties through **Run** → **Debug**).
  - Tracing for XSLT debugging can be enabled by modifying the options file in <rad_home>\rwd\eclipse\plugins\com.ibm.debug.xsl_6.0.0.
  - Increase debugger timeout values if timeout errors occur by configuring through **Window** → **Preferences** → **Java** → **Debug**.
  - Verify that problems encountered when debugging Java in Mixed Language debugger also occur in Java debugger (mixed Language debugger is based on Java debugger).
  - Verify Java process used to launch transformation has correct arguments (view JVM launch properties by right-clicking the process in the Debug view and selecting **Properties**).

Archived

- XML capabilities need to be enabled in order to use XSLT debugging functionality.
- Configure through **Window** → **Preferences** → **Workbench** → **Capabilities**.

Note: For an example of using the XSLT debugger, refer to 21.1.6, “XSLT debugger” on page 1128.

21.2 Prepare for the sample

This section describes how to set up the environment in preparation for the debug sample. We will use the ITSO RedBank Web application sampled developed in Chapter 11, “Develop Web applications using JSPs and servlets” on page 499, to demonstrate the debug facilities.

Import the sample application

To import the ITSO RedBank JSP and Servlet Web application Project Interchange file (BankBasicWeb.zip), do the following:

1. Open the Web perspective Project Explorer view.
2. Right-click **Dynamic Web Projects**, and select **Import** → **Import**.
3. When the Import dialog appears, select **Project Interchange** and then click **Next**.
4. In the Import Projects screen, browse to the c:\6449code\web folder and select **BankBasicWeb.zip**. Click **Open**.
5. Check the **BankBasicWeb** and **BankBasicWebEAR** projects, and click **Finish**.

Verify the sample application

To verify the sample application was imported properly, we recommend that you publish and run the sample Web application on the WebSphere Application Server V6.0 test server as follows:

1. Open the Web perspective.
2. Expand **Dynamic Web Projects** → **BankBasicWeb** → **WebContent**.
3. Right-click **index.html**, and select **Run** → **Run on Server**.
4. When the Server Selection dialog appears, select **Choose and existing server**, select **WebSphere Application Server v6.0**, and click **Finish**.

   This operation will start the server, and publish the application to the server.

5. When the Login page appears, enter 111-11-1111 in the Customer SSN field, and then click the **Login** button.

21.3 Debug a Web application on a local server

This section includes a Web application scenario in which the application will run, and shows hot to debug the local WebSphere Application Server V6.0 Test Environment. We use the expiring page sample included as part of the JSP, a Servlet Web application imported in 21.2, “Prepare for the sample” on page 1131.

The local debug example includes the following tasks to demonstrate the debug tooling and features of Rational Application Developer:

- Set breakpoints in a servlet.
- Set breakpoints in a JSP.
- Start the application for debugging.
- Debug view with stack frames.
- Debug functions.
- Breakpoints view.
- Watch variables.
- Inspect variables.
- Evaluate an expression.
- Debug a JSP.

21.3.1 Set breakpoints in a servlet

Breakpoints are indicators to the debugger that it should stop execution at specific places in the code, and let you step through it. Breakpoints can be set to trigger always or when a certain condition has been met.

In the ITSO RedBank sample application, before allowing the withdrawal of funds from an account, the amount requested to be withdrawn is evaluated with the amount that exists in the account. If there are adequate funds, the withdrawal will complete. If there are not enough funds in the account an InsufficientFundsException should be thrown. In this example, we set a breakpoint on the condition that tests the amount to withdraw does not exceed the amount that exists in the account.

To add a breakpoint in the code, do the following:

1. Select and expand the **Dynamic Web Projects** → **BankBasicWeb** → **Java Resources** → **JavaSource** → **itso.bank.facade**.
2. Double-click **MemoryBank.java** to open the file in the Java editor.

Archived

3. Place the cursor in the gray bar (along the left edge of the editor area) on the following line of code:

```
if (account.getBalance() > amount)
```

> **Tip:** Use the Outline view to find the withdraw method to quickly find the source code listed.

4. Double-click to set a breakpoint marker, as seen in Figure 21-7 on page 1133.

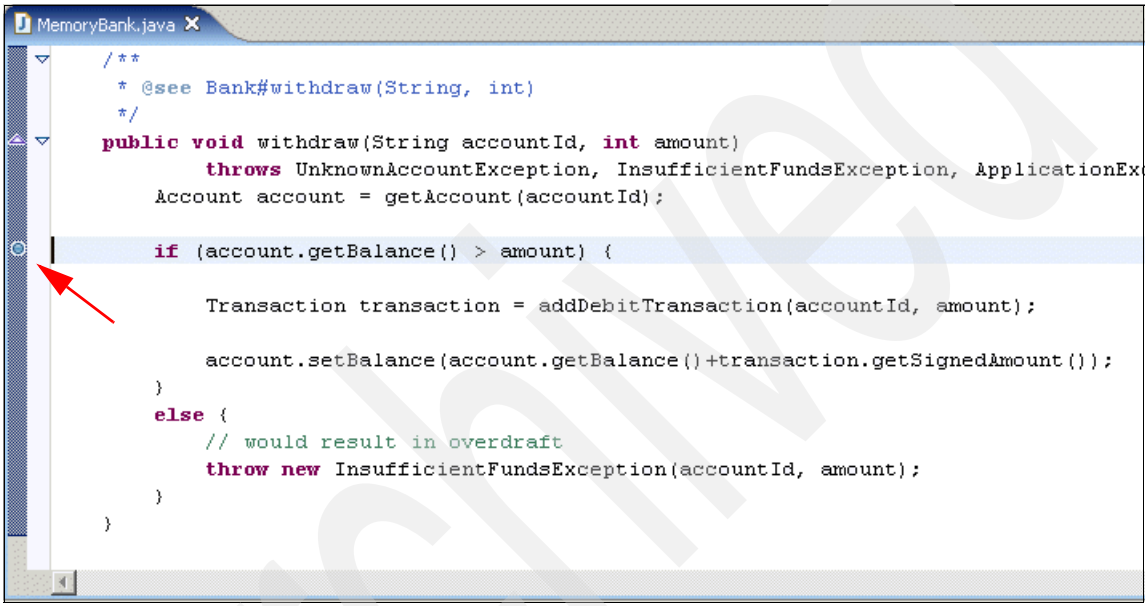


Figure 21-7 Add a breakpoint

> **Note:** Enabled breakpoints are indicated with a blue circle. If the enabled breakpoint is successfully installed in a class in the VM at runtime, it is indicated with a check mark overlay.

5. Right-click the breakpoint in the breakpoint view, and select **Breakpoint Properties** from the context menu.

   The Breakpoint Properties window should appear with more detailed options about the breakpoint, as seen in Figure 21-8 on page 1134.

Archived

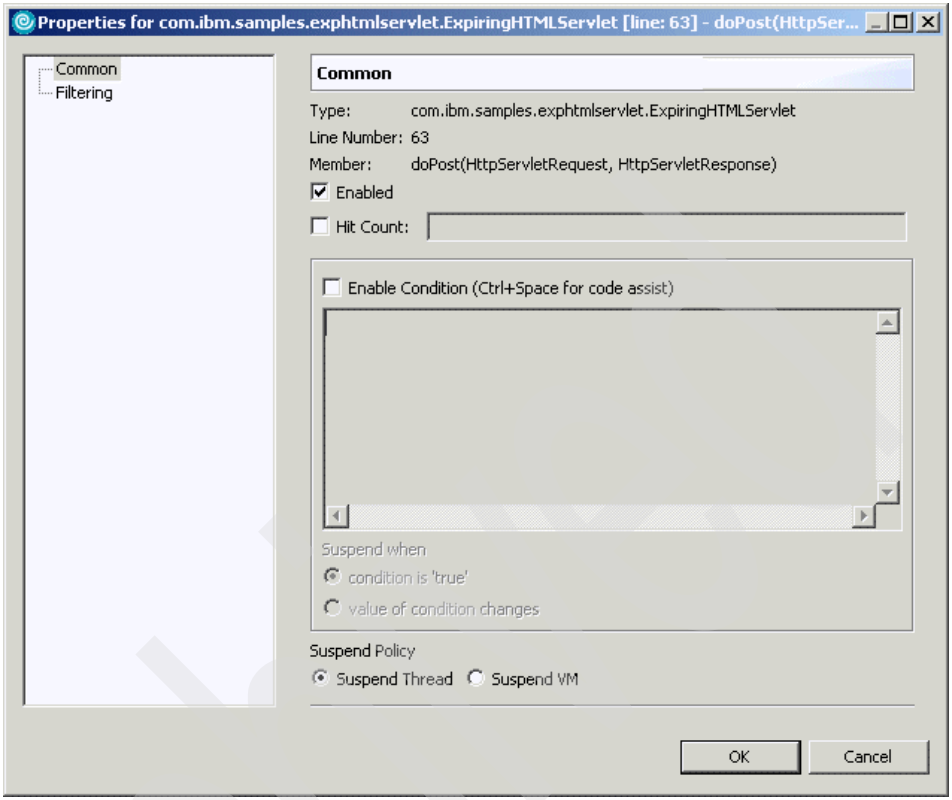


Figure 21-8 Breakpoint properties

- The Hit Count property, when set, causes the breakpoint to be triggered only when the lines have been executed as many times as the hit count specified. Once triggered, the breakpoint is disabled.
- The other property of interest here is *Enable Condition*. If set, then the breakpoint is reached only when the condition specified in the entry field evaluates to *true*. This condition is a Java expression. When this is enabled the breakpoint will now be marked with a question mark on the breakpoint, which indicates that it is a conditional breakpoint.

> **Note:** For example, check **Enable Condition**, select **condition is true**, and enter `amount==1000` in the Enable Condition text box, and then click **OK**. Remember, in this application 1000 is really $10.00. When the application is run and 1000 is entered to withdraw, this conditional breakpoint will be hit in the debugger.

6. Click **OK** to close the breakpoint properties.

21.3.2 Set breakpoints in a JSP

You can also set breakpoints in the JSP source code. However, you can only set breakpoints inside Java scriptlets or other JSP tags, such as JSTL tags.

In the following example, we set a breakpoint in the listAccounts.jsp at the point where the JSP displays a list of accounts for the customer.

1. Expand **Dynamic Web Projects** → **BankBasicWeb** → **WebContent**.
2. Double-click **listAccounts.jsp** to open the file in the editor.
3. Click the **Source** tab.
4. Set a breakpoint as shown in Figure 21-9 on page 1135 by double-clicking in the grey area next to the desired line of code.

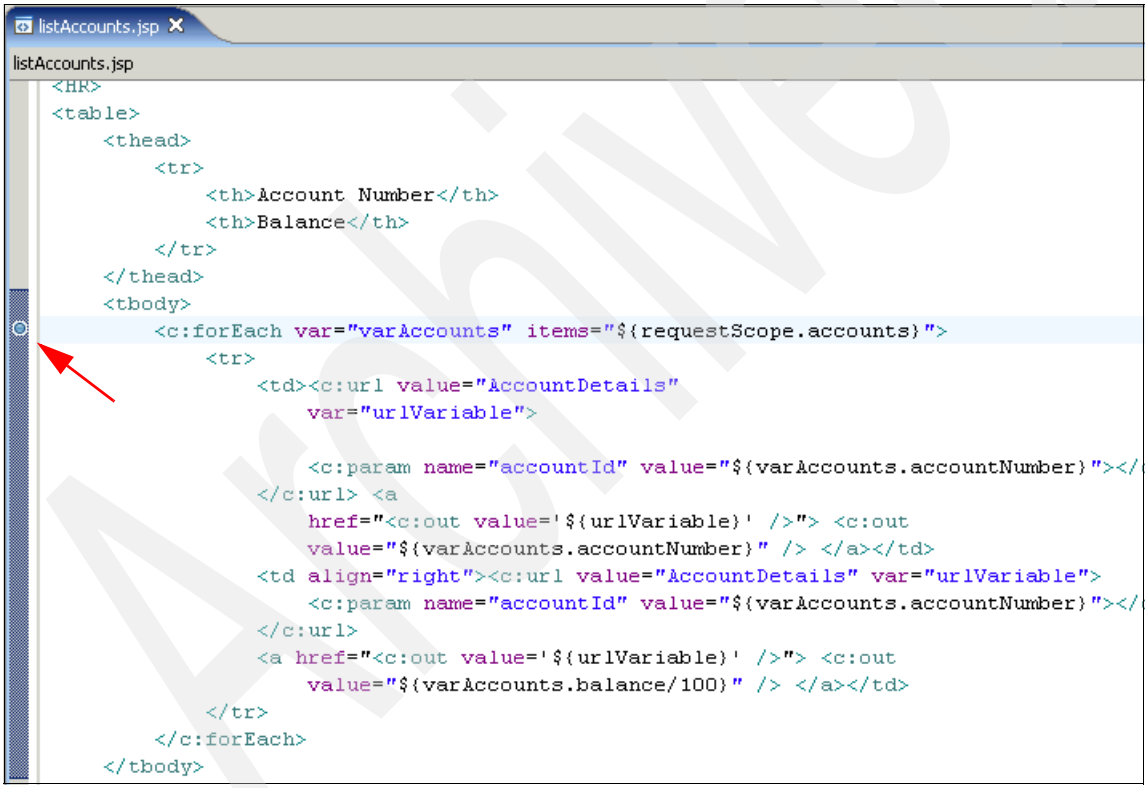


Figure 21-9 Set a breakpoint in a JSP

Archived

21.3.3 Start the application for debugging

Once you have set the breakpoint(s), the Web application can be started for debugging.

Note: The server used for testing (for example, WebSphere Application Server V6.0 (default test server)), must be either stopped or started in debug mode. Otherwise, an error message will be displayed.

1. Stop the WebSphere Application Server V6.0 test server.
2. From the Web perspective, expand **Dynamic Web Projects**.
3. Right-click **BankBasicWeb**, and select **Debug → Debug on Server**.

   Note: If the server was started it will prompt you to restart.

4. When the Define a New Server dialog appears, select **Choose and existing Server**, select **WebSphere Application Server V6.0**, and then click **Finish**.

   Tip: To debug a Java application, select **Run → Debug As → Java Application** to start the debugger. The startup is the primary difference between debugging a Web application and Java application.

21.3.4 Run the application in the debugger

After starting the application in debug mode, as described in 21.3.3, “Start the application for debugging” on page 1136, you should now see `index.html` displayed in the Web Browser view, as shown in Figure 21-10 on page 1137.

Archived

Figure 21-10 ITSO RedBank index.html page

1. Click **RedBank** in the horizontal navigation bar, as seen in Figure 21-10.
2. When prompted for the customer ID (SSN), enter `111-11-1111` (as seen in Figure 21-11 on page 1138), and then click **Submit**.

Archived

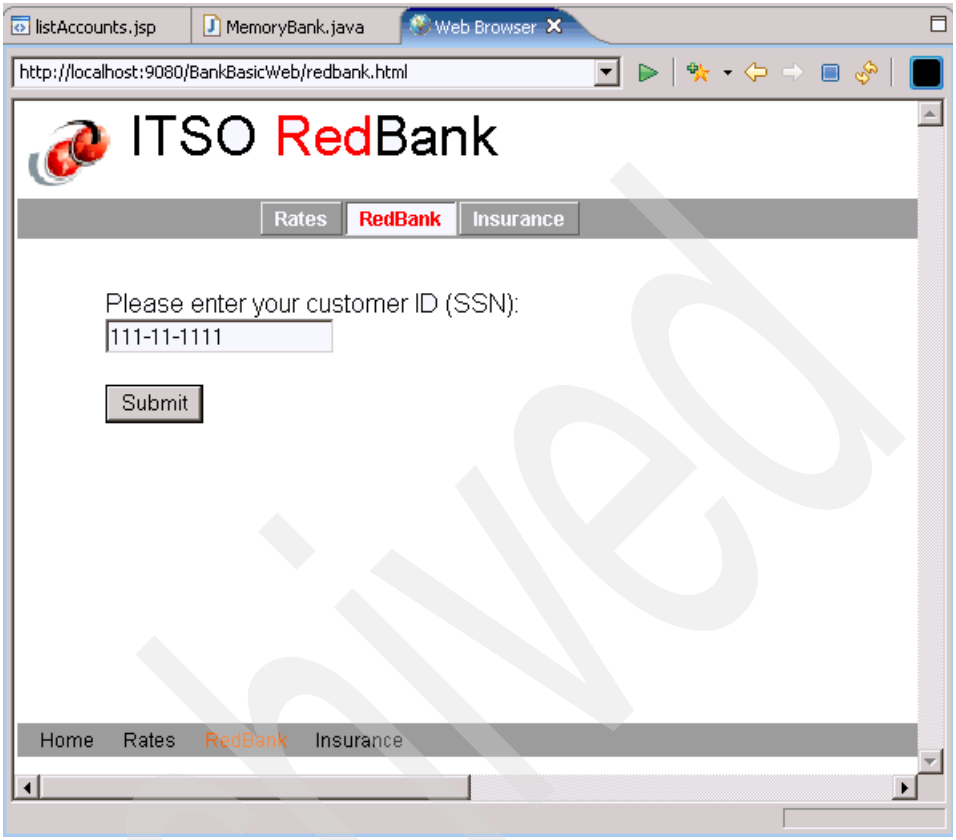


Figure 21-11 Enter customer ID (SSN)

3. When the Confirm Perspective Switch window appears, click **Yes** to switch to the Debug perspective.

   The sample will now be run in the Debug perspective.

4. Execution should stop at the breakpoint set in the listAccounts.jsp, since clicking Submit in the application will display the accounts. The thread is suspended in debug, but other threads might still be running (see Figure 21-12 on page 1139).

Archived

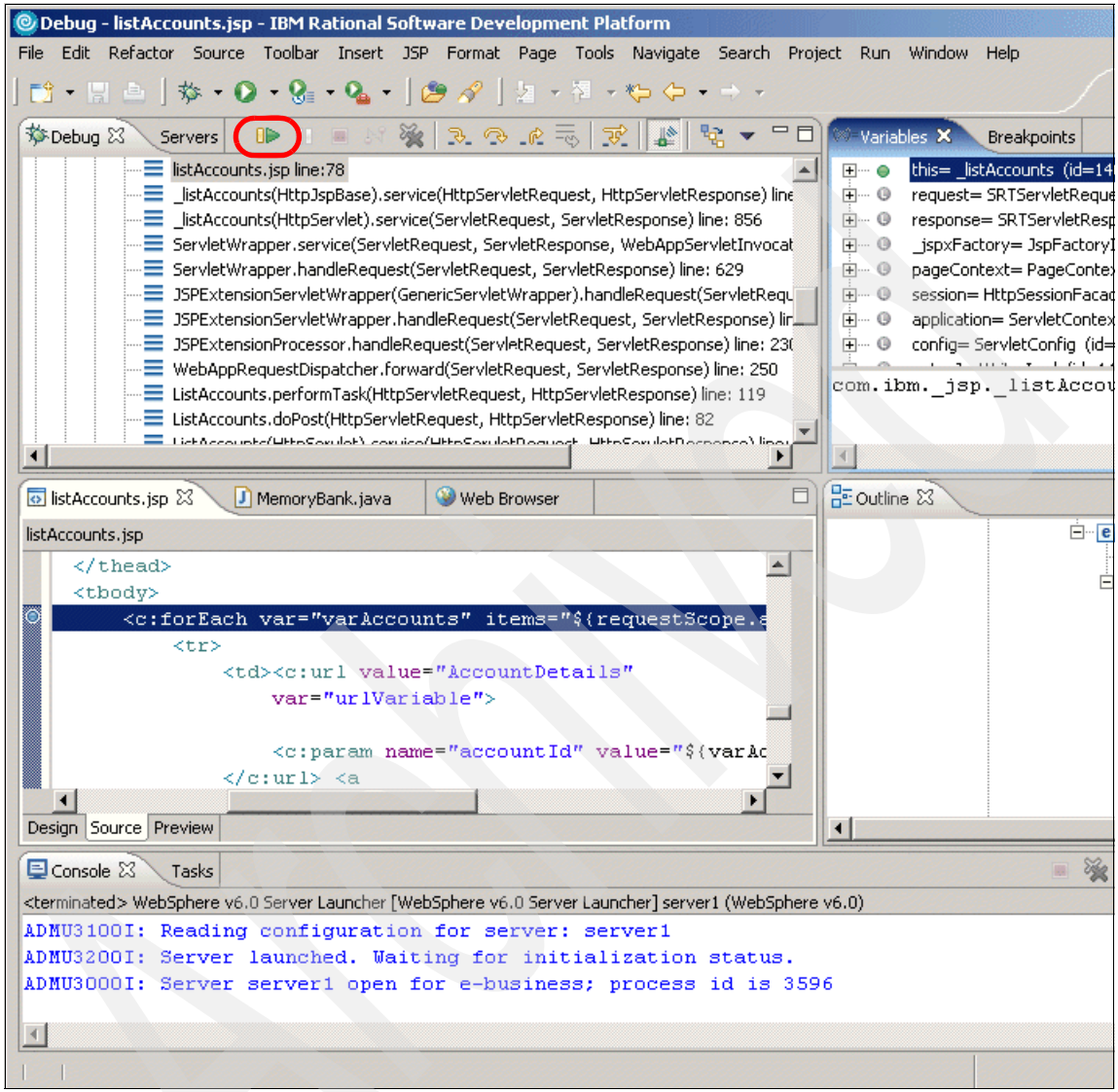


Figure 21-12 Breakpoint in listAccounts.jsp in the Debug perspective

5. Once a breakpoint is hit, you can proceed in a number of ways. For our example, we click the Resume icon highlighted in Figure 21-12.
6. Click the **Web Browser** session and resize the page as needed. Click the **001-999000888** account.

7. When the Account: 001-999000888 page appears, select **Withdraw**, enter 1000 (value of conditional breakpoint in MemoryBank.java), and then click **Submit**.

   The breakpoint in MemoryBank.java should be hit and displayed in the Debug perspective, similar to Figure 21-13.

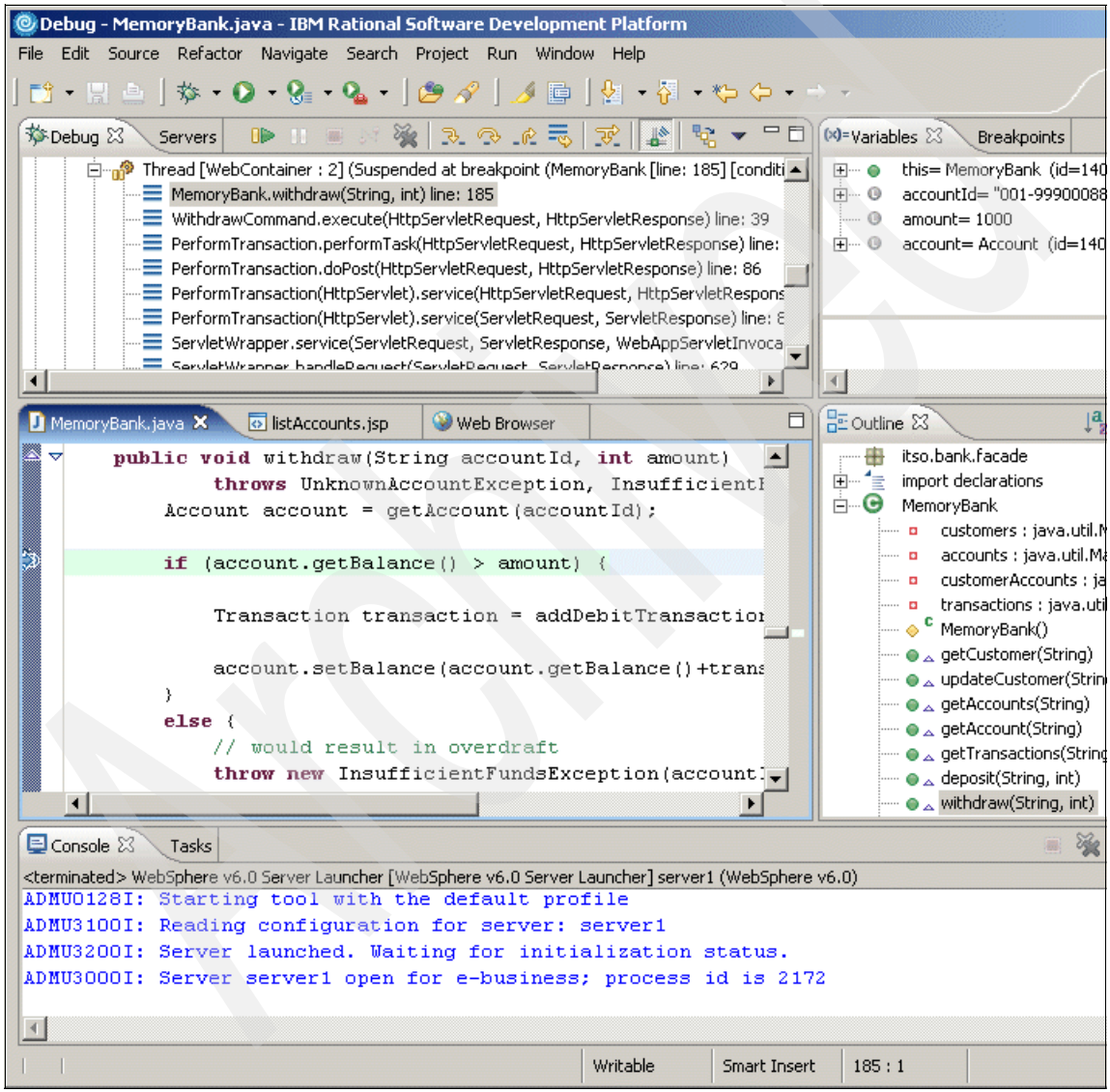


Figure 21-13 Breakpoint in MemoryBank.java in the Debug perspective

Next we discuss the different views of the Debug perspective.

21.3.5 Debug view with stack frames

When a breakpoint is reached, the debugger displays a list of stack frames before the breakpoint occurred. Each frame corresponds to a called method. The entire list is in reverse chronological order. Figure 21-14 shows the stack frame listing for the breakpoint in the MemoryBank.java withdraw method.

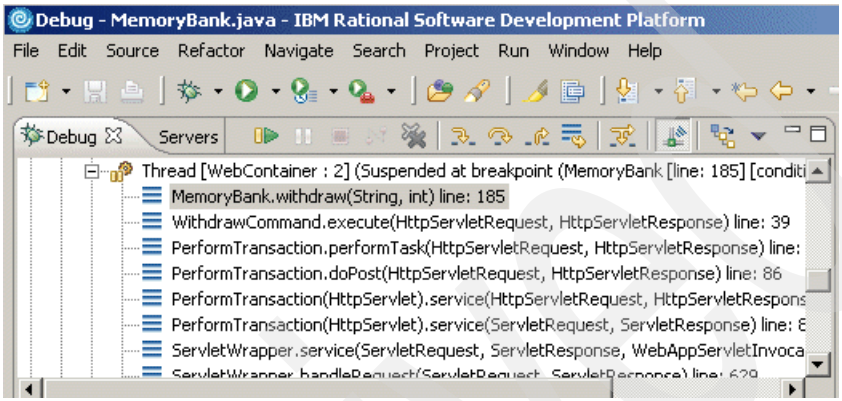


Figure 21-14 Stack frame listing in Debug view

When a thread suspends, the top stack frame is automatically selected. If you select another stack frame, all visible variables in that frame are shown in the Variables view.

21.3.6 Debug functions

From the Debug view, which should now be displayed in the top left pane, you can use the functions available from its icon bar to control the execution of the application. The following icons are available:

- Resume: Runs the application to the next breakpoint.
- Suspend: Suspends a running thread.
- Terminate: Terminates a process.
- Disconnect: Disconnects from the target when debugging remotely.
- Remove All Terminated Launches: Removes terminated executions.
- Step Into: Steps into the highlighted statement.
- Step Over: Steps over the highlighted statement.
- Step Return: Steps out of the current method.
- Drop to Frame: Drops to the Debug Frame view and highlights code.
- Step Filter: Enable/disable the filtering for the *step* functions.

- Step-By-Step Mode: Once the step-by-step debug feature is enabled in the Run/Debug preferences, this icon can be used to toggle the feature.
- Show Qualified Names: Toggle option to show the full package name.

In the upper right pane you can see the various debugging views that are available.

21.3.7 Breakpoints view

The Breakpoints view displays all the breakpoints set in the Workbench (see Figure 21-15).

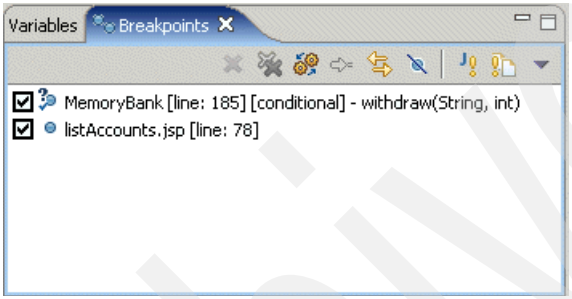


Figure 21-15 Debugging views

You can use the Breakpoints view to display and manipulate the breakpoints that are currently set. You can open the properties, remove the breakpoint, or open its source file.

21.3.8 Watch variables

The Variables view displays the current values of the variables in the selected stack frame. Follow these steps to see how you can track the state of a variable.

Click the Step Over icon to execute the current statement.

Click Step Over again and the year variable is added. The plus sign (+) next to a variable indicates that it is an object.

In our example, we set a conditional break point on amount==1000. Notice the year input variable value 1000 in Figure 21-16 on page 1143.

Archived

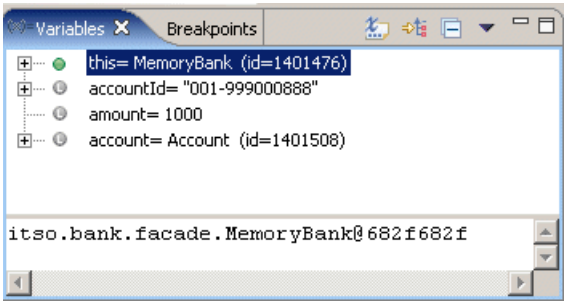


Figure 21-16 Displaying variables

If you want to test the code with some other value for any of these instance variables, you can change one of them by selecting **Change Variable Value** from its context menu. An entry field opens where you can change the value; for example, you can change the value of the year to 2002 and then click Resume .

21.3.9 Inspect variables

To view more details about a variable, select the variable (for example, **amount=1000**), right-click, and select **Inspect** from the context menu. The result opens in the Expressions view, as seen in Figure 21-17.

Both the Variables and the Expressions views can be split into two panes by selecting **Show Detail Pane** from the context menu.

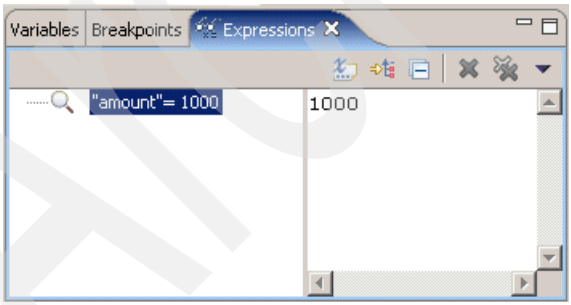


Figure 21-17 Inspecting a variable in Expressions view

21.3.10 Evaluate an expression

To evaluate an expression in the context of the currently suspended thread, use the Display view.

Archived

1. While in the debugger at the breakpoint, press F6 twice to step through code until you reach the following line:

   ```
   account.setBalance(account.getBalance()+transaction.getSignedAmount());
   ```

2. From the Workbench select **Windows** → **Show view** → **Display**.
3. Enter the expression `transaction.getTimestamp()`, then highlight the expression, right-click, and select **Display** from the context menu.

   Note: When entering the expression, we used the code assist (Ctrl+spacebar) to simplify entry and enter the proper method.

   Each expression is executed, and the result is displayed (Figure 21-18).

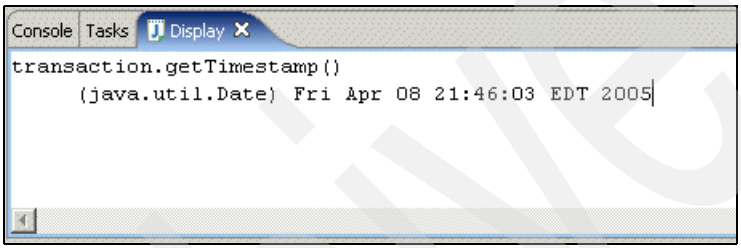


Figure 21-18 Expression and evaluated result in display view

4. The results of the Java expression can also be inspected by selecting **Inspect** from the context menu, as seen in Figure 21-19.

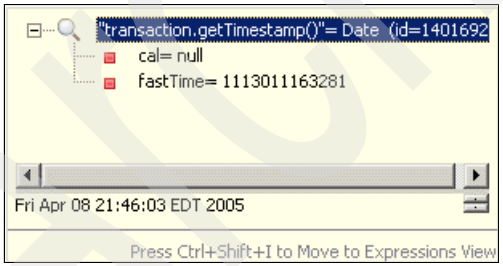


Figure 21-19 Display Inspect expression

Note: To move the results to the Expression view, press Ctrl+Shift+I.

5. You can also highlight any expression in the source code, right-click, and select **Display** or **Inspect** from the context menu. The result is shown either in the Display or the Expressions view.

Archive

This is a useful way to evaluate Java expressions during debugging, without having to make changes in your code and recompile.

6. Select **Remove** from the context menu to remove expressions or variables from the Expressions views. In the Display view just select the text and delete it.

21.3.11 Debug a JSP

Step through the code or click the Resume icon ( ) to progress to the breakpoint in the JSP (see Figure 21-12 on page 1139).

Note: If you have two JSPs in different Web applications, the wrong JSP source may be displayed. Open the correct JSP to see its source code.

Watch the JSP variables in the Variables view. The same functions as for servlets are available for JSP debugging. A JSP is compiled into a servlet. The difference is that the debugger shows the JSP source code and not the generated Java code.

When you step through JSP code, the debugger only stops at Java code; HTML statements are skipped.

Resume execution to see the next Web page, then close the Debug perspective and stop the server.

21.4 Debug a Web application on a remote server

It is possible to connect to and debug a Java Web application that has been launched in debug mode on a remote application server, and the application server has been configured to accept remote connections. Debugging a remote program is similar to debugging a local application, except that the program has already been launched and could be running on a remote host.

This example scenario will include a node where IBM Rational Application Developer V6.0 is installed (Developer node), and a separate node where IBM WebSphere Application Server V6.0 is installed (Application Server node).

21.4.1 Export the BankBasicWeb project to a WAR file

This section describes how to export the BankBasicWeb project to a WAR file so that it can be deployed on a remote WebSphere Application Server.

1. Open the Web perspective in Rational Application Developer.

2. Expand **Dynamic Web Projects**.
3. Select **BankBasicWeb**, right-click, and select **Export** → **WAR file**.
4. When the WAR Export window appears, enter the following and then click **Finish**:
   - Web project: `BankBasicWeb`
   - Destination: `c:\temp\BankBasicWeb.war`
5. Verify that the c:\temp\BankBasicWeb.war exists.

21.4.2 Deploy the BankBasicWeb.war

This section describes how to deploy the BankBasicWeb.war to a remote system where IBM WebSphere Application Server V6.0 has been installed.

1. Copy the BankBasicWeb.war from the node where Rational Application Developer is installed to the node where WebSphere Application Server is installed (for example, c:\temp).
2. Ensure that the WebSphere Application Server - server1 application server is started.
3. Start the WebSphere Application Server Administrative Console by entering the following in a Web browser and logging on:

   `http://<hostname>:9060/ibm/console`
4. Select **Applications** → **Install New Application**.
5. Enter the path to the war file and the context root, and then click **Next**. For example, we entered:
   - Specify path: `c:\temp\BankBasicWeb.war`
   - Context root: `BankBasicWeb`
6. We accepted the default options and clicked **Next**.
7. When you see an Application Security Warning, click **Continue**.
8. When the Step 1: Select installation options page appears, accept the default and click **Next**.
9. When the Step 2: Map modules to servers page appears, check the **BankBasicWeb** module, and then click **Next**.
10. When the Step 3: Map virtual hosts for Web modules page appears, check the **BankBasicWeb** Web module, select **default_host** from the Virtual host drop-down list, and then click **Next**.
11. When the Step 4: Summary page appears, accept the defaults and then click **Finish**.

Archived

You should see the following message if successfully deployed:

```
Application BankBasicWeb _war installed successfully.
```

12. Click **Save to Master**. Click **Save**.
13. Check **BankBasicWeb_war**, and click **Start**.
14. Click **Logout**.
15. Verify the application is working properly by entering the following URL:

```
http://<hostname>:9080/BankBasicWeb/
```

21.4.3 Install the IBM Rational Agent Controller

The IBM Rational Agent Controller provides several plug-ins for debugging, logging, profiling, and testing.

If the remote system is running WebSphere Application Server V6.0 and you only intend to use remote debug, the IBM Rational Agent Controller is not required, since the required functionality is built-in to WebSphere Application Server V6.0.

If the remote system is running WebSphere Application Server V5.1 or V5.0 and you intend to use remote debug, the IBM Rational Agent Controller is required. You will be prompted to provided the installation path for WebSphere Application Server V5.1 or V5.0 during the IBM Rational Agent Controller installation.

If you intend to use the profiling and testing features of Rational Application Developer, the IBM Rational Agent Controller is required for WebSphere Application Server V6.0, V5.1 and 5.0.

For the redbook scenario, although we are planning on debugging on a WebSphere Application Server V6.0 server, we also want to perform profiling and testing, so we installed all plug-ins for the IBM Rational Agent Controller.

For details on installing the IBM Agent Controller refer to “IBM Rational Agent Controller V6 installation” on page 1382.

21.4.4 Configure debug on remote WebSphere Application Server

The following steps explain how to configure WebSphere Application Server V6.0 to start in debug mode:

1. Start the application server.

```
<was_home>\bin\startServer.bat server1
```

2. Start the WebSphere Administrative Console by entering the following in a Web browser and then logging in:

   `http://<hostname>:9060/ibm/console`

3. In the left-hand frame, select **Servers** → **Application Servers**.
4. In the Application Servers page, click **server1**.
5. On the Configuration tab, select **Debugging Service** in the Additional Properties section to open the Debugging Service configuration page.
6. In the General Properties section of the Configuration tab, check **Enable service at startup**. This enables the debugging service when the server starts.

   Note: The value of the JVM debug port (default 7777) is needed when connecting to the application server with the debugger.

7. Click **OK** to make the changes to your local configuration.
8. Click **Save** to apply the configuration changes.
9. Click **Logout**.
10. You must restart the application server before the changes that you have made take effect.

21.4.5 Attach to the remote server in Rational Application Developer

To attach to the remote WebSphere Application Server V6.0 from within Rational Application Developer V6.0, do the following:

1. From the Workbench, open the J2EE perspective.
2. In the Servers view, right-click **WebSphere Application Server v6.0** and select **Stop**.
3. Create a new remote WebSphere Application Server V6 server.
   a. In the Servers view, right-click **New** → **Server**.
   b. When the Define a New Server window appears, we entered the following, as seen in Figure 21-20, and then clicked **Next**:
      - Host name: `was6win1.itso.ral.ibm.com`
      - Server type: Select **WebSphere V6.0 Server**.

Archived

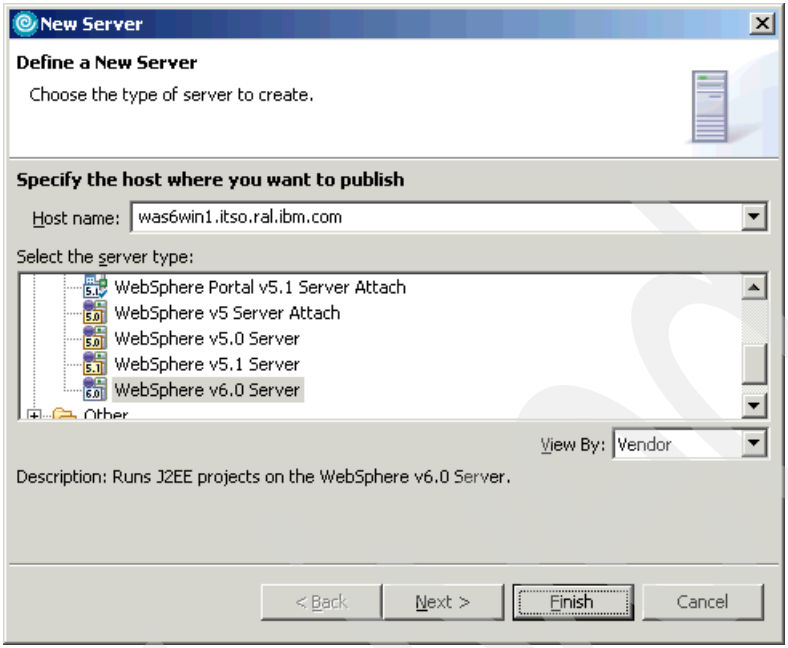


Figure 21-20 Define new remote server

c. When the WebSphere Server Settings page appears, we entered the following, as seen in Figure 21-21 on page 1150:

- Server admin port number (SOAP connector port): 8880

> **Note:** The SOAP connector port is defined in the WebSphere Profile.

- Server name: `server1`
- Server type: Select **Base or Express server**.

> **Note:** We recommend that you test the connection to the server by clicking the **Detect** button (see Figure 21-21 on page 1150).

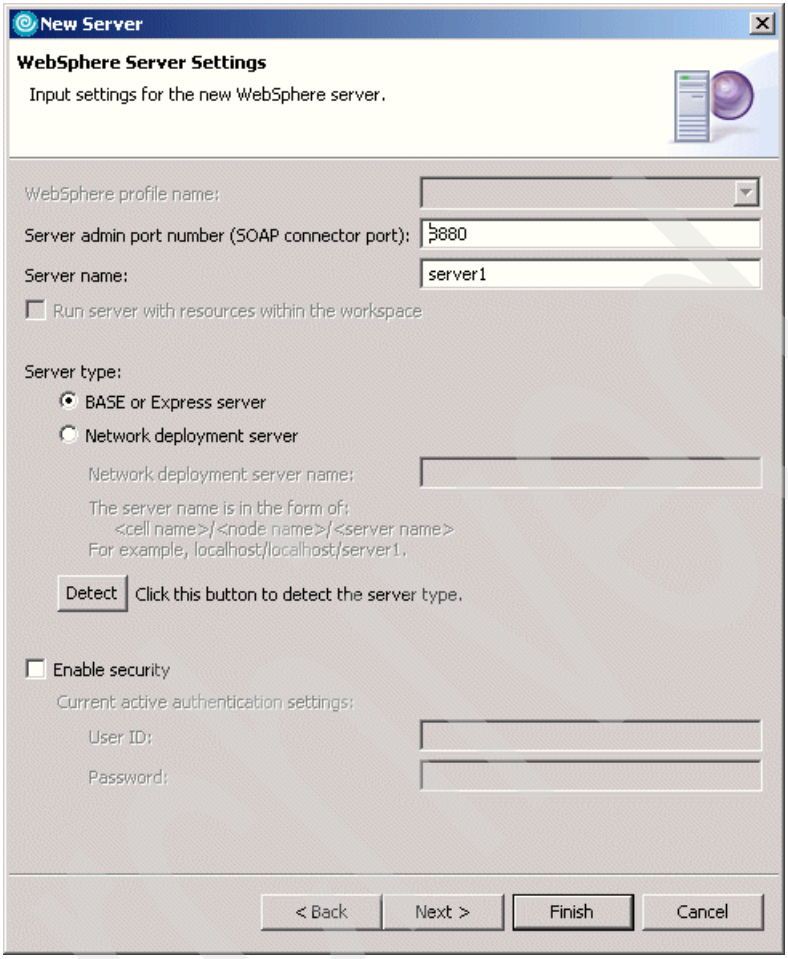


Figure 21-21 WebSphere Server Settings

d. Click **Finish**.

When complete, the configuration should look like Figure 21-22 on page 1151. Notice the status is `Debugging`.

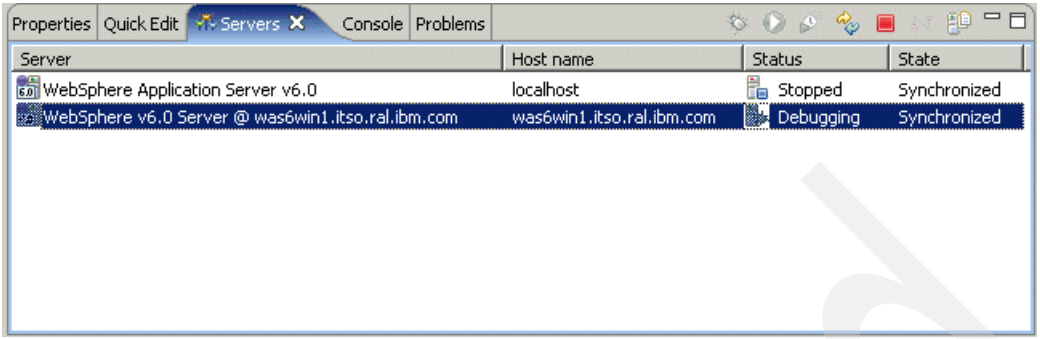


Figure 21-22 Remote server configuration

21.4.6 Debug the application on the remote server

Now that the environment is configured, we demonstrate how to debug an application running on the remote server within Rational Application Developer.

Debug sample application on remote server

The behavior of debugging for remote servers is slightly different than locally. For example, when debugging locally you can select a project and Debug on Server. This will cause the application to be invoked in the built-in Web browser and start the server. Also, the user will automatically be prompted if they want to switch to the Debug perspective.

When debugging on a remote server, some of these steps are a bit more manual in the way that they are initiated.

1. Open the Debug perspective.
2. Add the Web browser to the Debug perspective.
   a. From the tool bar of the Debug perspective, right-click **Customize Perspective**.
   b. Click the **Commands** tab.
   c. Scroll down and check **Web Browser**.
   d. Click **OK**.
3. Click the Web browser icon ( ) found on the tool bar.
4. Enter the URL for the remote Web application in the Web browser. For example, we entered the following:

   `http://was6win1.itso.ral.ibm.com:9080/BankBasicWeb`

5. Run the sample.

Archived

Once the application is started, the steps to run the application are similar to those in 21.3.4, “Run the application in the debugger” on page 1136.

Note: When the page of the breakpoint is loaded, you will see an option to locate the source. Select **Folder**, then select the project **WebContent** directory (for example, in this case for a JSP).

6. Provided that a breakpoint is set, the code will be displayed at the breakpoint. You can also step through the code.
7. Define source location.
   a. You will see the message `Source not found`. Click the **Edit Source Lookup Path** button.
   b. When the Edit Source Lookup Path dialog appears, click **Add**.
   c. When the Add Source dialog appears, select **Workspace** and click **OK**.
   d. Click **OK**.

Attention: From within Rational Application Developer, you can stop the remote server by right-clicking the remote server in the Servers view, and selecting **Stop**. However, you cannot start the remote server from within Rational Application Developer. You must start the remote server from the remote WebSphere Application Server node.

Archived

Part 4

Deploy and profile applications

© Copyright IBM Corp. 2005. All rights reserved.

Archived

22

Build applications with Ant

Traditionally, application builds are performed by using UNIX/Linux shell scripts or Windows batch files in combination with tools such as *make*. While these approaches are still valid, new challenges exist for when developing Java applications, especially in a heterogeneous environment. Traditional tools are limited in that they are closely coupled to a particular operating system. With Ant you can overcome these limitations and perform the build process in a standardized fashion regardless of the platform.

This chapter provides an introduction to the concepts and features of Ant within IBM Rational Application Developer V6.0. The focus of the chapter is to demonstrate how to use the Ant tooling included in Rational Application Developer to build projects (applications).

This chapter is organized into the following sections:

- Introduction to Ant
- New features
- Build a simple Java application
- Build a J2EE application
- Run Ant outside of Application Developer

© Copyright IBM Corp. 2005. All rights reserved.

22.1 Introduction to Ant

Ant is a Java-based, platform-independent, open source build tool. It was formerly a sub-project in the Apache Jakarta project, but in November 2002 it was migrated to an Apache top-level project. Ant's function is similar to the *make* tool. Since it is Java-based and does not make use of any operating system-specific functions, it is platform independent, thus allowing you to build your projects using the same build script on any Java-enabled platform.

The Ant build operations are controlled by the contents of the XML-based script file. This file not only defines what operations to perform, but also defines the order in which they should be performed, and any dependencies between them.

Ant comes with a large number of built-in tasks sufficient to perform many common build operations. However, if the tasks included are not sufficient, you also have the ability to extend Ant's functionality by using Java to develop your own specialized tasks. These tasks can then be plugged into Ant.

Not only can Ant be used to *build* your applications, but it can also be used for many other operations such as retrieving source files from a version control system, storing the result back in the version control system, transferring the build output to other machines, deploying the applications, generating Javadoc, and sending messages when a build is finished.

22.1.1 Ant build files

Ant uses XML *build files* to define what operations must be performed to build a project. We have listed the main components of a build file:

- project: A build file contains build information for a single project. It may contain one or more *targets*.
- target: A target describes the *tasks* that must be performed to satisfy a goal. For example, compiling source code into class files may be one target, and packaging the class files into a JAR file may be another target.

  Targets may depend upon other targets. For example, the class files must be up-to-date before you can create the JAR file. Ant can resolve these dependencies.
- task: A task is a single step that must be performed to satisfy a target. Tasks are implemented as Java classes that are invoked by Ant, passing parameters defined as attributes in the XML. Ant provides a set of standard tasks (core tasks), a set of optional tasks, and an API, which allows you to write your own tasks.

- property: A property has a name and a value pair. Properties are essentially variables that can be passed to tasks through task attributes. Property values can be set inside a build file, or obtained externally from a properties file or from the command line. A property is referenced by enclosing the property name inside ${}, for example ${basedir}.
- path: A path is a set of directories or files. Paths can be defined once and referred to multiple times, easing the development and maintenance of build files. For example, a Java compilation task may use a path reference to determine the classpath to use.

22.1.2 Ant tasks

A comprehensive set of built-in tasks is supplied with the Ant distribution. The tasks that we use in our example are as follows:

- delete: Deletes files and directories
- echo: Outputs messages
- jar: Creates Java archive files
- javac: Compiles Java source
- mkdir: Creates directories
- tstamp: Sets properties containing date and time information

To find out more about Ant, visit the Ant Web site at:

`http://ant.apache.org/`

This chapter provides a basic outline of the features and capabilities of Ant. For complete information you should consult the Ant documentation included in the Ant distribution or available on the Internet at:

`http://ant.apache.org/manual/index.html`

Note: IBM Rational Application Developer V6.0 includes Ant V1.4.1.

22.2 New features

IBM Rational Application Developer V6.0 includes the following new features to aid in the development and use of Ant scripts:

- The ability to run the build process in a background task like other tasks within IBM Rational Application Developer V6.0.
- The Ant editor now also offers code assist with the ability to insert snippets.
- The Ant editor now has a format function that will allow you to format your Ant files base on your preferences.

Archived

- A problems view is now available in the Ant editor to highlight syntax errors in your Ant files.

In this section we highlight the following Ant-related features in Rational Application Developer:

- Code Assist
- Code snippets
- Format an Ant script
- Define format of an Ant script
- Problem view

Note: The new features outlined in this section can be explored hands on by importing the BankAnt.zip Project Interchange file, as described in 22.3.1, “Prepare for the sample” on page 1168.

22.2.1 Code Assist

To access the new features such as Code Assist in the Ant editor, do the following:

1. Open the Java perspective.
2. Expand the **BankAnt** project.
3. Double-click **build.xml** to open the file in an editor.
4. Place the cursor in the file and enter <prop, and then press Ctrl+Spacebar.
5. The Code Assist dialog will be presented, as shown in Figure 22-1 on page 1159. You can then use the up and down arrow keys to select the tag that you want.

Archived

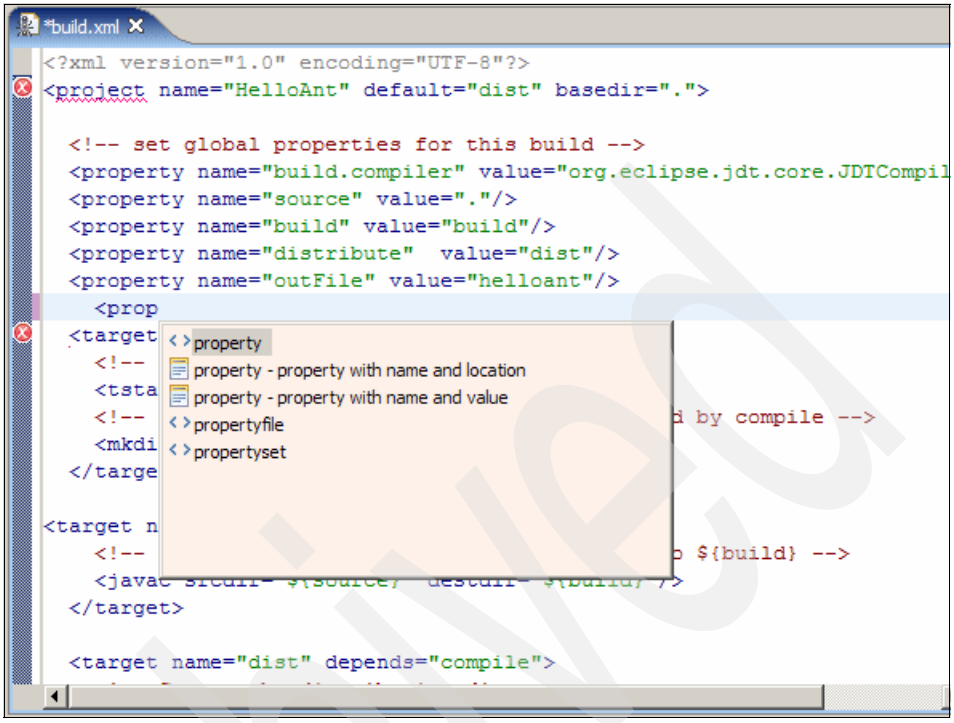


Figure 22-1 Code Assist in Ant editor

22.2.2 Code snippets

IBM Rational Application Developer V6 provides the ability to create code snippets that contain commonly used code to be inserted into files rather than typing the code in every time.

To create code snippets, do the following:

1. Open the Snippets view by selecting **Window** → **Show View** → **Other**, and the Show View dialog will be displayed.
2. Expand the **Basic** folder, select the **Snippets** view (as shown in Figure 22-2 on page 1160), and then click **OK**.

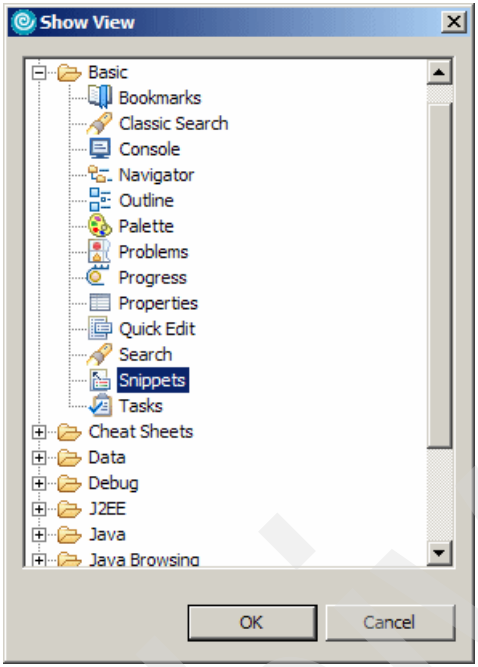


Figure 22-2 Show View dialog

3. Right-click the Snippets view and select **Customize**, as seen in Figure 22-3.

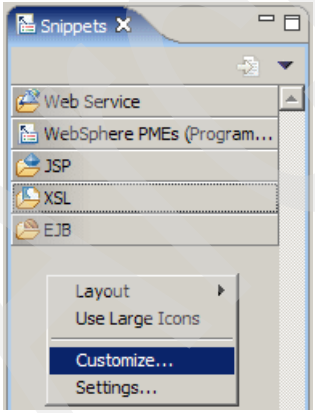


Figure 22-3 Customizing snippets

4. When the Customize Palette dialog appears, select **New** → **New Category**.

Archived

5. When the New Customize Palette dialog appears (as seen in Figure 22-4 on page 1161), do the following:
   - Name: `Ant`
   - Description: `Ant Snippets`
   - Select **Custom**.

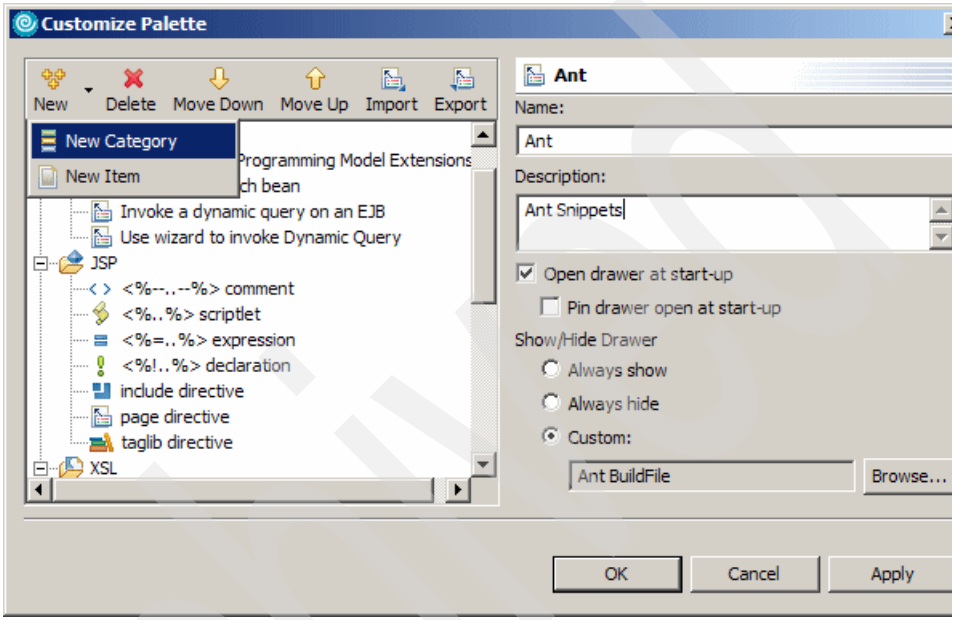


Figure 22-4 New Customize Palette dialog

6. Click **Browse** next to Custom, check **Ant Buildfiles** (as seen in Figure 22-5 on page 1162), and click **OK** to return to the Customize Palette dialog.

Archive

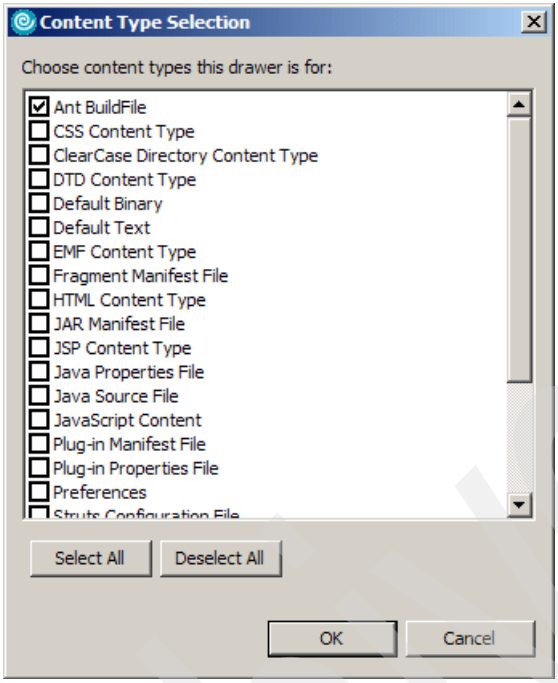


Figure 22-5 Content Type Selection view

7. From the Customize Palette dialog, select **New** → **New Item**.
8. When the Unamed Template dialog appears, enter the following:
   - Name: `Comment Tag`
   - Click **New** in the variables section.
   - Variable Name: `comment`
   - Template Pattern: `<!-- ${comment} -->`
9. Click **OK** on the Customize Palette dialog.

Use the code snippet

Now that you have created a code snippet you can use it in any Ant build file. To use a code snippet, do the following:

1. Double-click the **build.xml** file to open it in the editor.
2. Place the cursor under the <project tag, double-click the **Comment Tag** in the Snippets view, and the Insert Template dialog will be displayed (as shown in Figure 22-6 on page 1163).

Archived

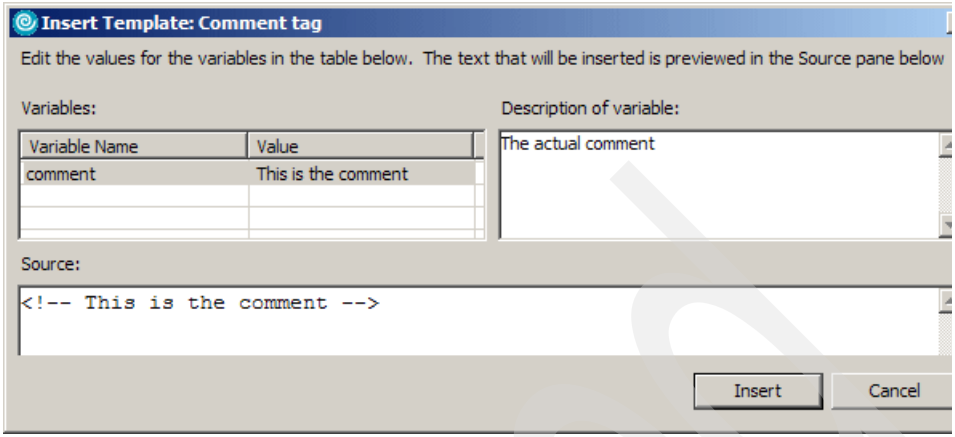


Figure 22-6 Insert Template dialog

3. In the variables table, enter `Surf's up!` in the comment variable.
4. Click the **Insert** button.

22.2.3 Format an Ant script

Rational Application Developer now offers you the ability to format Ant scripts in the Ant editor. To format the Ant script, do the following:

1. Double-click **build.xml** to open it in the Ant editor.
2. Right-click the editor and select **Format** from the context menu, as shown in Figure 22-7 on page 1164.

   Alternatively, you can press Ctrl+Shift+F.

Archived

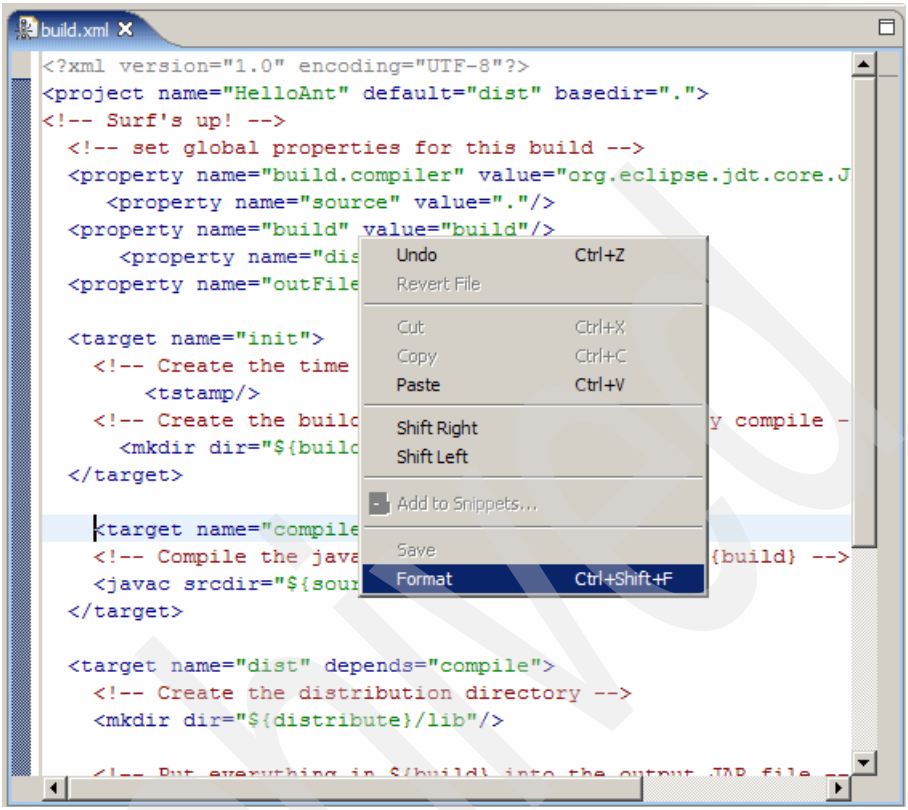


Figure 22-7 Formatting the Ant file

22.2.4 Define format of an Ant script

To define the format of an Ant script, do the following:

1. Select **Window** → **Preferences**.
2. When the Preferences dialog appears, select **Ant**.
3. When the Ant preferences dialog appears, as seen in Figure 22-8 on page 1165, you can specify the console colors.

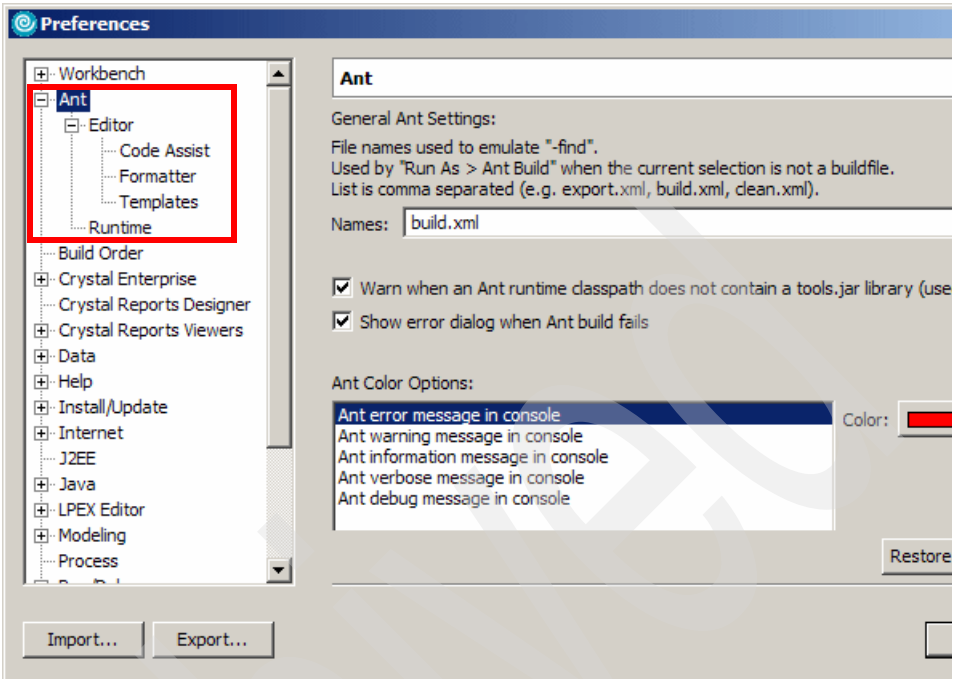


Figure 22-8 Ant preferences

4. Expand the **Ant** folder and select **Editor**.
   a. On the Appearance tab you can change the layout preferences of your Ant file.
   b. On the Syntax tab you can change the syntax highlighting preferences with a preview of the results, and on the Problem tab you can define how certain problems should be handled.
5. Expand **Editor**.
   a. In the Code Assist window you can define the code assist preferences.
   b. In the Formatter window you can define the preferences for the formatting tool for the Ant files.
   c. In the Templates window you can create, edit, and delete templates for Ant files.
6. Select **Runtime**.

   In this window you can define your preferences such as classpath, tasks, types, and properties.

22.2.5 Problem view

Rational Application Developer now offers you the problems view for the Ant file. The editor will present an error in the view by placing a *red X* on the left of the line with the problem as well as a line marker in the file on the right of the window, as shown in Figure 22-9. The problem view will list the problems as seen in Figure 22-10.

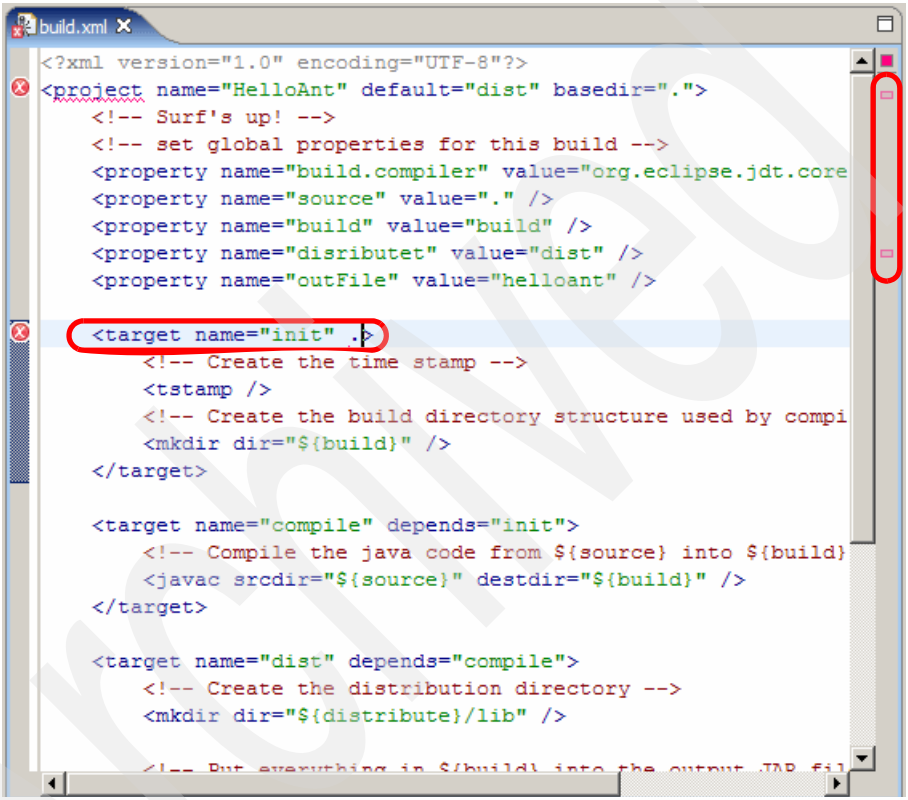


Figure 22-9 Problems in the Ant editor

Archive

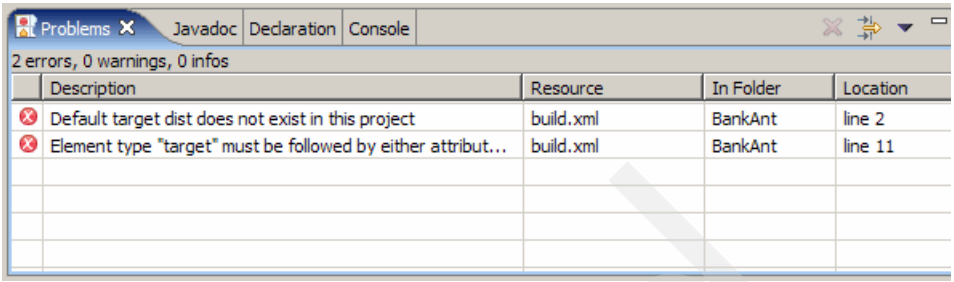


Figure 22-10 Problems view displaying Ant problems

22.3 Build a simple Java application

We created a simple build file that compiles the Java source for our HelloAnt application and generates a JAR file with the result. The build file is called build.xml, which is the default name assumed by Ant if no build file name is supplied.

The example simple build file has the following targets:

- init: Performs build initialization tasks. All other targets depend upon this target.
- compile: Compiles Java source into class files.
- dist: Creates the deliverable JAR for the module, and depends upon the compile target.
- clean: Removes all generated files. Used to force a full build.

Each Ant build file may have a default target. This target is executed if Ant is invoked on a build file and no target is supplied as a parameter. In our example, the default target is dist. The dependencies between the targets are illustrated in Figure 22-11.

Archive

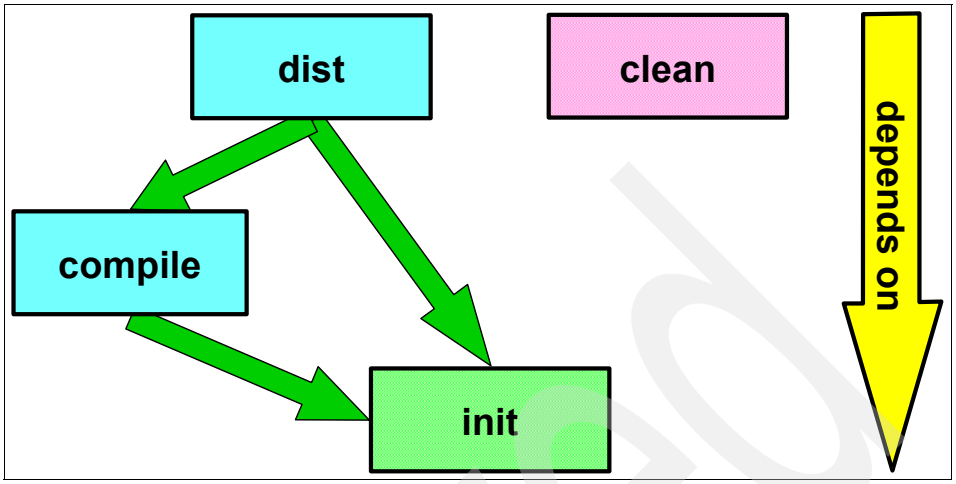


Figure 22-11 Ant example dependencies

22.3.1 Prepare for the sample

To demonstrate the basic concepts of Ant, we wrote a very simple Java application named HelloAnt, which prints a message to stdout.

We created a new Java project for this that we called BankAnt. In this project we created a Java package called itso.ant.hello and a class called HelloAnt. Since these steps are basic Rational Application Developer tasks and the application does nothing but a simple System.out.println("G'day from Australia."), we do not show them here. In addition, the source code for the chapter is included in the BankAnt project.

To import the BankAnt.zip Project Interchange file, do the following:

1. Start Rational Application Developer.
2. From the Workbench, select **File** → **Import**.
3. From the Import dialog, select **Project Interchange** and then click **Next**.
4. When prompted for the Project Interchange path and file name, and target workspace location, we entered the following:
   - From zip file: `c:\6449code\ant\BankAnt.zip`
   - Project location root: `c:\workspace`

     Enter the location of the desired workspace (for example, our workspace is found in c:\workspace).
5. After entering the zip file, check **BankAnt** and then click **Finish**.

22.3.2 Create a build file

The BankAnt project already includes the build.xml file we will create in this section.

To create the simple build file, do the following:

1. Select the **BankAnt** project in the Package Explorer view.
2. Right-click and select **New** → **File** from its context menu.
3. When the New File dialog appears, enter `build.xml` as the filename, as seen in Figure 22-12 on page 1169, and click **Finish**.

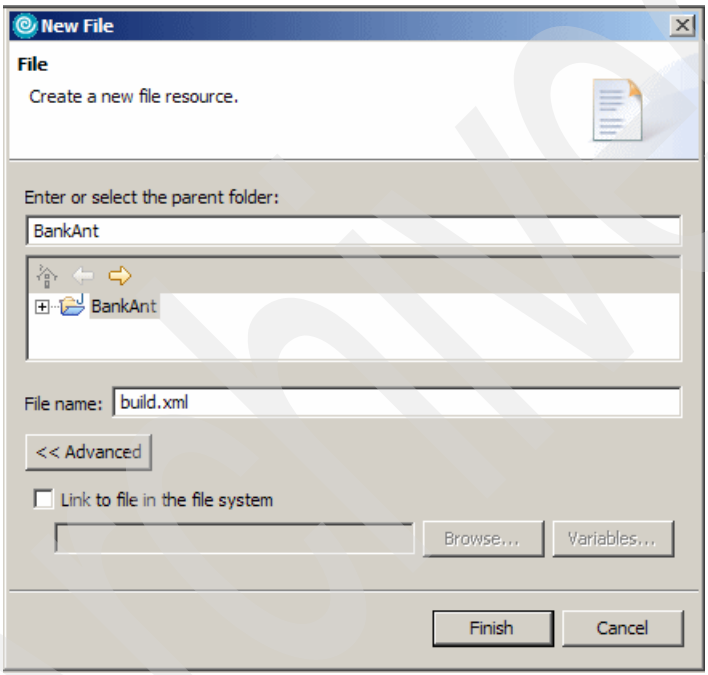


Figure 22-12 Create a build.xml file

> **Note:** IBM Rational Application Developer V6 now has the ability to link to external files on the file system. The Advance button on the New File dialog allows you to specify the location on the file system that the new file is linked to.

4. Cut and paste the text in Example 22-1 into the file.

Example 22-1 Example build.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<project name="HelloAnt" default="dist" basedir=".">

  <!-- set global properties for this build -->
  <property name="build.compiler" value="org.eclipse.jdt.core.JDTCompilerAdapter"/>
  <property name="source" value="."/>
  <property name="build" value="c:\temp\build"/>
  <property name="distribute"  value="c:\temp\BankAnt”/>
  <property name="outFile" value="helloant"/>

  <target name="init">
    <!-- Create the time stamp -->
    <tstamp/>
    <!-- Create the build directory structure used by compile -->
    <mkdir dir="${build}"/>
  </target>

<target name="compile" depends="init">
    <!-- Compile the java code from ${source} into ${build} -->
    <javac srcdir="${source}" destdir="${build}"/>
  </target>

  <target name="dist" depends="compile">
    <!-- Create the distribution directory -->
    <mkdir dir="${distribute}/lib"/>

    <!-- Put everything in ${build} into the output JAR file -->
    <!-- Add a timestamp to the output filename as well -->
    <jar jarfile="${distribute}/lib/${outFile}-${DSTAMP}.jar" basedir="${build}">
      <manifest>
         <attribute name="Main-Class" value="itso.ant.hello.HelloAnt"/>
      </manifest>
    </jar>
  </target>

  <target name="clean">
    <!-- Delete the ${build} and ${distribute} directory trees -->
    <delete dir="${build}"/>
    <delete dir="${distribute}"/>
  </target>

</project>
```

We will now walk you through the various sections of this file, and provide an explanation for each of them.

Archived

22.3.3 Project definition

The <project tag in the build.xml file defines the project name and the default target. The project name is an arbitrary name; it is not related to any project name in your Application Developer workspace.

The project tag also sets the working directory for the Ant script. All references to directories throughout the script file are based on this directory. A dot (.) means to use the current directory, which, in Application Developer, is the directory where the build.xml file resides.

22.3.4 Global properties

Properties that will be referenced throughout the whole script file can be placed at the beginning of the Ant script. Here we define the property build.compiler that tells the **javac** command what compiler to use. We will tell it to use the Eclipse compiler.

We also define the names for the source directory, the build directory, and the distribute directory. The source directory is where the Java source files reside. The build directory is where the class files end up, and the distribute directory is where the resulting JAR file is placed:

- We define the source property as ".", which means it is the same directory as the base directory specified in the project definition above.
- The build and distribute directories will be created as c:\temp\build and c:\temp\BankAnt directories.

Properties can be set as shown below, but Ant can also read properties from standard Java properties files or use parameters passed as arguments on the command line:

```
<!-- set global properties for this build -->
<property name="build.compiler"
value="org.eclipse.jdt.core.JDTCompilerAdapter"/>
<property name="source" value="."/>
<property name="build" value="c:\temp\build"/>
<property name="distribute"  value="c:\temp\BankAnt"/>
<property name="outFile" value="helloant"/>
```

22.3.5 Build targets

The build file contains four build targets:

- init
- compile
- dist

- clean

Initialization target (init)

The first target we describe is the init target. All other targets (except clean) in the build file depend upon this target. In the init target we execute the tstamp task to set up properties that include timestamp information. These properties are then available throughout the whole build. We also create a build directory defined by the build property.

```
<target name="init">
    <!-- Create the time stamp -->
    <tstamp/>
    <!-- Create the build directory structure used by compile -->
    <mkdir dir="${build}"/>
</target>
```

Compilation target (compile)

The compile target compiles the Java source files in the source directory and places the resulting class files in the build directory.

```
<target name="compile" depends="init">
    <!-- Compile the java code from ${source} into ${build} -->
    <javac srcdir="${source}" destdir="${build}"/>
</target>
```

With these parameters, if the compiled code in the build directory is up-to-date (each class file has a timestamp later than the corresponding Java file in the source directory), the source will not be recompiled.

Distribution target (dist)

The distribution target creates a JAR file that contains the compiled class files from the build directory and places it in the lib directory under the dist directory. Because the distribution target depends on the compile target, the compile target must have executed successfully before the distribution target is run.

```
<target name="dist" depends="compile">
    <!-- Create the distribution directory -->
    <mkdir dir="${distribute}/lib"/>

    <!-- Put everything in ${build} into the output JAR file -->
    <!-- We add a time stamp to the filename as well -->
   <jar jarfile="${distribute}/lib/${outFile}-${DSTAMP}.jar"
basedir="${build}">
      <manifest>
         <attribute name="Main-Class" value="itso.ant.hello.HelloAnt"/>
      </manifest>
    </jar>
```

```
</target>
```

Cleanup target (clean)

The last of our standard targets is the cleanup target. This target removes the build and distribute directories, which means that a full recompile is always performed if this target has been executed.

```
<target name="clean">
    <!-- Delete the ${build} and ${distribute} directory trees -->
    <delete dir="${build}"/>
    <delete dir="${distribute}"/>
</target>
```

Note that our build.xml file does not call for this target to be executed. It has to be specified when running Ant.

22.3.6 Run Ant

Ant is a built-in function to Rational Application Developer. You can launch it from the context menu of any XML file, although it will run successfully only on valid Ant XML build script files. When launching an Ant script, you are given the option to select which targets to run and whether you want to view the output in a special Log Console window.

To run our build script:

1. Open the Java perspective.
2. Expand the **BankAnt** project.
3. Right-click **build.xml** in either the Package Explorer or Outline view.
4. Select **Run** → **3.Run Ant...**, as seen in Figure 22-13.

Archived

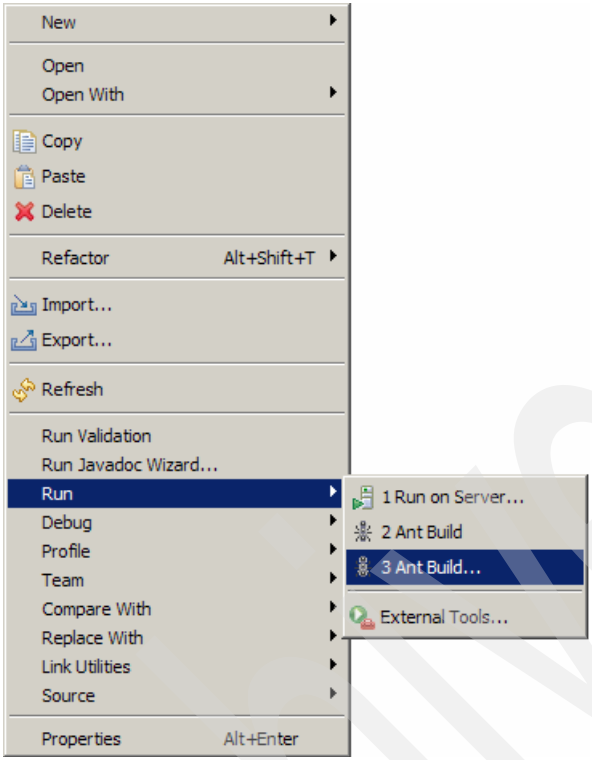


Figure 22-13 Launching Ant

5. When the Modify Attributes and Launch dialog appears, as seen in Figure 22-13 on page 1174, select the desired attributes. For example, select the **JRE** tab, select **Run in the same JRE as the workspace**, and then click **Run**.

   The default target specified in the build file is already selected as one target to run. You can check, in sequence, which ones are to be executed, and the execution sequence is shown in the Target execution order field.

   Note: Since dist depends on compile, even if you only select dist, the compile target is executed as well.

Archived

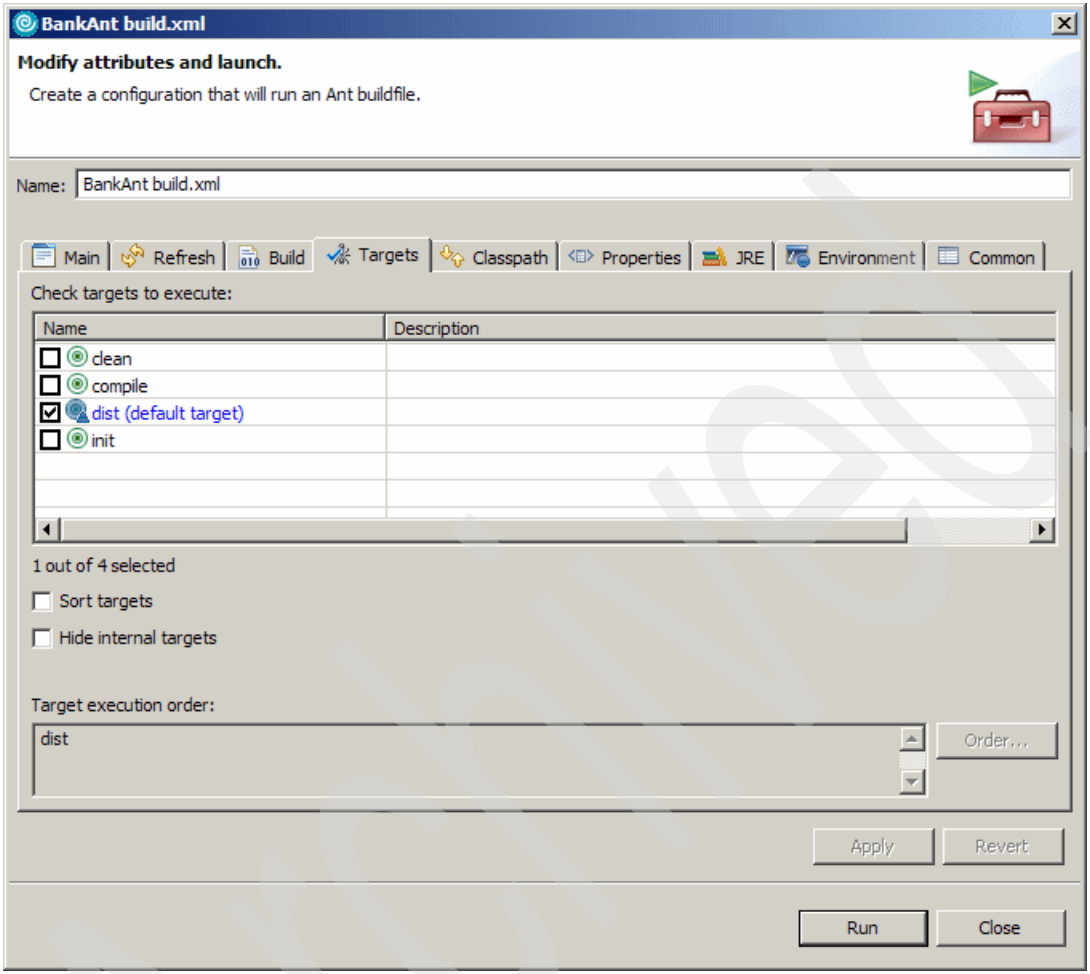


Figure 22-14 Selecting Ant targets to run

The Run Ant wizard gives you several tabs to configure or run the Ant process. The tabs allow you to do the following:

- Main: This tab allows you to select the build file, base directory, and arguments to pass to the Ant process.
- Refresh: This tab allows you to set some refresh options when the Ant process has finished running.
- Build: This tab allows you to set some build options before the Ant process is run.
- Target: This tab allows you to select the targets and the sequences the targets are to run.

– Classpath: This tab allows you to customize the classpath for the Ant process.
– Properties: This tab allows you to add, edit, or remove properties to be used by the Ant process.
– JRE: This tab allows you to select the Java Runtime Environment to use to run the Ant process.
– Environment: This tab allows you to define environmental variables to be used by the Ant process. This tab is only relevant when running in an external JRE.
– Common: This tab allows you to define the launch configuration for the Ant process.

When Ant is running, you will see output in the Log Console (Figure 22-15).

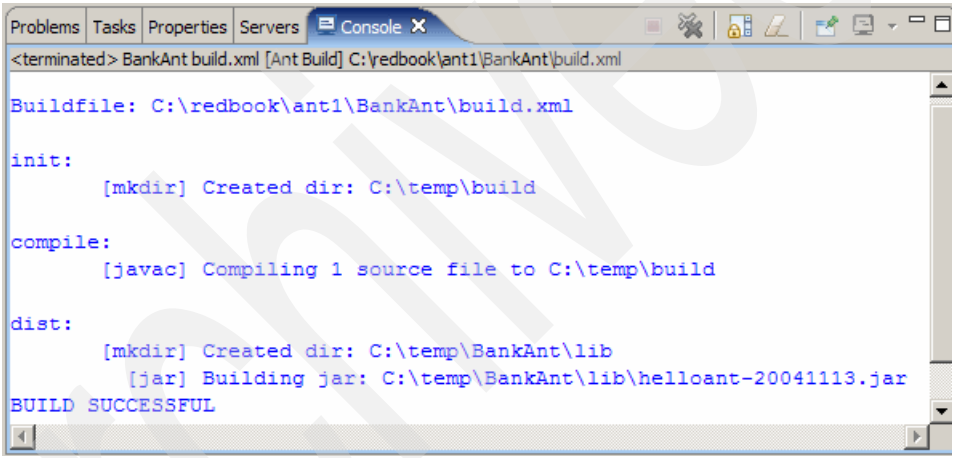


Figure 22-15 Log Console

22.3.7 Ant Log Console

The Log Console view opens automatically when running Ant, but if you want to open it manually, select **Window** → **Show view** → **Log Console**.

The Log Console shows that Ant has created the c:\temp\build directory, compiled the source files, created the c:\temp\BankAnt\lib directory, and generated a JAR file.

Archived

22.3.8 Rerun Ant

If you launch Ant again with the same target selected, Ant will not do anything at all since the c:\temp\build and c:\temp\BankAnt\lib directories were already created, and the class files in the build directory were already up-to-date.

22.3.9 Forced build

To generate a complete build, select the clean target as the first target and the dist target as the second target to run. You have to de-select dist, select clean, and then select dist again to get the execution order right (see Figure 22-16).

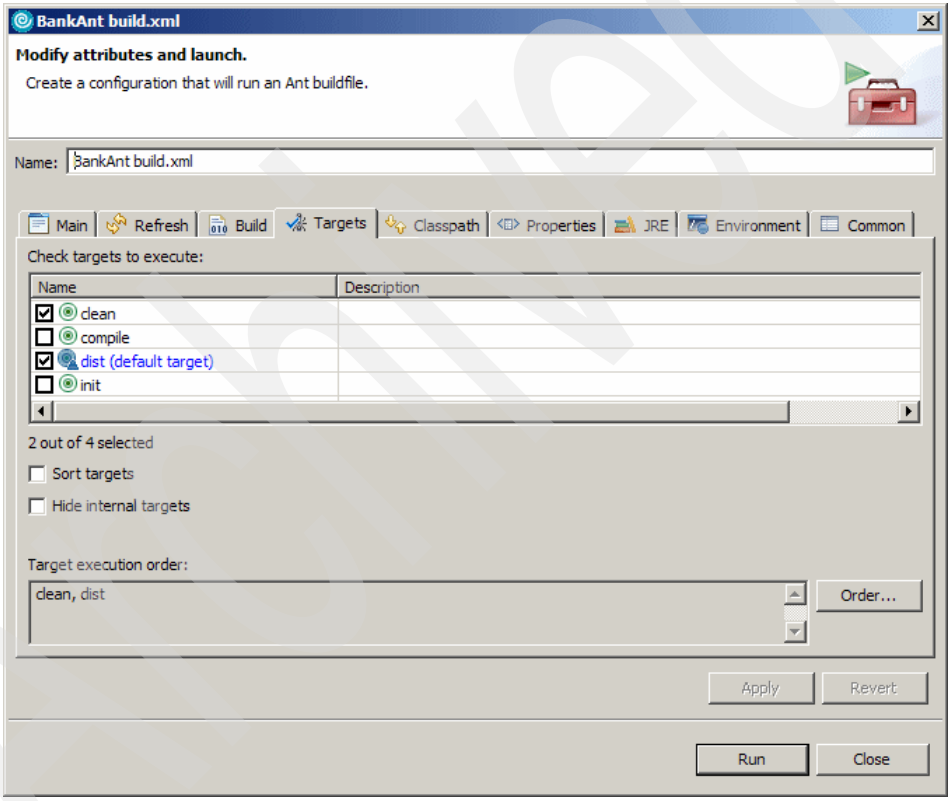


Figure 22-16 Launching Ant to generate complete build

22.3.10 Classpath problem

The classpath specified in the Java build path for the project is, unfortunately, not available to the Ant process. If you are building a project that references another project, the classpath for the javac compiler must be set up in the following way:

```
<javac srcdir="${source}" destdir="${build}" includes="**/*.java">
   <classpath>
      <pathelement location="../MyOtherProject"/>
      <pathelement location="../MyThirdProject"/>
   </classpath>
</javac>
```

22.3.11 Run the sample application to verify the Ant build

Now that you have completed the Ant build, we recommend that you verify the build by running the sample application as follows:

1. Open a Windows command window.
2. Navigate to the output directory of the Ant build (for example, c:\temp\BankAnt\lib).
3. Set the Java path by entering the following command:

   ```
   set PATH=%PATH%;c:\Program
   Files\IBM\Rational\SDP\6.0\runtimes\base_v6\java\bin
   ```

4. Enter the following to run the program:

   ```
   java -jar helloant-20050216.jar
   ```

   Where the timestamp in the jar filename will be dependent on when it is built.

   You should see the following output:

   ```
   G'day from Australia!
   ```

22.4 Build a J2EE application

As we have just demonstrated in the previous section, building a simple Java application using Ant is quite easy. In this section we demonstrate how to build a J2EE application from existing J2EE-related projects.

This section is organized as follows:

- J2EE application deployment packaging.
- Prepare for the sample.
- Create the build script.
- Run the Ant J2EE application build.

Archived

22.4.1 J2EE application deployment packaging

EAR, WAR, and EJB JAR files contain a number of deployment descriptors that control how the artifacts of the application are to be deployed onto an application server. These deployment descriptors are mostly XML files and are standardized within the J2EE specification.

While working in Application Developer, some of the information in the deployment descriptor is stored in XML files. The deployment descriptor files also contain information in a format convenient for interactive testing and debugging. This is one of the reasons it is so quick and easy to test J2EE applications in the integrated WebSphere Application Server V6.0 Test Environment included with Rational Application Developer.

The actual EAR being tested, and its supporting WAR, EJB, and client application JARs, are not actually created as a standalone file. Instead, a special EAR is used that simply points to the build contents of the various J2EE projects. Since these individual projects can be anywhere on the development machine, absolute path references are used.

When an enterprise application project is exported, a true standalone EAR is created, including all the module WARs, EJB JARs, and Java utility JARs it contains. Therefore, during the export operation, all absolute paths are changed into self-contained relative references within that EAR, and the internally optimized deployment descriptor information is merged and changed into a standard format. To create a J2EE-compliant WAR or EAR, we therefore have to use Application Developer's export function.

22.4.2 Prepare for the sample

For the purposes of demonstrating how to build a J2EE application using Ant, we will use the J2EE applications developed in Chapter 15, "Develop Web applications using EJBs" on page 827.

To import the BankEJB.zip Project Interchange file containing the sample code into Rational Application Developer, do the following:

1. Open the J2EE perspective Project Explorer view.
2. Select **File** → **Import**.
3. Select **Project Interchange** from the list of import sources and then click **Next**.
4. When the Import Projects dialog appears, click the **Browse** button next to zip file, navigate to and select the **BankEJB.zip** from the c:\6449code\ejb folder, and click **Open**.

Note: For information on downloading and unpacking the redbook sample code, refer to Appendix B, “Additional material” on page 1395.

5. Click **Select All** to select all projects and then click **Finish**.

After importing the BankEJB.zip Project Interchange file you should see the following projects:

- BankEJBEAR
- BankEJB
- BankEJBClient
- BankBasicWeb

22.4.3 Create the build script

To build the BankEJBEAR enterprise application, we created an Ant build script (build.xml) that utilizes the J2EE Ant tasks provided by Rational Application Developer.

Note: The completed version the build.xml can be found in the c:\6449code\ant\j2ee directory.

To add the Ant build script to the project, do the following:

1. Open the J2EE perspective Project Explorer view.
2. Expand **Enterprise Applications** → **BankEJBEAR**, and select **META-INF**.
3. Select **File** → **New** → **Other**.
4. When the New File dialog appears, select **Simple** → **File**, and click **Next**.
5. Enter `build.xml` in the File name field, and click **Finish**.
6. Enter the code in Example 22-2 into the build.xml file.

Tip: For simplicity, we suggest that you simply import the completed build.xml or cut and paste the contents from the c:\6449code\ant\j2ee\build.xml file.

7. Modify the value for the work.dir property to match your desired working directory (for example, c:/BankEAR_workdir), as highlighted in Example 22-2.

Example 22-2 J2EE Ant build.xml script

```
<?xml version="1.0" encoding="UTF-8"?>
<project name="ITSO RAD Pro Guide Ant" default="Total" basedir=".">
```

```
   <!-- Set global properties -->
   <property name="work.dir" value="c:/BankEAR_workdir" />
   <property name="dist" value="${work.dir}/dist" />
   <property name="project.ear" value="BankEJBEAR" />
   <property name="project.ejb" value="BankEJB" />
   <property name="project.war" value="BankBasicWeb" />
   <property name="type" value="incremental" />
   <property name="debug" value="true" />
   <property name="source" value="true" />
   <property name="meta" value="false" />
   <property name="noValidate" value="false" />

   <target name="init">
      <!-- Create the time stamp -->
      <tstamp />
      <!-- Create the dist directory where the output files are placed -->
      <mkdir dir="${dist}" />
   </target>

   <target name="info">
      <!-- Displays the properties for this run -->
      <echo message="debug=${debug}" />
      <echo message="type=${type}" />
      <echo message="source=${source}" />
      <echo message="meta=${meta}" />
      <echo message="noValidate=${noValidate}" />
      <echo message="Output directory=${dist}" />
      <echo message="project.ear=${project.ear}" />
      <echo message="project.ejb=${project.ejb}" />
      <echo message="project.war=${project.war}" />
   </target>

   <target name="deployEjb">
      <!-- Generates deployed code for the EJBs -->
      <ejbDeploy EJBProject="${project.ejb}" NoValidate="${noValidate}" />
   </target>

   <target name="buildEjb" depends="deployEjb">
      <!-- Builds the EJB project -->
      <projectBuild ProjectName="${project.ejb}" BuildType="${type}"
DebugCompilation="${debug}" />
      <projectBuild ProjectName="${project.ejb}" BuildType="${type}"
DebugCompilation="${debug}" />
   </target>

   <target name="buildWar">
      <!-- Builds the WAR project -->
```

Archived

```
      <projectBuild ProjectName="${project.war}" BuildType="${type}"
DebugCompilation="${debug}" />
   </target>

   <target name="buildEar">
      <!-- Builds the EAR project -->
      <projectBuild ProjectName="${project.ear}" BuildType="${type}"
DebugCompilation="${debug}" />
   </target>

   <target name="exportEjb" depends="init">
      <!-- Exports the EJB JAR -->
      <ejbExport ejbprojectname="${project.ejb}" ejbexportfile="${dist}/${project.ejb}.jar"
exportsource="${source}" overwrite="true" />
   </target>

   <target name="exportWar" depends="init">
      <!-- Exports the WAR file -->
      <warExport warprojectname="${project.war}" warexportfile="${dist}/${project.war}.war"
exportsource="${source}" overwrite="true" />
   </target>

   <target name="exportEar" depends="init">
      <!-- Exports the EAR file -->
      <echo message="Exported EAR files to ${dist}/${project.ear}.ear" />
      <earExport earprojectname="${project.ear}" earexportfile="${dist}/${project.ear}.ear"
exportsource="${source}" IncludeProjectMetaFiles="${meta}" overwrite="true" />
   </target>

   <target name="buildAll" depends="buildEjb,buildWar,buildEar">
      <!-- Builds all projects -->
      <echo message="Built all projects" />
   </target>

   <target name="exportAll" depends="exportEjb,exportWar,exportEar">
      <!-- Exports all files -->
      <echo message="Exported all files" />
   </target>

   <target name="Total" depends="buildAll,exportAll">
      <!-- Buidl all projects and exports all files -->
      <echo message="Total finished" />
   </target>

   <target name="clean">
      <!-- Delete the output files -->
      <delete file="${dist}/${project.ejb}.jar" failonerror="false" />
      <delete file="${dist}/${project.war}.war" failonerror="false" />
      <delete file="${dist}/${project.ear}.ear" failonerror="false" />
```

Archived

```
    </target>

</project>
```

The build.xml script includes the following Ant targets, which correspond to common J2EE application build.

- deployEjb: This generates the deploy code for all EJBs in the project.
- buildEjb: This builds the EJB project (compiles resources within project).
- buildWar: This builds the Web project (compiles resources within project).
- buildEar: This builds the Enterprise Application project (compiles resources within project).
- exportEjb: This exports the EJB project to a jar file.
- exportWar: This exports the Web project to a WAR file.
- exportEar: This exports the Enterprise Application project to an EAR file.
- buildAll: This invokes the buildEjb, buildWar, and buildEar targets.
- exportAll: This invokes the exportEjb, exportWar, and exportEar targets to create the BankEJBEAR.ear used for deployment.

In the global properties for this script we define a number of useful variables, such as the project names and the target directory. We also define a number of properties that we pass on to the Application Developer Ant tasks. These properties allow us to control whether the build process should perform a full or incremental build, whether debug statements should be included in the generated class files, and whether Application Developer's metadata information should be included when exporting the project.

When launching this Ant script, we can also override these properties by specifying other values in the arguments field, allowing us to perform different kinds of builds with the same script.

22.4.4 Run the Ant J2EE application build

When launching the build.xml script, you can select which targets to run and the execution order.

To run the Ant build.xml to build the J2EE application, do the following:

1. Open the J2EE perspective Project Explorer view.
2. Expand **Enterprise Applications** → **BankEJBEAR** → **META-INF**.
3. Right-click **build.xml**, and select **Run** → **3 Ant Build**.

Note: From the content menu for the Ant build script, the following two build options exist:

- 2 Ant Build: This will invoke the default target for the Ant build. In our example, this is the Total target, which in turn invokes buildAll and exportAll targets.
- 3 Ant Build: This will launch a dialog where you can select the targets and order, and provide parameters, as seen in Figure 22-17 on page 1185.

The Modify attributes and launch dialog will appear, as seen in Figure 22-17 on page 1185.

4. Click the **Main** tab.
   - To build the J2EE EAR file with debug, source files, and meta data, enter the following in the Arguments text area:

     ```
     -DDebug=true -Dsource=true -Dmeta=true
     ```

   Or:

   - To build the J2EE EAR for production deployment (without debug support, source code, and meta data), enter the following in the Arguments text area:

     ```
     -Dtype=full
     ```

5. Click the **Targets** tab, and check **Sort Targets**. Ensure that **Total** is checked (default).
6. Click the **JRE** tab. Select **Run in the same JRE as the workspace**.
7. Click **Apply** and then click **Run**.
8. Change to the following directory to ensure that the BankEJBEAR.ear file was created:

   ```
   c:\BankEAR_workdir\dist
   ```

The Console view will display the operations performed and their results.

Archived

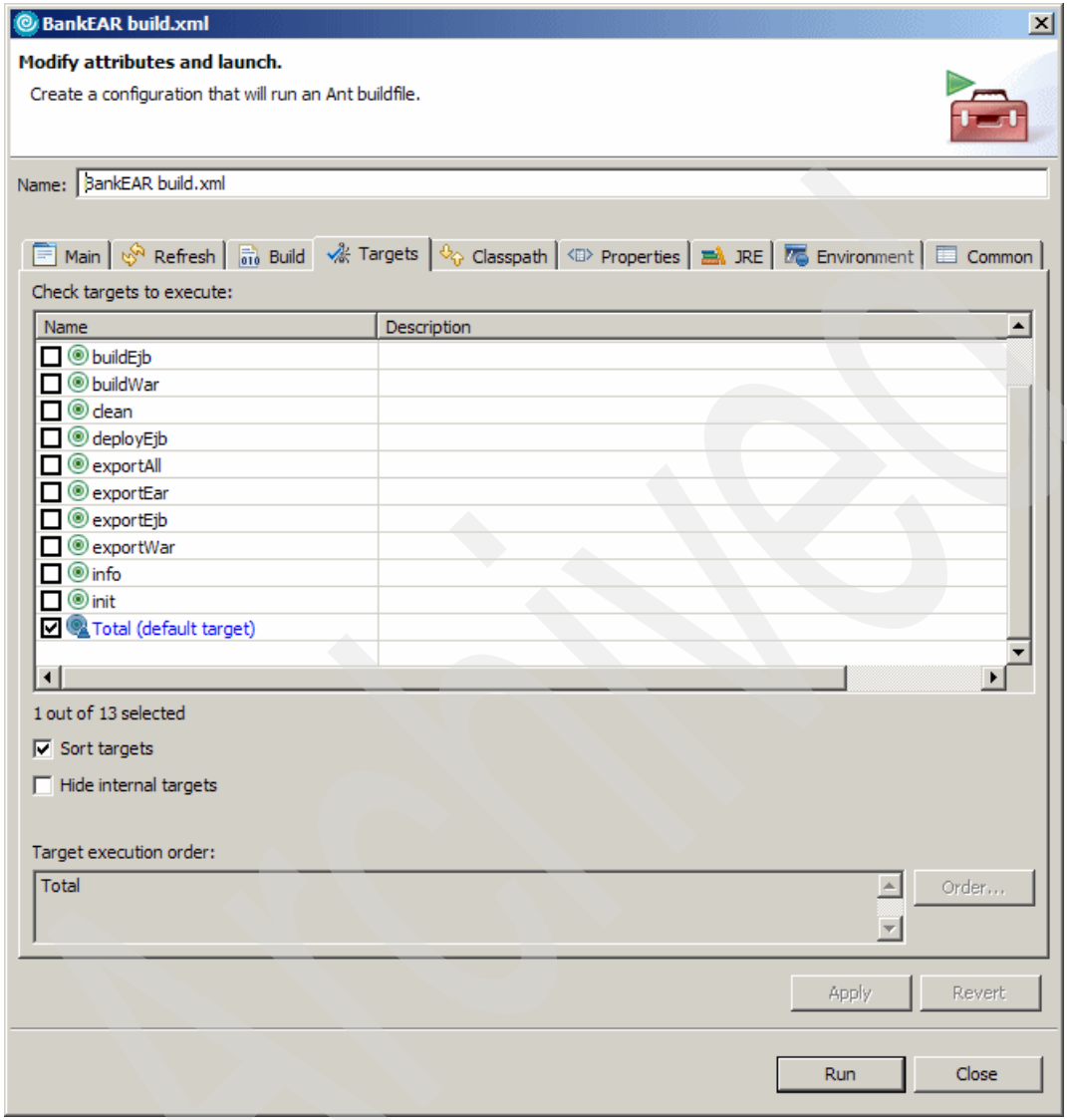


Figure 22-17 Launch Ant to build and export a J2EE project (EAR)

22.5 Run Ant outside of Application Developer

To automate the build process even further, you may want to run Ant outside of Application Developer by running Ant in *headless mode*.

22.5.1 Prepare for the headless build

Rational Application Developer includes a runAnt.bat file that can be used to invoke Ant in headless mode and passes the parameters that you specify. This will need to be customized for your environment.

The runAnt.bat file included with Rational Application Developer is located in the following directory:

```
<rad_home>\rwd\eclipse\plugins\com.ibm.etools.j2ee.ant_6.0.0
```

To create a headless Ant build script for J2EE project, do the following:

1. Copy the runAnt.bat file to a new file called itsoRunAnt.bat.
2. Modify the following values in the itsoRunAnt.bat, as seen in Example 22-3:

```
set STUDIO_DIR=C:\Program Files\IBM\Rational\SDP\6.0
set WORKSPACE=C:\workspace
```

Example 22-3 Snippet of the itsoRunAnt.bat (modified runAnt.bat)

```
@echo off
setlocal
REM RUNANT_DIR=This directory (which may, or may not, be your current working directory)
set RUNANT_DIR=%~dp0

:studio
REM The root directory of your Studio installation
set STUDIO_DIR=C:\Program Files\IBM\Rational\SDP\6.0
if not exist "%STUDIO_DIR%"\eclipse\jre  set STUDIO_DIR=%RUNANT_DIR%..\..\..
if not exist "%STUDIO_DIR%"\eclipse\jre  echo ERROR: incorrect STUDIO_DIR=%STUDIO_DIR%
if not exist "%STUDIO_DIR%"\eclipse\jre  goto done

:java
if not $%JAVA_DIR%$==$$ goto workspace
set JAVA_DIR=%STUDIO_DIR%\eclipse\jre\bin

:workspace
if not $%WORKSPACE%$==$$ goto check
REM ######################################################
REM ##### you must edit the "WORKSPACE" setting below #####
REM ######################################################
REM *********** The location of your workspace ************
set WORKSPACE=C:\workspace
```

Archived

22.5.2 Run the headless Ant build script

Attention: Prior to running Ant in headless mode, Rational Application Developer must be closed. If you do not close Rational Application Developer, you will get build errors when attempting to run Ant build in headless mode.

To run the itsoRunAnt.bat command file, do the following:

1. Ensure you have closed Rational Application Developer.
2. Open a Windows command prompt.
3. Navigate to the location of the itsoRunAnt.bat file.
4. Run the command file by entering the following:

   ```
   itsoRunAnt -buildfile c:\workspace\BankEJBEAR\META-INF\build.xml clean
   Total -DDebug=true -Dsource=true -Dmeta=true
   ```

   The -buildfile parameter should specify the fully qualified path of the build.xml script file. We can pass the targets to run as parameters to itsoRunAnt and we can also pass Java environment variables by using the -D switch.

   In this example we chose to run the clean and exportWar1 targets, and we chose to include the debug, Java source, and metadata files in the resulting EAR file.

   Note: We have included a file named output.txt, which contains the output from the headless Ant script for review purposes. The file can be found in the c:\6449code\ant\j2ee directory.

5. There are several build output files, which can be found in the c:\BankEARTest\dist directory:
   - BankBasicWeb.war, which is the BankBasicWeb project
   - BankEJB.war, which is the BankEJB project
   - BankEJBEAR.ear, which is the BankEJBEAR project with all the dependant projects included

Archived

Archived

23

Deploy enterprise applications

The term *deployment* can have many different meanings depending on the context. In this chapter we start out by defining the concepts of application deployment. The remainder of the chapter provides a working example for packaging and deploying the ITSO Bank enterprise application to a standalone IBM WebSphere Application Server V6.0 (Base Edition).

The application deployment concepts and procedures described in this chapter apply to IBM WebSphere Application Server V6.0 Base, Express, and Network Deployment editions, as well as the Rational Application Developer integrated WebSphere Application Server V6.0 Test Environment.

This chapter is organized into the following sections:

- Introduction to application deployment
- Prepare for the sample
- Package the application for deployment
- Deploy the enterprise application
- Verify the application

© Copyright IBM Corp. 2005. All rights reserved.

23.1 Introduction to application deployment

Deployment is a critical part of the J2EE application development cycle. Having a solid understanding of the deployment components, architecture, and process is essential for the successful deployment of the application.

In this section we review the following concepts of the J2EE and WebSphere deployment architecture:

- Common deployment considerations
- J2EE application components and deployment modules
- Java and WebSphere class loader
- Deployment descriptors
- WebSphere deployment architecture

Note: Further information on the IBM WebSphere Application Server deployment can be found in the following sources:

- *WebSphere V6 Planning and Design*, SG24-6446
- *WebSphere Application Server V6 Systems Management and Configuration*, SG24-6451
- *WebSphere Application Server V6 Scalability and Performance*, SG24-6392
- IBM WebSphere Application Server V6.0 InfoCenter found at:

  http://publib.boulder.ibm.com/infocenter/ws60help/index.jsp?topic=/com.ibm.websphere.base.doc/info/welcome_base.html

23.1.1 Common deployment considerations

Some of the most common factors that impact the deployment of a J2EE application are as follows:

- Deployment architecture: How can you create, assemble, and deploy an application properly if you do not understand the deployment architecture.

  Note: This chapter focuses on this aspect of deployment.

- Infrastructure: What the hardware and software constraints are for the application.
- Security: What security will be imposed on the application and what is the current security architecture.
- Application requirements: Do they imply a distributed architecture?

- Performance: How many users are using the system (frequency, duration, and concurrency).

23.1.2 J2EE application components and deployment modules

Within the J2EE application development life cycle, the application components are created, assembled, and then deployed. In this section, we explore the application component types, deployment modules, and packaging formats to gain a better understanding of what is being packaged (assembled) for deployment.

Application component types

In J2EE V1.4, there are four application component types supported by the runtime environment:

- Application Clients: Run in the Application Client Container.
- Applets: Run in the Applet Container.
- Web applications (servlets, JSPs, HTML pages): Run in the Web Container.
- EJBs: Run in the EJB Container.

Deployment modules

The J2EE deployment components are packaged for deployment as modules:

- Web application module
- EJB module
- Resource adapter module
- Application client module

Packaging formats (WAR and EAR)

There are two key packaging formats used to package J2EE modules for deployment, namely Web Application Archive (WAR) and Enterprise Application Archive (EAR). The packaging technology for both is similar to that of a jar file. WAR files can include servlets, JSPs, HTML, images, etc. Enterprise Application Archive (EAR) can be used to package EJB modules, resource adapter modules, and Web application modules.

23.1.3 Java and WebSphere class loader

Class loaders are responsible for loading classes, which may be used by an application. Understanding how Java and WebSphere class loaders work is an important element of WebSphere Application Server configuration needed for the application to work properly after deployment. Failure to set up the class loaders properly will often result in class loading exceptions such as ClassNotFoundException when trying to start the application.

Archived

Java class loader

Java class loaders enable the Java virtual machine (JVM) to load classes. Given the name of a class, the class loader should locate the definition of this class. Each Java class must be loaded by a class loader.

The the JVM is started; three class loaders are used:

- Bootstrap class load: The bootstrap class loader is responsible for loading the core Java libraries (that is, core.jar, server.jar, etc.) in the <JAVA_HOME>/lib directory. This class loader, which is part of the core JVM, is written in native code.

  Note: Beginning with JDK 1.4, the core Java libraries in the IBM JDK are no longer packaged in rt.jar as was previously the case (and is the case for the Sun JDKs), but instead split into multiple JAR files.

- Extensions class loader: The extensions class loader is responsible for loading the code in the extensions directories (<JAVA_HOME>/lib/ext or any other directory specified by the java.ext.dirs system property). This class loader is implemented by the sun.misc.Launcher$ExtClassLoader class.
- System class loader: The system class loader is responsible for loading the code that is found on java.class.path, which ultimately maps to the system CLASSPATH variable. This class loader is implemented by the sun.misc.Launcher$AppClassLoader class.

Delegation is a key concept to understand when dealing with class loaders. It states that a custom class loader (a class loader other than the bootstrap, extension, or system class loaders) delegates class loading to its parent before trying to load the class itself. The parent class loader can either be another custom class loader or the bootstrap class loader. Another way to look at this is that a class loaded by a specific class loader can only reference classes that this class loader or its parents can load, but not its children.

The Extensions class loader is the parent for the System class loader. The Bootstrap class loader is the parent for the Extensions class loader. The class loaders hierarchy is shown in Figure 23-1 on page 1193.

If the System class loader needs to load a class, it first delegates to the Extensions class loader, which in turn delegates to the Bootstrap class loader. If the parent class loader cannot load the class, the child class loader tries to find the class in its own repository. In this manner, a class loader is only responsible for loading classes that its ancestors cannot load.

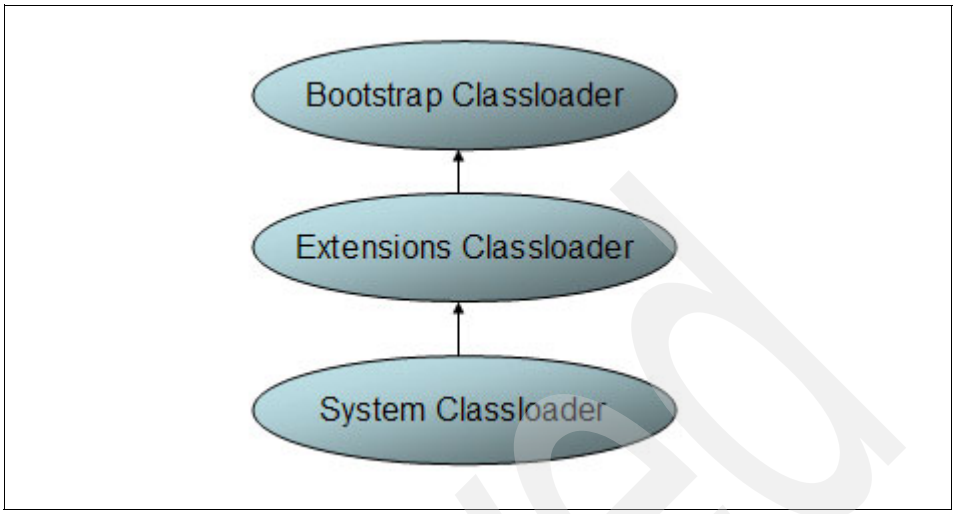


Figure 23-1 Java class loaders hierarchy

WebSphere class loader

It is important to keep in mind when reading the following material on WebSphere class loaders, that each Java Virtual Machine (JVM) has its own setup of class loaders. This means that in a WebSphere environment hosting multiple application servers (JVMs), such as a Network Deployment configuration, the class loaders for the JVMs are completely separated even if they are running on the same physical machine.

WebSphere provides several custom delegated class loaders, as shown in Figure 23-2 on page 1194.

Archived

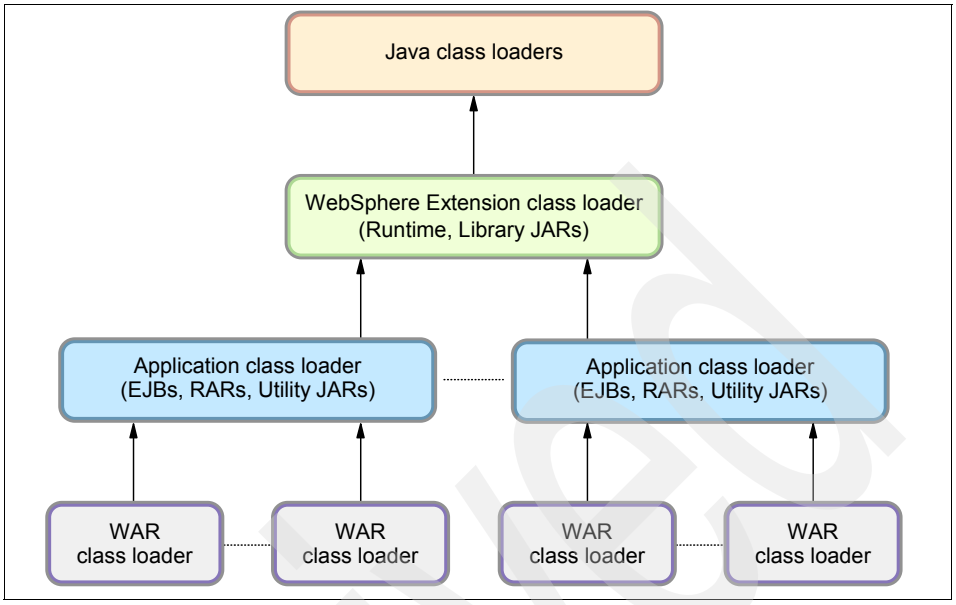


Figure 23-2 WebSphere class loaders hierarchy

In Figure 23-2, the top box represents the Java (Bootstrap, Extension, and System) class loaders. WebSphere does not load much here, just enough to get itself bootstrapped and initialize the WebSphere extension class loader.

WebSphere extensions class loader

The WebSphere extensions class loader is where WebSphere itself is loaded. It uses the following directories to load the required WebSphere classes:

- <JAVA_HOME>\lib
- <WAS_HOME>\classes (Runtime Class Patches directory, or RCP)
- <WAS_HOME>\lib (Runtime class path directory, or RP)
- <WAS_HOME>\lib\ext (Runtime Extensions directory, or RE)
- <WAS_HOME>\installedChannels

The WebSphere runtime is loaded by the WebSphere extensions class loader based on the ws.ext.dirs system property, which is initially derived from the WS_EXT_DIRS environment variable set in the setupCmdLine script file. The default value of ws.ext.dirs is the following:

```
SET
WAS_EXT_DIRS=%JAVA_HOME%\lib;%WAS_HOME%\classes;%WAS_HOME%\lib;%WAS_HOME%\i
nstalledChannels;%WAS_HOME%\lib\ext;%WAS_HOME%\web\help;%ITP_LOC%\plugins\c
om.ibm.etools.ejbdeploy\runtime
```

The RCP directory is intended to be used for fixes and other APARs that are applied to the application server runtime. These patches override any copies of the same files lower in the RP and RE directories. The RP directory contains the core application server runtime files. The bootstrap class loader first finds classes in the RCP directory, then in the RP directory. The RE directory is used for extensions to the core application server runtime.

Each directory listed in the ws.ext.dirs environment variable is added to the WebSphere extensions class loaders class path. In addition, every JAR file and/or ZIP file in the directory is added to the class path.

You can extend the list of directories/files loaded by the WebSphere extensions class loaders by setting a ws.ext.dirs custom property to the Java virtual machine settings of an application server.

Application and Web module class loaders

J2EE applications consist of five primary elements: Web modules, EJB modules, application client modules, resource adapters (RAR files), and Utility JARs. Utility JARs contain code used by both EJBs and/or servlets. Utility frameworks (like log4j) are a good example of a utility JAR.

EJB modules, utility JARs, resource adapters files, and shared libraries associated with an application are always grouped together into the same class loader. This class loader is called the application class loader. Depending on the application class loader policy, this application class loader can be shared by multiple applications (EAR) or be unique for each application (the default).

By default, Web modules receive their own class loader (a WAR class loader) to load the contents of the WEB-INF/classes and WEB-INF/lib directories. The default behavior can be modified by changing the application's WAR class loader policy (the default being Module). If the WAR class loader policy is set to Application, the Web module contents are loaded by the *application class loader* (in addition to the EJBs, RARs, utility JARs, and shared libraries). The application class loader is the parent of the WAR class loader.

The application and the Web module class loaders are reloadable class loaders. They monitor changes in the application code to automatically reload modified classes. This behavior can be altered at deployment time.

Handling JNI code

Due to a JVM limitation, code that needs to access native code via a Java Native Interface (JNI) must not be placed on a reloadable class path, but on a static class path. This includes shared libraries for which you can define a native class path, or the application server class path. So if you have a class loading native

code via JNI, this class must not be placed on the WAR or application class loaders, but rather on the WebSphere extensions class loader.

It may make sense to break out just the lines of code that actually load the native library into a class of its own and place this class on a static class loader. This way you can have all the other code on a reloadable class loader.

23.1.4 Deployment descriptors

Information describing a J2EE application and how to deploy it into a J2EE container is stored in XML files called deployment descriptors. An EAR file normally contains multiple deployment descriptors, depending on the modules it contains. Figure 23-3 shows a schematic overview of a J2EE EAR file. In this figure the various deployment descriptors are designated with DD after their name.

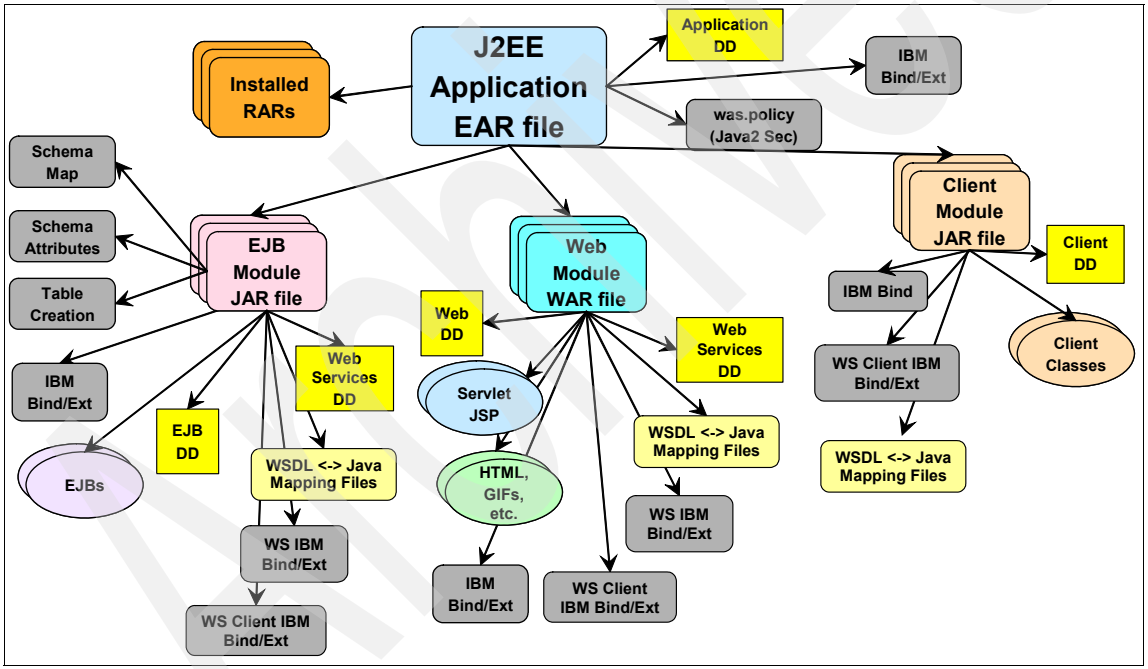


Figure 23-3 J2EE EAR file structure

The deployment descriptor of the EAR file itself is stored in the META-INF directory in the root of the EAR and is called application.xml. It contains information about the modules making up the application.

The deployment descriptors for each module are stored in the META-INF directory of the module and are called web.xml (for Web modules), ejb-jar.xml

(for EJB modules), ra.xml (for resource adapter modules), and application-client.xml (for Application client modules). These files describe the contents of a module and allow the J2EE container to configure things like servlet mappings, JNDI names, etc.

Classpath information specifying which other modules and utility JARs are needed for a particular module to run, is stored in the manifest.mf file, also in the META-INF directory of the modules.

In addition to the standard J2EE deployment descriptors, EAR files produced by Rational Application Developer or the Application Server Toolkit can also include additional WebSphere-specific information used when deploying applications to WebSphere environments. This supplemental information is stored in an XMI file, also in the META-INF directory of the respective modules. Examples of information in the IBM-specific files are IBM extensions like servlet reloading and EJB access intents.

New in WebSphere Application Server V6 is also the information contained in the enhanced EAR files. This information, which includes settings for the resources required by the application, is stored in an ibmconfig subdirectory of the application's (EAR file's) META-INF directory. In the ibmconfig directory are the well-known directories for a WebSphere cell configuration.

Rational Application Developer and the Application Server Toolkit have easy-to-use editors for working with all deployment descriptors. The information that goes into the different files is shown on one page in the GUI, eliminating the need to be concerned about what information is put into what file. However, if you are interested, you can click the Source tab of the deployment descriptor editor to see the text version of what is actually stored in that descriptor.

For example, if you open the EJB deployment descriptor, you will see settings that are stored across multiple deployment descriptors for the EJB module, including:

- The EJB deployment descriptor, ejb-jar.xml
- The extensions deployment descriptor, ibm-ejb-jar-ext.xmi
- The bindings file, ibm-ejb-jar-bnd.xmi files
- The access intent settings, ibm-ejb-access-bean.xmi

Archived

The deployment descriptors can be modified from within Rational Application Developer, by double-clicking the file to open the Deployment Descriptor Editor, as seen in Figure 23-4.

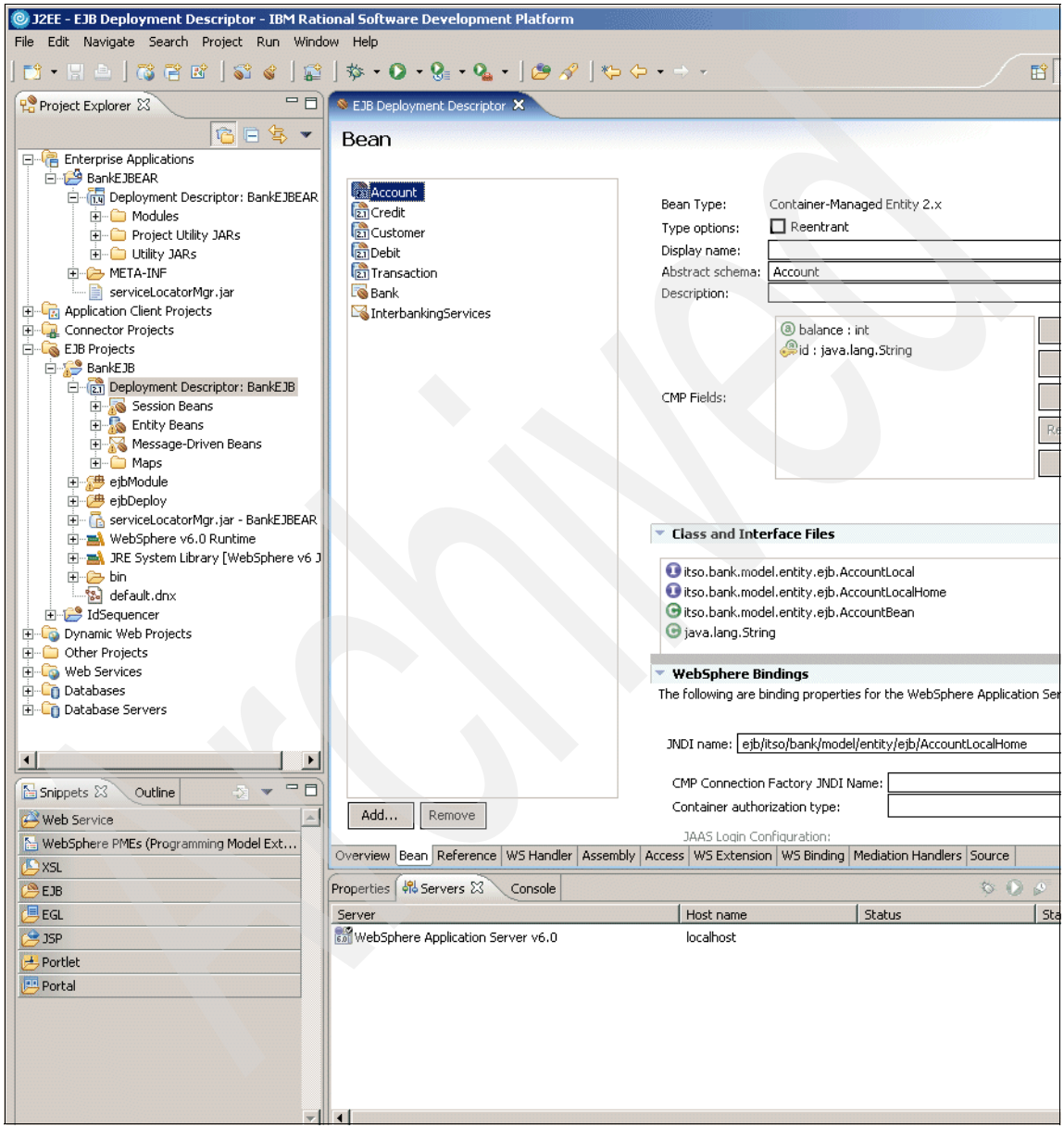


Figure 23-4 Deployment Descriptor Editor in Rational Application Developer

While the editor will show you information stored in all the relevant deployment descriptor files on the appropriate tabs, the Source tab will only show you the source of the deployment descriptor itself (for example, ejb-jar.xml or web.xml) and not the IBM extensions and bindings stored in the WebSphere-specific deployment descriptor files. If you want to view the results of updates to those files in the source, you will need to open each file individually. By hovering over the EJB Deployment Descriptor Caption tab you can see the different files that make up the EJB deployment descriptor you are editing. The descriptor files are kept in the META-INF directory of the module you are editing.

When you have made changes to a deployment descriptor, save it by pressing Ctrl+S and then close it.

Note: "Customize the deployment descriptors" on page 1220 provides an example of customizing the ITSO Bank EJB deployment descriptors for the desired database server type (Cloudscape or DB2 Universal Database).

23.1.5 WebSphere deployment architecture

This section provides an overview of the IBM WebSphere Application Server V6.0 deployment architecture.

IBM Rational Application Developer V6.0 includes an integrated WebSphere Application Server V6.0 Test Environment. Administration of the server and applications is performed by using the WebSphere Administrative Console for such configuration tasks as:

- J2C authentication aliases
- Datasources
- Service buses
- JMS queues and connection factories

Due to the loose coupling between Rational Application Developer and WebSphere Application Server, applications can deploy in the following ways:

- Deploy from a Rational Application Developer project to the integrated WebSphere Application Server V6.0 Test Environment.
- Deploy from a Rational Application Developer project to a separate WebSphere Application Server runtime environment.
- Deploy via EAR to an integrated WebSphere Application Server V6.0 Test Environment.
- Deploy via EAR to a separate WebSphere Application Server runtime environment.

Archived

Administration of the application server is performed through the use of an Internet Web browser.

In addition, the WebSphere Application Server V6 can obtain an EAR from external tools without the deployment code generated and be loaded into the Rational Application Developer or Application Server Toolkit (AST). Rational Application Developer or AST will generate the deployment code for the WebSphere Application Server and a new EAR will be saved to deploy out to the application server.

A diagram of the deployment architecture and the various mechanisms to deploy out an application are provided in Figure 23-5.

Details on how to configure the servers in IBM Rational Application Developer V6 are documented in Chapter 19, “Servers and server configuration” on page 1043.

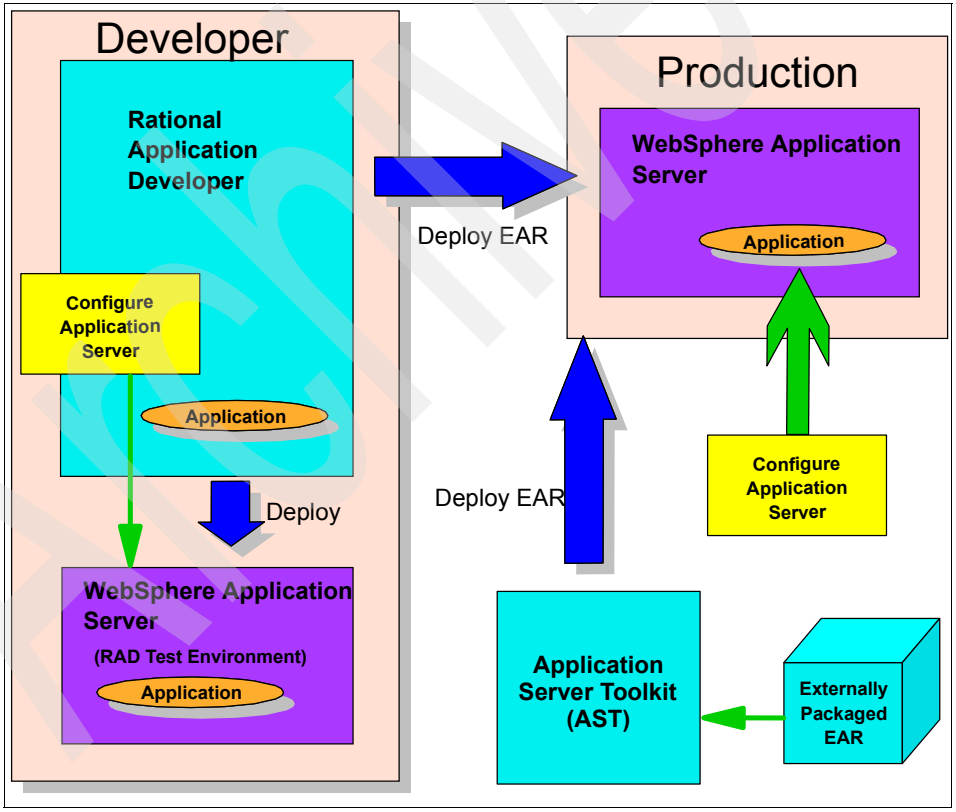


Figure 23-5 Deployment architecture

Profiles in WebSphere Application Server

WebSphere Application Server V6.0 has been split into two separate components:

- A set of shared read-only product files
- A set of configuration files known as WebSphere Profiles

The first component is the runtime part of WebSphere Application Server V6.0, while the second component is a new concept called WebSphere Profiles.

A WebSphere Profile is the set of configurable files including WebSphere Application Server configuration, applications, and properties files that constitute a new application server. Having multiple profiles equates to having multiple WebSphere Application Server instances for use with a number of applications.

Note: This function is similar to the **wsinstance** command that was available in WebSphere Application Server V5.x, creating multiple runtime configurations using the same installation.

In IBM Rational Application Developer V6 this allows a developer to configure multiple application servers for various applications that they may be working with. Separate WebSphere Profiles can then be set up as test environments in Rational Web/Application Developer (see Chapter 19, “Servers and server configuration” on page 1043).

WebSphere enhanced EAR

The WebSphere enhanced EAR is a feature of IBM WebSphere Application Server V6.0 that provides an extension of the J2EE EAR with additional configuration information for resources typically required by J2EE applications. This information is not mandatory to be supplied at packaging time, but it can simplify the deployment of applications to WebSphere Application Server for selected scenarios.

The Enhanced EAR Editor can be used to edit several WebSphere Application Server V6 specific configurations, such as data sources, class loader policies, substitution variables, shared libraries, virtual hosts, and authentication settings. The configuration settings can be made simply within the editor and published with the EAR at the time of deployment.

The upside of the tool is that it makes the testing process simpler and repeatable, since the configurations can be saved to files and then shared within a team’s repository. Even though it will not let you configure every possible runtime setting, it is a good tool for development purposes because it eases the process of configuring the most common ones.

The downside is that the configurations the tool makes will be attached to the EAR, and will not be visible from the WebSphere Administrative Console. The WebSphere Administrative Console is only able to edit settings that belong to the cluster, node, and server contexts.

When you change a configuration using the Enhanced EAR Editor, these changes are made within the application context. The deployer can still make changes to the EAR file using the Application Server Toolkit (AST), but it still requires a separate tool. Furthermore, in most cases these settings are dependent on the node the application server is installed in anyway, so it may not make sense to configure them at the application context for production deployment purposes.

Table 23-1 lists the supported resources that the enhanced EAR provides and the scope in which they are created.

Table 23-1 Enhanced EAR resources supported and their scope

Resource Type	Scope
JDBC Providers	Application
DataSources	Application
Substitution variables	Application
Class loader policies	Application
Shared libraries	Server
JAAS authentication aliases	Cell
Virtual hosts	Cell

The following example demonstrates how to use the Extended EAR Editor to create a data source, as an alternative to using the WebSphere Administrative Console.

1. Start Rational Application Developer.
2. Open the J2EE perspective Project Explorer view.
3. Expand **Enterprise Applications** → **BankEJBEAR**.
4. Double-click the **Deployment Descriptor: BankEJBEAR** to open in the editor.
5. Configure JAAS Authentication for DB2 Universal Database.

> **Note:** This step is not required for Cloudscape (no authentication required).

a. Click the **Deployment** tab, as seen in Figure 23-6 on page 1203.

b. Scroll down the page until you see the Authentication property. This allows you to define a login configuration used by JAAS Authentication.

c. Select the **dbuser** entry (default) and click **Edit**.

   Alternatively, click **Add** to create a new entry.

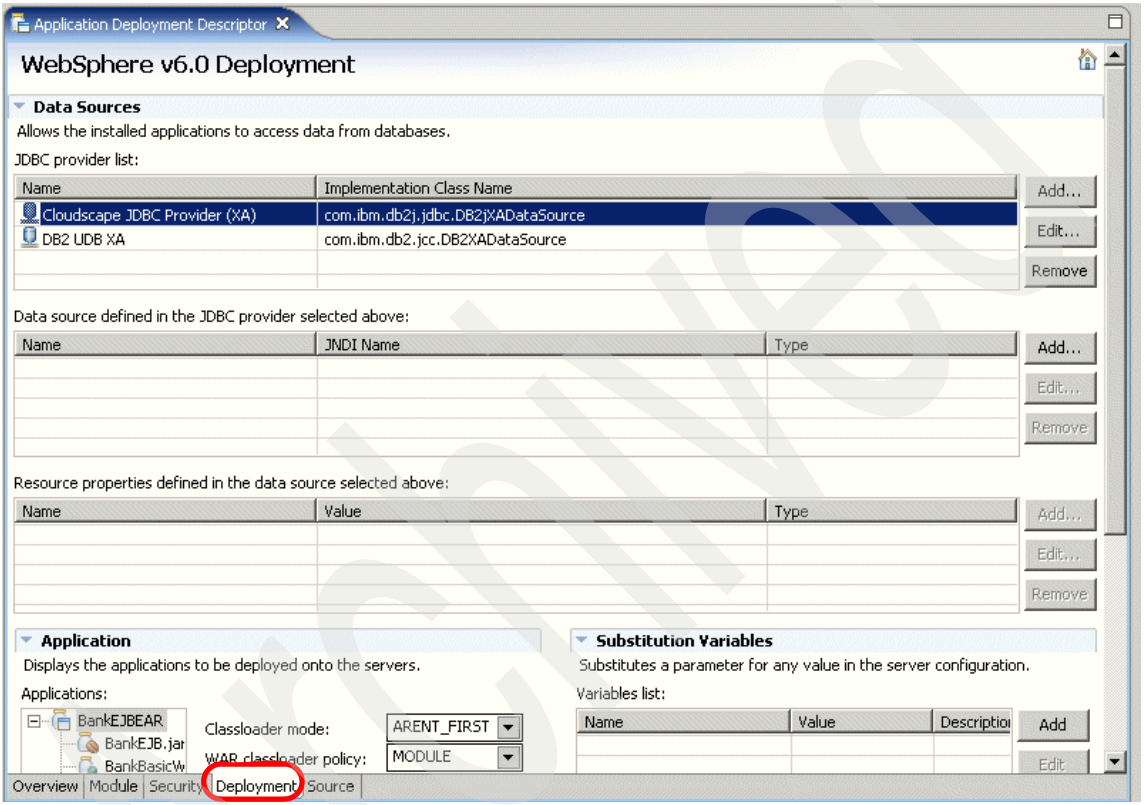


Figure 23-6 Application Deployment Descriptor - Deployment tab

d. When the Add JAAS Authentication Entry dialog appears, enter the user ID and password for the DB2 Universal Database, as seen in Figure 23-7 on page 1204.

Archived

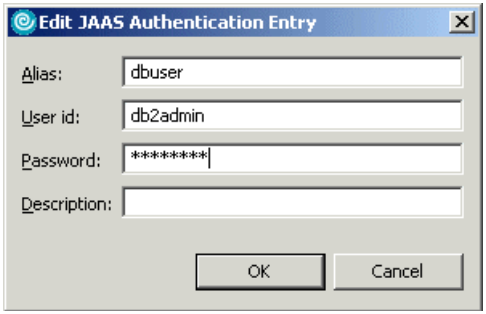


Figure 23-7 JAAS Authentication Entry

6. Add a JDBC provider.
   a. From the Enhanced EAR Editor, scroll back to the JDBC Provider section at the top of the page.
   b. Click **Add** next to the JDBC Provider list.
   c. When the Create a JDBC Provider dialog appears, select **IBM DB2** as the Database type, select **DB2 Universal JDBC Driver Provider (XA)** from the JDBC provider type (as seen in Figure 23-8 on page 1205), and then click **Next**.

> **Note:** Note that for our development purposes, the DB2 Universal JDBC Driver Provider (non XA) would work just as well, because we will not use the XA (two-phase commit) capabilities.

Archived

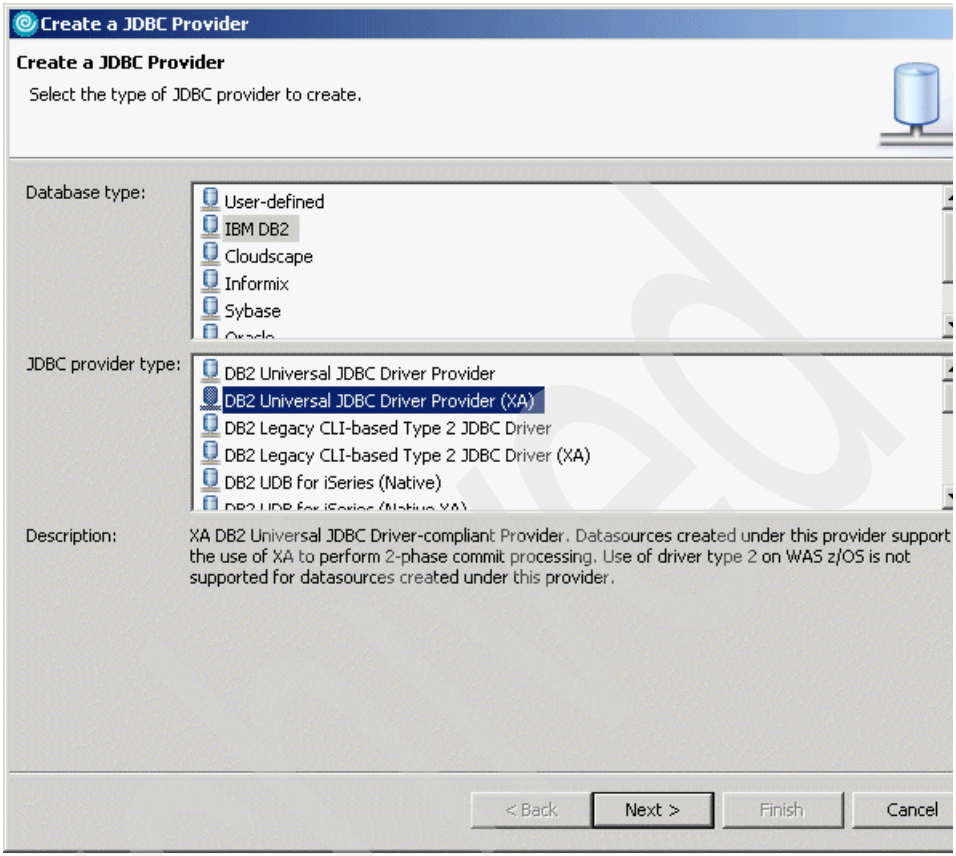


Figure 23-8 Create a JDBC Provider

d. Enter `DB2 XA JDBC Provider` in the Name field, accept the defaults for remaining fields (as seen in Figure 23-9 on page 1206), and click **Finish**.

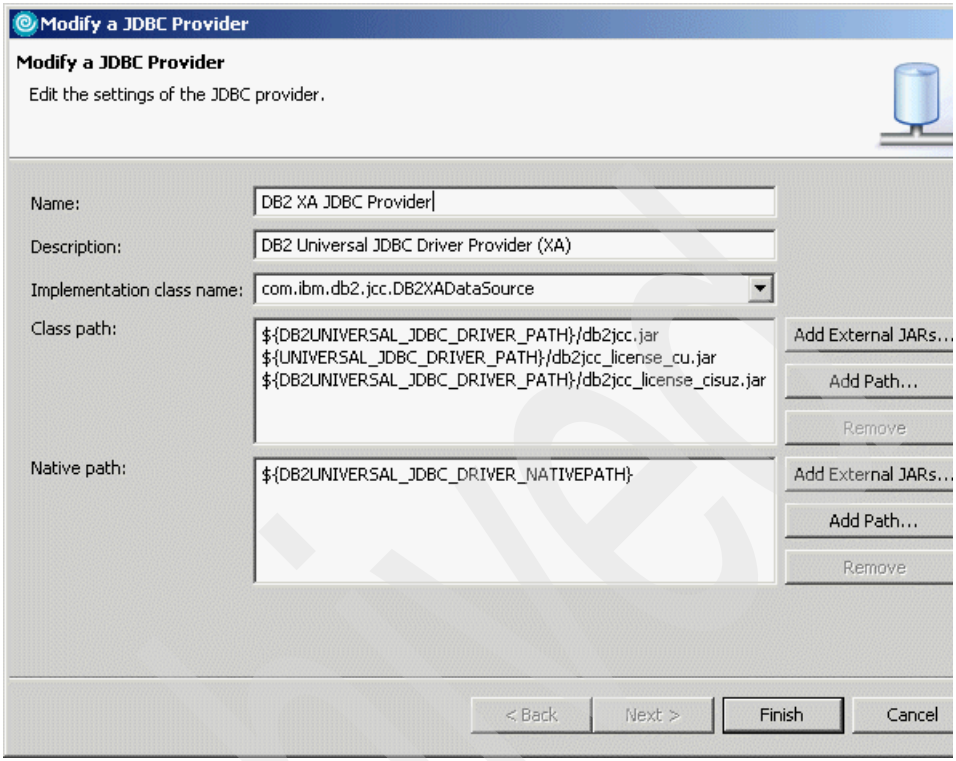


Figure 23-9 Create a JDBC Provider - Name

7. Add a data source.
   a. Select the **DB2 XA JDBC Provider** you created in the previous step under the JDBC provider list.
   b. Click **Add** next to the Data source section.
   c. When the Create a Data Source (page 1) dialog appears, select **DB2 Universal JDBC Driver Provider (XA)** (as seen in Figure 23-10 on page 1207), and then click **Next**.

Archive

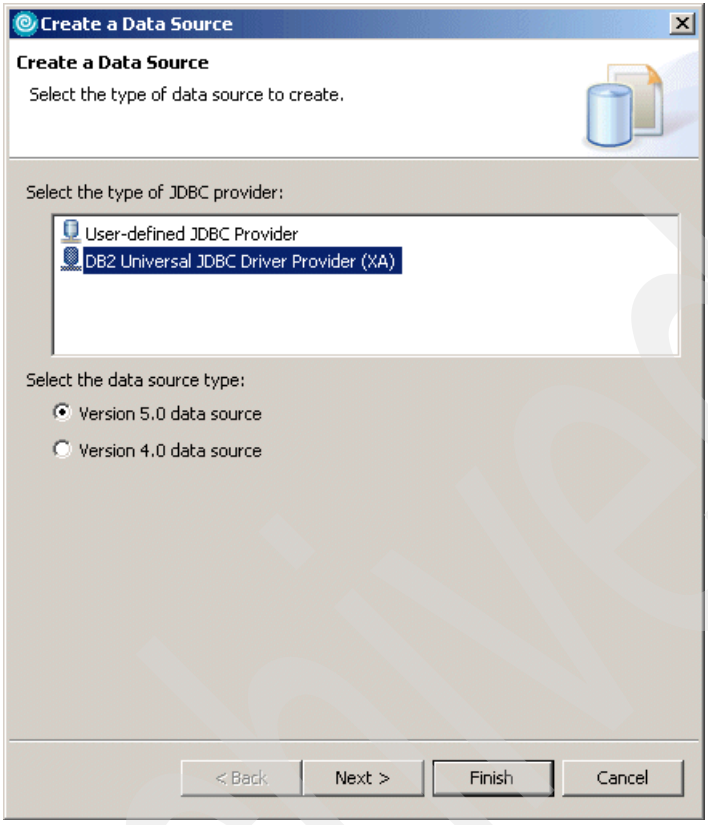


Figure 23-10 Create a Data Source (page 1)

d. When the Create Data Source (page 2) dialog appears, we entered the following (as seen in Figure 23-11 on page 1208) and then clicked **Next**:

- Name: `BankDS`
- JNDI name: `jdbc/bankDS`
- Description: Bank DataSource
- Component-managed authenticated alias: Select **dbuser**.

Archived

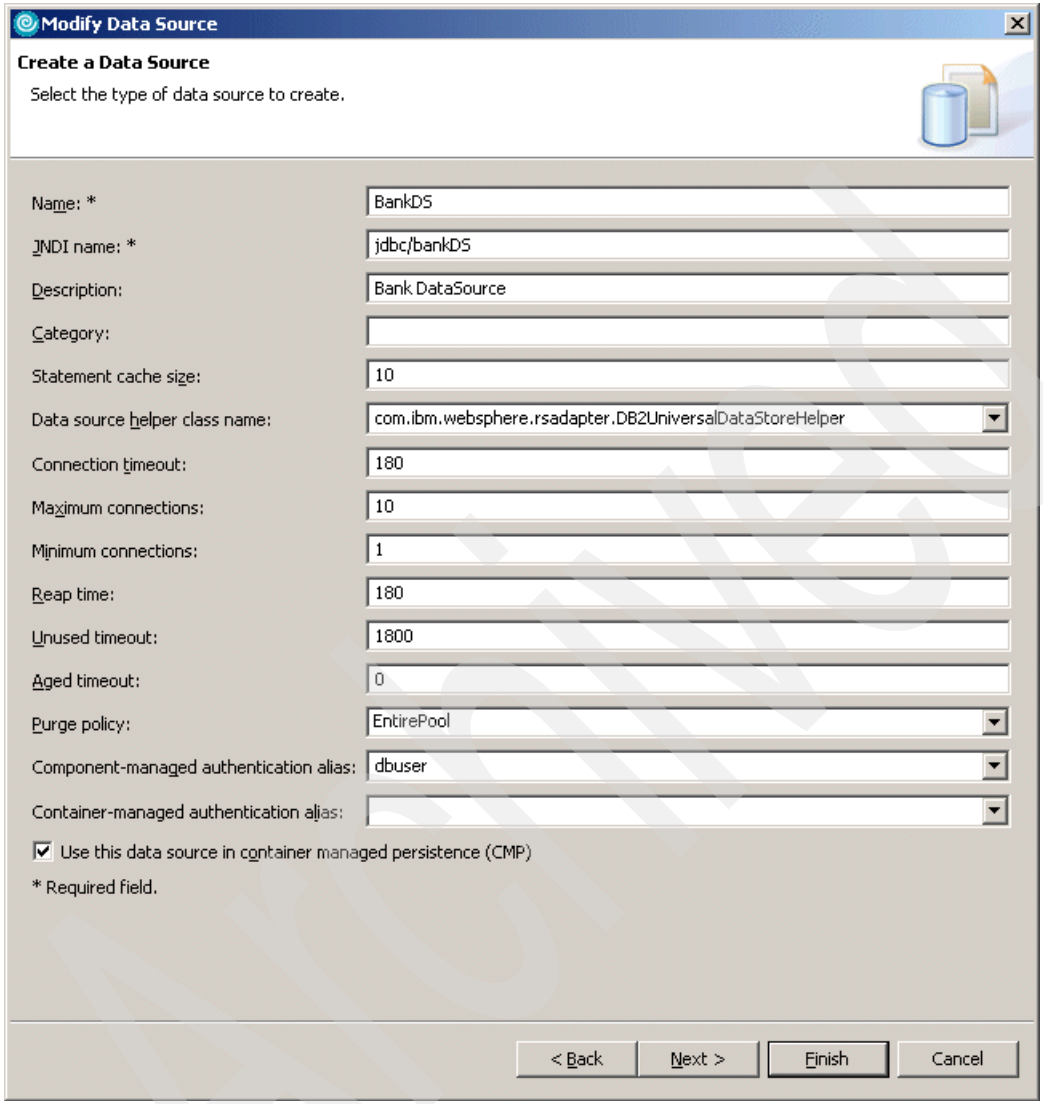


Figure 23-11 Create a Data Source (page 2)

e. When the Create Resource Properties dialog appears, select the **databaseName** variable from the Resource Properties list, enter `BANK` in the variable value field (as seen in Figure 23-12 on page 1209), and then click **Finish**.

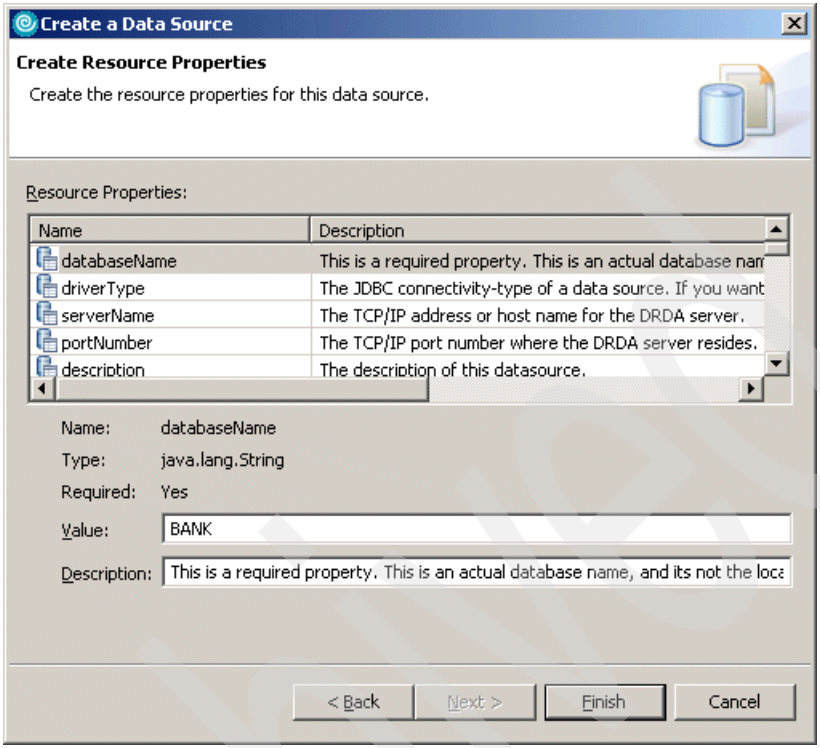


Figure 23-12 Create a Data Source (page 3)

8. Save the Deployment Descriptor.

The settings configured using the Enhanced EAR Editor will be packaged as part of the standard EAR export process and published at the time of installing the enterprise application in WebSphere Application Server.

Note: For more detailed information and an example of using the WebSphere enhanced EAR refer to:

- *Packaging applications* chapter in the *WebSphere Application Server V6 Systems Management and Configuration*, SG24-6451
- IBM WebSphere Application Server V6.0 InfoCenter found at:

  http://publib.boulder.ibm.com/infocenter/ws60help/index.jsp?topic=/com.ibm.websphere.base.doc/info/welcome_base.html

WebSphere Rapid Deployment

WebSphere Rapid Deployment is a collection of tools and technologies introduced in IBM WebSphere Application Server V6.0 that makes application development and deployment easier than ever before.

WebSphere Rapid Deployment consists of the following elements:

- Annotation-based programming (Xdoclet)
- Rapid deployment tools
- Fine-grained application updates

Annotation-based programming (Xdoclet)

Annotation-based programming speeds up application development by reducing the number of artifacts that you need to develop and manage on your own. By adding metadata tags to the Java code, the WebSphere Rapid Deployment tools can automatically create and manage the artifacts to build a J2EE-compliant module and application.

Rapid deployment tools

Using the rapid deployment tools part of WebSphere Rapid Deployment you can:

- Create a new J2EE application quickly without the overhead of using an integrated development environment (IDE).
- Package J2EE artifacts quickly into an EAR file.
- Deploy and test J2EE modules and full applications quickly on a server.

For example, you can place full J2EE applications (EAR files), application modules (WAR files, EJB JAR files), or application artifacts (Java source files, Java class files, images, JSPs, etc.) into a configurable location on your file system, referred to as the *monitored,* or *project,* directory. The rapid deployment tools then automatically detect added or changed parts of these J2EE artifacts and perform the steps necessary to produce a running application on an application server.

There are two ways to configure the monitored directory, each performing separate and distinct tasks (as depicted in Figure 23-13 on page 1212):

- Free-form project
- Automatic application installation project

With the free-form approach you can place in a single project directory the individual parts of your application, such as Java source files that represent servlets or enterprise beans, static resources, XML files, and other supported application artifacts. The rapid deployment tools then use your artifacts to automatically place them in the appropriate J2EE project structure, generate any

additional required artifacts to construct a J2EE-compliant application, and deploy that application on a target server.

The automatic application installation project, on the other hand, allows you to quickly and easily install, update, and uninstall J2EE applications on a server. If you place EAR files in the project directory they are automatically deployed to the server. If you delete EAR files from the project directory, the application is uninstalled from the server. If you place a new copy of the same EAR file in the project directory, the application is reinstalled. If you place WAR or EJB JAR files in the automatic application installation project, the rapid deployment tool generates the necessary EAR wrapper and then publishes that EAR file on the server. For RAR files, a wrapper is not created. The standalone RAR files are published to the server.

The advantage of using a free-form project is that you do not need to know how to package your application artifacts into a J2EE application. The free-form project takes care of the packaging part for you. The free-form project is suitable when you just want to test something quickly, perhaps write a servlet that performs a task.

An automatic application installation project, on the other hand, simplifies management of applications and relieves you of the burden of going through the installation panels in the WebSphere administrative console or developing wsadmin scripts to automate your application deployment.

The rapid deployment tools can be configured to deploy applications either onto a local or remote WebSphere Application Server.

Archive

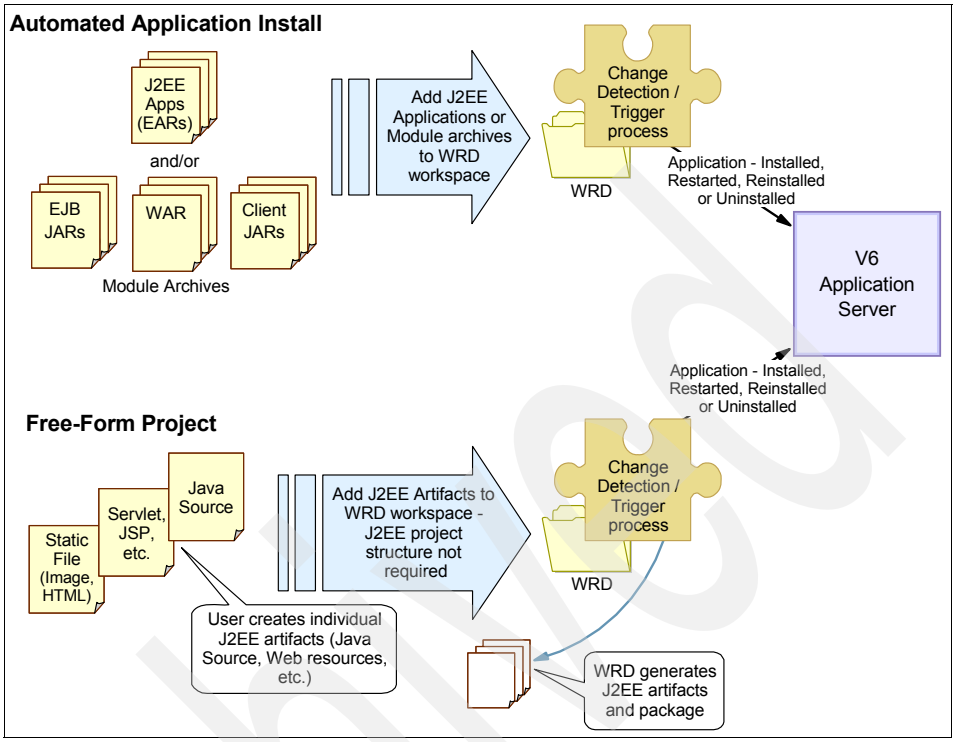


Figure 23-13 WebSphere Rapid Deployment modes

Note: For more detailed information on WebSphere Rapid Deployment refer to the following:

- *WebSphere V6 Planning and Design*, SG24-6446
- *WebSphere Application Server V6 Systems Management and Configuration*, SG24-6451
- IBM WebSphere Application Server V6.0 InfoCenter found at:

  `http://publib.boulder.ibm.com/infocenter/ws60help/index.jsp?topic=/com.ibm.websphere.base.doc/info/welcome_base.html`

23.2 Prepare for the sample

This section describes the steps required to prepare the environment for the deployment sample. We will use the ITSO Bank enterprise application developed in Chapter 15, “Develop Web applications using EJBs” on page 827, to demonstrate the deployment process.

This section includes the following tasks:

- Review the deployment scenarios.
- Install prerequisite software
- Import the sample application Project Interchange file.
- Set up the sample database.

23.2.1 Review the deployment scenarios

Now that the Rational Application Developer integration with WebSphere Application Server V6.0 is managed the same as a standalone WebSphere Application Server, the procedure to deploy the ITSO Bank sample application is nearly identical to that of a standalone WebSphere Application Server.

There are several possible configurations in which the sample can be installed. The deployment process in this chapter accounts for the following two scenarios:

- Deploy ITSO Bank to a separate production IBM WebSphere Application Server V6.0. This scenario uses two nodes (Developer node, Application Server node).

  or

- Deploy ITSO Bank to a Rational Application Developer - integrated WebSphere Application Server V6.0 Test Environment.

23.2.2 Install prerequisite software

The application deployment sample requires that you have the software defined in this section installed. Within the example, you can choose between DB2 Universal Database and Cloudscape as your database server.

The sample for the working example environment consists of two nodes:

- Developer node (see Table 23-2 on page 1214 for product mapping)

  This node will be used by the developer to import the sample code and package the application in preparation for deployment.

- Application Server node (see Table 23-3 on page 1214 for product mapping)

  This node will be used as the target server where the enterprise application will be deployed.

Note: For detailed information on installing the required software for the sample, refer to Appendix A, “IBM product installation and configuration tips” on page 1371.

Archive

Table 23-2 Developer node product mapping

Software	Version
Microsoft Windows	XP + Service Pack 1a + Critical fixes and security patches.
IBM Rational Application Developer * IBM WebSphere Application Server V6.0 Test Environment **Note**: Cloudscape is installed by default.	6.0 + Interim Fix 0004
IBM DB2 Universal Database, Express Edition	8.2

Table 23-3 Application Server node product mapping

Software	Version
Microsoft Windows	2000 + Service Pack 4 + Critical fixes and security patches.
IBM WebSphere Application Server	6.0
IBM DB2 Universal Database, Express Edition	8.2

Note: DB2 Universal Database or Cloudscape

The procedures in this chapter provide information on using either DB2 Universal Database or Cloudscape. Cloudscape is installed by default as part of the Rational Application Developer installation (part of WebSphere Application Server V6.0 Test Environment).

23.2.3 Import the sample application Project Interchange file

This section describes how to import the BankEJB.zip Project Interchange file into Rational Application Developer. The BankEJB.zip contains the following projects for the ITSO Bank enterprise application developed in Chapter 15, “Develop Web applications using EJBs” on page 827:

- BankBasicWeb
- BankEJB
- BankEJBClient
- BankEJBEAR

To import the BankEJB.zip Project Interchange file containing the sample code into Rational Application Developer, do the following:

1. Start Rational Application Developer.
2. Open the J2EE perspective Project Explorer view.

3. Select **File** → **Import**.
4. Select **Project Interchange** from the list of import sources and then click **Next**.
5. When the Import Projects dialog appears, click the **Browse** button next to zip file, navigate to and select the **BankEJB.zip** from the c:\6449code\ejb folder, and click **Open**.
6. Click **Select All** to select all projects and then click **Finish**.

23.2.4 Set up the sample database

The ITSO Bank application requires the BANK database. The default procedure uses Cloudscape; however, we provide instructions for DB2 Universal Database as an alternative.

Important: In our scenario, we set up the BANK database on both Developer Node and on the Application Server node, since this will be the target for deploying the application.

Depending on which database server platform you choose, the appropriate table and sample data files can be found as follows:

- Cloudscape:

  ```
  c:\6449code\deploy\database\cloudscape\Bank\Table.ddl
  c:\6449code\deploy\database\cloudscape\Bank\Loaddata.sql
  ```

  or

- DB2 Universal Database:

  ```
  c:\6449code\deploy\database\db2\Bank\Table.ddl
  c:\6449code\deploy\database\db2\Bank\Loaddata.sql
  ```

Attention: The database setup procedure should be performed on the target node such as the Application Server node for our scenario, and the Developer node where Rational Application Developer is installed.

To create the BANK database, create the connections, create tables for the databases, and populate the BANK database with sample data, do the following:

1. Create the BANK database on the Application Server node and Developer node.
   - Cloudscape

     For details refer to “Create Cloudscape database via Cloudscape CView” on page 344.

or

– DB2 Universal Database

  For details refer to “Create DB2 UDB database via a DB2 command window” on page 346.

2. Create the BANK database tables on the Application Server node and Developer node.

   – Create Cloudscape tables on the Application Server node.

     For details refer to “Create Cloudscape database tables via Cloudscape CView” on page 351.

   or

   – Create DB2 Universal Database tables on the Application Server node.

     For details refer to “Create DB2 UDB database tables via a DB2 command window” on page 351.

   or

   – Create Cloudscape tables from Rational Application Developer.

     For details refer to “Create database tables via Rational Application Developer” on page 350.

3. Populate the BANK database tables with sample data on the Application Server node and Developer node.

   – Populate Cloudscape BANK database tables with sample data on the Application Server node.

     For details refer to “Populate the tables via Cloudscape CView” on page 353.

   or

   – Populate DB2 Universal Database BANK database tables with sample data on the Application Server node.

     For details refer to “Populate the tables via a DB2 UDB command window” on page 354.

   or

   – Populate Cloudscape BANK database tables from Rational Application Developer.

     For details refer to “Populate the tables within Rational Application Developer” on page 352.

4. Create the connection to the BANK database on the Developer node from within Rational Application Developer.

   For details refer to “Create a database connection” on page 347.

Archived

Note: This step does not apply for setup on the Application Server node.

This step only applies if you are setting up the database server within Rational Application Developer. In our example, we are preparing the database on the Application Server node, and thus this step is not needed.

5. Add the JDBC driver as a variable to the Java build path on the Developer node from within Rational Application Developer.

   To add a Cloudscape JDBC driver as a classpath variable, do the following.

   Note: For DB2 Universal Database, a DB2_DRIVER_PATH variable already exists. We suggest that you double check that the path to the driver is correct for your DB2 Universal Database installation.

   a. Select **Window** → **Preferences**.

   b. Select **Java** → **Build Path** → **Classpath variable**.

   c. Click **New**.

   d. When the New Variable Entry dialog appears, enter `CLOUDSCAPE_DRIVER_JAR` in the Name field. Click **File** and navigate to the directory of the driver. Select the **db2j.jar** and then click **Open**.

      For example, see Figure 23-14 path:

      ```
      C:/Program Files/IBM/Rational/SDP/6.0/runtimes/base_v6/cloudscape
      /lib/db2j.jar
      ```

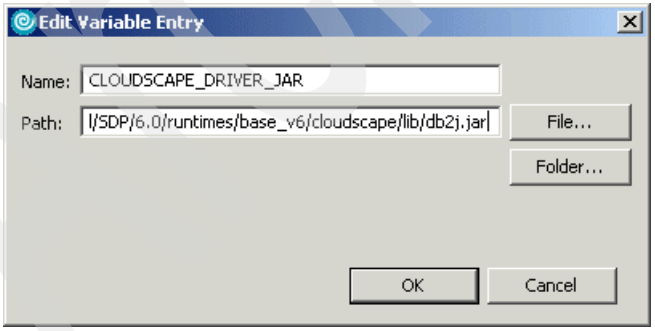


Figure 23-14 CLOUDSCAPE_DRIVER_JAR variable

   e. Click **OK** to add the variable.

   f. Click **OK** to exit the preferences window.

g. Recompile the code. By default, the Workbench preferences are configured to *Build automatically*.

23.3 Package the application for deployment

This section describes the steps in preparation for packaging, as well as how to export the enterprise application from Rational Application Developer to an EAR file, which will be used to deploy on the WebSphere Application Server.

This section includes the following:

- Packaging recommendations.
- Generate the EJB to RDB mapping.
- Customize the deployment descriptors.
- Remove the Enhanced EAR datasource.
- Generate the deploy code.
- Export the EAR.

23.3.1 Packaging recommendations

We have included some basic guidelines to consider when packaging an enterprise application:

- The EJB JAR modules and Web WAR modules comprising an application should be packaged together in the same EAR module.
- When a Web module accesses an EJB module, you should not package the EJB interfaces and stubs in the WAR modules. Thanks to the class loading architecture, EJB stubs and interfaces are visible by default to WAR modules.
- Utility classes used by a single Web module should be placed within its WEB-INF/lib folder.
- Utility classes used by multiple modules within an application should be placed at the root of the EAR file.
- Utility classes used by multiple applications can be placed on a directory referenced via a shared library definition.

23.3.2 Generate the EJB to RDB mapping

This section describes how to generate the EJB to RDB mapping for either Cloudscape or DB2 Universal Database. If you are using Cloudscape as your database server, this section is not required, but included for reference purposes. If using DB2 Universal Database, you will need to complete the DB2 section to generate the EJB to RDB mapping.

Archived

The following procedure describes how to generate the EJB to RDB mapping for the BankEJB EJBs.

1. If you are using Cloudscape, ensure the Cloudscape JDBC driver variable has been defined.

   For details refer to “Set up the sample database” on page 1215.

   For DB2 Universal Database the corresponding variable is already defined.

2. From the Data perspective, right-click the connection that you created in 23.2.4, “Set up the sample database” on page 1215, and select **Copy to Project**.

3. When the Copy to Project dialog appears, click **Browse**, expand and select **BankEJB** → **ejbModule** → **META-INF**, and click **OK**.

4. Check **Use default schema folder for EJB projects**. The folder path is then automatically entered for the appropriate database type.

5. Click **Finish**.

6. When the Confirm folder create message box appears, click **Yes**.

7. Switch to the J2EE perspective Packaging Explorer view.

8. Expand **EJB Projects**, right-click the **BankEJB** project, and select **EJB to RDB Mapping** → **Generate Map**.

9. Select **Use an existing backend folder**, select the backend folder that you created in the previous step (for example, CLOUDSCAPE_V51_1 for Cloudscape, or DB2EXPRESS_V82_1 for DB2 Universal Database), and click **Next**.

Archived

Note: Location of generated mapping

The mapping is generated in the following folders:

- BankEJB\Deployment Descriptor: BankEJB\Maps\Cloudscape V5.1\BANK: CLOUDSCAPE_V51_1
- BankEJB\ejbModule\META-INF\backends\CLOUDSCAPE_V51_1

  The Map.mapxmi file is found in this folder and will be opened as a result of generating the mapping.

The mapping will increment the name of the map by one each time another map is created. In our example, the map is named CLOUDSCAPE_V51_1 since it is the first for Cloudscape. If we added another map for Cloudscape it would be named CLOUDSCAPE_V51_2.

To delete a mapping, select the mapping (for example, BANK: CLOUDSCAPE_V51_1) from the **BankEJB** → **Deployment Descriptor: BankEJB** → **Maps** → **Cloudscape V5.1** folder, right-click, and select **Delete**.

10. Select **Meet-In-The-Middle** and click **Next**.
11. Select **Match by Name** and click **Next**.
12. Click **Finish**.
13. The Mapping editor will open. Follow the instructions in “Completing the EJB-to-RDB mapping” on page 897 to complete the mapping.

23.3.3 Customize the deployment descriptors

Depending on your development process you may customize your deployment descriptors prior to exporting the enterprise application to an EAR file. For example, you may wish to customize the context paths for the target WebSphere Application Server the EAR will be deployed to or change the target database from Cloudscape to DB2 Universal Database.

Alternatively, the deployment descriptors of the exported EAR could be customized using the Application Server Toolkit provided with IBM WebSphere Application Server V6.0.

Note: For more information refer to “Deployment descriptors” on page 1196.

Customize the EJB deployment descriptors for Cloudscape

If you are using Cloudscape, the deployment descriptors for the sample application are already configured to use Cloudscape, thus the section is optional.

To customize the EJB deployment descriptors for use with Cloudscape, do the following:

1. Open the J2EE perspective Packaging Explorer view.
2. Expand **Enterprise Applications**.
3. Customize the BankEJB deployment descriptor.
   a. Expand the **BankEJB** project.
   b. Double-click **Deployment Descriptor: BankEJB** to open in the Deployment Descriptor Editor.
   c. Scroll down near the bottom to the section marked Backend ID.
   d. Select **CLOUDSCAPE_V51_1**.
   e. Save the Deployment Descriptor: BankEJB file.

Customize the EJB deployment descriptors for DB2 UDB

To customize the EJB deployment descriptors for use with DB2 Universal Database, do the following:

1. Open the J2EE perspective Packaging Explorer view.
2. Expand **Enterprise Applications**.
3. Customize the BankEJB deployment descriptor.
   a. Expand **EJB Projects** → **BankEJB**.
   b. Double-click **Deployment Descriptor: BankEJB** to open in the Deployment Descriptor Editor.
   c. Scroll down near the bottom to the section marked Backend ID.
   d. Select **DB2UDBNT_V82_1**.
   e. Save the Deployment Descriptor: BankEJB file.

23.3.4 Remove the Enhanced EAR datasource

The BankEJBEAR application deployment descriptor, contained in the BankEJB.zip Project Interchange that was imported in 23.2.3, “Import the sample application Project Interchange file” on page 1214, contains enhanced EAR settings to configure the datasource. These settings are needed when running the application within Rational Application Developer.

Archived

When deploying the application to a remote WebSphere Application Server system, we chose to configure the datasource using the WebSphere Application Server Administrative Console for demonstration purposes. Since the enhanced EAR datasource configuration overrides the Administrative Console configuration, the enhanced EAR datasource settings must be removed.

To remove the enhanced EAR datasource settings, do the following:

1. Expand **Enterprise Applications** → **BankEJBEAR**.
2. Double-click **Application Deployment Descriptor: BankEJBEAR**.
3. Select the **Deployment** tab.
4. Select the JDBC provider, and click **Remove**.
5. Save the deployment descriptor.

23.3.5 Generate the deploy code

Prior to exporting the EAR, we recommend that you generate the deploy code as follows:

1. Open the J2EE Perspective Project Explorer view.
2. Expand **Enterprise Applications**.
3. Right-click **BankEJBEAR** → **Deploy**.

Note: After generating deployed code for the BankEJB project, you may see a number of warnings about unused imports in the Problems view. These can be safely ignored. If you wish, you can remove these warnings from the Problems view by following the process described in 15.3.4, “Configure the EJB projects” on page 849.

23.3.6 Export the EAR

This section describes how to filter the contents (include, exclude) of an EAR file, and provides a procedure to export an EAR.

Filtering the content of the EAR

When creating assets, such as diagrams, under the JavaSource folder, we found that the assets are automatically copied to the WEB-INF\classes folder. Since the WAR export utility will include all resources located in the WebContent tree, any assets located in the JavaSource tree will be included in the WAR file, regardless of the setting of the Export source files check box. For example, this is likely not a desired behavior for diagrams such as a sequence diagram.

The following procedure describes a procedure to filter the contents of an EAR for export:

1. Right-click the **BankBasicWeb** project, and select **Properties** (where BankBasicWeb is the project you wish to configure).
2. Select **Java Build Path**.
3. Select the **Source** tab.
4. Expand the **BankBasicWeb/JavaSource**, select **Excluded**, and click **Edit**.
5. When the Inclusion and Exclusion Patterns window appears, you will be presented with several options that can be used to filter contents.
   - Exclude by pattern (**Add** button)
   - Exclude individual files (**Add Multiple** button)

   For example, we used the Exclude by pattern option to filter out diagrams as follows:

   a. Click **Add**.

   b. When the Add Exclusion Patterns dialog appears, enter `**/*.dnx`, and click **OK**.

   c. Repeat the previous steps for the following patterns:

      - **/*.iex
      - **/*.ifx
      - **/*.tpx

      Note: Where ** is any sub folder of the JavaSource folder and the JavaSource folder itself. The /* means all files.

6. Click **OK** to close the Inclusion and Exclusion Patterns window.

Export an EAR

To export the enterprise application from Rational Application Developer to an EAR file, do the following:

1. Open the J2EE perspective Project Explorer view.
2. Expand **Enterprise Applications**.
3. Right-click **BankEJBEAR** → **Export** → **EAR file**.
4. When the EAR Export dialog window appears as shown in Figure 23-15, we entered the destination path (for example, c:\ibm\BankEJBEAR.ear) and then click **Finish**.

Archived

Note: Make sure that the three options in the dialog (see Figure 23-15 on page 1224) are unchecked. Because the EAR file will be used in a production system, we do not want to include the source files or the Application Developer metadata.

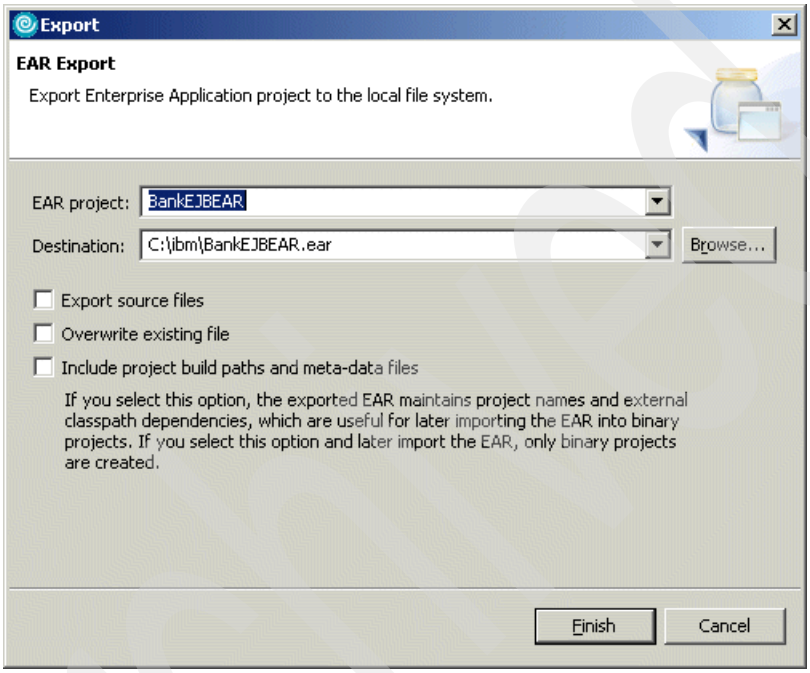


Figure 23-15 Exporting an EAR file

5. Verify that the BankEJBEAR.ear file exists in the c:\ibm folder specified during the export.

23.4 Deploy the enterprise application

Now that the application has been packaged to an EAR file, the enterprise application can be deployed to WebSphere Application Server. This section describes the steps required to configure the target WebSphere Application Server and deploy the BankEJBEAR.ear file.

Note: You can deploy the BankEJBEAR.ear exported in the previous section. Alternatively, the BankEJBEAR.ear found in the c:\6449code\deploy directory of the ITSO sample code can be used.

23.4.1 Configure the data source in WebSphere Application Server

The data source for WebSphere Application Server can be created in several ways including:

- Enhanced EAR

  For details on using the enhanced EAR to configure the JDBC provider, data source, authentication, and database name refer to “WebSphere enhanced EAR” on page 1201.

- wsadmin command line interface

  For details refer to the WebSphere Application Server V6.0 InfoCenter.

- WebSphere Application Server Administrative Console

  Note: Our example uses this method.

The high-level configuration steps are as follows to configure the data source within WebSphere Application Server for the ITSO Bank application sample:

- Start the server1 application server.
- Configure the environment variables.
- Configure J2C authentication data.
- Configure the JDBC Provider.
- Create the datasource.

Start the server1 application server

Ensure that the server1 application server is started on the Application Server node. If it is, not start it as follows:

- Click **Start** → **Programs** → **IBM WebSphere** → **Application Server V6** → **Profiles** → **default** → **Start server** to start the application server named server1.

  or

- Enter the following in a command window:

```
cd \Program Files\IBM\WebSphere\AppServer\profiles\default\bin
startServer.bat server1
```

  or

- Start the Windows service named IBM WebSphere Application Server v6 - was6win1Node01 (where was6win1 is the host name of the system).

Archived

Note: To verify that the server1 application server has started properly, you can look for the message `Server server1 open for e-business` in the SystemOut.log found in the following directory:

```
C:\Program Files\IBM\WebSphere\AppServer\profiles\default\logs\server1
```

Configure the environment variables

Prior to configuring the data source, ensure that the environment variables are defined for the desired database server type. This step does not apply to Cloudscape since we are using the embedded Cloudscape, which already has the variables defined. For example, if you choose to use DB2 Universal Database, you must update the path of the driver for DB2 Universal Database.

1. Launch the WebSphere Administrative Console:
   - Click **Start** → **Programs** → **IBM WebSphere** → **Application Server V6** → **Profiles** → **default** → **Administrative Console**.

   or

   - Enter the following URL in a Web browser:

     `http://<hostname>:9060/ibm/console`
2. Enter a login ID (for example, admin) and then click **Log in**. The user ID can be anything at this point, since WebSphere security is not enabled.
3. Select and expand **Environment** → **WebSphere Variables**.
4. Scroll down the page and click the desired variable and update the path accordingly for your installation.
   - CLOUDSCAPE_JDBC_DRIVER_PATH

     By default this variable is already configured since Cloudscape is installed with WebSphere Application Server.

   or

   - DB2UNIVERSAL_JDBC_DRIVER_PATH

     For example, if you have decided to use DB2 Universal Database, click **DB2UNIVERSAL_JDBC_DRIVER_PATH** to enter the path value to the DB2 Java (db2java.zip). For example, we entered C:\Program Files\IBM\SQLLIB\java. Click **OK**.
5. Click **Save**, and then when prompted click **Save** to Save to Master Configuration.

Archive

Configure J2C authentication data

This section describes how to configure the J2C authentication data (database login and password) for WebSphere Application Server from the WebSphere Administrative Console. This step is required for DB2 UDB and optional for Cloudscape.

If using DB2 UDB, configure the J2C authentication data (database login and password) for WebSphere Application Server from the Administrative Console:

1. Ensure the server1 application server is started.
2. Start the WebSphere Administrative Console.

   `http://<hostname>:9060/ibm/console`
3. Select **Security** → **Global Security**.
4. Under the Authentication properties, expand **JAAS Configuration** → **J2C Authentication data**.
5. Click **New**.
6. Enter the following on the J2C Authentication data page if you are using DB2 UDB and then click **OK**:
   - Alias: `dbuser`
   - User ID: `db2admin`
   - Password: `<password>`
7. Click **Save** and then when prompted click **Save** to Save to Master Configuration.

Configure the JDBC Provider

This section describes how to configure the JDBC Provider for the selected database type. The following procedure demonstrates how to configure the JDBC Provider for Cloudscape, with notes on how to do the equivalent for DB2 Universal Database.

To configure the JDBC Provider from the WebSphere Administrative Console, do the following:

1. Select **Resources** → **JDBC Providers**.
2. Click **New**.
3. From the New JDBC Providers page, do the following and then click **Next**:
   a. Select the Database Type: Select **Cloudscape**.

> **Note:** For DB2 UDB, select **DB2**.

Archived

b. Select the JDBC Provider: Select **Cloudscape JDBC Provider**.

> **Note:** For DB2 UDB, select **DB2 Universal JDBC Driver Provider**.

c. Select the Implementation type: Select **XA data source**.

> **Note:** For DB2 UDB, select **XA data source**.

4. We accepted the default for the JDBC Provider name. Click **OK**.

   The JDBC Provider default names are as follows:

   – Cloudscape: Cloudscape JDBC Provider (XA)
   – DB2: DB2 Universal JDBC Driver Provider (XA)

5. Click **Save** and then when prompted click **Save** to Save to Master Configuration.

Create the datasource

To create the datasource from the WebSphere Administrative Console, do the following:

1. Select **Resources** → **JDBC Providers**.
2. Click the JDBC Provider created in "Configure the JDBC Provider" on page 1227. For example:

   – For Cloudscape click **Cloudscape JDBC Provider (XA)**.

   or

   – For DB2 click **DB2 Universal JDBC Driver Provider (XA)**.

3. Under Additional Properties (right-hand side of page), click **Data sources**.
4. Create the BankDS datasource for the BANK database.

   a. Click **New** from the Data sources page.

   b. Enter the following and then click **OK**:

   - Name: `BankDS`
   - JNDI name: `jdbc/BankDS`
   - Database name (Cloudscape): `c:\databases\BANK`

   > **Note:** For DB2 enter `BANK` for the database name.

   c. For DB2 you must also set the authentication. Select **dbuser** (user defined in "Configure J2C authentication data" on page 1227) from the Component-managed authentication alias drop-down list.

Archived

5. Click **Save**, and then when prompted click **Save** to Save to Master Configuration.
6. Verify the connection by checking **BankDS** and then click **Test connection**.

23.4.2 Deploy the EAR

To deploy the BankEJBEAR.ear exported in 23.3.6, “Export the EAR” on page 1222, to the WebSphere Application Server, do the following:

1. Copy the BankEJBEAR.ear from the Developer node where you exported the EAR from Rational Application Developer to a temporary directory on the Application Server node.
2. Ensure the server1 application server is started.
3. Start the WebSphere Administrative Console by opening the following location in a Web browser:

   `http://<hostname>:9060/ibm/console`
4. Select **Applications** → **Enterprise Applications**.
5. Click **Install**.
6. Enter the following and then click **Next**:
   - Select **Local file system**.
   - Specify path: `c:\temp\BankEJBEAR.ear`
7. We accepted the defaults on the generate bindings page. Click **Next**.
8. When the Install New Application wizard appears, you will need to perform the following sequence of steps to deploy the EAR.
   a. Select installation options. Accept defaults and click **Next**.
   b. Select modules to servers. Accept defaults and click **Next**.
   c. Select current backend ID. Select the currentBankendID for the specific database type then click **Next**:
   - For Cloudscape, select **CLOUDSCAPE_V51_1**.
   - For DB2 UDB, select **DB2UDBNT_V82_1**.

   d. Provide JNDI names for beans. Accept defaults and click **Next**.
   e. Provide default data source mapping for modules containing 2.x entity beans. Accept defaults and click **Next**.
   f. Map data sources for all 2.x CMP beans. Accept defaults and click **Next**.
   g. Map EJB references to beans. Accept defaults and click **Next**.
   h. Map virtual hosts for Web modules. Accept defaults and click **Next**.

Archived

i. Ensure all unprotected 2.x methods have the correct level of protection. Accept defaults and click **Next**.

j. Summary. Click **Finish**.

You should see the a number of messages, concluded by the following message:

```
Application BankEJBEAR installed successfully.
```

9. Click **Save to Master Configuration** and click **Save**.
10. Select **Applications** → **Enterprise Applications**.
11. Check **BankEJBEAR** and then click **Start**.

The status should change to `started`.

23.5 Verify the application

This section provides some basic procedures to verify the ITSO Bank was deployed properly and is working. Table 23-4 on page 1230 lists the sample data for the ITSO Bank loaded via the loaddata.sql in “Set up the sample database” on page 1215.

Table 23-4 ITSO Bank sample data (loaddata.sql)

Customer name	Customer SSN	Account number	Account balance
John Ganci	111-11-1111	001-999000777 (wife) 001-999000888 (kids) 001-999000999 (dad)	$1,234,567.89 $6,543.21 $98.76
George Kroner	222-22-2222	002-999000777 (wife) 002-999000888 (kids) 002-999000999 (dad)	$65,484.23 $87.96 $654.65
Daniel Farrell	333-33-3333	003-999000777 (hush $) 003-999000888 (slush) 003-999000999 (mush)	$9,876.52 $568.79 $21.56
Juha Nevalainen	444-44-4444	004-999000777 (fish) 004-999000888 (cats) 004-999000999 (builder)	$9,876.52 $1,456,456.46 $23,156.46
Ed Gondek	555-55-5555	005-999000777 (food) 005-999000888 (food) 005-999000999 (food)	$65.89 $72,213.41 $897.55
Fabio Ferraz	666-66-6666	006-999000777 (beef) 006-999000888 (more beef) 006-999000999 (barbecue)	$500.00 $100.00 $100,000.00

Archived

Customer name	Customer SSN	Account number	Account balance
Kiriya Keat	777-77-7777	007-999000777 (Vegas) 007-999000888 (beverage) 007-999000999 (ice)	$2,500,000.00 $1,000,000.00 $1.23
Nicolai Nielsen	999-99-9999	009-999000999 (Danny's)	$658,600.42

To verify that the ITSO Bank sample is deployed and working properly, do the following:

1. Ensure that the server1 application server is started.
2. Enter the following URL to access the ITSO Bank application:

   `http://<hostname>:9080/BankBasicWeb/index.html`

Archived

Figure 23-16 ITSO Bank home page

3. You should see something like Figure 23-16. Click **RedBank**.
4. You should see something like Figure 23-16. Enter customer ID `111-11-1111` and then click **Submit**.

Archived

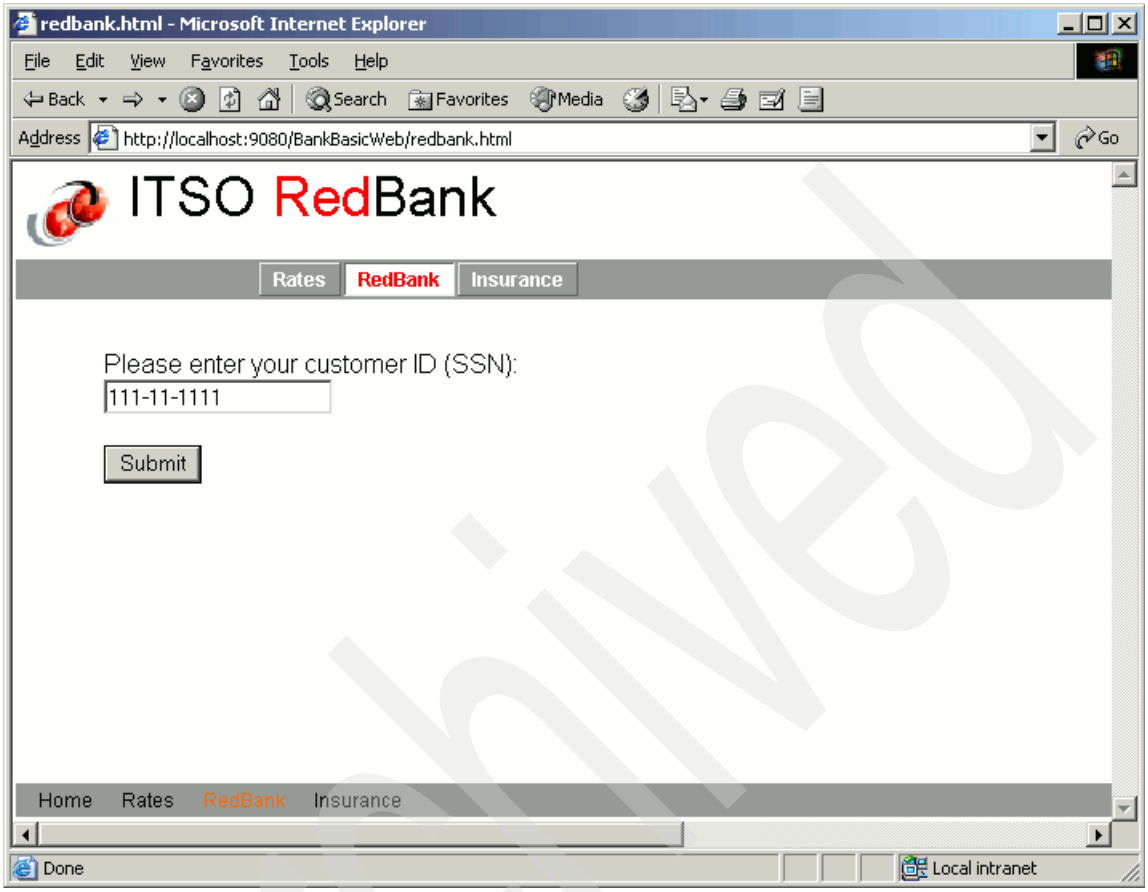


Figure 23-17 ITSO Bank login page

5. When the Accounts page appears as seen in Figure 23-18 on page 1234, click **001-999000888**.

Archived

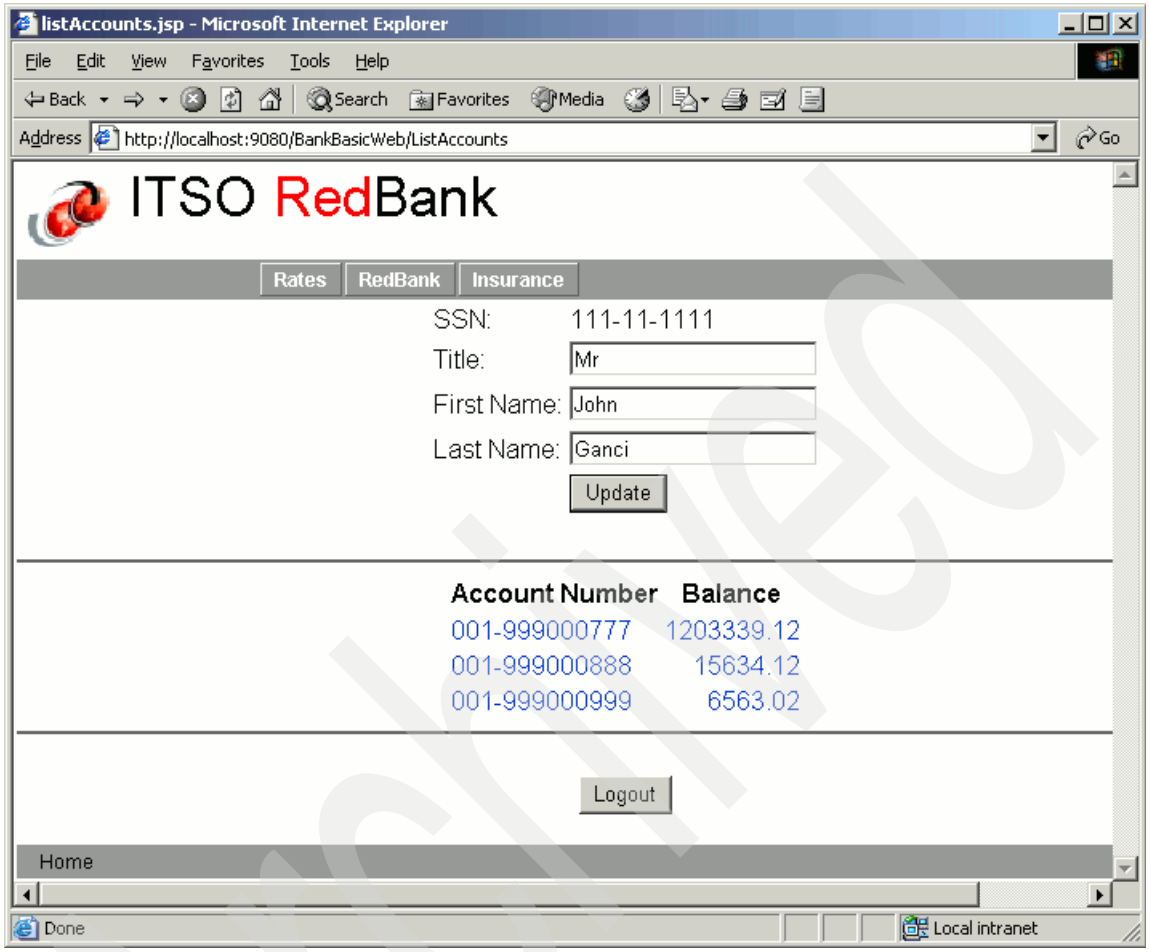


Figure 23-18 ITSO Bank - Accounts page

6. When the Account Maintenance page appears (as seen in Figure 23-19 on page 1235), do the following and then click **Submit**:
   - Select **Transfer**.
   - Amount: 54000

   Note: This transfers 54000 cents, or 540 dollars.

   - Destination account: 004-999000888

Archived

Note: This transfers funds to the account associated with user SSN 444-44-44444.

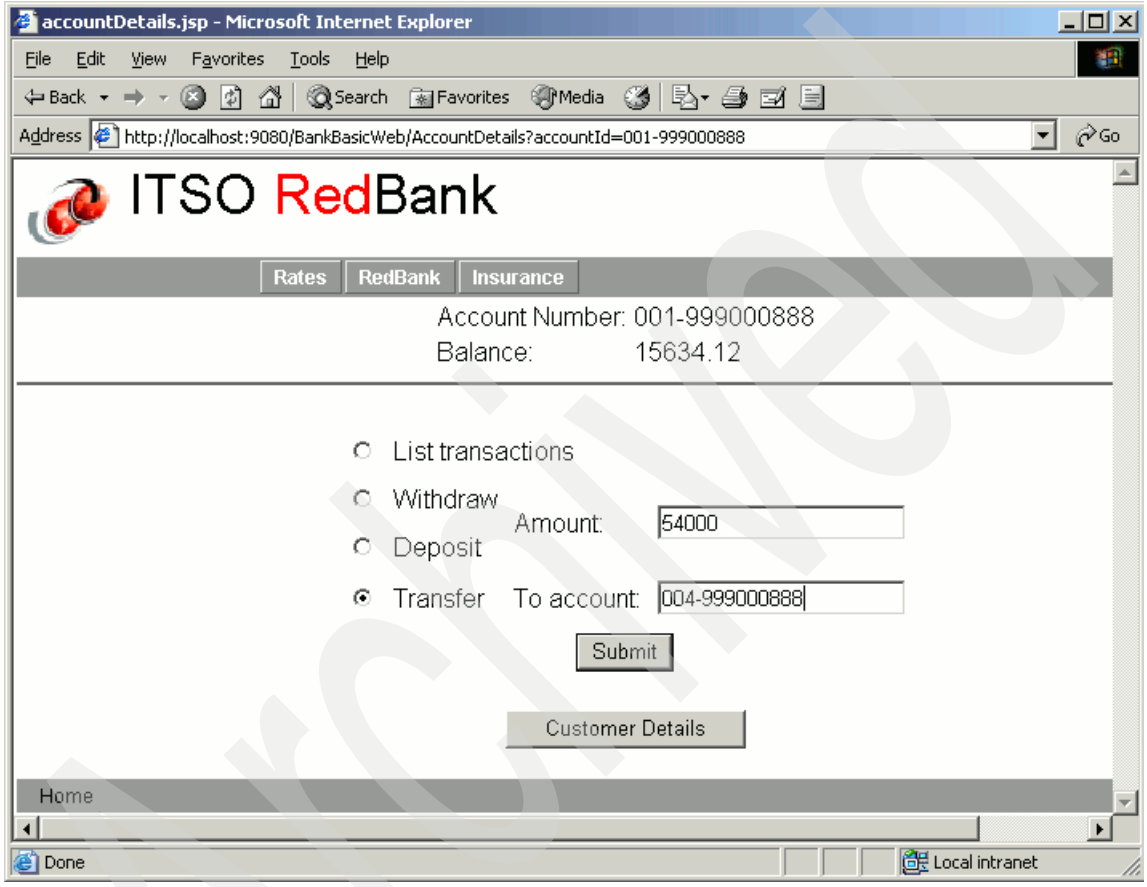


Figure 23-19 ITSO Bank - Transfer

After the transfer you should see a page like Figure 23-20 on page 1236. Notice the account balance has been updated (subtracted transfer amount).

Archive

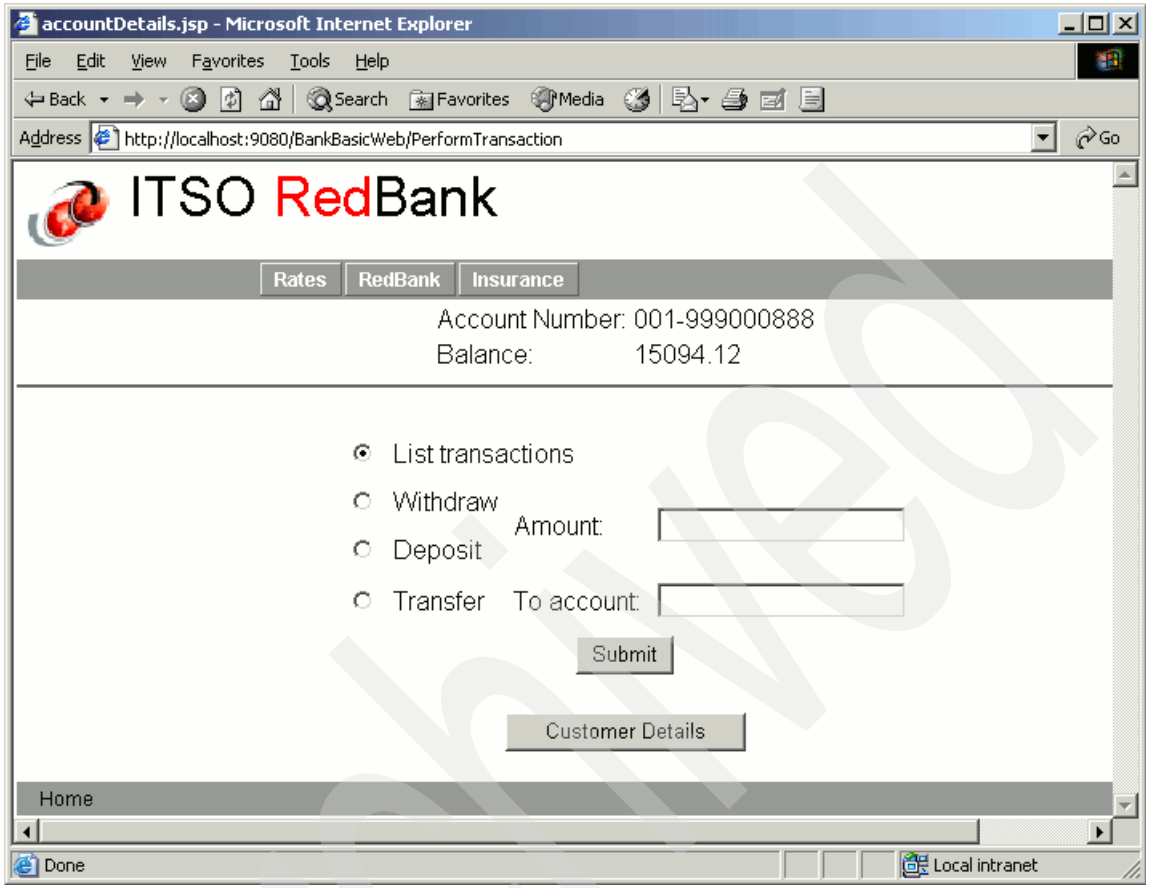


Figure 23-20 ITSO Bank - After transfer

7. Start another Web browser session, and enter the following URL to access the ITSO Bank application to simulate the other customer and account:

   `http://<hostname>:9080/BankBasicWeb/index.html`

8. Enter customer ID `444-44-4444` and then click **Submit**.
9. Click **004-999000888**.
10. Select **List transactions** and click **Submit**.

    You should see that the account has had $540.00 transferred to it. The original account balance was $1,456,456.46 and is now $1,456,996.46.

Archived

24

Profile applications

Profiling is a technique used by developers to detect and isolate application problems such as memory leaks, performance bottlenecks, excessive object creation, and exceeding system resource limits during the development phase.

This chapter introduces the features, architecture, and process for profiling applications using the profiling tooling included with IBM Rational Application Developer V6.0. We have included a working example for code coverage analysis for the Web application developed in Chapter 11, “Develop Web applications using JSPs and servlets” on page 499.

The chapter is organized into the following sections:

- Introduction to profiling
- Prepare for the profiling sample
- Profile the sample application

© Copyright IBM Corp. 2005. All rights reserved.

24.1 Introduction to profiling

Traditionally, performance analysis is performed once an application is getting close to deployment or after it has already been deployed. The profiling tools included with Rational Application Developer allow the developer to move the performance analysis to a much earlier phase in the development cycle, thus providing more time for changes to the application that may effect the architecture of the application before they become critical production environment issues.

The types of problems that IBM Rational Application Developer V6.0 profiling tooling can assist in detecting include:

- Memory leaks
- Performance bottlenecks
- Excessive object creation
- System resource limits

The profiling tools can be used to gather information on applications that are running:

- Inside an application server, such as WebSphere Application Server
- As a standalone Java application
- On the same system as Rational Application Developer
- On a remote WebSphere Application Server with the IBM Rational Agent Controller installed
- In multiple JVMs

24.1.1 Profiling features

Within Rational Application Developer, there are several profiling types. Each profiling type includes predefined profiling sets used to detect and analyze common problems such as memory leaks, performance bottlenecks, and excessive object creation.

This section describes the capabilities of the following profiling types:

- Memory analysis
- Thread analysis
- Execution time analysis
- Code coverage analysis
- Probekit analysis

Archived

Memory analysis

Memory analysis is used to detect memory management problems. In IBM Rational Application Developer V6 there is new support to provide automatic detection of memory leaks. Memory analysis can help developers identify memory leaks as well as excessive object allocation that may cause performance problems.

The memory analysis capability in IBM Rational Application Developer V6.0 has been enhanced with the addition of new views described in Table 24-1, and new capabilities found in Table 24-2.

Table 24-1 New memory analysis views

View name	Description
Leak Candidates view	A tabular view to assist the developer in identifying the most likely objects responsible for leaking memory.
Object Reference Graph view	A graphical view that shows the referential relationship of objects in a graph highlighting the allocation path of leak candidates.

Table 24-2 New memory analysis capabilities

Capability	Description
Memory Leak Analysis - Manual	Allows at the discretion of the developer to capture memory heap dumps after application warm-up; that is, when classes are loaded and initialized.
Memory Leak Analysis - Automatic	Provides timed memory heap dumps at specified intervals during the running of the Java application.

Thread analysis

Thread analysis is used to help identify thread contention and deadlock problems in a Java application. Thread contention issues can cause performance problems, while deadlocks are a correctness issue that can cause a critical runtime issue. The thread analysis capabilities provide analysis data for detecting both of these types of problems.

The thread analysis has been enhanced with the additional view displayed in Table 24-3.

Archived

Table 24-3 New thread analysis views

View name	Description
Thread View	A graphical view of all threads available, their states, and which thread is holding locks. It assists in identifying thread contentions.

Execution time analysis

Execution time analysis is used to detect performance problems by highlighting the most time intensive areas in the code. This type of analysis helps developers identify and remove unused or inefficient coding algorithms.

The execution time analysis has been enhanced with the additional views described in Table 24-4.

Table 24-4 New execution time analysis views

View name	Description
Performance Call Graph View	A graphical view focusing on data that indicates potential performance problems including statistical information.
Method Details View	A view that provides complete performance data for the currently displayed method, including information about its callers and descendants.

Code coverage analysis

Code coverage is a new capability in IBM Rational Application Developer V6.0. It is used to detect areas of code that have not been executed in a particular scenario that is tested. This capability is a useful analysis tool to integrate with component test scenarios and can be used to assist in identifying test cases that may be missing from a particular test suite or code that is redundant.

Note: The profiling working example found in the following sections of this chapter demonstrate the end-to-end process of profiling a Web application for a code coverage:

- 24.2, “Prepare for the profiling sample” on page 1246
- 24.3, “Profile the sample application” on page 1249

New views associated with this capability are shown in Table 24-5.

Table 24-5 New views associated with code coverage

View name	Description
Coverage Navigator	A graphical view that shows coverage levels of packages, classes, and methods and their coverage statistics.
Annotated Source	Includes displays that: ► Have a copy of the code marked indicated tested, untested, and partially tested lines. ► Shows at the class and method level a pie chart with the line coverage statistic.
Coverage Statistics	A tabular view showing the coverage statistics.

Probekit analysis

Probekit analysis provides a new capability that has been introduced into IBM Rational Application Developer V6.0. It is a scriptable byte-code instrumentation (BCI) framework, to assist in profiling runtime problems by inserting Java code fragments into an application. The framework is used to collected detailed runtime information in a customized way.

A probekit file can be contain one or more probes with each containing one or more probe fragments. These probes can be specified when to be executed or on which program they will be used. The probe fragments are a set of Java methods that are merged with standard boilerplate code with a new Java class generated and compiled. The functions generated from the probe fragments appear as static methods of the generated probe class.

The probekit engine called the BCI engine is used to apply probe fragments by inserting the calls into the target programs. The insertion process of the call statements into the target methods is referred to as *instrumentation*. The data items requested by a probe fragment are passed as arguments (for example, method name and arguments). The benefit of this approach is that the probe can be inserted into a large number of methods with small overhead.

Probe fragments can be executed at the following points (see IBM Rational Application Developer V6's online help for a complete list):

- On method entry or exit
- At exception handler time
- Before every executable code when source code is available
- When specific methods are called, not inside the called method

Each of the probe fragments can access the following data:

- Package, class, and method name
- Method signature
- This object
- Arguments
- Return value
- The exception object that caused an exception handler exit to execute, or an exception exit from the method

There are two major types of probes available to the user to create, as described in Table 24-6.

Table 24-6 Types of probes available with Probekit

Type of probe	Description
Method Probe	Probe can be inserted anywhere within the body of a method with the class or jar files containing the target methods instrumented by the BCI engine.
Callsite Probe	Probe is inserted into the body of the method that calls the target method. The class or jar files that call the target instrumented by the BCI engine.

A tutorial on using Probekit is available in IBM Rational Application Developer V6 via the menu by performing the following steps:

1. Start IBM Rational Application Developer V6.
2. Select **Help** → **Tutorials Gallery**.
3. When the Tutorials Gallery window displays, expand **Do and Learn**.
4. Select **Use Probekit to customize Java profiling**.

This tutorial is estimated to take 45 minutes and will assist the developer in creating and using the Probekit in IBM Rational Application Developer V6.

24.1.2 Profiling architecture

The profiling architecture that exists in IBM Rational Application Developer V6.0 is based upon the data collection engine feature provided by the open source Eclipse Hyades project. More detailed information on the Eclipse Hyades project can be found at:

`http://www.eclipse.org/hyades`

Archived

Hyades provides the IBM Rational Agent Controller daemon with a process for enabling client applications to launch host processes and interact with agents that exist within host processes. Figure 24-1 depicts the profiling architecture.

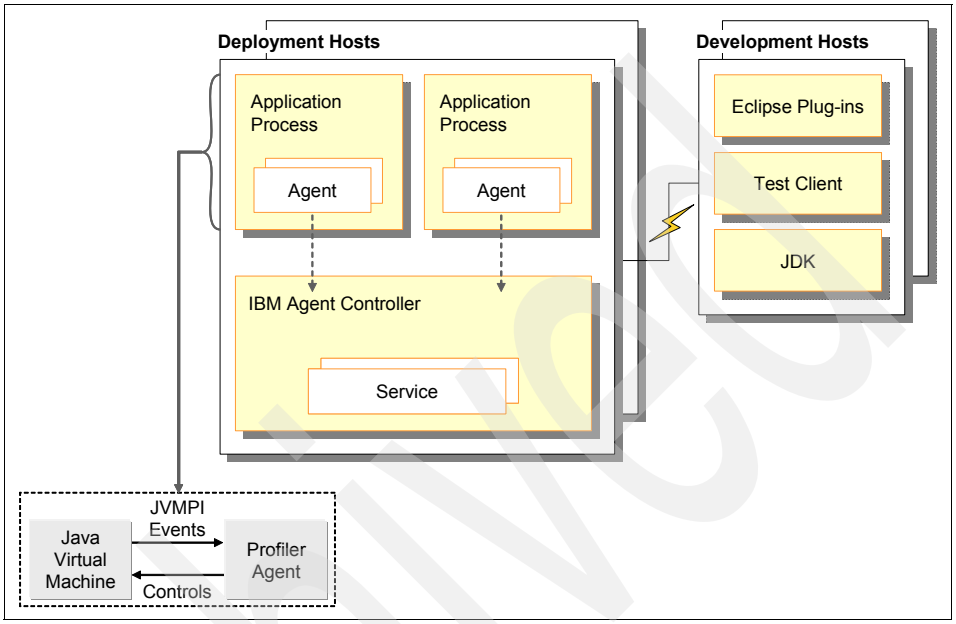


Figure 24-1 Profiling architecture of IBM Rational Application Developer V6.0

The definitions for the profiling architecture are as follows:

- Application process: The process that is executing the application consisting of the Java Virtual Machine (JVM) and the profiling agent.
- Agent: The profiling component installed with the application that provides services to the host process, and more importantly, provides a portal by which application data can be forwarded to attached clients.
- Test Client: A local or remote application that is the destination of host process data that is externalized by an agent. A single client can be attached to many agents at once, but does not always have to be attached to an agent.
- Agent Controller: A daemon process that resides on each deployment host providing the mechanism by which client applications can either launch new host processes, or attach to agents coexisting within existing host processes. The Agent Controller can only interact with host processes on the same node.
- Deployment hosts: The host that an application has been deployed to and is being monitored for the capture of profiling agent.

Archive

- Development hosts: The host that runs an Eclipse-compatible architecture such as IBM Rational Application Developer V6 to receive profiling information and data for analysis.

Each application process shown in Figure 24-1 on page 1243 represents a JVM that is executing a Java application that is being profiled. A profile agent will be attached to each application to collect the appropriate runtime data for a particular type of profiling analysis. This profiling agent is based on the Java Virtual Machine Profiler Interface (JVMPI) architecture. More details on the JVMPI specification can be found at:

`http://java.sun.com/j2se/1.4.2/docs/guide/jvmpi`

The data collected by the agent is then sent to the agent controller, which then forwards this information to IBM Rational Application Developer V6 for analysis and visualization.

There are two types of profiling agents available in IBM Rational Application Developer V6:

- Java Profiling Agent: This agent is based on the JVMPI architecture and is shown in Figure 24-1 on page 1243. This agent is used for the collection of both standalone Java applications as well as applications running on an application server.
- J2EE Request Profiling Agent: This agent resides in an application server process and collects runtime data for J2EE applications by intercepting requests to the EJB or Web containers.

> **Note:** There is only one instance of the J2EE Request Profiling agent that is active in a process that hosts WebSphere Application Server.

24.1.3 Profiling and Logging perspective

After installing Rational Application Developer, you will first need to enable the Profiling and Logging capability (see “Enable the Profiling and Logging capability” on page 1247 for details).

The Profiling and Logging perspective can be accessed by selecting **Window** → **Open Perspective** → **Other** → **Profiling and Logging** and then clicking **OK**. If it is not listed click **Show all**.

There are many supporting views for the Profiling and Logging perspective. When selecting **Window** → **Show View** while in the Profiling and Logging perspective, you will see the following views:

- Console

- Log Navigator
- Log View
- Navigator
- Package Statistics
- Problems
- Profile Monitor
- Properties
- Statistical Data
- Other

Note: If you select Other, you will see a listing of many more views in support of profiling.

24.1.4 Profiling sets

The profiling in IBM Rational Application Developer V6 has been structured around profiling sets and associated views. These profiling sets focus on providing the user with the ability to concentrate on particular types of analysis while profiling an application. Users who require more extensive analysis can create their own unique profiling sets to satisfy their particular needs.

The predefined profiling sets available in IBM Rational Application Developer V6 are outlined in Table 24-7.

Table 24-7 Profiling sets and available views

Profiling set	Options selected	Views available
Memory/Leak Analysis	N/A	► Package Statistics view ► Class Statistics view ► Object References view
	Instance Level information check box selected	► Package Statistics view ► Class Statistics view ► Statistics ► Object References view

Archived

Profiling set	Options selected	Views available
Execution Time Analysis	Show execution Statistics (compressed data)	► Package Statistics view ► Class Statistics view ► Method Statistics view ► Coverage Statistics
	Show execution graphical details	► Package Statistics view ► Class Statistics view ► Method Statistics view ► Coverage Statistics ► Execution Flow view ► UML2 Sequence diagrams views (object, class, thread)
	Show Instance level information, Show execution graphical details	► Package Statistics view ► Class Statistics view ► Method Statistics view ► Instance Statistics ► Coverage Statistics ► Object References view ► Execution Flow view ► UML2 Sequence diagrams views (object, class, thread)
Method Code Coverage	N/A	► Package Statistics view ► Class Statistics view ► Method Statistics view ► Coverage Statistics

Important: For the Object References view you will need to ensure that *Collect Object References* is enabled to view the profiling data using the Object References view.

24.2 Prepare for the profiling sample

This redbook includes a code coverage profiling example. We will use the ITSO RedBank sample Web application developed in Chapter 11, “Develop Web applications using JSPs and servlets” on page 499, as our sample application for profiling.

Complete the following tasks in preparation for the profiling sample:

- Prerequisites hardware and software.
- Enable the Profiling and Logging capability.

- Import the sample project interchange file.
- Publish and run sample application.

24.2.1 Prerequisites hardware and software

This section outlines the hardware and software we used to run the profiling working example.

Prerequisite hardware

When developing the working example, we used the following hardware:

- IBM ThinkCentre™ M50 (8189-E1U)
  - Intel Pentium 4, 2.8 GHz CPU
  - 2 GB RAM

    Note: We found memory profiling to be very resource intensive.

  - 40 GB 7200 RPM IDE HDD

Prerequisite software

The working example requires the following software be installed:

1. IBM Rational Application Developer V6.0

   For details refer to “Rational Application Developer installation” on page 1372.
2. IBM Rational Application Developer V6.0 - Interim Fix 0004

   For details refer to “Rational Application Developer Product Updater - Interim Fix 0004” on page 1380.
3. IBM Rational Agent Controller

   For details refer to “IBM Rational Agent Controller V6 installation” on page 1382.

24.2.2 Enable the Profiling and Logging capability

To enable the Profiling and Logging capability in the preferences, do the following:

1. Select **Window** → **Preferences**.
2. Expand **Workbench** → **Capabilities**.
3. Expand **Tester**, and check **Profiling and Logging** (as seen in Figure 24-2), and then click **OK**.

Archived

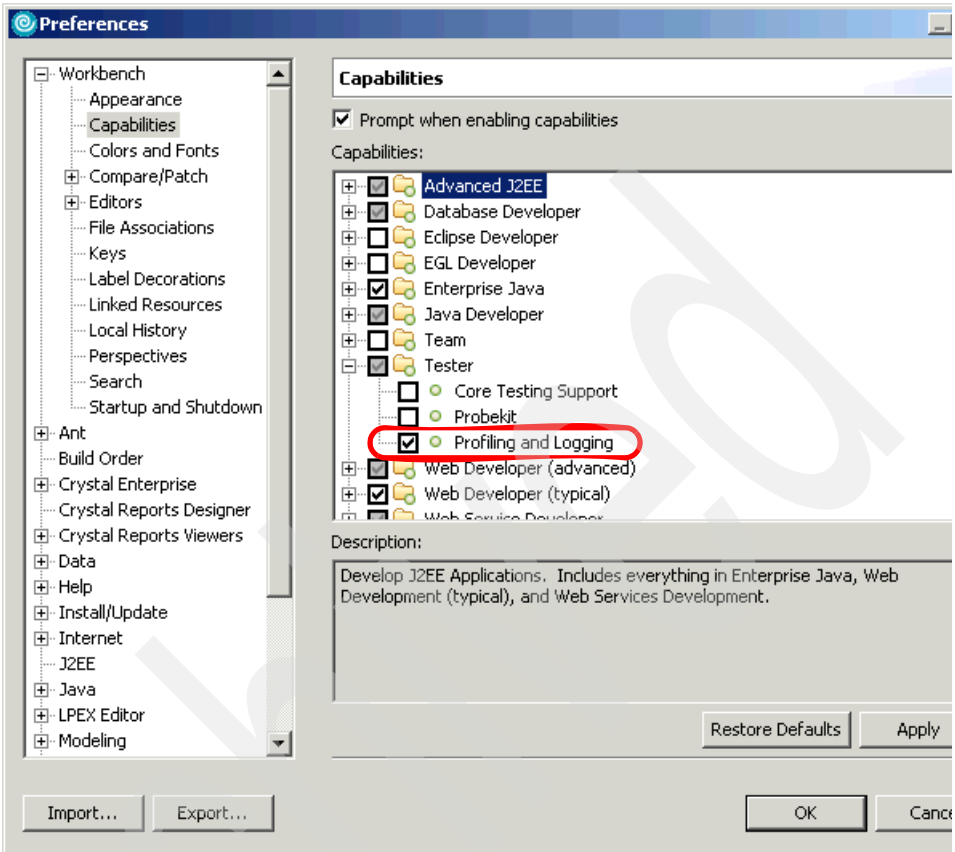


Figure 24-2 Enable Profiling and Logging capability

Note: If you want to use the Probekit, you will need to enable this capability by checking the Probekit check box, as seen in Figure 24-2.

24.2.3 Import the sample project interchange file

To import the ITSO RedBank JSP and Servlet Web application Project Interchange file (BankBasicWeb.zip), do the following:

1. Open the Web perspective Project Explorer view.
2. Right-click **Dynamic Web Projects**, and select **Import** → **Import**.
3. When the Import dialog appears, select **Project Interchange** and then click **Next**.

4. In the Import Projects screen, browse to the c:\6449code\web folder and select **BankBasicWeb.zip**. Click **Open**.
5. Check the **BankBasicWeb** and **BankBasicWebEAR** projects, and click **Finish**.

24.2.4 Publish and run sample application

The sample application needs to be published to the WebSphere Application Server, prior to running the application server in profile mode.

To publish and run the sample application on the WebSphere Application Server V6.0 test server, do the following:

1. Open the Web perspective.
2. Expand **Dynamic Web Projects** → **BankBasicWeb** → **WebContent**.
3. Right-click **index.html**, and select **Run** → **Run on Server**.
4. When the Server Selection dialog appears, select **Choose and existing server**, select **WebSphere Application Server v6.0**, and click **Finish**.

   This operation will start the server and publish the application to the server.

24.3 Profile the sample application

This section demonstrates the code coverage analysis feature of profiling for the ITSO RedBank Web application.

24.3.1 Start server in profile mode

To start the WebSphere Application Server V6.0 in profile mode, do the following:

1. Ensure the IBM Rational Agent Controller Windows service is started.
2. Open the Web perspective.
3. In the Servers view, do the following to start the server in profile mode:
   - If the WebSphere Application Server V6.0 is started, right-click and select **Restart** → **Profile**.
   - If the WebSphere Application Server V6.0 is not started, right-click the server and select **Profile**.
4. After the server has started, the Profile on server dialog should appear presenting the user with a listing of agents to attach to.

Expand **PID** as seen in Figure 24-3, and select the > to move the PID to the Selected agents pane.

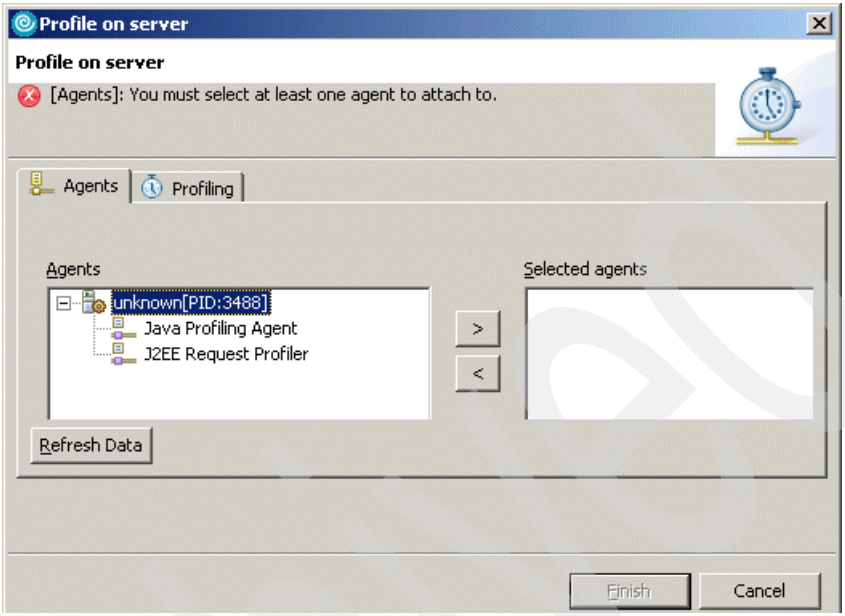


Figure 24-3 Identify Agents to attach to

5. Click the **Profiling** tab as seen in Figure 24-4 on page 1251.

Archive

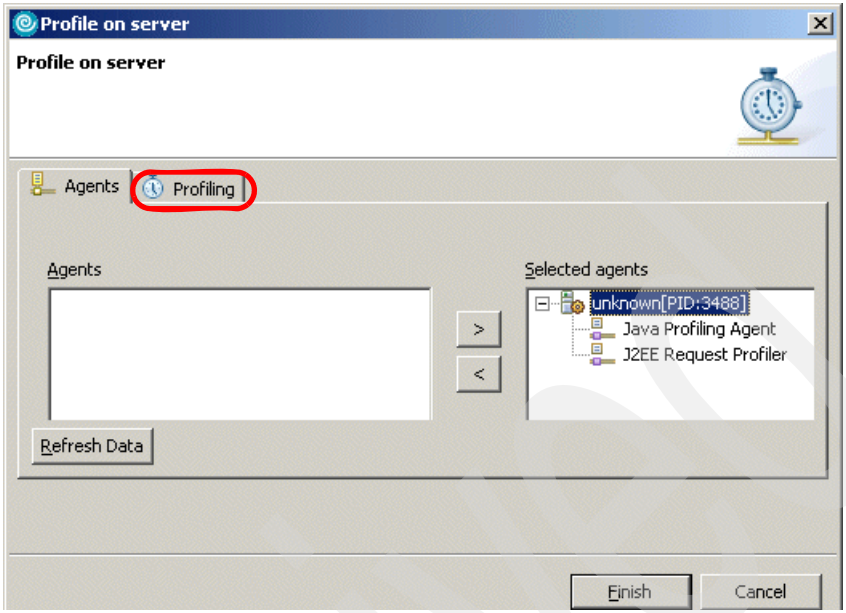


Figure 24-4 Addition of PID to the list of selected agents

6. When the Overview tab appears, check **Method Coverage Information** (as seen in Figure 24-5 on page 1252), and click **Finish**.

   When you click **Finish**, the Profiling and Logging perspective will appear.

> **Note:** In most cases, the pre-defined profile sets will be adequate for testing; however, you may want to create a new profile set that chains together the several existing profile sets.
>
> To add a new profile set, click **Add** as seen in Figure 24-5 on page 1252, and enter the name of the new profile set. Click **Edit** and check the desired existing profiling sets that you wish to include in your new profiling set.

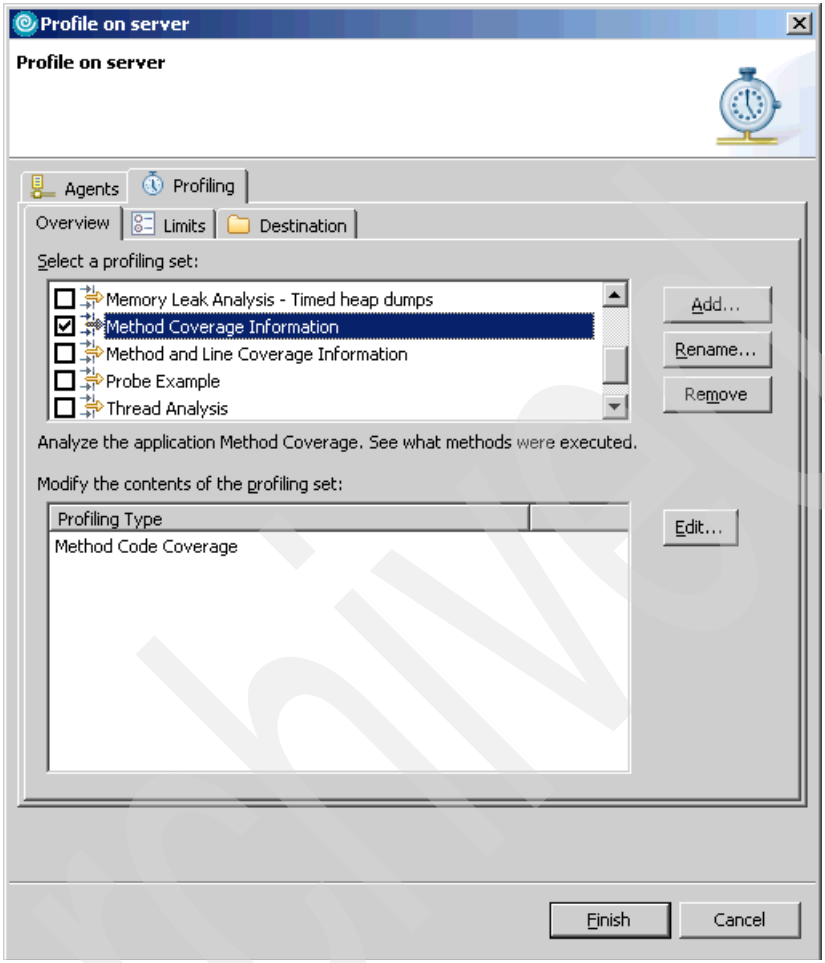


Figure 24-5 Profile on server

7. When the Profiling tips dialog appears as seen in Figure 24-6, click **OK**.

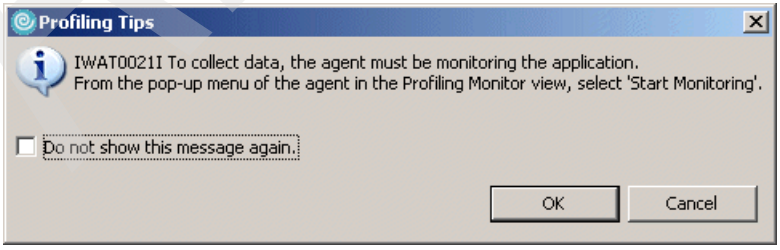


Figure 24-6 Profiling Tips

24.3.2 Collect profile information

To collect the profile information, do the following:

1. In the Profiling Monitor view of the Profiling and Logging perspective, select the two attached profiler process, as shown in Figure 24-7. Right-click and select **Start Monitoring**.

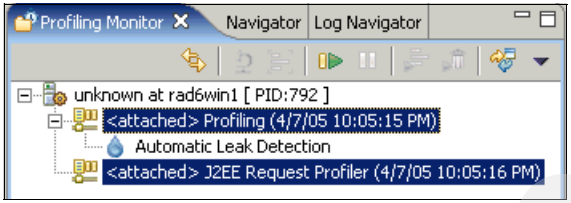


Figure 24-7 Selecting active process in the Profile Monitor view

2. Run a test of a part of the application.

> **Note:** This step requires that you have published the project to the server as described in 24.2.4, “Publish and run sample application” on page 1249.

   a. Enter the following URL to launch the application in a Web browser:

      `http://localhost:9080/BankBasicWeb/index.htm`

   b. Click **RedBank**.

   c. When prompted, enter `111-11-1111` in the customer SSN field and then click **Submit**.

We have now run the part of the application that we want to obtain code coverage information for. In the next section, we will analyze this information.

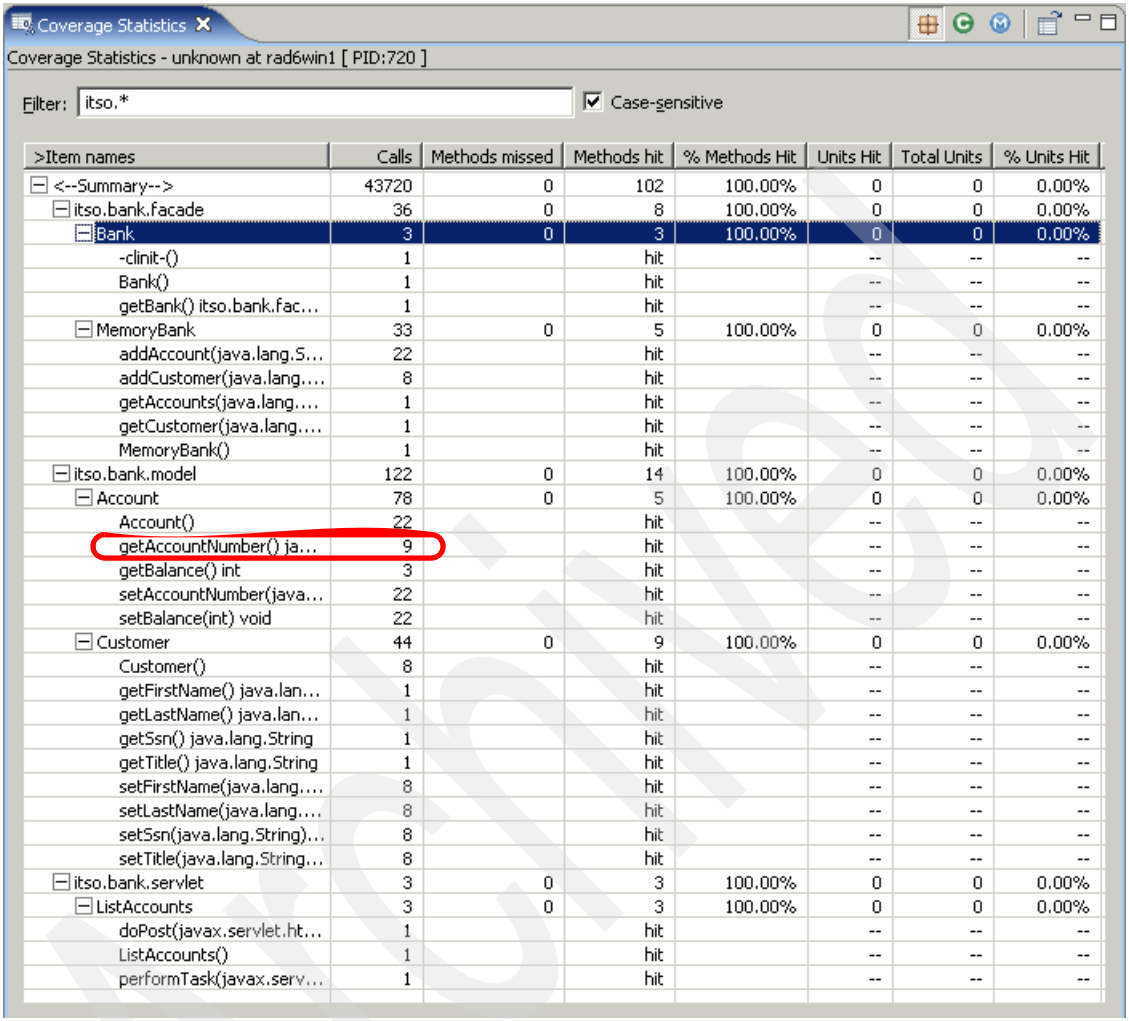
24.3.3 Analysis of code coverage information

To analyze the code coverage information for our sample application, do the following:

1. In the Profiling Monitor view, right-click **Method Code Coverage**, and select **Open With** → **Coverage Statistics**.
2. The Coverage Statistics view will open. Enter `itso.*` in the Filter field and press Enter.

   The Coverage Statistics view should look similar to Figure 24-8 on page 1254.

Archived



>Item names	Calls	Methods missed	Methods hit	% Methods Hit	Units Hit	Total Units	% Units Hit
<--Summary-->	43720	0	102	100.00%	0	0	0.00%
itso.bank.facade	36	0	8	100.00%	0	0	0.00%
Bank	3	0	3	100.00%	0	0	0.00%
-clinit-()	1		hit		--	--	--
Bank()	1		hit		--	--	--
getBank() itso.bank.fac...	1		hit		--	--	--
MemoryBank	33	0	5	100.00%	0	0	0.00%
addAccount(java.lang.S...	22		hit		--	--	--
addCustomer(java.lang....	8		hit		--	--	--
getAccounts(java.lang....	1		hit		--	--	--
getCustomer(java.lang....	1		hit		--	--	--
MemoryBank()	1		hit		--	--	--
itso.bank.model	122	0	14	100.00%	0	0	0.00%
Account	78	0	5	100.00%	0	0	0.00%
Account()	22		hit		--	--	--
getAccountNumber() ja...	9		hit		--	--	--
getBalance() int	3		hit		--	--	--
setAccountNumber(java...	22		hit		--	--	--
setBalance(int) void	22		hit		--	--	--
Customer	44	0	9	100.00%	0	0	0.00%
Customer()	8		hit		--	--	--
getFirstName() java.lan...	1		hit		--	--	--
getLastName() java.lan...	1		hit		--	--	--
getSsn() java.lang.String	1		hit		--	--	--
getTitle() java.lang.String	1		hit		--	--	--
setFirstName(java.lang....	8		hit		--	--	--
setLastName(java.lang....	8		hit		--	--	--
setSsn(java.lang.String)...	8		hit		--	--	--
setTitle(java.lang.String...	8		hit		--	--	--
itso.bank.servlet	3	0	3	100.00%	0	0	0.00%
ListAccounts	3	0	3	100.00%	0	0	0.00%
doPost(javax.servlet.ht...	1		hit		--	--	--
ListAccounts()	1		hit		--	--	--
performTask(javax.serv...	1		hit		--	--	--

Figure 24-8 Coverage Statistics view

The Coverage Statistics view shows which methods have been called during the tests performed on the sample application. For example, the Account.getAccountNumber method was executed nine times. This type of information is useful for function testing.

You can examine the source code for a class or method by right-clicking it in the Coverage Statistics view and selecting **Open Source**.

Part 5

Team development

Archived

© Copyright IBM Corp. 2005. All rights reserved.

Archived

25

Rational ClearCase integration

This chapter introduces the features and terminology of IBM Rational ClearCase with respect to Rational Application Developer. In addition, we provide a basic scenario with two developers working in parallel on a common Web project using Rational Application Developer and ClearCase. The focus of the example is to demonstrate the tooling and integration Rational Application Developer with ClearCase.

The chapter is organized into the following topics:

- Introduction to IBM Rational ClearCase
- Integration scenario overview
- ClearCase setup for a new project
- Development scenario

© Copyright IBM Corp. 2005. All rights reserved.

25.1 Introduction to IBM Rational ClearCase

This section provides an introduction to the IBM Rational ClearCase product as well as basic information on the IBM Rational Application Developer V6.0 integration features for ClearCase.

We have organized the section into the following topics:

- IBM Rational Application Developer ClearCase overview
- IBM Rational ClearCase terminology
- IBM Rational ClearCase LT installation
- New V6 integration features for ClearCase
- IBM Rational Application Developer integration for ClearCase

25.1.1 IBM Rational Application Developer ClearCase overview

Rational ClearCase is a software configuration management (SCM) product that helps to automate the tasks required to write, release, and maintain software code.

Rational ClearCase offers the essential functions of version control, workspace management, process configuration, and build management. By automating many of the necessary and error-prone tasks associated with software development, Rational ClearCase helps teams of all sizes build high-quality software.

ClearCase incorporates Unified Change Management (UCM), Rational's best practices process for managing change at the activity level, and control for workflow.

UCM can be applied to projects *out-of-the-box*, enabling teams to get up and running quickly. However, it can be replaced with any other process that you already have in place at your site.

ClearCase provides support for parallel development. With *automatic branching* and *merge* support, it enables multiple developers to design, code, test, and enhance software from a common, integrated code base.

Snapshot views support a *disconnected use* model for working away from the office. All changes since the last snapshot are automatically updated once you are connected again.

IBM offers two versions of the Rational ClearCase product:

- ClearCase
- ClearCase LT

Archived

ClearCase LT is a *light* version for support of small teams that do not need the full functionality of the complete ClearCase product (distributed servers, database replication, advanced build management, transparent file access). A product license for IBM Rational ClearCase LT is included with IBM Rational Application Developer V6.0.

For the *full-sized* IBM Rational ClearCase, the product also provides an add-on *MultiSite* feature.

Note: Rational Application Developer includes entitlement for ClearCase LT.

More information on the IBM Rational ClearCase products can be found at:

http://www.ibm.com/software/awdtools/clearcase

25.1.2 IBM Rational ClearCase terminology

We have outlined some of the key terminology used in IBM Rational ClearCase:

- Activity: A unit of work performed by an individual. In UCM an activity tracks a change set, that is, a list of versions of files created to perform the work (for example, Developer 1 fixing problem report #123). When you work on an activity, all versions you create are associated with that activity.
- Component: A set of related directory and file elements. Typically, elements that make up a component are developed, integrated, and released together. In Application Developer, a component contains one or more projects.
- Baseline: A version of a project.
- Development stream: Each developer's own working area.
- Integration stream: A shared working area for the team, containing the versions of the components that are available to all developers.
- Deliver stream: The act of making a developer's development stream available to the integration stream, publishing a developer's work.
- Rebase: The act of retrieving a project to work on locally, or to synchronize your development stream with what is available in the integration stream.
- Check in and check out: A file that is to be edited must be checked out. This lets other developers know that the file is opened by another developer. Once a developer completes any edits on a file, it must be checked back in before making the files available to others.
- VOB (versioned object base): The permanent data repository where ClearCase stores files, directories, and metadata.
- View: A selection of resources in a VOB, a window to the VOB data.

25.1.3 IBM Rational ClearCase LT installation

IBM Rational Application Developer V6.0 includes a license for IBM Rational ClearCase LT.

Information on obtaining a copy of IBM Rational ClearCase LT can be found in TechNote-Installing_CCLT.html of the IBM Rational Application Developer V6.0 CD1. This technote describes how to obtain ClearCase LT, as well as the need to apply a patch to work properly with IBM Rational Application Developer V6.0.

For information on the IBM Rational ClearCase LT installation, refer to “IBM Rational ClearCase LT installation” on page 1385.

Figure 25-1 shows a typical development environment when using ClearCase with two developers.

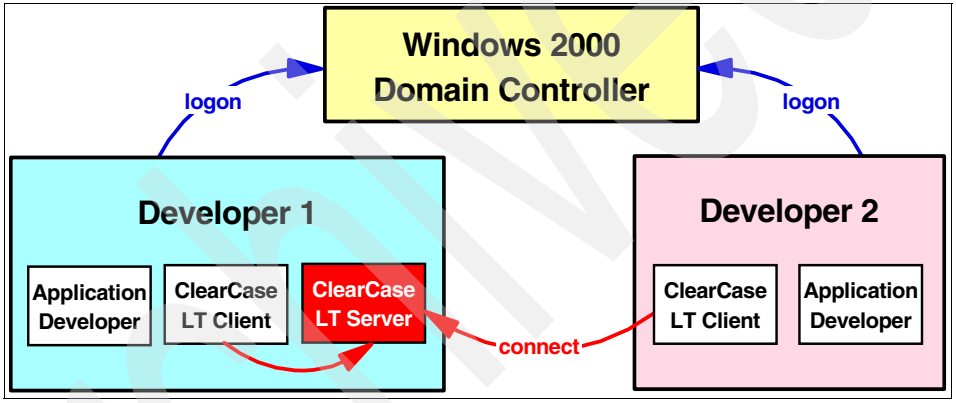


Figure 25-1 Sample ClearCase LT setup between two developers

In this chapter, we are simulating this flow by using two workspaces on the same machine.

25.1.4 IBM Rational Application Developer integration for ClearCase

IBM Rational Application Developer V6.0 includes integration support for IBM Rational ClearCase (as well as ClearCase LT), allowing easy access to ClearCase features. The ClearCase adapter is automatically activated when you start Rational Application Developer the next time after ClearCase installation.

New V6 integration features for ClearCase

Integration with Rational ClearCase via the source control management (SCM) adaptor has been enhanced in IBM Rational Application Developer V6.0 as follows:

- Dynamic views are now fully supported. Note though that ClearCase LT supports snapshot views only and the dynamic view capability is provided in ClearCase or ClearCase MultiSite®.
- Improved compare and merge, including integration with Eclipse compare/merge framework.
- Improved support for working in disconnected mode.
- Better workspace/view management.
- Better usage guidance and documentation.

ClearCase help in Rational Application Developer

Rational Application Developer provides two links to documentation for using ClearCase.

To access the help documentation in Rational Developer, select **Help** → **Help Contents** to open the new help window. Select **Developing applications in a team environment** from the Contents view.

To access the Rational ClearCase Help system, click **ClearCase Help** or click the ClearCase Help icon ( ). This icon becomes active when you connect to ClearCase.

ClearCase preferences

Ensure that the ClearCase SCM Adapter capability is enabled. Refer to 25.3.1, “Enable Team capability in preferences” on page 1264, for details.

There are a number of ClearCase preferences that you can modify by selecting **Window** → **Preferences** → **Team** → **ClearCase SCM Adapter** (see Figure 25-2 on page 1262).

We recommend that you check out files from Rational ClearCase before you edit them. However, if you edit a file that is under ClearCase control but is not checked out, Rational Developer can automatically check it out for you if you select the **Automatically checkout** option for the setting “Checked-in files are saved by an internal editor.”

You can specify if you want to automatically connect to ClearCase when you start Application Developer. Select **Automatically connect to ClearCase on startup** to enable this option.

Archived

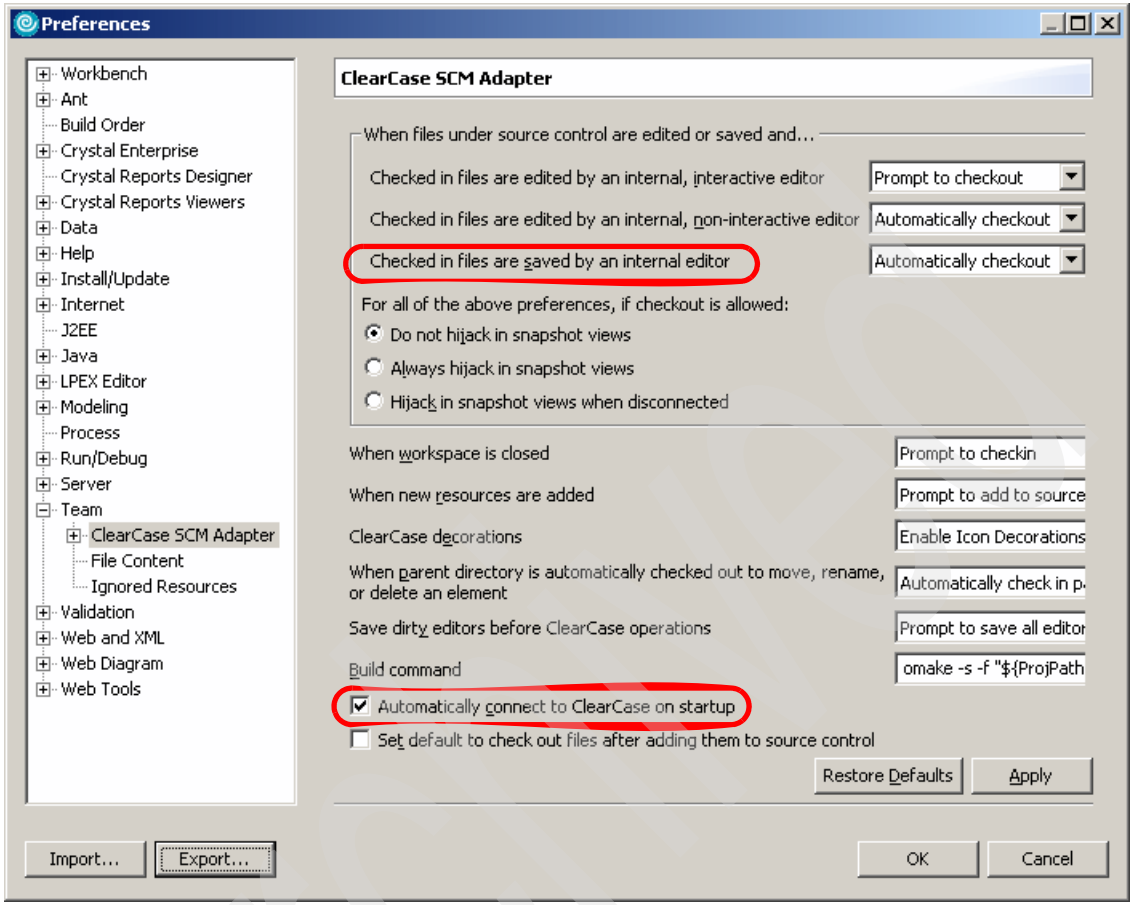


Figure 25-2 ClearCase preferences dialog

If you click **Advanced Options** and then select the **Operations** tab, there is a preference for generating backup copies when you cancel a checkout. This is enabled by default, and specifies that ClearCase generates backup copies when you perform an undo checkout operation. The backup files will have a .keep extension.

Note: You must be connected to ClearCase for the Advanced Options button to be active.

The ClearCase online help in Rational Developer contains a detailed description of each option of the preferences page.

25.2 Integration scenario overview

This section describes a scenario with two developers, developer 1 and 2, working on a Web project called ITSO_ProGuide_UCM. Developer 1 is assigned the role of project integrator and is responsible for setting up the environment.

Table 25-1 Development activities

Developer 1 activities	Developer 2 activities
► Creates a new ClearCase project, ITSO_Project ► Joins the ClearCase project by creating views ► Creates a new Web project, ITSO_ProGuide_UCM ► Moves the project under ClearCase source control ► Adds a servlet, ServletA ► Delivers the work to the integration stream ► Makes a baseline	
	► Joins the ClearCase project by creating views ► Rebases his view to match the integration stream ► Imports the project into Rational Application Developer workspace
► Checks out ServletA ► Makes changes ► Checks in the servlet	► Checks out ServletA ► Makes changes ► Checks in the servlet ► Delivers the work to the integration stream
► Delivers the work to the integration stream ► Resolves conflicts by using the Merge Tool	

The setup of this scenario and its flow is shown in Figure 25-3 on page 1264. ClearCase terminology is used for the tasks.

Archived

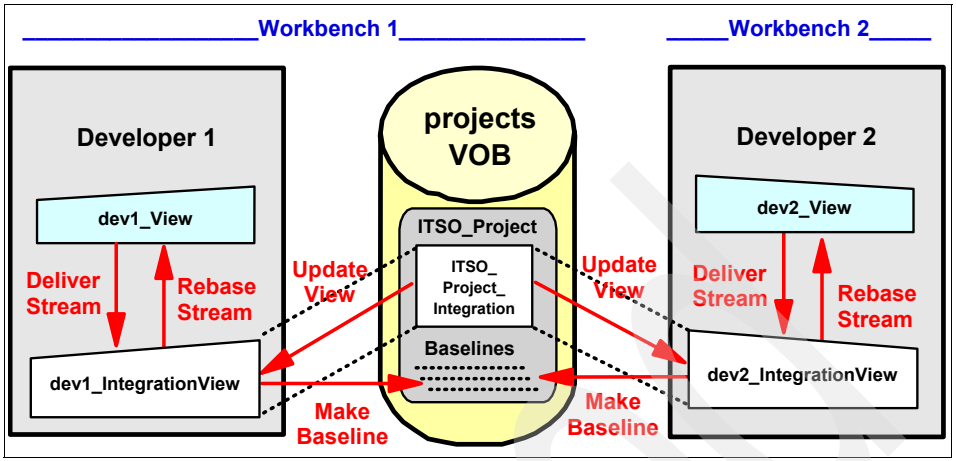


Figure 25-3 Scenario setup

Note that the integration view is like a window to the integration stream. The integration stream should be reserved for only that code that has passed the developer's inspection and is sharable to the entire team.

When finished with the changes, the developer delivers his or her development stream back to the integration stream. A project integrator (or any of the developers) can then make a new baseline freezing the latest code changes.

25.3 ClearCase setup for a new project

In this example scenario, developer 1 and developer 2 work on the same machine by switching workspaces to simulate multiple users. Alternatively, you could have developer 1 working on a machine where a ClearCase LT Server is installed, and other developers working on machines where only the ClearCase LT Client is installed. The steps are basically the same in both cases.

25.3.1 Enable Team capability in preferences

By default, the ClearCase SCM Adapter capability is disabled. To enable, do the following:

1. Select **Window** → **Preferences**.
2. Select **Workbench** → **Capabilities**.
3. Expand **Team**.
4. Check **ClearCase SCM Adapter** and **Core Team Support**.

25.3.2 Create new ClearCase project

Developer 1 creates a new project under ClearCase control as follows:

1. As developer 1, select **Start** → **Programs** → **Rational Software** → **Rational ClearCase** → **Project Explorer**.
2. Right-click **projects** and select **New** → **Project** from the context menu.
3. When the New Project Wizard appear, enter `ITSO_Project` in the Project name field as seen in Figure 25-4, and then click **Next**.

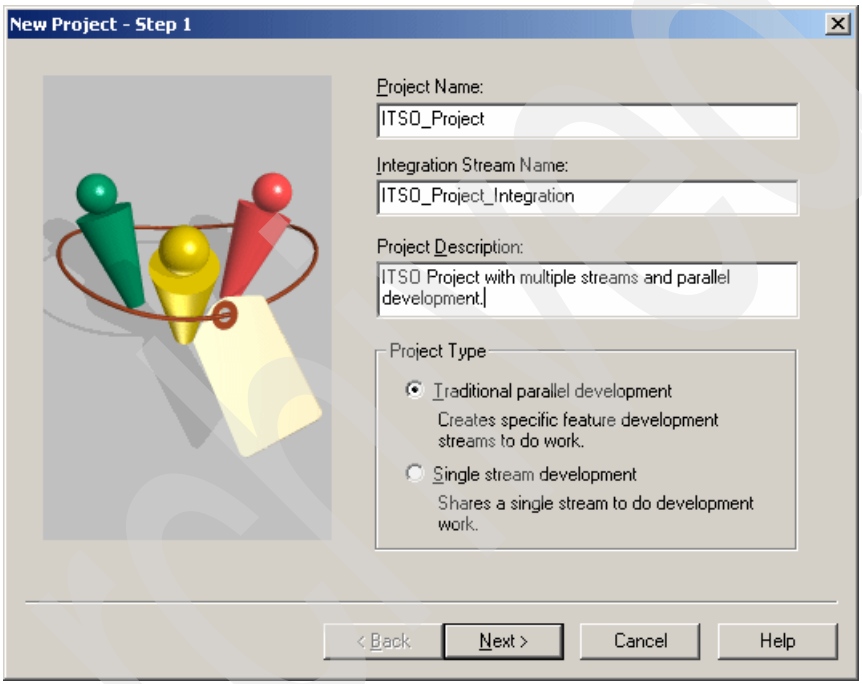


Figure 25-4 Creating new project: Step 1

4. In the Step 2 dialog ensure **No** is selected, as seen in Figure 25-5 on page 1266, and then click **Next**.

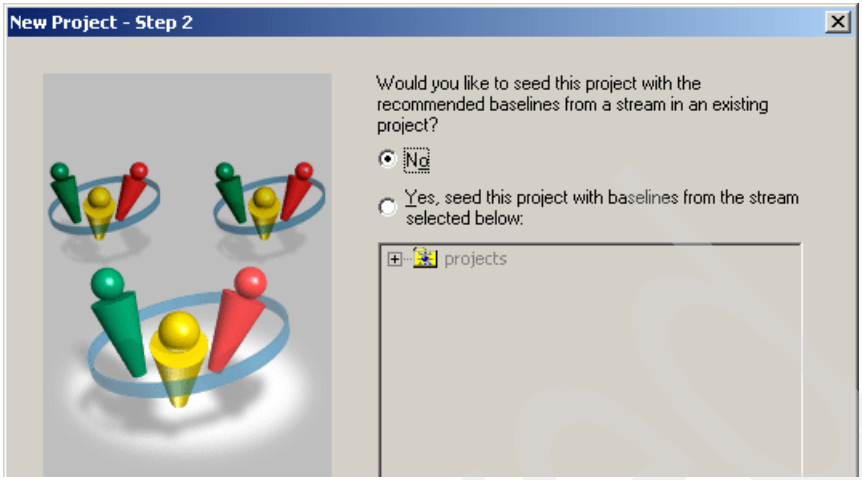


Figure 25-5 Creating new project: Step 2

5. In the Step 3 dialog (see Figure 25-6 on page 1267) click **Add**:
   a. In the Add Baseline dialog, first click **Change >>** and select **all streams.**
   b. Select the component **InitialComponent** from the drop-down list and select **InitialComponent_INITIAL** under Baselines.
   c. Click **OK**.
6. Click **Next**.

Archive

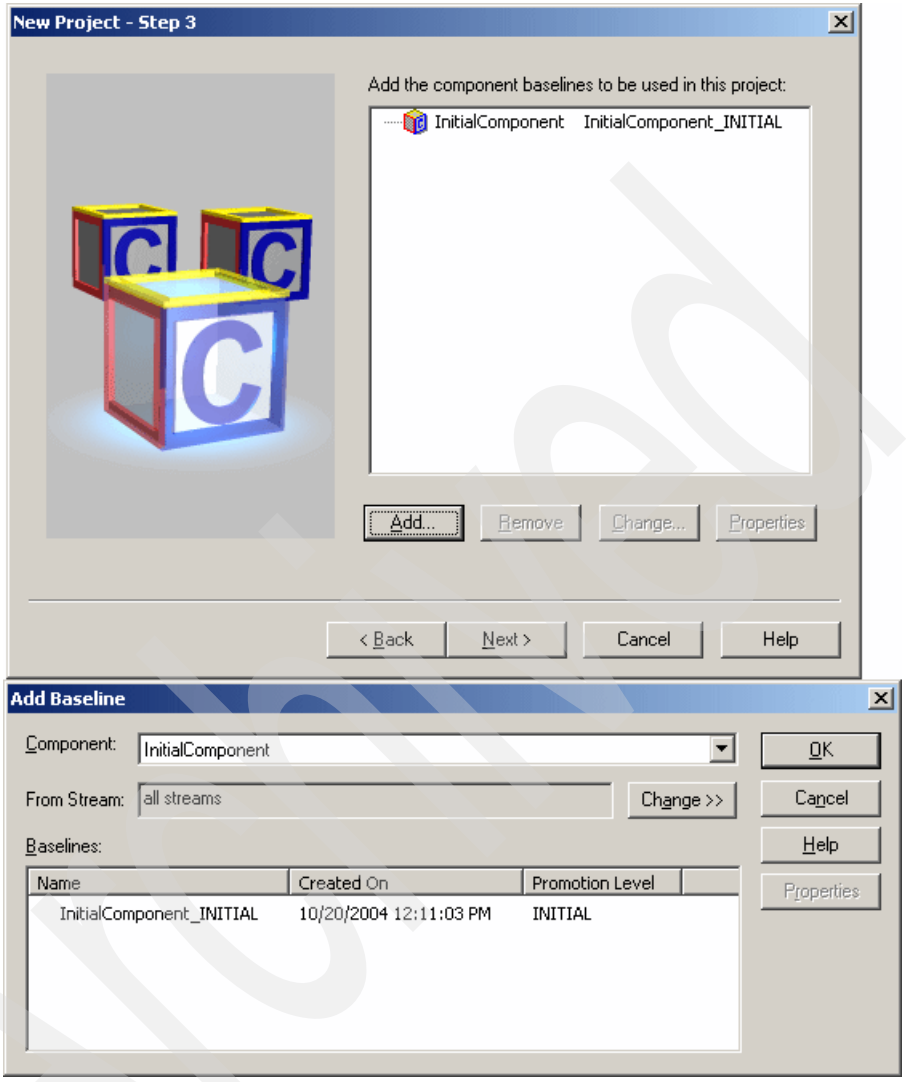


Figure 25-6 Creating new project: Step 3

7. In the Step 4 dialog (Figure 25-7 on page 1268) select **InitialComponent** under Make the following components modifiable. Leave the other values as their defaults. Click **Next**.

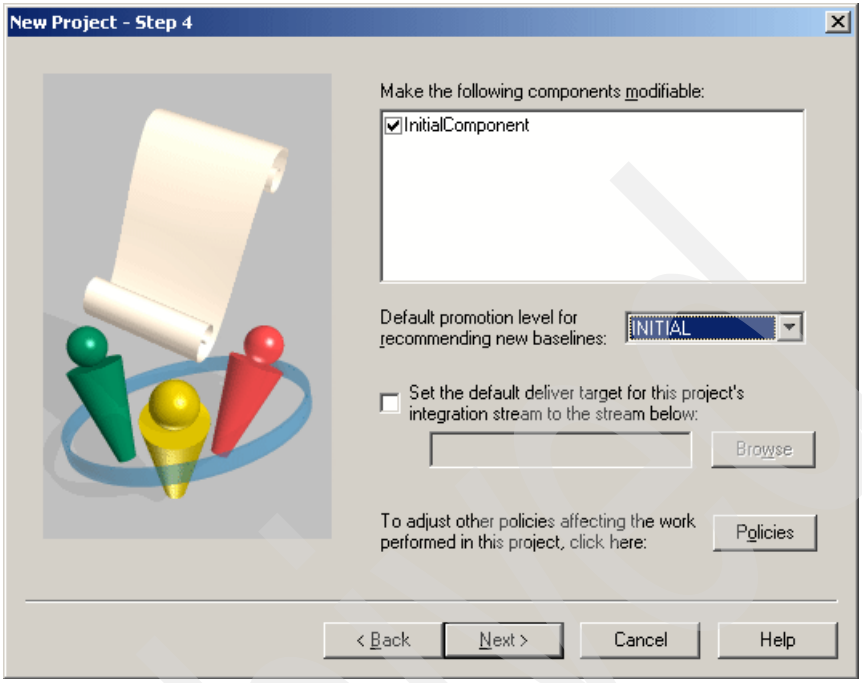


Figure 25-7 Creating new project: Step 4

8. In the Step 5 dialog, select **No** and click **Finish**.
9. Click **OK** on the confirmation dialog.

   ClearCase now creates the project and it shows up in the Project Explorer.

25.3.3 Join a ClearCase project

The next step for developer 1 is to join the project and create a new Web project in Rational Application Developer.

1. Start Rational Application Developer and enter C:\dev1_workspace as the name of the workspace for developer 1 (Figure 25-8 on page 1269).

   Do not enable the default workspace option since we will be switching workspaces several times during this exercise. Click **OK**.

   – If the Workspace Launcher dialog did not appear, change your Rational Application Developer startup setting under **Window** → **Preferences** → **Workbench** → **Startup** and shut down. Make sure the **Prompt for workspace on startup** option is selected.

   – If the Auto Launch Configuration Change Alert dialog appears during startup, informing of an update to your auto launch settings, click **Yes**.

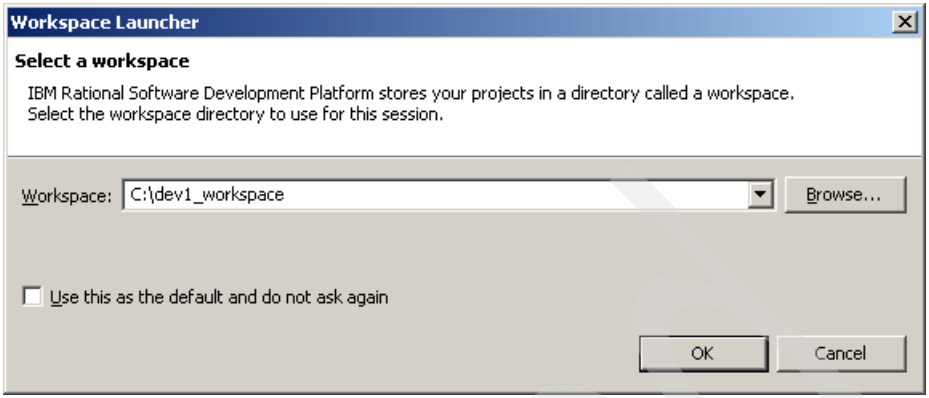


Figure 25-8 Selecting workspace for developer 1

2. Close or minimize the Welcome view.
3. Select **ClearCase** → **Connect to Rational ClearCase** (unless you specified to automatically connect to ClearCase when Rational Application Developer starts). You may also click the ClearCase Connect icon ( ).
4. Select **ClearCase** → **Create New View**.
5. In the View Creation Wizard (Figure 25-9 on page 1270), select **Yes** to indicate that we are working on a ClearCase project. Expand **projects** and select the **ITSO_Project**. Click **Next**.

Archive

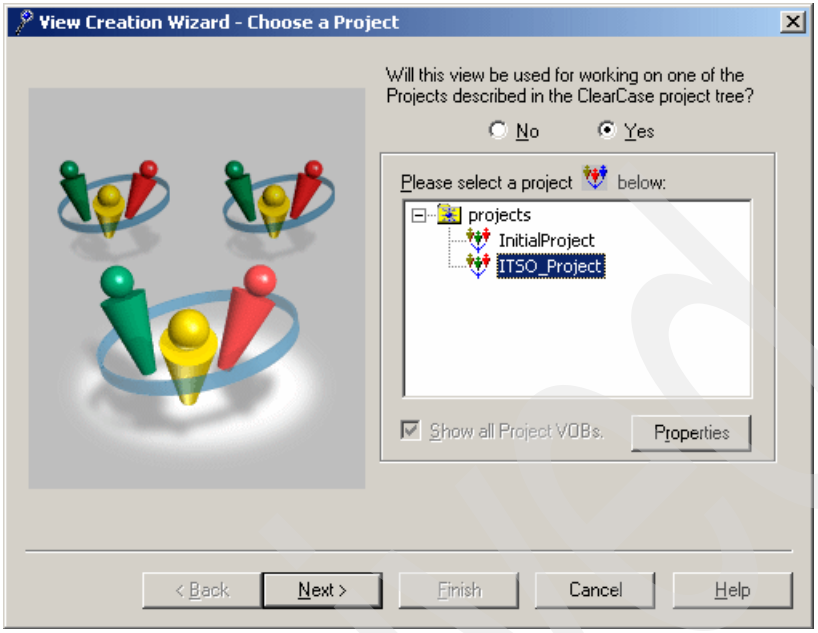


Figure 25-9 Creating a new view

6. In the Create a Development Stream dialog (Figure 25-10 on page 1271) enter `dev1_view` as the development stream name and make sure the integration stream name is ITSO_Project_Integration. Click **Next**.

Archive

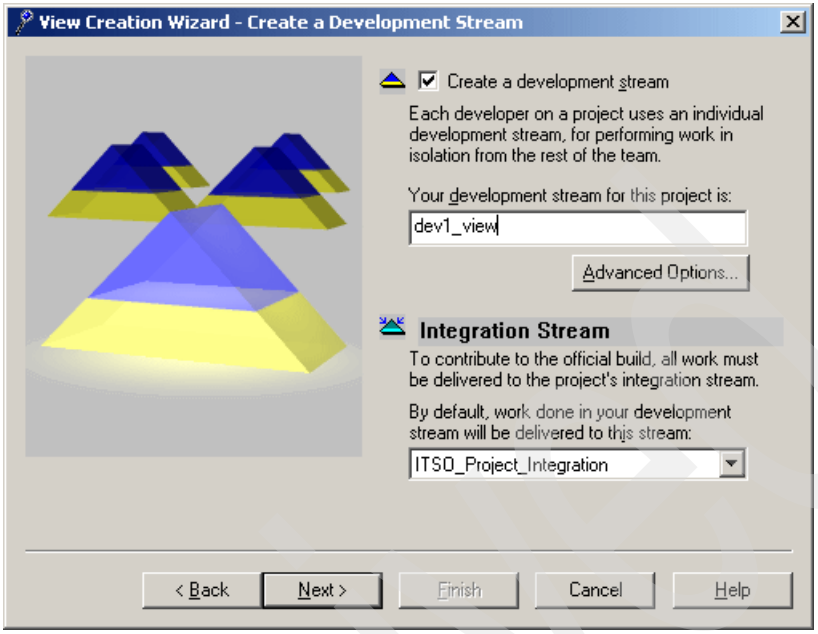


Figure 25-10 Creating a development stream

7. In the Choose Location for a Snapshot View (Development View) dialog (Figure 25-11 on page 1272) change the location to `C:\ITSO\dev1_view`. Click **Next** and click **Yes** on the confirmation dialog.

Archive

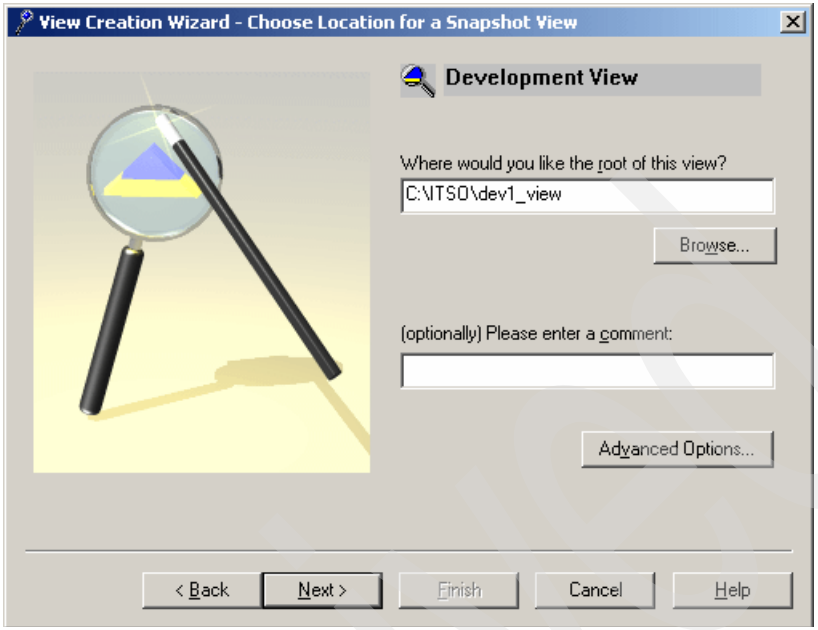


Figure 25-11 Specifying location for a development view

8. In the Choose Location for a Snapshot View (Integration View) dialog (Figure 25-12 on page 1273) change the location to `C:\ITSO\Integration_view`. Click **Next**.

Archive

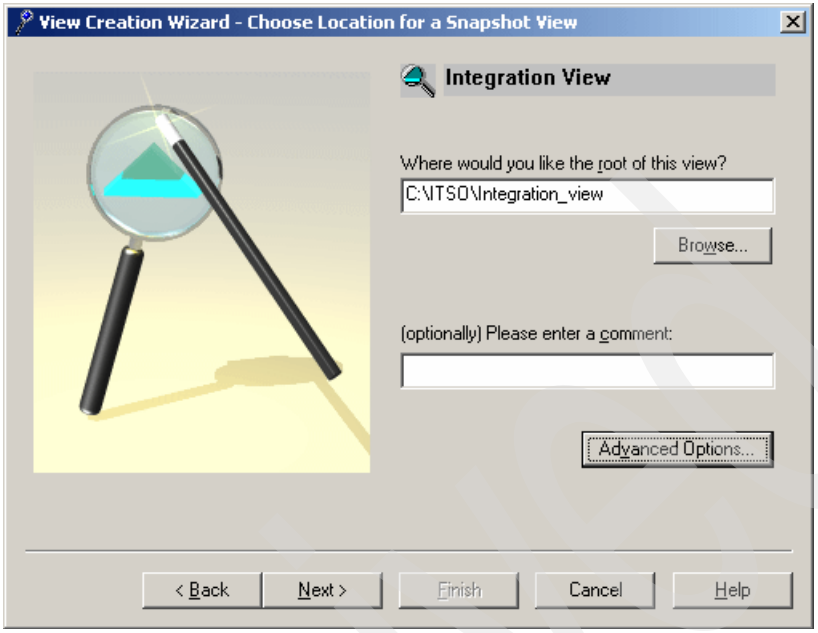


Figure 25-12 Specifying location for the integration view

9. In the Choose Components dialog, leave the **Initial_Component** selected. Click **Finish**.
10. In the Confirmation dialog (Figure 25-13) click **OK**.

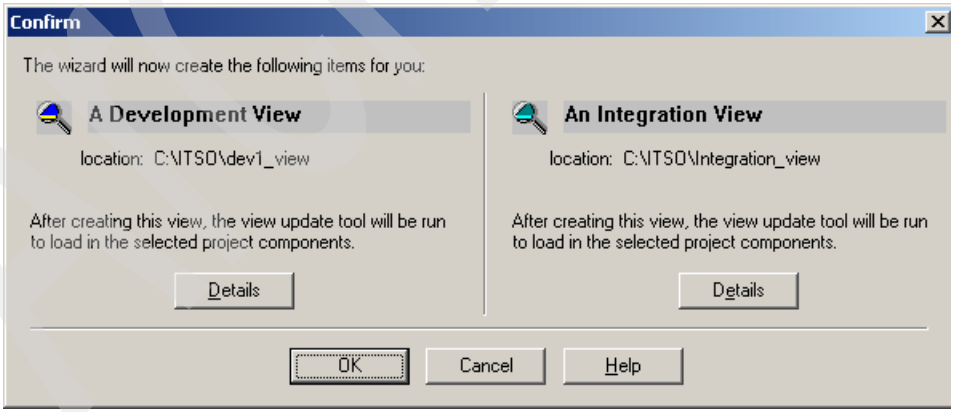


Figure 25-13 View creation confirmation dialog

11. The View Creation Status dialog is displayed after the views have been created. Click **OK**.

25.3.4 Create a Web project

Developer 1 has now created the necessary views and joined the project. The next task is to start actual development and add a new project to ClearCase control. We will create a dynamic Web project containing just one servlet for this purpose.

1. As developer 1, select **File** → **New** → **Dynamic Web Project**. In the New Dynamic Web Project dialog (Figure 25-14), enter `ITSO_ProGuide_UCM` as a project name and click **Finish**.
2. After the project is created, click **Yes** to switch to the Web perspective.

   You should now have a new project under the Dynamic Web Projects folder and the related EAR project under the Enterprise Applications folder.

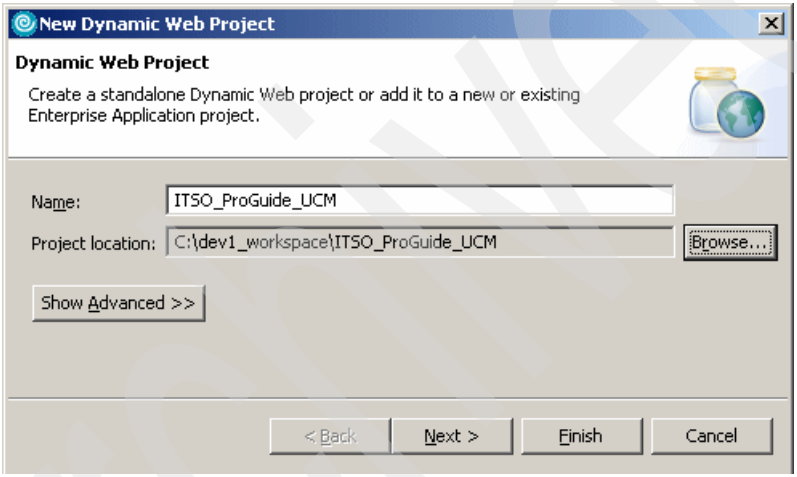


Figure 25-14 Creating dynamic Web project

25.3.5 Add a project to ClearCase source control

To add the new project under ClearCase control, do the following:

1. As developer 1, right-click the **ITSO_ProGuide_UCM** project in the Project Explorer view and select **Team** → **Share Project...** from the context menu.
2. In the Share Project dialog, select **ClearCase SCM Adapter** and click **Next**.
3. In the Move Project Into a ClearCase VOB dialog (Figure 25-15 on page 1275) click **Browse** and select the **C:\ITSO\dev1_view\sources\InitialComponent** directory. Click **OK** and then click **Finish**.

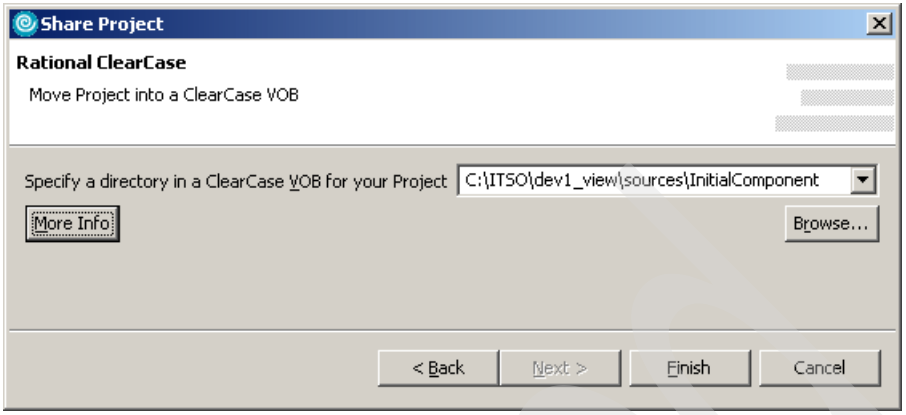


Figure 25-15 Moving project into ClearCase

4. In the Add Elements to Source Control dialog (Figure 25-16), leave all items selected and deselect **Keep checked out**. Click **OK**.

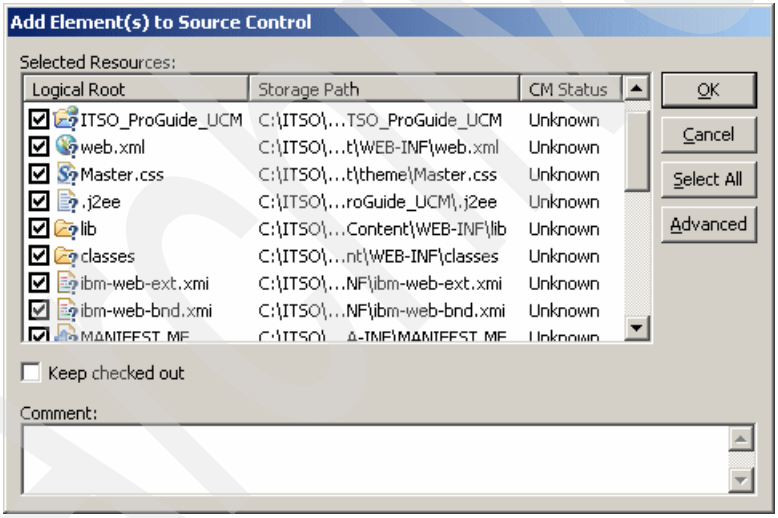


Figure 25-16 Specifying elements to add to source control

5. In the Select Activity dialog (Figure 25-16) select **New...** and enter `Developer 1 adds project to source control`. Click **OK** to return, and then click **OK** to continue. The Web project is now added to ClearCase source control.

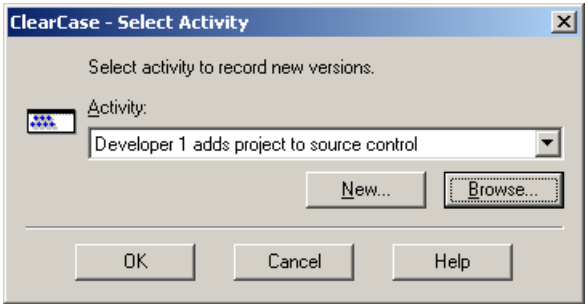


Figure 25-17 Specifying activity

In Figure 25-18 you can see that the icons belonging to the Web project now have a blue background, and the project name has a view name attached to it. This indicates that the resources are under ClearCase source control.

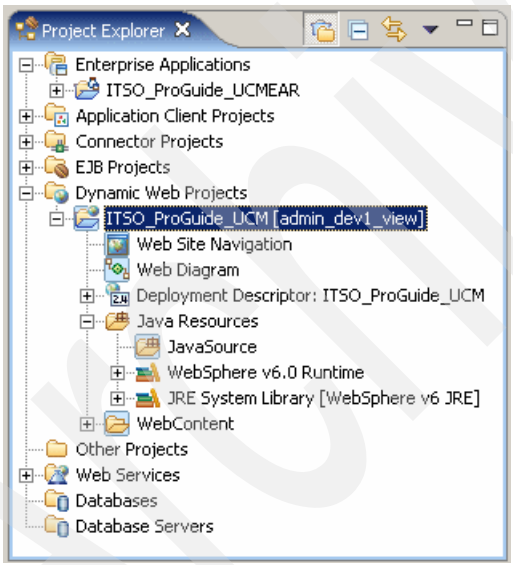


Figure 25-18 Resources under ClearCase source control

Project contents have been *moved* from the workspace to developer 1 view. (C:\dev1_workspace Æ C:\ITSO\dev1_view\sources\InitialComponent in our example). Open Windows Explorer and verify this result.

Note that the ITSO_ProGuide_UCMEAR project still resides under the dev1 workspace.

6. Under Enterprise Applications, select the **ITSO_ProGuide_UCMEAR** project and add this project to source control using the same method (do not create a

new activity, use the activity created when adding the Web project to source control).

Both projects are now under ClearCase source control.

25.4 Development scenario

To show how to work with ClearCase we use a simple scenario where two developers work in parallel on a common project. We will use a servlet to illustrate handling a situation when adding an element (the servlet) generates a potential update to some other element(s), like the deployment descriptor.

25.4.1 Developer 1 adds a servlet

Developer 1 defines the servlet in the Web project as follows:

1. As developer 1, right-click **ITSO_ProGuide_UCM** and select **New** → **Other** → **Web** → **Servlet** from the context menu.
2. Enter `ServletA` as the servlet name and click **Next**. Enter `itso.ucm` as the Java package name and click **Finish**.
3. Adding a servlet to the project causes an update to the deployment descriptor (web.xml) and the binding and extension information (the ibm-web-bnd.xmi and ibm-web-ext.xmi files). You are prompted to check them out (Figure 25-19). On the Check Out Elements dialog, ensure all three files are selected and click **OK**.

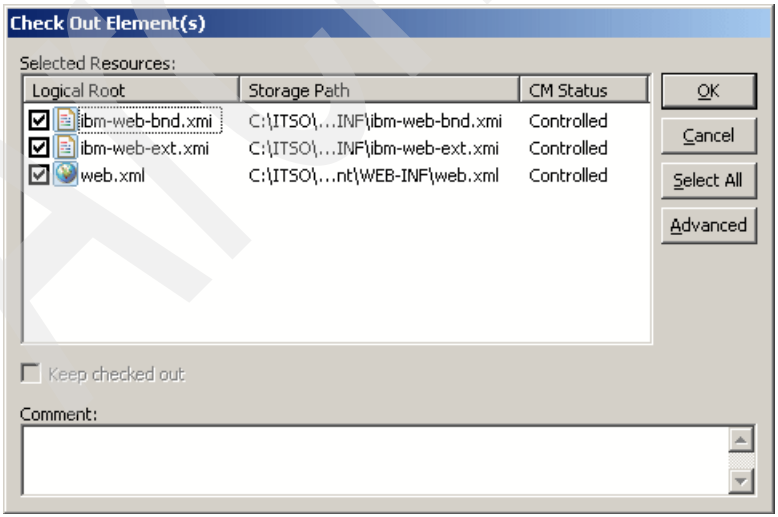


Figure 25-19 Checking out dependent elements

4. On the Select Activity dialog, select **New** and enter `Developer 1 adds servletA` as the name of the activity. Click **OK** twice. The servlet is generated.
5. On the Add Element(s) to Source Control dialog (see Figure 25-20) make sure the packages and the servlet are selected. Leave “Keep checked out” deselected and click **OK**.

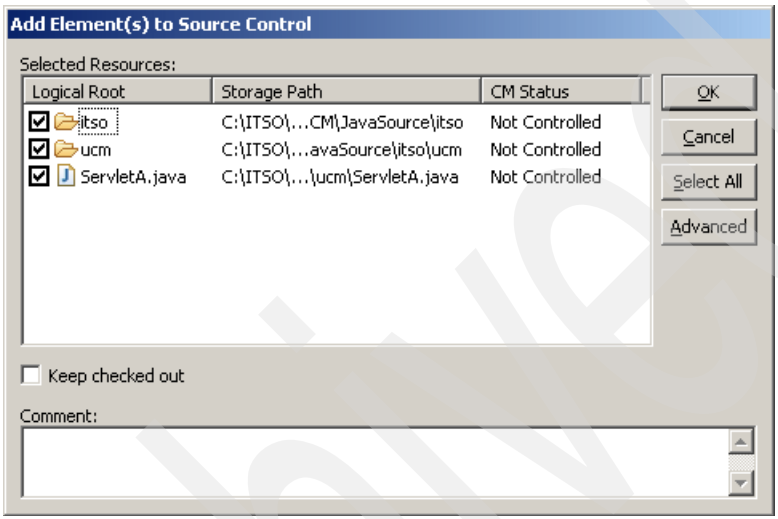


Figure 25-20 Adding new elements to source control

6. On the Select Activity dialog, select **Developer 1 adds ServletA** and click **OK**. The servlet is added to the project and to ClearCase source control.
7. Click **Yes** on the File Changed dialog.

   The three dependent files are still checked out. Expand **WebContent\WEB-INF** and note the green check marks. Before we deliver the work to the integration stream, we want to check them back in.
8. Select **web.xml** and select **Team** → **Check in** from the context menu.
9. On the Check in Elements dialog, ensure that the element is selected and click **OK**. The green check mark on the resource icon is removed, indicating that the file is no longer checked out.
10. Before we can deliver the pro**ject to the stream, the `ibm-web-bnd.xmi` and `ibm-web-ext.xmi` files must be** checked in as well. As their contents have not changed, we will simply undo their checkout. Select both of them and select **Team** → **Undo Check Out...** from the context menu.
11. Before delivering to the stream, it is also good practice to make sure that nothing else is checked out. Select **ClearCase** → **Find Checkouts...**. On the

Find Criteria dialog (see Figure 25-21) click **Browse** and select the **C:\ITSO\dev1_view\sources** directory. Keep the other defaults as shown and click **OK**.

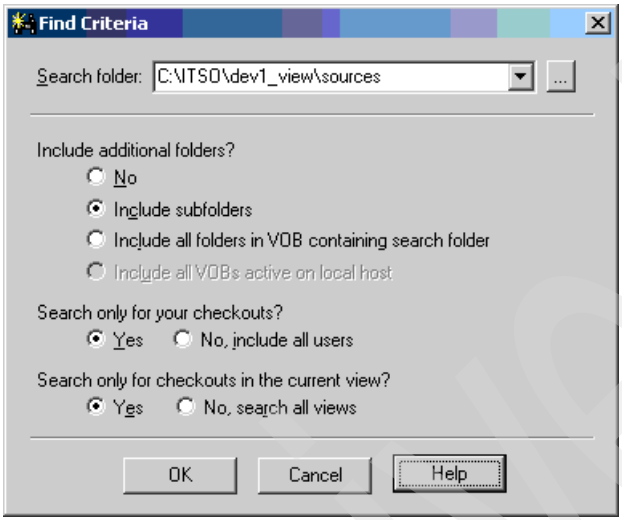


Figure 25-21 Finding checkouts

12. No checked out files should be found. Click **OK** to dismiss the dialog and then close the Find Checkouts window.

25.4.2 Developer 1 delivers work to the integration stream

Follow these steps to deliver the work into the integration stream:

1. Select **ClearCase** → **Deliver Stream**. On the Deliver from Stream dialog (Figure 25-22 on page 1280) select the development stream **dev1_view** and click **OK**.

Archived

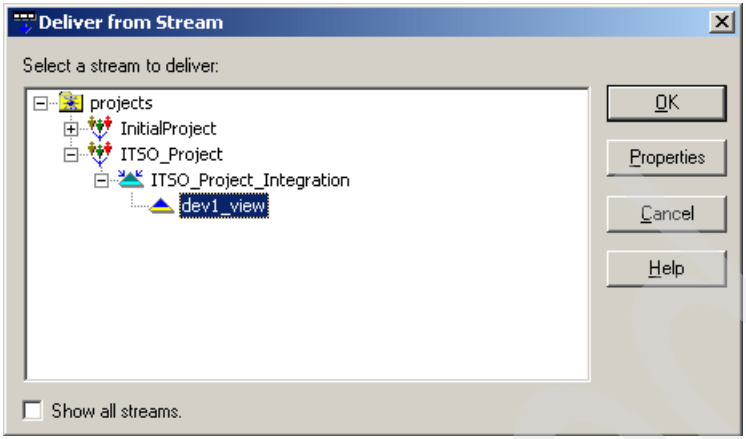


Figure 25-22 Delivering to integration stream

2. In the Deliver from Stream Preview dialog (Figure 25-23) make sure both activities are selected and that the view to deliver to is <userid>_Integration_view. Click **OK**.

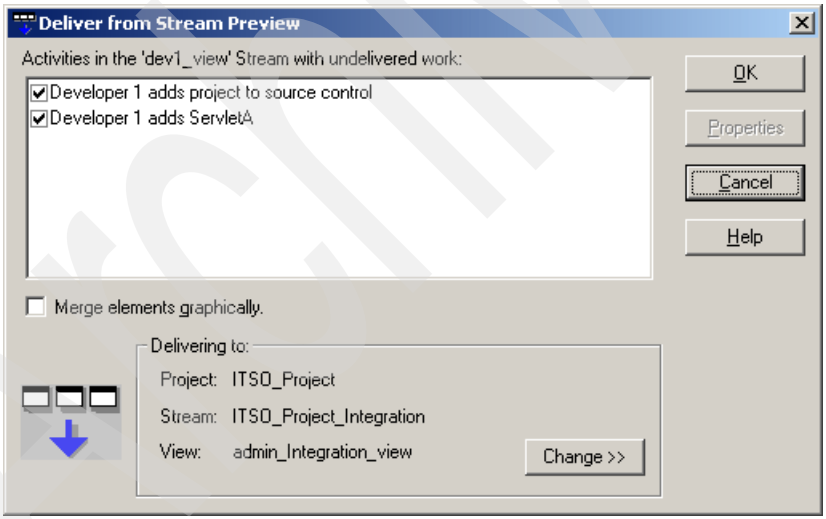


Figure 25-23 Deliver from Stream Preview dialog

3. After a while the Deliver from Stream - Merges Complete dialog (see Figure 25-24 on page 1281) is shown. Deselect **Open a ClearCase Explorer** and click **OK**.

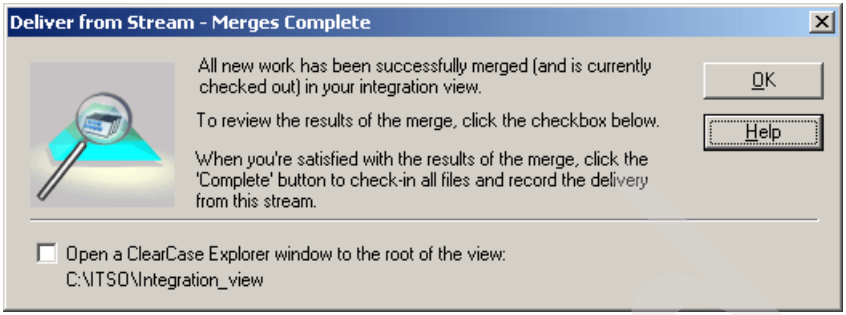


Figure 25-24 Deliver from Stream - Merges Complete dialog

4. On the Delivering to View dialog (Figure 25-25) click **Complete**.

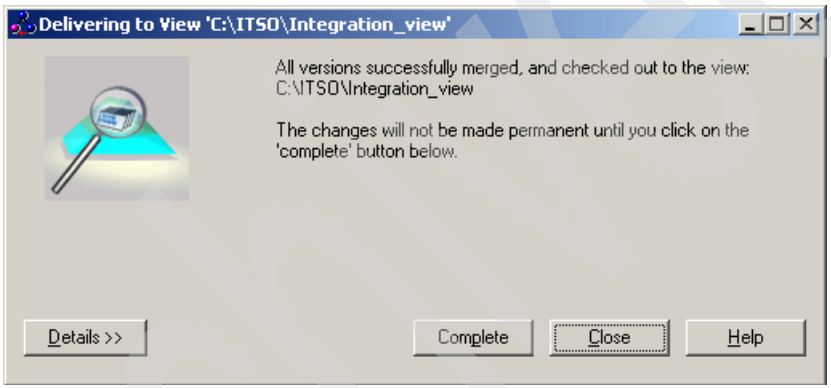


Figure 25-25 Delivering to View dialog

5. Optionally, you can click **Details** to see a list of the files delivered (see Figure 25-26 on page 1282), then click **Close**.

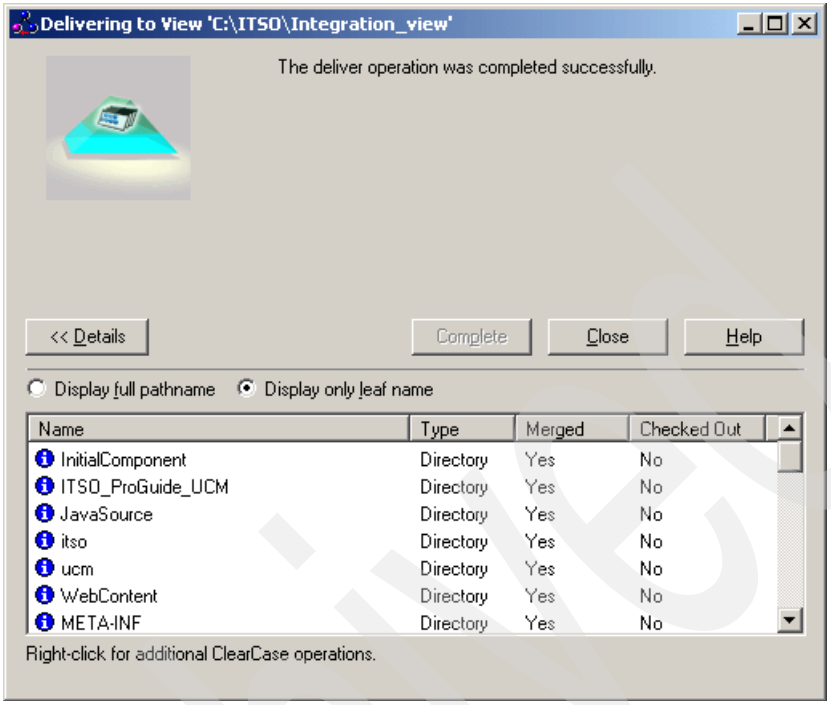


Figure 25-26 Showing files delivered

25.4.3 Developer 1 makes a baseline

To make a baseline, do the following:

1. Select **Start** → **Programs** → **Rational Software** → **Rational ClearCase** → **Project Explorer**.
2. In the left pane right-click **ITSO_Project_Integration** and select **Make Baseline** from the context menu (see Figure 25-27 on page 1283).

Archive

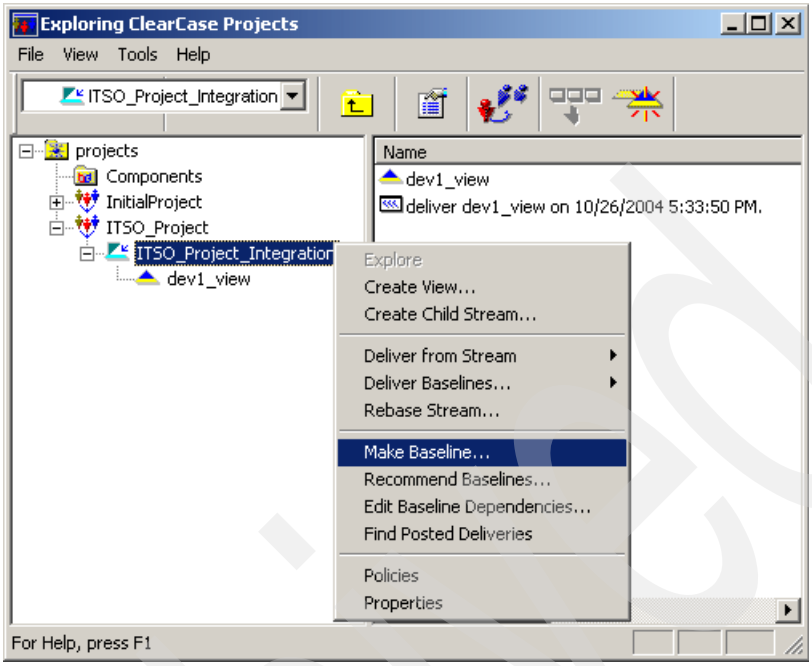


Figure 25-27 Making a baseline

3. In the Make Baseline dialog (see Figure 25-28) click **OK**.

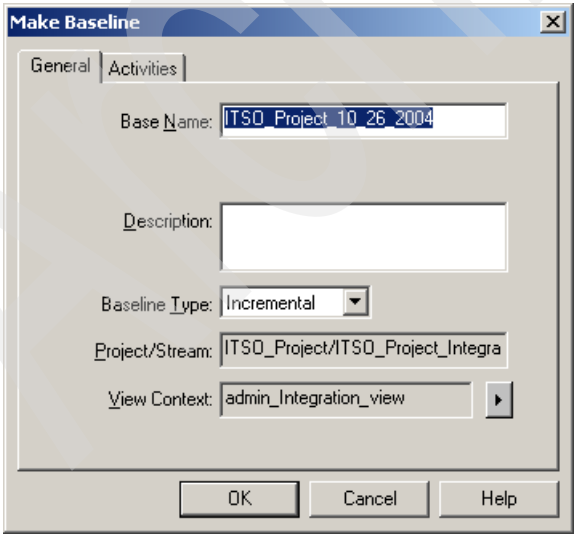


Figure 25-28 Make baseline settings

4. Click **OK** on the confirmation dialog (one new baseline was created) and then close the Make Baseline dialog.

You can now close the ClearCase Project Explorer. Developer 1 has now finished the current task. Developer 2 will now join the project and make changes to the servlet.

25.4.4 Developer 2 joins the project

Developer 2 now joins the ClearCase project and adds it to his Rational Application Developer workspace.

1. Select **File** → **Switch Workspace...** in Rational Application Developer for the scenario simulation purposes.
2. Enter `C:\dev2_workspace` as the workspace for developer 2. Click **OK**.
3. Close or minimize the Welcome view.
4. If you are not connected to ClearCase by the preference setting, select **ClearCase** → **Connect to Rational ClearCase** or click the ClearCase Connect icon ( ).
5. Select **ClearCase** → **Create New View**.
6. In the View Creation Wizard (Figure 25-29 on page 1285), select **Yes** to indicate that you are working on a ClearCase project. Expand **projects** and select the **ITSO_Project**. Click **Next**.

Archived

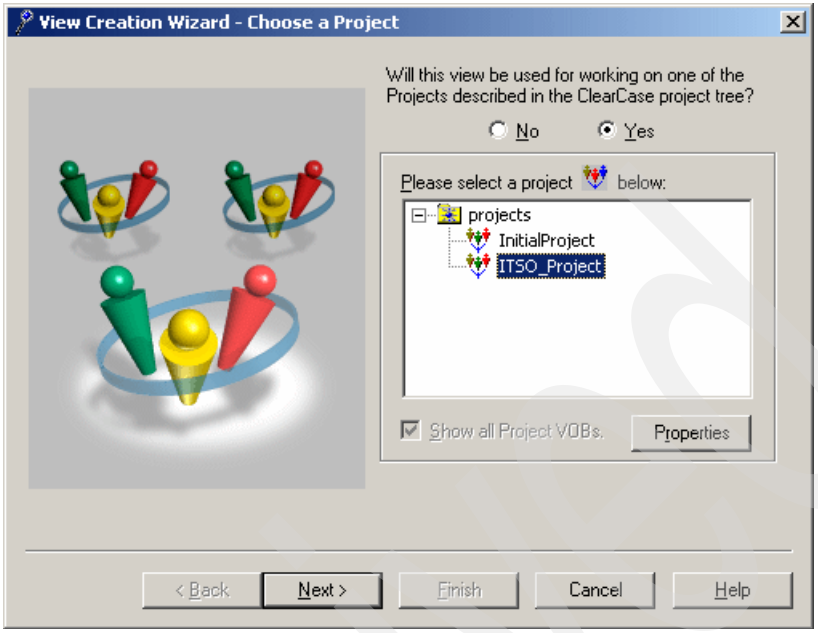


Figure 25-29 Creating a new view

7. The ClearCase View Tool dialog notifies you that there is already one stream defined for this Windows user (dev1_view). Click **OK**.
8. In the Create a Development Stream dialog (see Figure 25-30 on page 1286) enter `dev2_view` as the development stream name and verify that the integration stream name is ITSO_Project_Integration. Click **Next**.

Archive

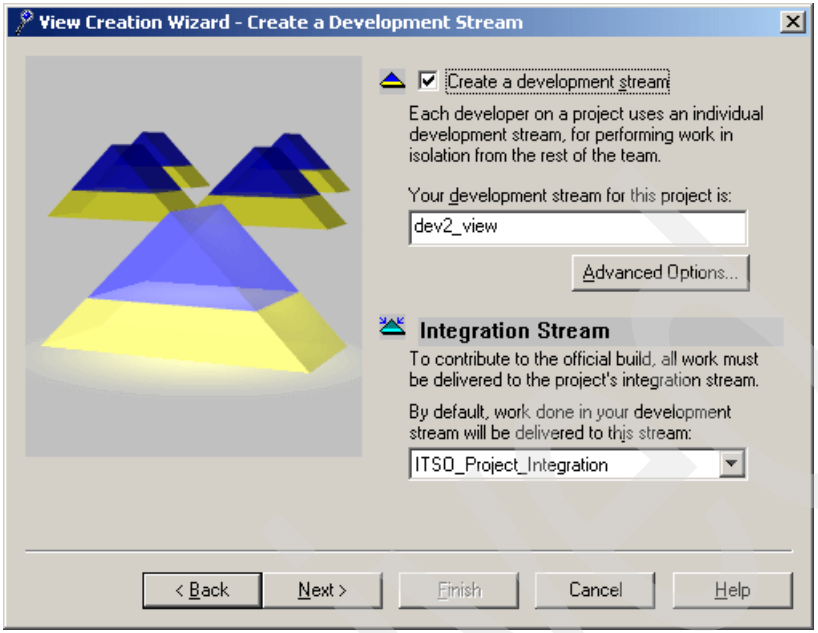


Figure 25-30 Creating a development stream - Developer 2

9. In the Review Types of Views dialog, ensure that “Create a Development view” is selected and click **Next**.
10. In the Choose Location for a Snapshot View (Development View) dialog (Figure 25-31 on page 1287) accept C:\ITSO\dev2_view as the path for the new development view and click **Next**.

Archive

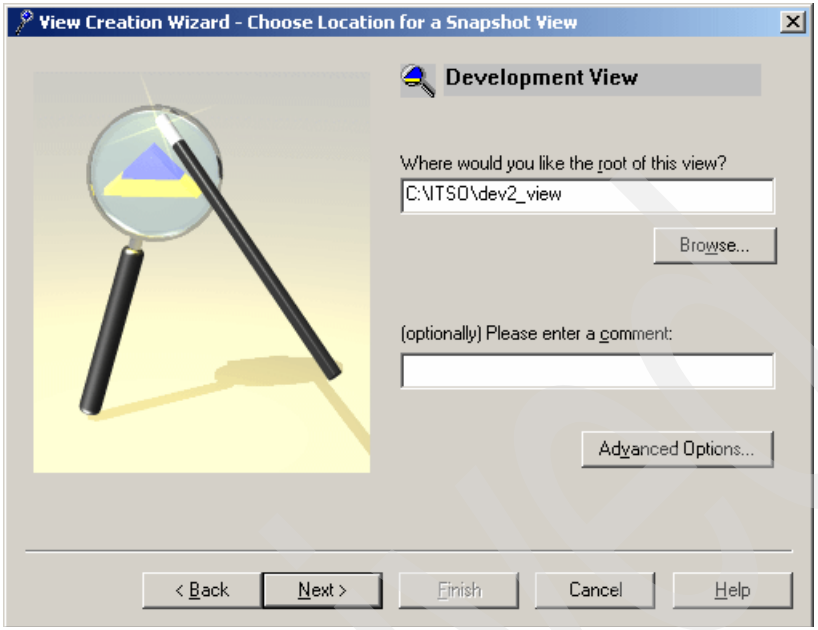


Figure 25-31 Select location for development view

11. In the Choose Components dialog, make sure **InitialComponent** is selected. Click **Finish** to create the development view for developer 2.
12. Click **OK** when the confirmation dialog is displayed and then click **OK** in the View Creation Status dialog.

25.4.5 Developer 2 imports projects into Application Developer

Developer 2 works on the same projects as developer 1 and has to import the projects:

1. As developer 2, select **ClearCase** → **Rebase Stream** in Rational Application Developer to update your development stream with the contents of the integration stream.
2. In the Rebase Stream dialog (see Figure 25-32 on page 1288) select **Projects** → **ITSO_Project** → **ITSO_Project_Integration** → **dev2_view** and click **OK**.

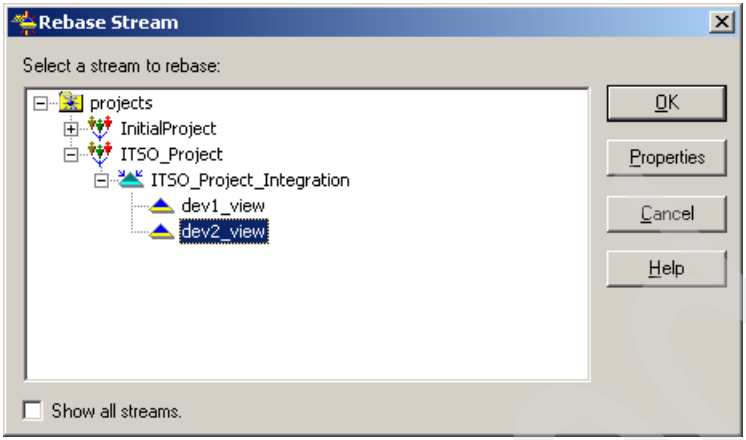


Figure 25-32 Rebase Stream dialog

3. In the Rebase Stream Preview dialog (Figure 25-33) select **InitialComponent_ITSO_Project_<date>** from the baseline drop-down list and verify that **<userid>_dev2_view** is selected as the target view. Click **OK**.

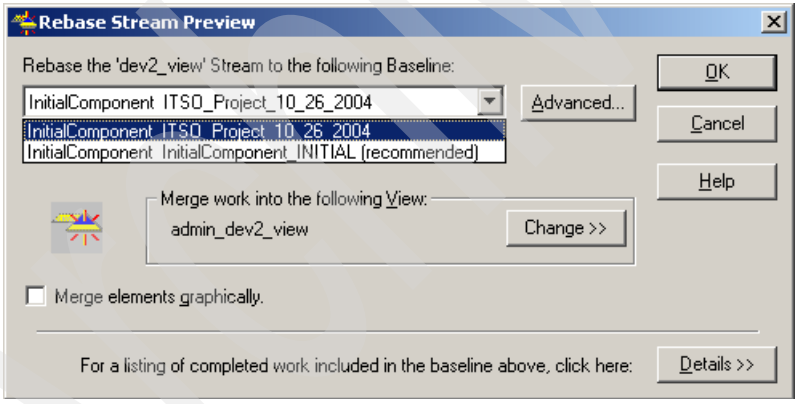


Figure 25-33 Rebase Stream Preview - Developer 2

4. Click **OK** to dismiss the Hijacked Files Warning dialog.
5. In the Rebasing in View dialog, click **Complete** to perform the rebase action. After this is done, click **Close**.

   The contents of the integration view have now been copied to the developer 2 view, but do not yet appear on this workspace.
6. In Rational Application Developer, select **File** → **Import** → **Existing Project into Workspace** and click **Next** (see Figure 25-34 on page 1289).

7. Click **Browse**, select the EAR project, and click **Finish**.

   `C:\ITSO\dev2_view\sources\InitialComponent\ITSO_ProGuide_UCMEAR`

R

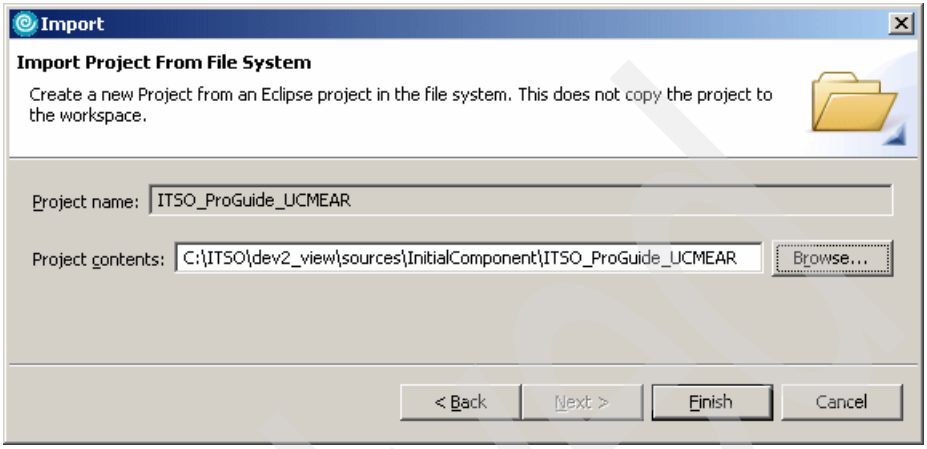


Figure 25-34 Import EAR project

8. Repeat the import process for the Web project:

   `C:\ITSO\dev2_view\sources\InitialComponent\ITSO_ProGuide_UCM`

Note: By now developer 1 and developer 2 are set up with a new shared project. Both can now check out files, work with these files, check them in, and deliver their work to the stream.

25.4.6 Developer 2 modifies the servlet

As we only want to show how to work with ClearCase, we do not need to add any real code to the servlet. Adding a simple comment to the servlet will work just as well.

1. Expand **ITSO_ProGuide_UCM** → **Java Resources** → **JavaSource** → **itso.ucm** and open **ServletA.java** by double-clicking it.
2. Start entering some text inside the green comment area, indicating that developer 2 made this change. As soon as you start typing in the editor, the servlet needs to be checked out. The dialog shown in Figure 25-35 on page 1290 is displayed, asking you to check out the servlet file. Click **OK**.

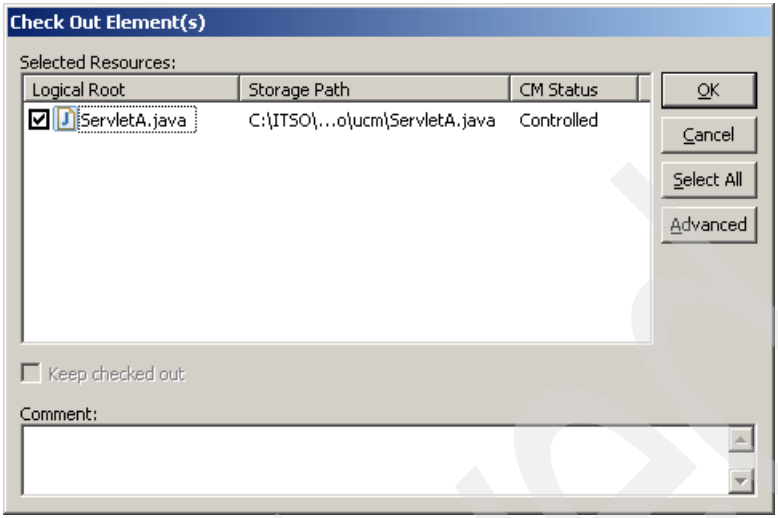


Figure 25-35 Check out elements

3. On the Select Activity dialog, create a new activity **Developer 2 updates ServletA** for this update and click **OK** to confirm. The file is now checked out, which is indicated by the green check mark on the servlet's icon .
4. After making your changes (for example, adding your name as the author), press Ctrl+S to save the servlet, and then close the editor.
5. The changed file must now be checked in. Right-click the servlet and select **Team** → **Check in** in the context menu (or use the Check in icon).
6. On the Check in Elements dialog, click **OK**. The green check marks of the resources icons are removed, indicating that the elements are no longer checked out.

   Developer 2 is now ready to deliver the changes to the stream and share the code with the other developers. Before doing this, it is a best practice to make sure that no other developer has made changes recently.
7. Select **ClearCase** → **Rebase Stream**. In the Rebase Stream dialog, select developer 2's development stream and click **OK** (Figure 25-36 on page 1291).

Archived

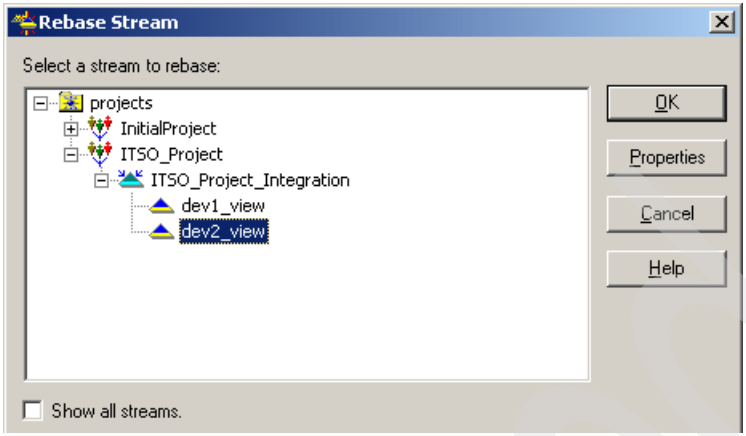


Figure 25-36 Selecting development stream to rebase - Developer 2

8. In the Rebase Stream Preview dialog (see Figure 25-37), select the latest baseline (top of the list), and make sure the <userid>_dev2_view view is selected as the target. Click **OK**.

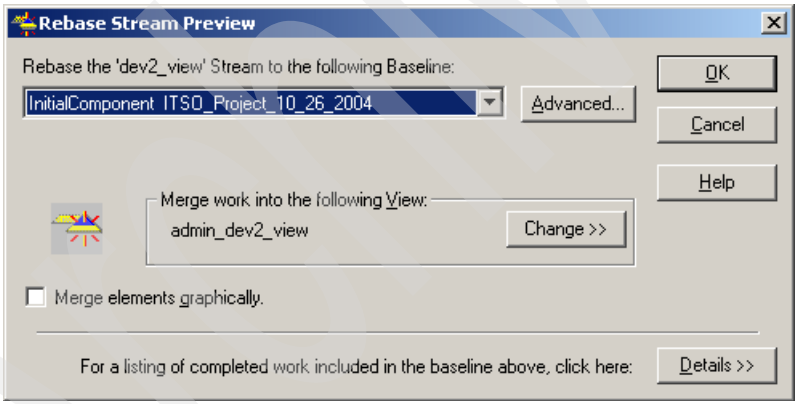


Figure 25-37 Rebase Stream Preview

9. The Rebase Stream dialog is displayed, and notifies you that the stream is currently up-to-date. Click **OK** to dismiss the information dialog and click **Cancel** to dismiss the Rebase Stream Preview dialog.

Note that what we did was to check to make sure developer 1 did not make any changes to the stream.

25.4.7 Developer 2 delivers work to the integration stream

Developer 2 is ready to integrate his work:

1. As developer 2, select **ClearCase** → **Deliver Stream**. In the Deliver from Stream dialog (Figure 25-38) select **Projects** → **ITSO_Project** → **ITSO_Project_Integration** → **dev2_view**. Click **OK**.

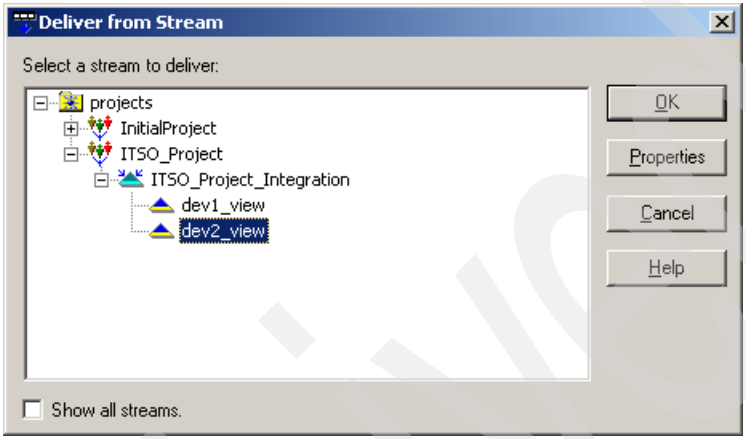


Figure 25-38 Deliver from Stream dialog

2. On the Deliver from Stream Preview dialog (see Figure 25-39) make sure the <userid>_Integration_view is selected as the integration view. Click **OK**.

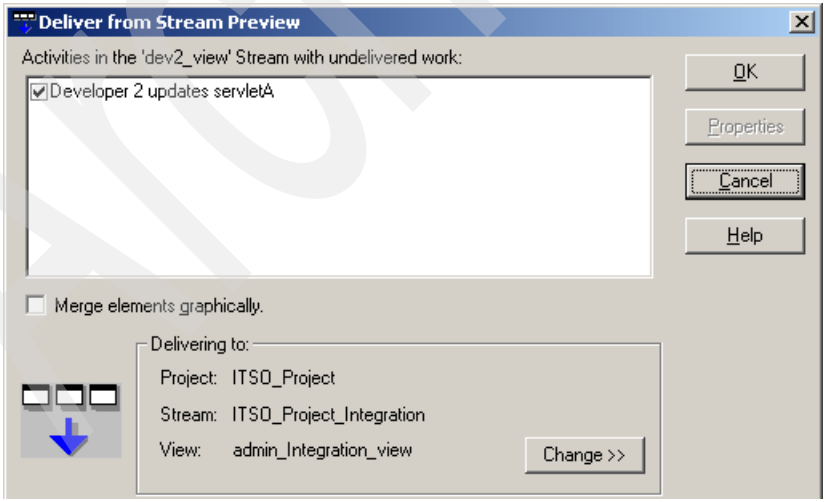


Figure 25-39 Deliver from Stream Preview

3. The integration view is now updated with the contents of the development view.
4. On the Delivering to View dialog (see Figure 25-40), click **Complete** and then **Close**.

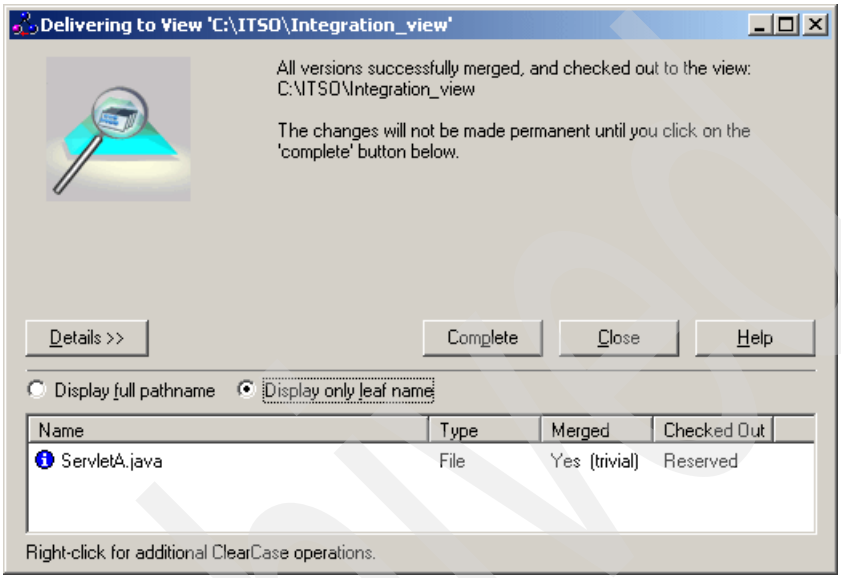


Figure 25-40 Delivering to integration view - Developer 2

25.4.8 Developer 1 modifies the servlet

ServletA has now been updated once in the integration stream since the original baseline was created. Let us see what happens when another developer makes changes to the same element and makes delivery.

1. Select **File** → **Switch Workspace...** in Rational Application Developer.
2. Enter `C:\dev1_workspace` as the workspace for developer 2. Click **OK**.
3. Expand **ITSO_ProGuide_UCM** → **Java Resources** → **JavaSource** → **itso.ucm** and open ServletA.java by double-clicking it.
4. Make an update to the comment as developer 1. The servlet will be checked out again.
5. On the Select Activity dialog, select **Developer 1 modifies ServletA** and click **OK**.
6. After making your changes (use some other text than previously), press Ctrl+S to save the servlet and then close the editor.
7. Check in the updated servlet.

25.4.9 Developer 1 delivers new work to the integration stream

Developer 1 is now ready to integrate his work as well:

1. As developer 1, select **ClearCase** → **Deliver Stream**. In the Deliver from Stream dialog (Figure 25-41) select **Projects** → **ITSO_Project** → **ITSO_Project_Integration** → **dev1_view**. Click **OK**.

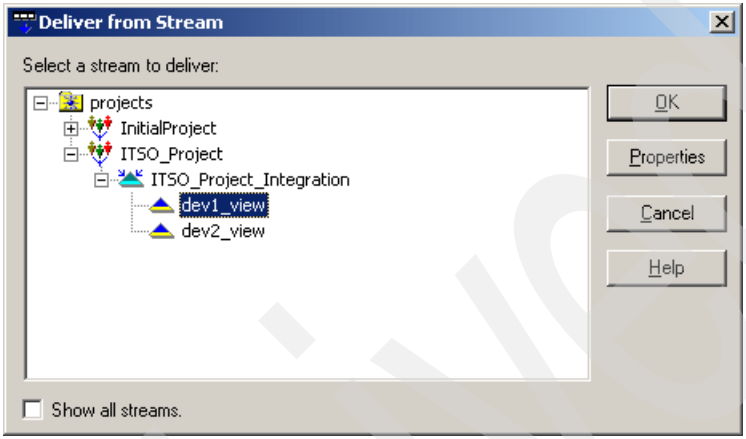


Figure 25-41 Deliver from Stream dialog - Developer 1

2. On the Deliver from Stream Preview dialog (see Figure 25-39 on page 1292) make sure the <userid>_Integration_view is selected as the integration view. Click **OK**.

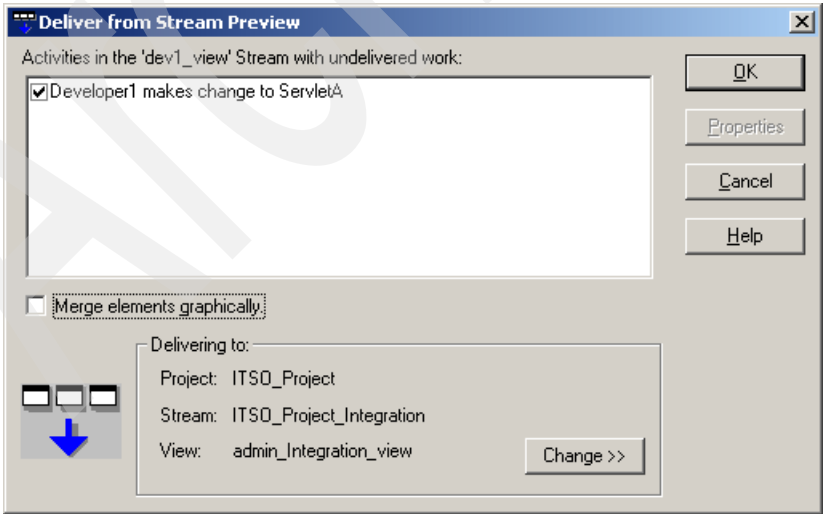


Figure 25-42 Deliver from Stream Preview

3. ClearCase notifies you that there is a conflict and the element cannot be merged automatically (Figure 25-43). Select **start the Diff Merge tool for this element** and click **OK**. Click **OK** again at the Diff Merge status dialog.

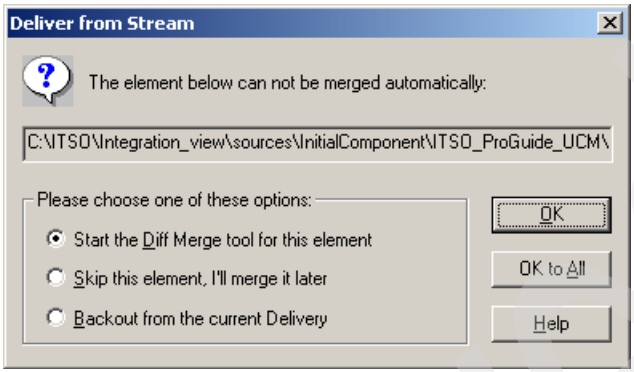


Figure 25-43 Deliver from Stream alert

4. The Diff Merge tool is now launched (see Figure 25-44). Take a few moments to become familiar with the information displayed and the options available to you.

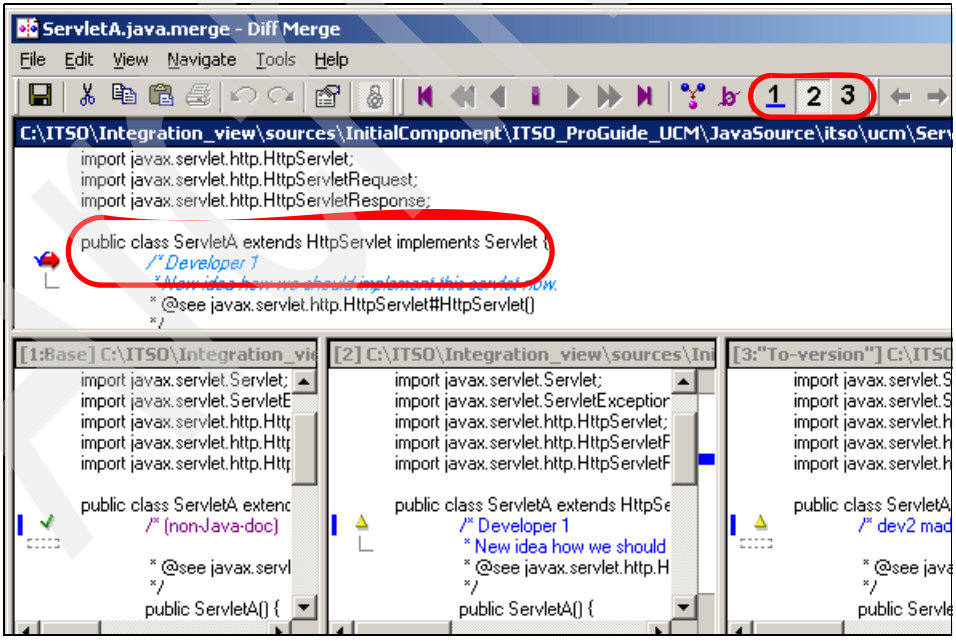


Figure 25-44 Diff Merge tool

Archived

- The top panel shows the merge result and areas of conflict. Click a line to focus on that particular conflict.
- The lower panels show the different versions that are available.
- The number icons on the toolbar (1, 2, 3) are used to indicate which version should be used in the merged result.

5. In the top panel displaying the merge result, click the line that has the arrow next to it, indicating the conflict.
6. On the toolbar, click **2** to determine that in this case the implementation by developer 1 should be used. Verify the change in the merge panel.
7. Close the Merge Tool by saving the changes (select **File** → **Save***)* and close the tool.
8. On the Delivering to View dialog, click **Complete** and then **Close**.

Now all the changes to the servlet have been applied to the integration stream. You can verify this by looking at the element in the Version Tree (see Figure 25-45 on page 1297).

You can invoke the Version Tree Browser from Rational Application Developer or from Windows Explorer. Right-click the servlet file and select the **Version Tree** option from the context menu.

Archived

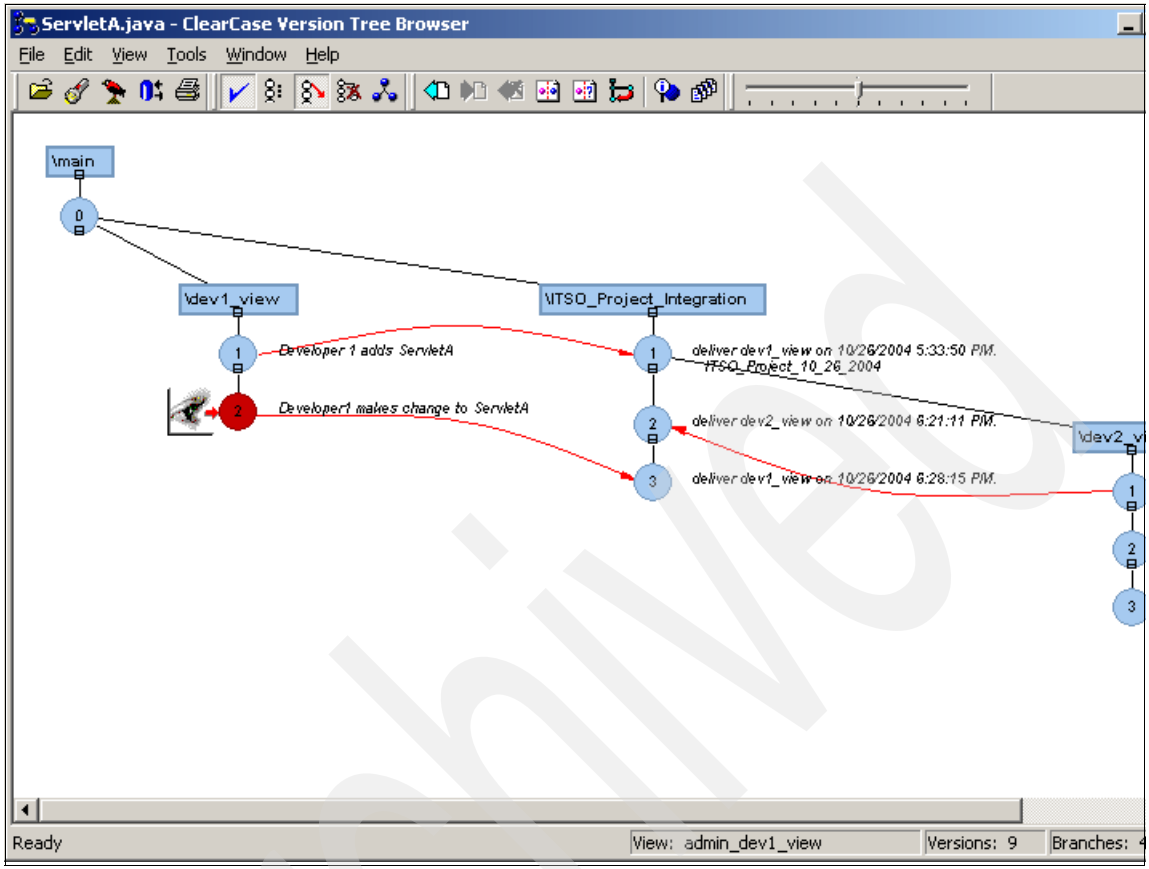


Figure 25-45 ClearCase Version Tree Browser

Note that looking at the same element as developer 2 will show the new element version in the tree, but we will not see the latest changes in that view because our view is set to look at the recommended base line. To see the delivered changes, developer 2 will need to wait for a new baseline to be created, and then rebase from that.

Archived

Archived

26

CVS integration

This chapter provides an introduction to the widely adopted open source Concurrent Version System (CVS). We will discuss the integration features of Rational Application Developer tooling for CVS by guiding the reader through an example implementation, as well as usage scenarios.

Note: The example in this chapter calls for two simulated developer systems. This can be accomplished by having two instances of Rational Application Developer or two Workspaces for demonstration purposes. Refer to 3.1, "Workbench basics" on page 76, for detailed instructions.

This chapter is organized into the following topics:

- Introduction to CVS
- CVSNT Server implementation
- CVS client configuration for Application Developer
- Configure CVS in Rational Application Developer
- Development scenario
- CVS resource history
- Comparisons in CVS
- Annotations in CVS
- Branches in CVS
- Work with patches
- Disconnecting a project
- Synchronize perspective

© Copyright IBM Corp. 2005. All rights reserved.

26.1 Introduction to CVS

Concurrent Version System (CVS) is a simple open source software configuration management (SCM) system. CVS only implements version control. CVS can be used by individual developers as well as by large, distributed teams.

26.1.1 CVS features

Some of the main features of CVS are as follows:

- Multiple client-server protocols over TCP/IP that let developers access the latest code from a wide variety of clients virtually anywhere an Internet connection exists.

  Note: IBM Rational Application Developer V6 supports three communication protocols:

  - pserver (password server
  - ext
  - extssh

  The protocols ext and extssh require additional configuration from the Rational Application Developer preferences:

  - ext: **Window → Preferences → Team → CVS → Ext Connection Method**
  - extssh: **Window → Preferences → Team → CVS → SSH2 Connection Method**

- It stores all the versions of a file in a single file using forward-delta versioning, which stores only the differences among the versions.
- It insulates the different developers from each other. Every developer works in his own directory, and CVS merges the work in the repository when each developer is done. Conflicts should be resolved in the process.

  Important: CVS and IBM Rational Application Developer V6 have a different understanding of what a conflict is:

  - For CVS, a conflict arises when two changes to the same base file are close enough to be noticed by the merge command.
  - For IBM Rational Web Developer V6, a conflict exists when the local copy of a revision of a modified file is different from the revision stored in the repository.

- It uses an unreserved checkout approach to version control that helps avoid artificial conflicts common when using an exclusive checkout model.
- It keeps shared project data in repositories. Each repository has a root directory on the file system.
- CVS maintains a history of the source code revisions. Each change is stamped with the time it was made and the user name of the person who made it. It is recommended that developers also provide a description of the change. Given that information, CVS can help you find answers to questions such as: Who made the change? When was it made, and why?

26.1.2 New V6 features for team development

The key new IBM Rational Application Developer V6 features for team development are as follows:

- New team synchronize view
- Multiple synchronization of repositories
- Scheduling of synchronization
- Setting up of a commit set of defined resources located anywhere in the workspace
- Background synchronization of CVS operations of the workspace
- Checkout wizard to allow checkout by using **New** → **Project** or **File** → **Import**
- Editor interface to specify CVS date tags in standardized format
- “Blame” view to identify what and who has changed a particular file
- Additional security options to communicate with the CVS server

This list does not include all the new features, but points to the major differences compared to previous versions of WebSphere Studio products.

26.2 CVSNT Server implementation

The CVS server code for Linux and UNIX platforms is available at the project’s site, as is installation and usage documentation:

`http://www.cvshome.org`

Since our development environment is a pure Windows environment, we choose to use CVS for NT (CVSNT) for the development environment.

Archived

The CVSNT Server implementation is organized as follows:

- CVS Server installation.
- CVS Server repository configuration.
- Create the Windows users and groups used by CVS.
- Verify the CVSNT installation.
- Create CVS users.

Important: CVSNT is not officially supported by IBM Rational Application Developer V6.

We chose to use CVSNT V2.0.58b since all of our samples where developed on the Windows platform and we found CVSNT easy to use. We did not experience any significant problems using CVSNT.

For information on CVS compatibility with Eclipse versions, refer to the following URL:

`http://dev.eclipse.org/viewcvs/index.cgi/%7Echeckout%7E/platform-vcm-home/docs/online/html-cvs/cvs-compatibility.html`

26.2.1 CVS Server installation

To install CVS on the Windows platform, do the following:

1. Before installing CVSNT, we recommend reading the installation tips:

   `http://www.cvsnt.org/wiki/InstallationTips`

   Important: The CVSNT software requires a user that has local system privileges to install and configure a service in Windows.

2. Download the CVSNT V2.0.58b Server (cvsnt-2.0.58b.exe) from the following URL to a temporary directory (for example, c:\temp) on the Build and SCM node:

   `http://www.cvsnt.org`

3. Execute the CVSNT installer by double-clicking the self-extracting **cvsnt-2.0.58b.exe** file from the temporary directory.
4. When the Welcome window appears, click **Next**.
5. When the License Agreement window appears, review the terms and if in agreement select **I accept the agreement** and click **Next**.
6. When the Select Destination Location window appears, we entered `c:\cvsnt` and then clicked **Next**.

7. When the Select Components window appears, we did the following and then clicked **Next**:
   - Select **Typical Installation**.
   - Under protocols, we additionally checked **Named Pipe (:ntserver) Protocol**.
8. When the Select Start Menu Folder window appears, accept default and click **Next**.
9. When the Select Additional Tasks window appears, we did the following and then clicked **Next**:
   - Check **Install cvsnt service**.
   - Check **Install cvsnt lock service**.
   - Check **Generate default certificate**.
10. When the Ready to Install window appears, review the installation options and then click **Install**.
11. When the Completing the cvsnt Setup Wizard window appears, accept the defaults and click **Finish**.
12. Restart your system.

    When the installation is complete, we recommend that you restart your system. This step will guarantee that the environment variables are set up properly and the CVSNT Windows services are started.

26.2.2 CVS Server repository configuration

After you have installed the CVS Server and restarted your system (Build and SCM node), do the following to create and configure the CVS Server repository:

1. Manually create the common root directory. For example, we created the c:\rep6449 directory using Windows Explorer or via a command window.
2. Stop the CVS services in order to create the repository, by clicking **Start** → **Programs** → **CVSNT** → **Service control panel**.
3. When the CVSNT control panel window appears, stop the following services by clicking stop:
   - Click **Stop** under CVS service.
   - Click **Stop** under CVS Lock service.
4. Click the **Repositories** tab.
5. Click **Add**.
6. When the Edit repository window appears, we entered the following (as seen in Figure 26-1 on page 1304), and then clicked **OK**:

Archived

– Location: `c:/rep6449`

> **Note:** We manually created this directory on the file system in step 1.

– Name: `/rep6449`

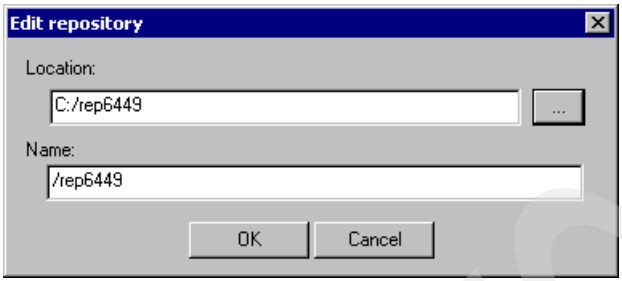


Figure 26-1 Add repository

7. When prompted with the message `c:/rep6449 exists, but is not a valid CVS repository. Do you want to initialise it?`, click **Yes**.

   When done, the Repository page should look like Figure 26-2.

Archived

Figure 26-2 CVSNT service configuration (Repository page)

8. Click **Apply**.
9. Click the **Compatibility** tab.
10. From the Compatibility tab, check the following options (as seen in Figure 26-3), and then click **OK**:
    - Select **Generic non-cvsnt**.
    - Check **Respond as cvs 1.11.2 to version request**.
    - Check **Emulate '-n checkout' bug**.
    - Check **Hide extended log/status information**.

    These settings ensure that CVSNT is compatible with clients such as IBM Rational Application Developer V6 and other CVS clients.

Archived

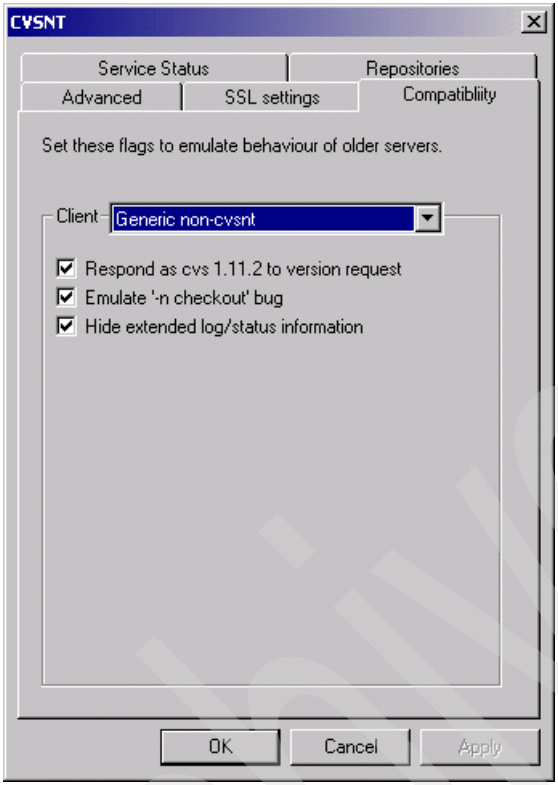


Figure 26-3 Compatibility settings required for interoperation with IBM Rational Application Developer V6 and other CVS clients

26.2.3 Create the Windows users and groups used by CVS

When setting up CVS users, we opted to use the *pserver protocol* commonly used to secure a CVS repository for the following reasons:

- The pserver protocol is desired when working from a multi operating system environment (for example, Windows and Linux clients).
- Provides a password facility that is independent of the operating system (for example, the pserver protocol does not use the password system of the native operating system).
- A single CVS administrator use can be set up and used to run the CVS commands minimizing the administration of permissions and security of all the users that will use the repository.

Archived

Add a Windows user (cvsadmin)

To add a Windows CVS administrator user do the following:

1. From the Windows desktop, right-click **My Computer**, and select **Manage**.
2. Expand **Local Users and Groups** and select **Users**.
3. From the menu bar, click **Action** → **New User**.
4. When the New user window appears, we entered the following and clicked **Create**:
   - User name: `cvsadmin`
   - Password: `<password>`
   - Uncheck **User must change password at next logon**.
5. Click **Close** on the New Users dialog, but do not exit the Computer Management tool (needed in next section).

Add Windows user (cvsadmin) to a group (Power Users)

To add the Windows user to a group that has the sufficient permissions, do the following:

1. Click **Groups** and double-click **Power Users**.
2. Click **Add**.
3. Select the **cvsadmin** and click **Add**.
4. Click **OK** to exit the Select Users or Groups window.
5. Click **OK** to exit the Power Users Properties window.
6. Exit the Computer Management console.

26.2.4 Verify the CVSNT installation

To verify the CVSNT installation, do the following:

1. Restart the system to ensure that the environment variables are loaded and the CVSNT services are started.
2. After the system has been restarted, verify that the following CVSNT Windows services have started:
   - CVSNT
   - CVSNT Lock Service

Archived

Tip: The CVSNT services can also be accessed from the CVSNT Service control panel.

When using the CVSNT Service control panel, we noticed that on occasion under the Service Status tab when clicking the **Stop** button for the *C*VS Lock Service and then clicking the **Start** button, that the lock service does not start running.

To resolve this open the Windows Task Manager, locate the process cvslock.exe, right-click, and select **End Process Tree**. Reattempt clicking the **Start** button for the CVS Lock Service and this should set it in the running state. If further problems occur, restart the system.

26.2.5 Create CVS users

To create CVS users to access the files in the repository, do the following:

1. Open a Windows command prompt.
2. Set the cvsroot as follows:

   ```
   set cvsroot=:pserver:cvsrep1.itso.ral.ibm.com:/rep6449
   ```

   Where cvsrep1.itso.ral.ibm.com is the host name of the repository node and /rep6449 is the repository located on this host name.

   Note: The host name must be specified. For example, specifying localhost will not work.

3. Log on to the CVS repository machine to manage the users, using the following command:

   ```
   cvs login cvsadmin
   ```

4. You will be prompted to enter the CVS password.

   We entered the password for the cvsadmin user created in 26.2.3, “Create the Windows users and groups used by CVS” on page 1306.

5. Enter the following CVS commands to add users:

   ```
   cvs passwd -a -r cvsadmin <cvs user id>
   ```

   – Where **`cvs passwd -a`** indicates that you wish to add a password for a user

   – **`-r cvsadmin`** indicates the alias or native user name that the user will run under when connecting to the repository (set up in “Create the Windows users and groups used by CVS” on page 1306).

– *<cvs user id>* is the user ID to be added in.

For example, to add user cvsuser1 enter the following:

```
cvs passwd -a -r cvsadmin cvsuser1
```

Note: The first occurrence of a user being added will create the file passwd in the directory, in our case c:\rep6449\CVSROOT. The following user will be appended into this file. It is recommended that this file not be edited directly by a text editor. It also must not be placed under CVS control.

6. A prompt will appear to enter the password, as shown in the following:

```
Adding user cvsuser1@cvsrep1.itso.ral.ibm.com
New password: ********
Verify password: ********
```

7. Repeat the previous step for additional CVS users.
8. Next, provide the development users their CVS account information, host name, and connection type to the CVS server so that they can establish a connection from WebSphere Studio Application Developer in a later configuration step on the Developer node. For example:
   – Account info: Developer CVS user created (for example, `cvsuser1`)
   – Host name: <CVS_Server_host_name> (for example, `cvsrep1.itso.ral.ibm.com`)
   – Connection type: `pserver`
   – Password: `<password>`
   – Repository path: `/rep6449`

26.3 CVS client configuration for Application Developer

This section describes CVS client configurations to be made for use with IBM Rational Application Developer V6.0.

26.3.1 Configure CVS Team Capabilities

IBM Rational Application Developer V6, by default, does not enable the *Team Capabilities* when creating a new workspace for a user.

To enable the Team Capabilities, do the following:

1. Open IBM Rational Application Developer V6.
2. Click **Windows** → **Preferences**.

3. Select and expand the **Workbench** and click **Capabilities**.
4. Select and expand the **Team** capability and select the items showing in Figure 26-4 on page 1310.

> **Note:** It is sufficient to set the *CVS Support* only, since the IBM Rational Application Developer V6 will automatically set *Core Team Support* as well.

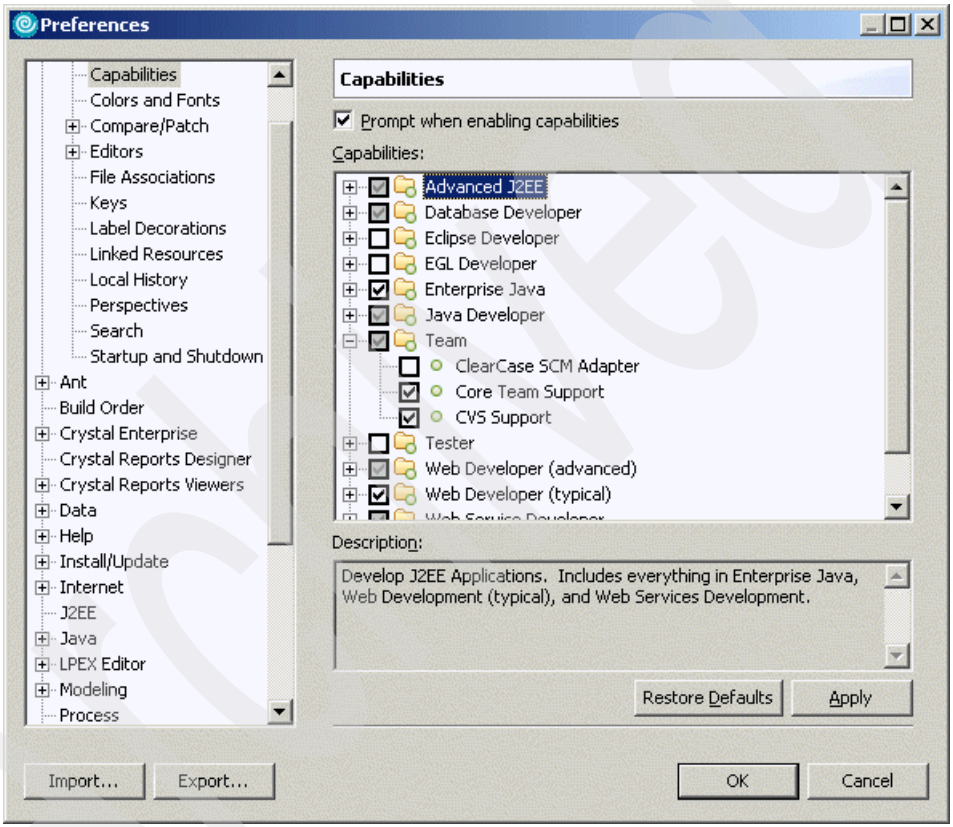


Figure 26-4 Setting the Team capability to be available

5. Click **Apply** and then click **OK**.

26.3.2 Access the CVS Repository

To access a repository that has been configured on a server for users to perform their version management, do the following:

1. Open IBM Rational Application Developer V6.

2. Click **Windows** → **Open Perspective** → **Other** → **CVS Repository Exploring**. Click **OK**.
3. In the CVS Repositories view, right-click and select **New** → **Repository Location**.
4. Add the parameters for the repository location as in Figure 26-5, check **Validate Connection Finish** and **Save Password**, and then click **Finish**.

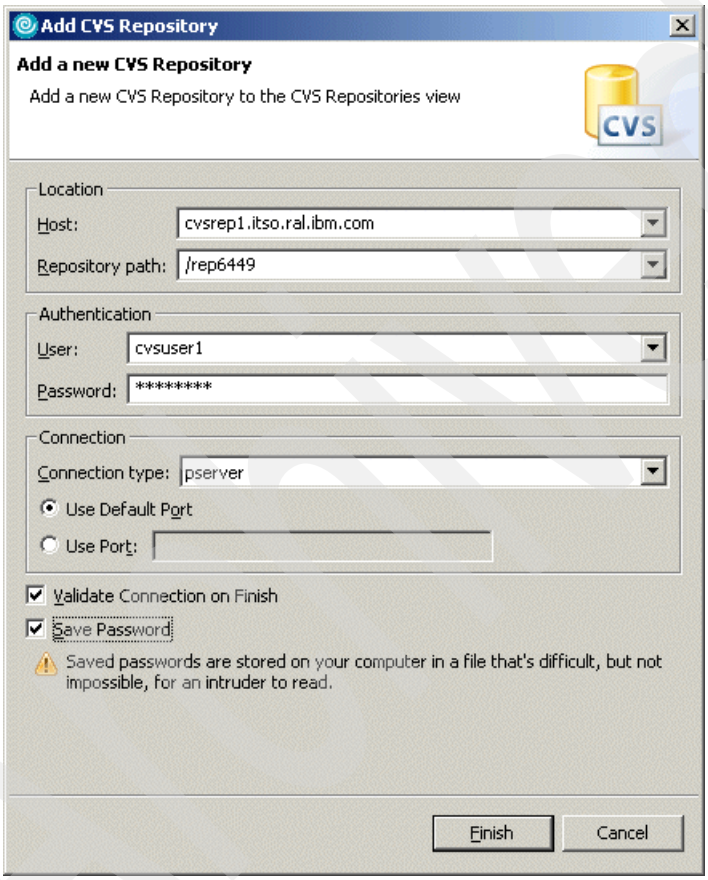


Figure 26-5 Add the CVS repository to the workspace

If everything worked correctly, you now should be able to see a repository location with HEAD, Branches, and Versions (see Figure 26-6 on page 1312).

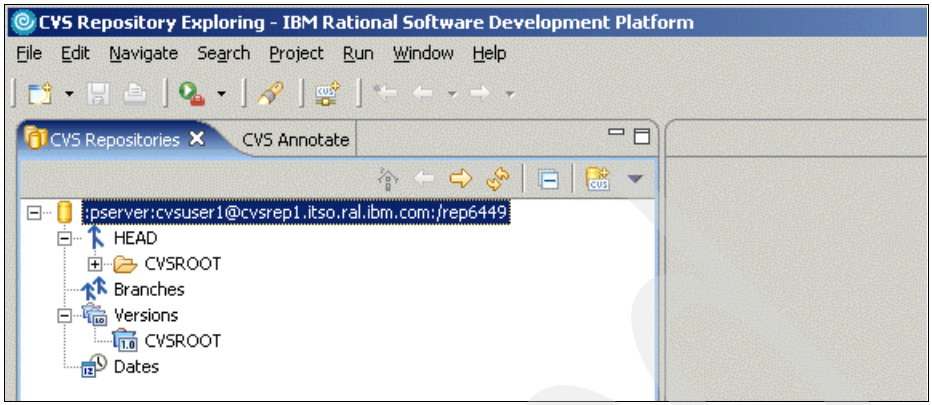


Figure 26-6 The CVS perspective of a successfully connected CVS repository

26.4 Configure CVS in Rational Application Developer

The team support for CVS had some major improvements in IBM Rational Application Developer V6.

26.4.1 Configure Rational Application Developer CVS preferences

Before you start working with CVS, you should look through the CVS preferences. Preferences that can be set for CVS include:

- Label decorations
- File content
- Ignored resources
- CVS-specific settings

Included in this section is a description of the keyword substitutions that CVS provides and how they can be used.

Label decorations

Label decorations are set to be on by default.

To view or change the label decorations, select **Windows** → **Preferences** and expand the **Label Decorations** section and select **CVS**.

CVS label decorations are set to be on if the check box is checked (see Figure 26-7 on page 1313).

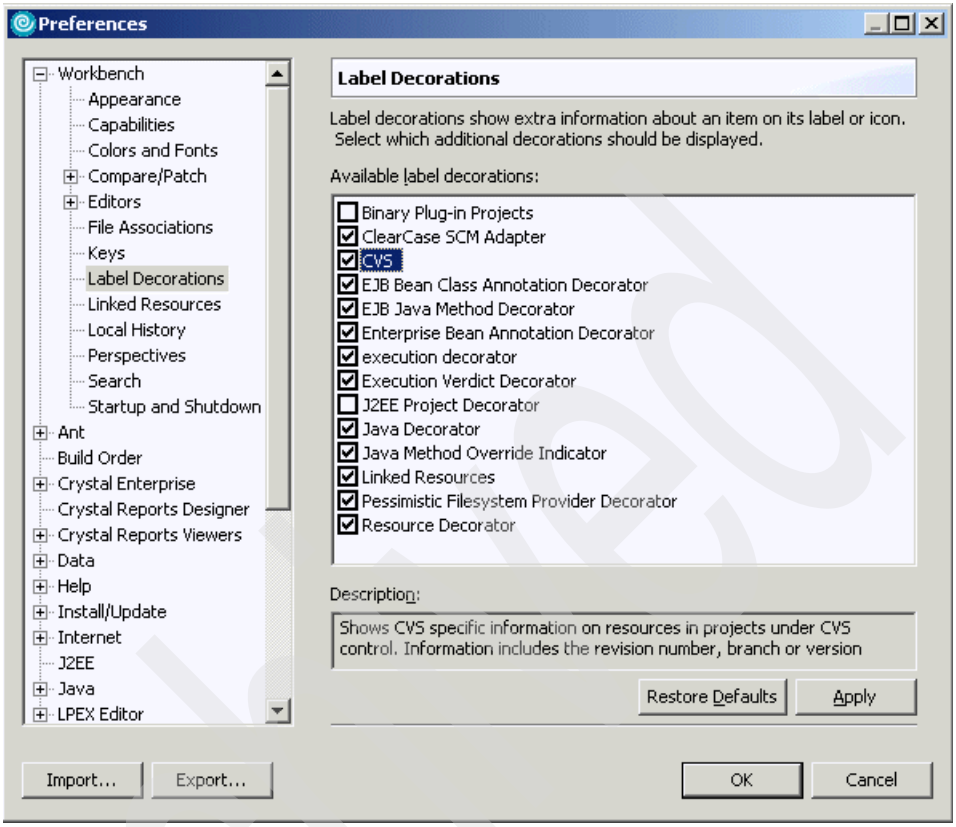


Figure 26-7 CVS decoration preferences

File content

The file content of resources can be changed to be saved into the repository as either ASCII or Binary. When working with file extensions that are not part of the file contents that are defined in IBM Rational Application Developer V6, then these files are saved into the repository as Binary by default.

To verify that a resource in the workspace is stored in the repository correctly, select **Windows** → **Preferences** and expand the **Team** → **File Content** section (see Figure 26-8 on page 1314). Verify that the file extension that you are using is present and stored in the repository as desired.

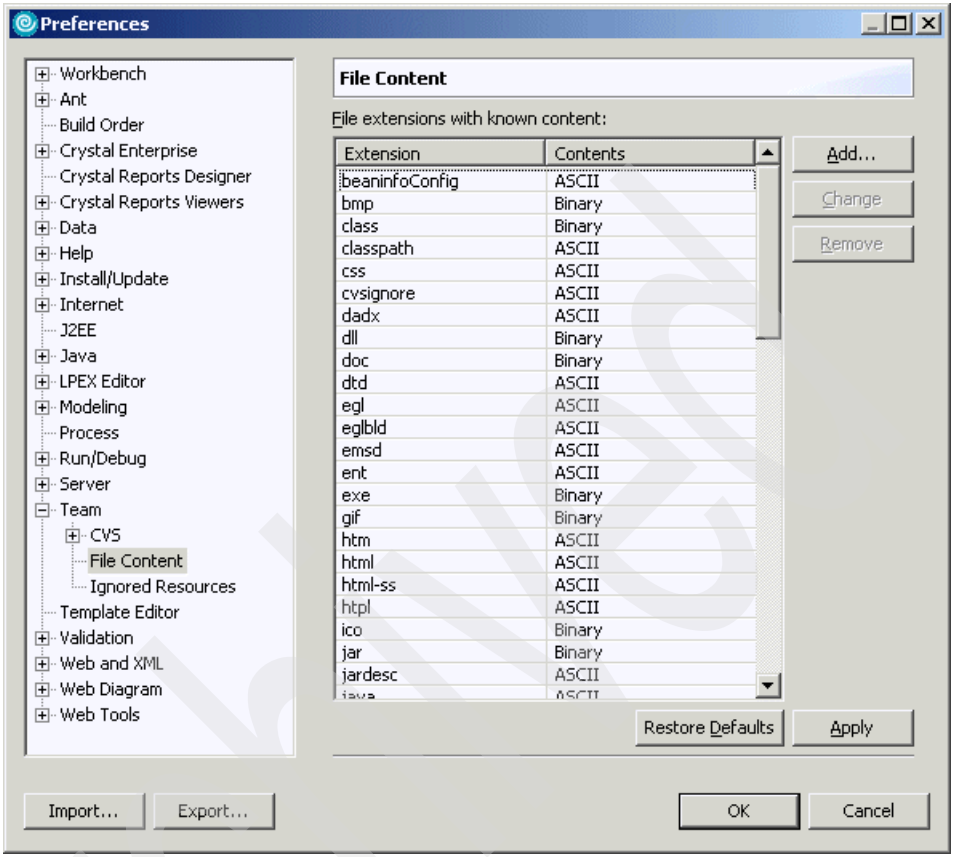


Figure 26-8 Team File Content preferences

If you find that a particular file extension is not in the list then you will need to add this extension if you do not want the resource stored with the default binary behavior. A common file that sometimes occurs when you are supplied a library is a Makefile.mak file used in making applications, which is an ASCII file.

To demonstrate adding this extension that is not present in this list, perform the following:

1. Select **Windows** → **Preferences** and expand the **Team** → **File Content** section.
2. Select the **Add** button.
3. Type the extension name `mak` (see Figure 26-9 on page 1315) and click **OK**.

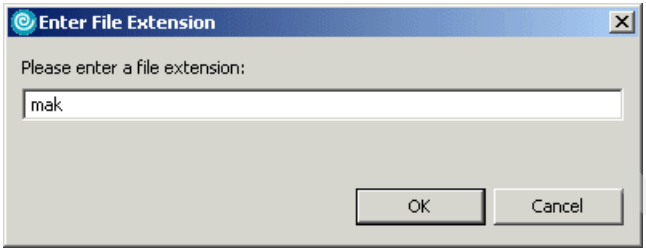


Figure 26-9 Entering a file extension

4. Find the extension in the list, click the **Content** column, and select **ASCI** from the drop-down.

> **Tip:** The content can also be changed by highlighting the extension and using the **Change** button.

5. Click **Apply** and then **OK** to exit the Windows preferences.

Ignored resources

Resources that are created or changed dynamically via mechanisms such as compilation or builds are not recommended to be saved in the repository. This may include class files, executables, lexer and parser code, and Enterprise Java Bean stub and implementation code.

IBM Rational Application Developer V6 has a list of these resources that is accessed by selecting **Windows** → **Preferences** and expanding the **Team** → **Ignored Resources** section.

Resources can be added to this list by specifying the pattern that will be ignored. The two wild card characters are a asterisk (*), which indicates a match of zero; or many characters and a question mark (?), which indicates a match of a character. For example, a pattern of _EJS*.java would match any file that begin with _EJS and had zero to many characters and ended in .java.

The addition of resources that need to be ignored for saving into the repository can be performed as follows using the example of a Windows dll file:

1. Select **Windows** → **Preferences** and expand the **Team** → **Ignored Resources** section.
2. Select the **Add** button.
3. Type the pattern `*.dll` (as shown in Figure 26-10 on page 1316) that will be ignored.

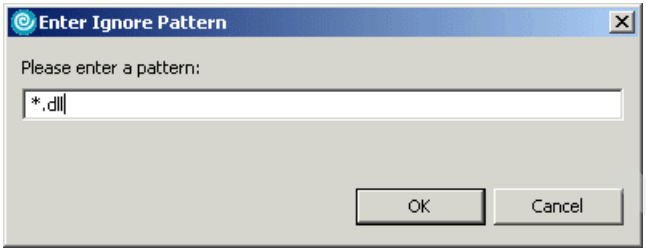


Figure 26-10 Ignored resource pattern to add

4. Click **OK** and ensure that the resource (*.dll) is checked (see Figure 26-11).

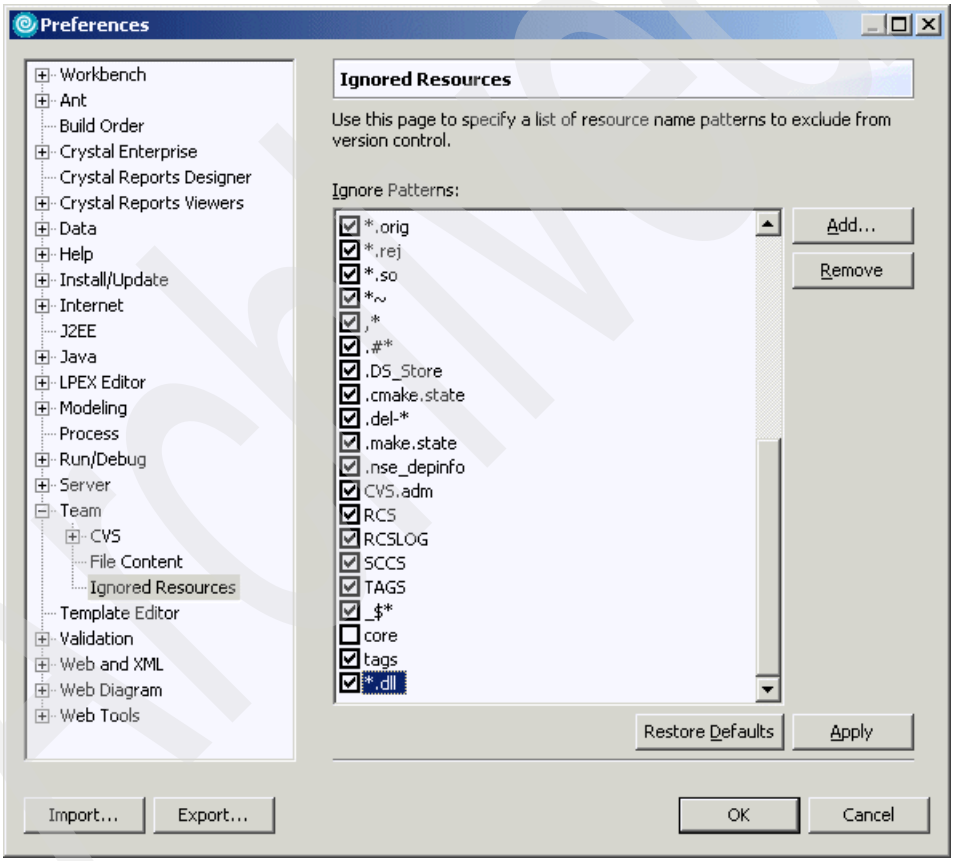


Figure 26-11 Resources that will be ignored when saving to the repository

Archived

Patterns to be removed from the ignore list can be selected and the **Remove** button clicked (see Figure 26-11 on page 1316). Alternatively, you can temporarily disable ignoring the file pattern by de-selecting it from the list, and you do not have to remove the specified file pattern from the list.

Additionally, there are two further facilities that can be used to exclude a file from version control:

- Resources marked as derived will automatically be ignored. This is implemented by builders in the Eclipse framework, such as the Java builder.
- Use of a .cvsignore file created in the same directory that it applies to. This file will contain a list of files or directories that should not be placed into the repository and is specific to CVS.

  This can be created by using the wizards to create a simple file or by selecting the resource, right-clicking, and selecting **Team** → **Add to .cvsignore**.

Further details on the syntax of .cvsignore can be found at:

`http://www.cvshome.org`

CVS-specific settings

The CVS settings in IBM Rational Application Developer V6 are extensive and cannot be covered in full here. A list of categories for the CVS settings is provided in Table 26-1 with short descriptions. Some of the more important settings are highlighted to assist users; a description of the remaining settings can be obtained from the IBM Rational Application Developer V6 help system.

Table 26-1 Category of CVS settings available

Category	Menu location	Description
General CVS Settings	**Windows** → **Preferences** → **Team** → **CVS**	Settings for the default behavior in communicating with CVS
Console	**Windows** → **Preferences** → **Team** → **CVS** → **Console**	Settings for the colors to display in the CVS console and the flag to set the console display
Ext Connection Method	**Windows** → **Preferences** → **Team** → **CVS** → **Ext Connection Method**	Settings to identify the ssh external program and associated parameters

Archived

Category	Menu location	Description
Label Decorations	**Windows → Preferences → Team → CVS → Label Decorations**	Settings for displaying the state of resources in IBM Rational Application Developer V6
Password Management	**Windows → Preferences → Team → CVS → Password Management**	Manages the passwords required to connect to multiple CVS repositories
SSH2 Connection Method	**Windows → Preferences → Team → CVS → SSH2 Connection Method**	Configuration settings for SSH2 protocol to the CVS repository
Watch/Edit	**Windows → Preferences → Team → CVS → Watch/Edit**	Settings for the CVS watch and edit functionality

CVS keyword substitution

The key attributes of software development require that configuration management and versions be maintained. CVS provides a mechanism for identifying the version of the source code and other related information that is stored in the repository. This information can be accessed by developers by the defined keywords. This is known as keyword expansion.

Keyword expansion is an effective mechanism for identifying what version a resource is in the repository versus what a user has checkout locally on their workspace.

IBM Rational Application Developer V6, by default, has the keyword substitution set to *ASCII with keyword expansion (-kkv)* under the selection **Windows → Preferences → Team → CVS → Console**. This setting expands out keyword substitution based on what CVS understands, and is performed wherever they are located in the file.

Some of the available keywords (case sensitive) are listed in Table 26-2.

Table 26-2 CVS keywords

Keyword	Description
$Author$	Expands to including the name of the author of the change in the file, for example: `$Author: itsodev $`
$Date$	Expands to the date and time of the change in UTC, for example: `$Date: 2004/10/29 18:21:32 $`

Keyword	Description
$Header$	Contains the RCS file in repository, revision, date (in UTC), author, state and locker, for example: `$Header: /rep6449/XMLExample/.project,v 1.1 2004/10/29 18:21:32 itsodev Exp itso $`
Id	Like $Header$ except without the full path of the RCS file, for example: `$Id: .project,v 1.1 2004/10/29 18:21:32 itsodev Exp itso $`
$Locker$	Name of the user that has a lock on this revision. (In CVS this is not applicable.)
Log	The log message of this revision. This does not get replaced but gets appended to existing log messages. In general, this is not recommended since files can become large for no real benefit.
$Name$	Expands to the name of the sticky tag, which is a file retrieved by date or revision tags, for example: `$Name: version_1_3 $`
$RCSFile$	Expands to the name of the RCS file in the repository, for example: `$RCSFile: .project,v $`
$Revision$	Expands to the revision number of the file, for example: `$Revision: 1.1 $`
$Source$	Expands to the full path of the RCS file in the repository, for example: `$Source: /rep6449/XMLExample/.project,v $`
$State$	Expands to the state of the revision, for example: `$State: Exp $`. This is not commonly used.

To ensure consistency between multiple users working on a team, it is recommended that a standard header is defined for each Java code file that is created and is filled inappropriately. A simple example is shown in Example 26-1 for what could be established.

Example 26-1 Example of CVS keywords used in Java

```
/**
 * class comment goes here.
 *
 * <pre>
 * Date $Date$
 * Id $Id$
 * </pre>
 * @author $Author$
```

```
 * @version $Revision$
 *
 * ${todo} To change the template for this generated type comment go to
 * Window - Preferences - Java - Code Style - Code Templates
 */
```

To ensure consistency across all files created, each user would need to cut and paste this into their document. Fortunately, IBM Rational Application Developer V6 offers a means to ensure that consistency. To set up a standard template do the following:

1. Select **Windows** → **Preferences** → **Java** → **Code Style** → **Code Templates**.
2. Expand out the **Comments** tree.
3. Select **Types** and click **Edit**.
4. Cut and paste or type what comment header you require, as shown in Example 26-2.

Example 26-2 Comment header to paste into IBM Rational Application Developer V6

```
/**
 * class comment goes here.
 *
 * <pre>
 * Date $$Date$$
 * Id $$Id$$
 * </pre>
 * @author $$Author$$
 * @version $$Revision$$
 *
 * ${todo} To change the template for this generated type comment go to
 * Window - Preferences - Java - Code Style - Code Templates
 */
```

Note: The double dollar sign ($$) (as shown in Figure 26-12 on page 1321) is required since IBM Rational Application Developer V6 treats a single dollar ($) as one of its own variables. The double dollar ($$) is used as a means of escaping the single dollar so that it can be post processed by CVS.

Archive

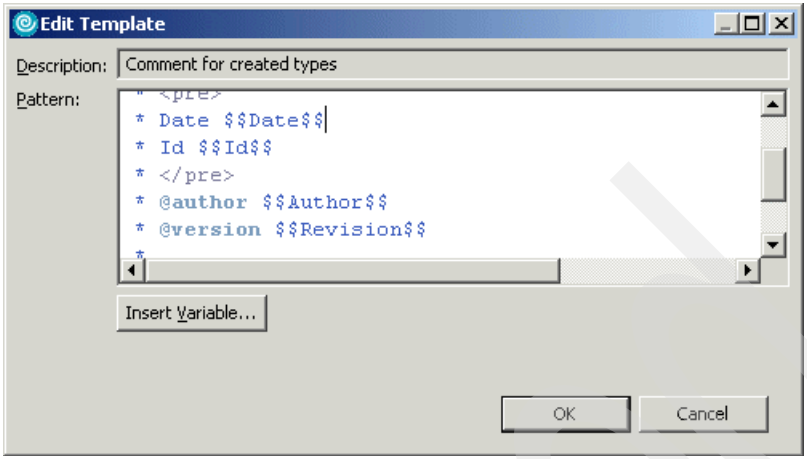


Figure 26-12 Setup of a common code template for Java files

5. Click **OK** to complete the editing.
6. Click **Apply** followed by **OK**.

After performing this operation, creating a new class, and checking in and checking out, the header will be displayed as shown in Example 26-3.

Example 26-3 Contents of Java file after check in and check out from CVS

```
/**
 * class comment goes here.
 *
 * <pre>
 * Date $Date: 2004/10/29 18:21:32 $
 * Id $Id: $Id: Example.java,v 1.1 2004/10/29 18:21:32 itsodev Exp itso $
 * </pre>
 * @author $Author: itsodev $
 * @version $Revision: 1.1 $
 *
 * ${todo} To change the template for this generated type comment go to
 * Window - Preferences - Java - Code Style - Code Templates
*/
```

26.5 Development scenario

To show you how to work with CVS in IBM Rational Application Developer V6, we will follow a simple but typical development scenario, shown in Table 26-3.

Archived

Two developers, cvsuser1 and cvsuser2, work together to create a Servlet *ServletA* and a view bean *View1*.

Table 26-3 Sample development scenario

Step	Developer 1 (cvsuser1)	Developer 2 (cvsuser2)
1	Creates a new Dynamic Web Project *ISTOCVSGuide* and adds it to the version control and the repository. Creates a servlet *ServletA* and commits it to the repository.	
2	Updates the servlet ServletA.	Imports the ISTOCVSGuide CVS module as a Workbench project. Creates a view bean View1, adds it to the version control, and synchronizes the project with the repository.
3	Synchronizes the project with the repository to commit his changes to repository and merges changes.	
4	Continues changing and updating servlet. Synchronizes the project with the repository to commit his changes to repository and merges changes.	Synchronizes the project with repository, and begins changes to servlet. Synchronizes the project after cvsuser1 has committed and needs to merge code from their workspace and the CVS repository.
5	Synchronizes and merges changes. Versions the project.	

Steps 1 through 3 are serial development—no parallel work is being done. During steps 4, 5, and 6, both developers work in parallel, resulting in inevitable conflicts. These conflicts are resolved using IBM Rational Application Developer V6 tooling.

In the sections that follow, we perform each of the steps and explain the team actions in detail.

26.5.1 Create and share the project (step 1 - cvsuser1)

IBM Rational Application Developer V6 offers a perspective specifically designed for viewing the contents of CVS servers: The CVS Repository Exploring.

Add a CVS repository

To add a CVS repository, do the following:

1. Open the CVS Repository Exploring perspective.
2. Select **New** → **Repository Location** and fill in the information (as shown in Figure 26-13) based on the repository set up in 26.2, “CVSNT Server implementation” on page 1301.

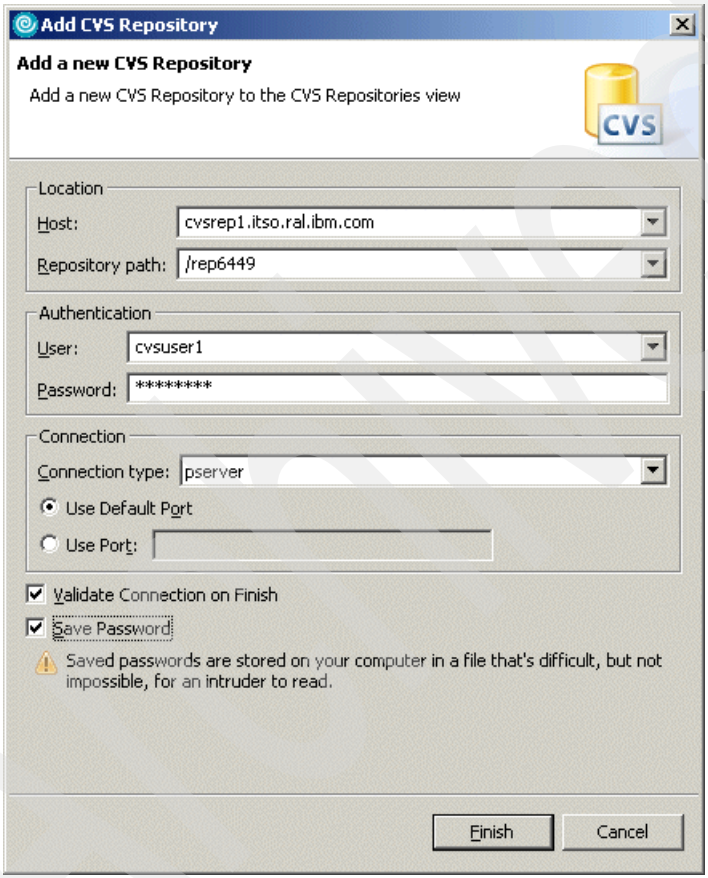


Figure 26-13 Adding a CVS repository

Important: With pserver, passwords are stored on the client side in a trivial encoding and transmitted in the same encoding. The encoding is done only to prevent inadvertent password compromises, and will not prevent an attacker from obtaining the password. The other supported protocol, ssh, does not have this problem, but has to be manually set up.

3. Click **Finish**, and the CVS Repositories view now contains the new repository location (Figure 26-14).

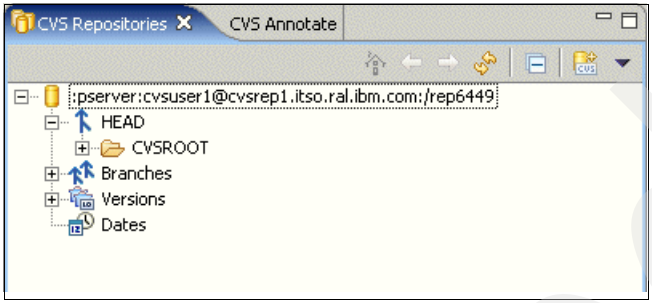


Figure 26-14 CVS Repositories view

Expanding a location in the CVS Repository view reveals branches and versions. A special branch, called HEAD, is shown detached because of its importance. It is the main integration branch, holding the project's current development state.

You can use the CVS Repositories view to check out repository resources as projects on the Workbench. You can also configure branches and versions, view resource histories, and compare resource versions and revisions.

We must first create a project and share it before making full use of the repository.

Create a project and servlet

To create a project and a servlet, do the following:

1. Switch to the Web perspective and create a new Dynamic Web Project by selecting **File** → **New** → **Dynamic Web Project**.
2. Type in the name of the project, `ITSOCVSGuide`, and click **Finish**.
3. In the Project Explorer, expand **Dynamic Web Projects** → **ITSOCVSGuide** → **Deployment Descriptor** → **Servlets**, right-click, and select **New** → **Servlet**.
4. Type the name of the servlet to be `ServletA` and click **Next**.
5. Specify the package as `itso.ral.ibm.com` (see Figure 26-15) and click **Finish**.

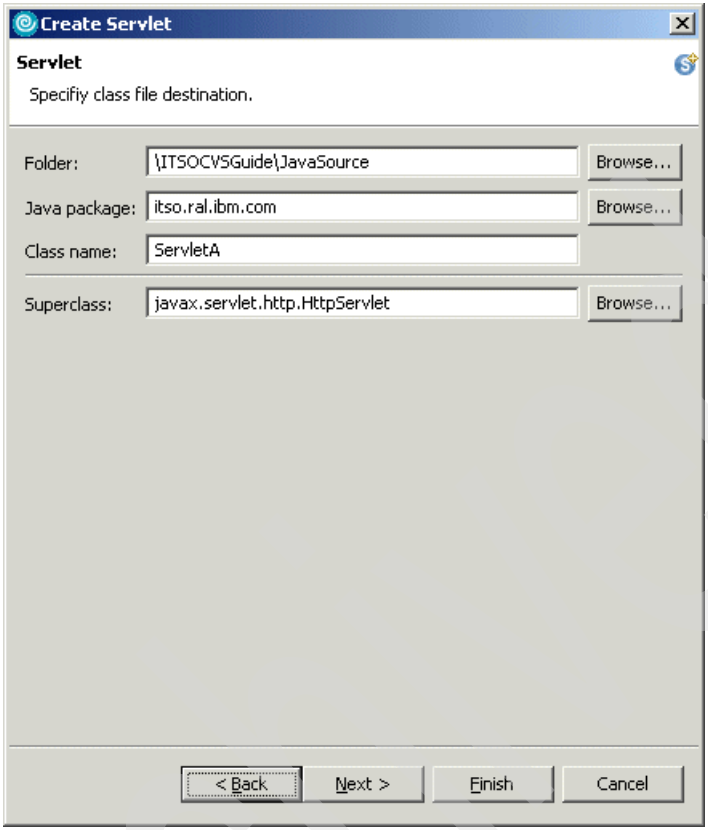


Figure 26-15 Specifying the package name for servlet

6. Expand the **Dynamic Web Projects** tree in the Project Explorer view and click on the project **ITSOCVSGuide**. Right-click and select **Team** → **Share Project**.
7. Select the option **CVS**, and click **Next**.
8. Select the radio button **Use existing repository location:** and the repository that was added previously, and click **Next**.
9. Select the radio button **Use project name as module name**, and click **Next**.
10. The window Share Project Resources (see Figure 26-16 on page 1326) will appear listing the resources to be added in. Click **Finish** to add this into the repository.

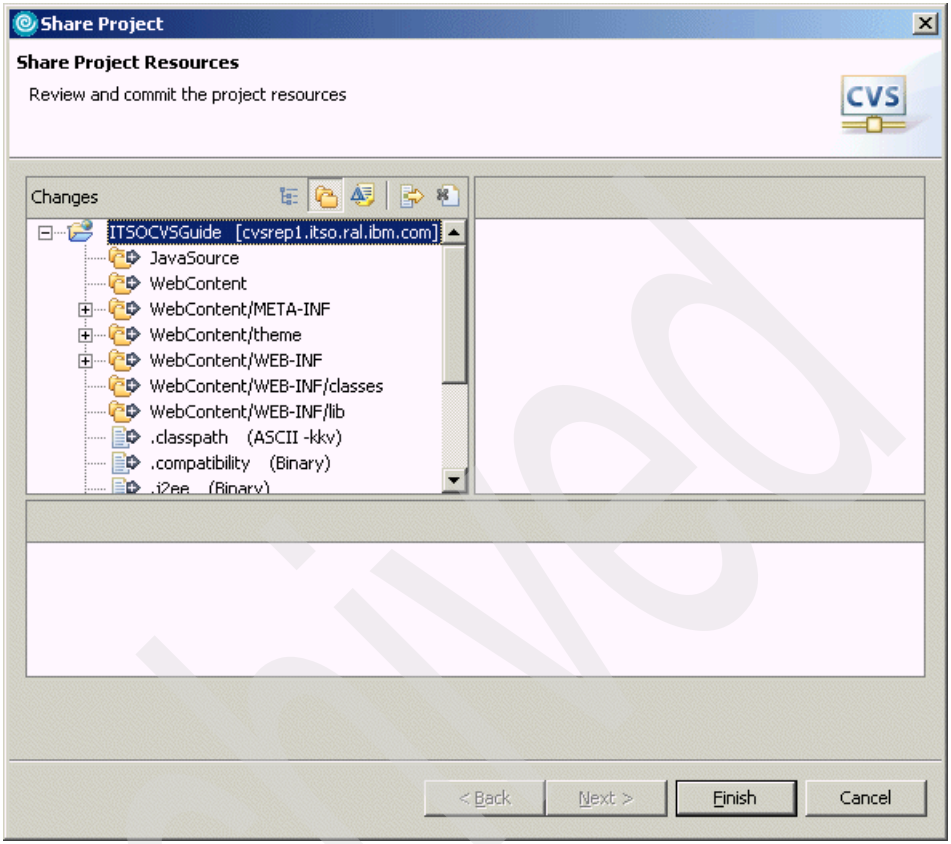


Figure 26-16 Verification of resources to commit under CVS revision control

11. A dialog will ask whether to commit uncommitted changes. Click **Yes**.
12. A dialog will ask whether to add resources to the repository. Click **Yes**.
13. A prompt for a commit comment will be presented. Type `Initial Version` and click **OK**.

> **Note:** A Run Background button is provided on the status when committing. This functionality has been introduced via the Eclipse 3.0 framework into IBM Rational Application Developer V6. This enhances productivity, allowing the user to perform other tasks while waiting for a checkin.

26.5.2 Add a shared project to the workspace (step 2 - cvsuser2)

The purpose of using CVS is to allow multiple developers to work as a team on the same project. We have created the project in one developer's workspace, shared it using CVS, and now wish to add the same project to a second developer's workspace.

1. The second developer must add the CVS repository location to the workspace using the CVS Repositories view in the CVS Repository Exploring perspective, as described in "Add a CVS repository" on page 1323.

   The difference is now that the HEAD branch in the repository, if expanded, contains the ITSOCVSGuide module, as shown in Figure 26-17.

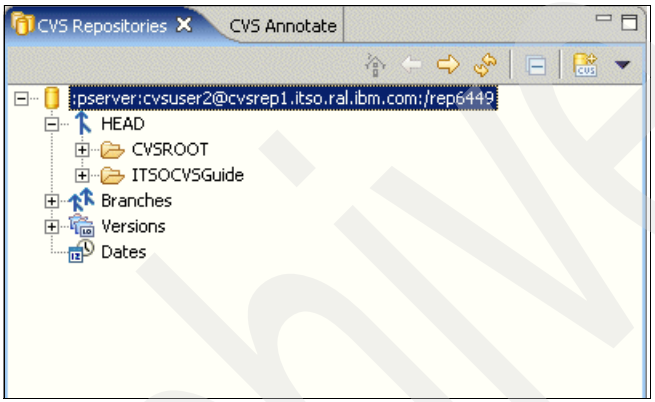


Figure 26-17 CVS Repository with ITSOCVSGuide project

2. Select the ItsoProGuideCVS module, right-click, and click **Check Out**. The current project in the HEAD branch is added to the workspace.

Develop the viewbean

Now that both developers have exactly the same synchronized HEAD branch of the ITSOCVSGuide project on their workspaces, it is time for the second developer to create the viewbean View1.

1. Select **Window** → **Open Perspective** → **Other** → **Java** and click **OK**.
2. Expand the project tree **ITSOCVSGuide** → **JavaSource** → **itso.ral.ibm.com** and select **itso.ral.ibm.com**. Right-click and select **New** → **Class**.
3. Type the name of the class to be `View1` and click **Finish** (see Figure 26-18 on page 1328).

Archived

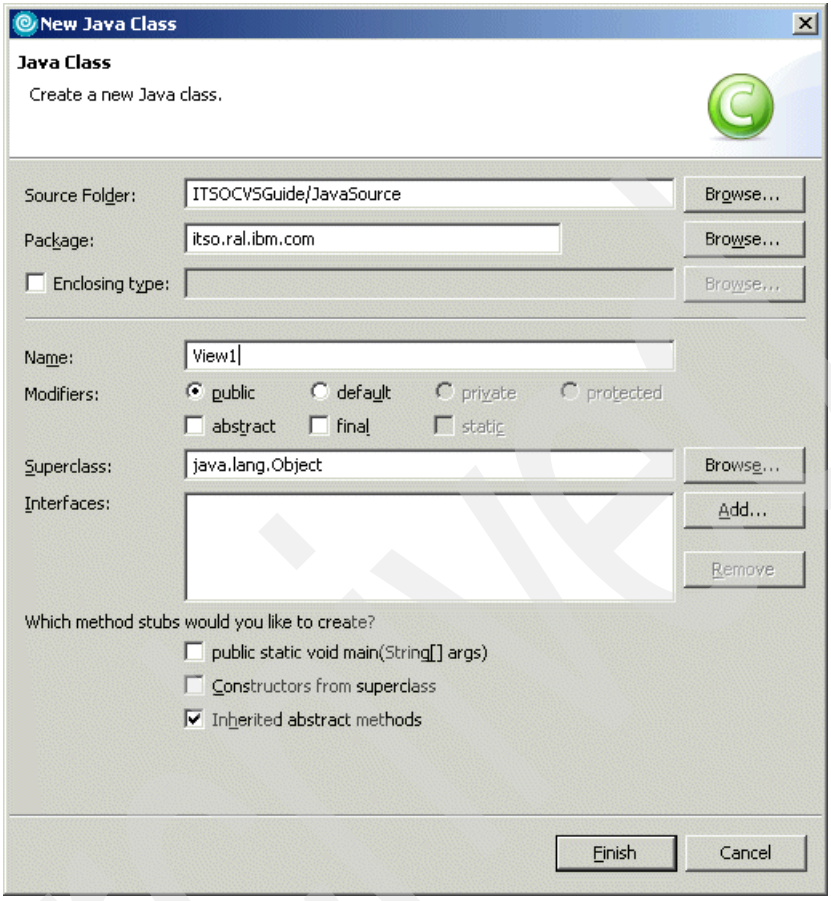


Figure 26-18 Creating the View1 viewbean

4. Create two private attributes in the class an integer named `count` and a string named `message`, as shown in Figure 26-19 on page 1329.

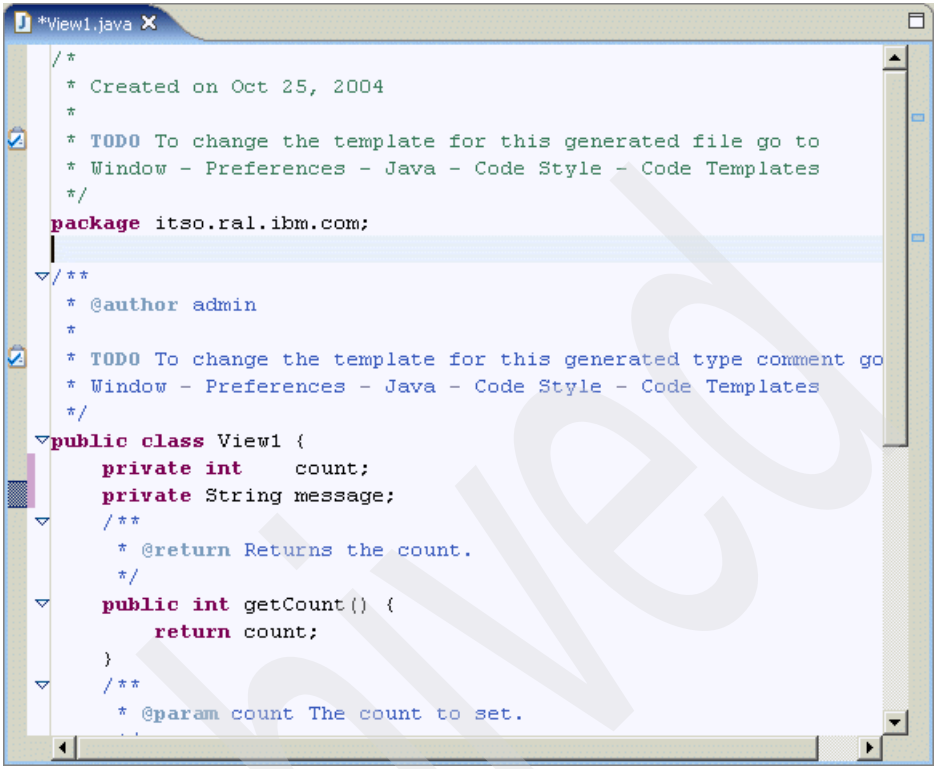


Figure 26-19 View1 code with attributes added in before a save

5. Double-click the **count** attribute to highlight it, right-click, select **Source** → **Generate Getters and Setters...,** and check the check box for the *message* and verify that the Access Modifier section has *public* set, as in Figure 26-20 on page 1330. Click **OK** to complete.

Archived

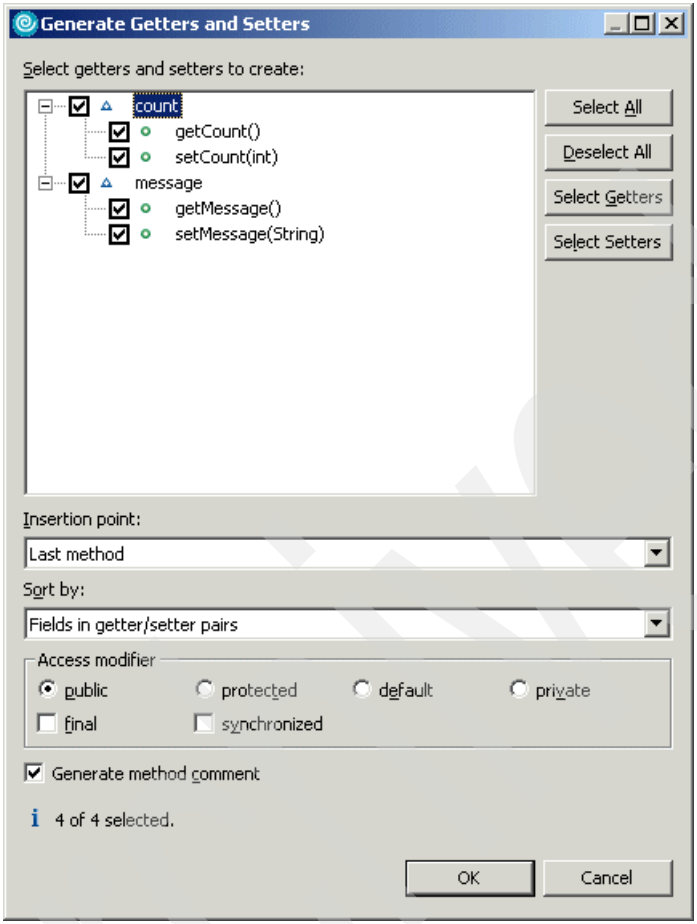


Figure 26-20 Creating setters and getters for class View1

6. Save the file by selecting **File** → **Save**.

Tip: The greater than sign (>) in front of a resource name means that the particular resource is not synchronized with the repository. You can always use this visual cue to determine when a project requires synchronization.

Synchronizing with the repository

To update the repository with these changes, perform the following:

1. Select the **ITSOCVSGuide** project and select **Team** → **Synchronize with Repository...** by clicking the right button. A dialog will prompt you to change to the Synchronize view. Click **Yes**. The project is compared with the

repository, and the differences are displayed in the Synchronize view (Figure 26-21).

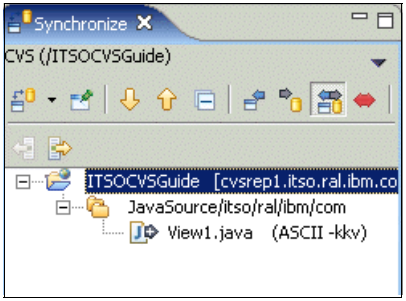


Figure 26-21 Synchronizing ITSOCVSGuide after creating the viewbean View1

This view allows you to update resources in the Workbench with newer content from the repository, commit resources from the Workbench to the repository, and resolve conflicts that may occur in the process. The arrow icons with a plus sign (+) indicate that the files do not exist in the repository.

2. Add these new resources to version control by selecting **ITSOCVSGuide** in this view by right-clicking and selecting **Commit...** .
3. When the **Add to CVS Version Control** dialog appears, click the **Details>>>** button to verify the changes and then click **Yes** (see Figure 26-22).

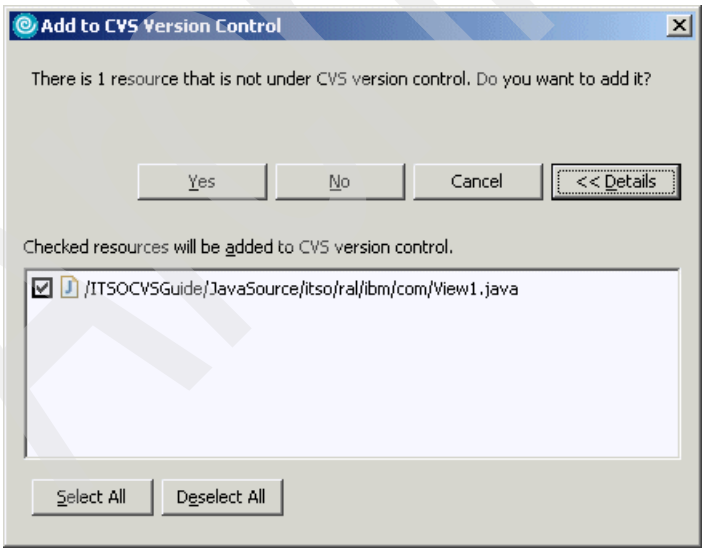


Figure 26-22 Verifying committing of resources into repository

4. The Commit dialog box will appear prompting for a comment. Enter `Added view bean for application` and click **OK**.

Note: The font in the Synchronize view turns to *italic* to indicate that there is an activity that is in progress on these files. This is useful when you are checking a large quantity of files into the CVS and have background mode on.

26.5.3 Modifying the Servlet (step 2 - cvsuser1)

While activities in the section Add a shared project to the workspace (step 2 - cvsuser2) occur, our original user, cvsuser1, is working on developing the servlet further.

1. Select the Web Perspective by clicking **Window** → **Open Perspective** → **Other** → **Web**.
2. In the Project Explorer expand **Dynamic Web Projects** → **ITSOCVSGUIDE** → **Java Resources** → **JavaSource** → **itso.ral.ibm.com** and double-click **ServletA.java** to open in an editor.
3. Create a static attribute called totalCount of type int and initialized to zero, as in Figure 26-23 on page 1333.

Archived

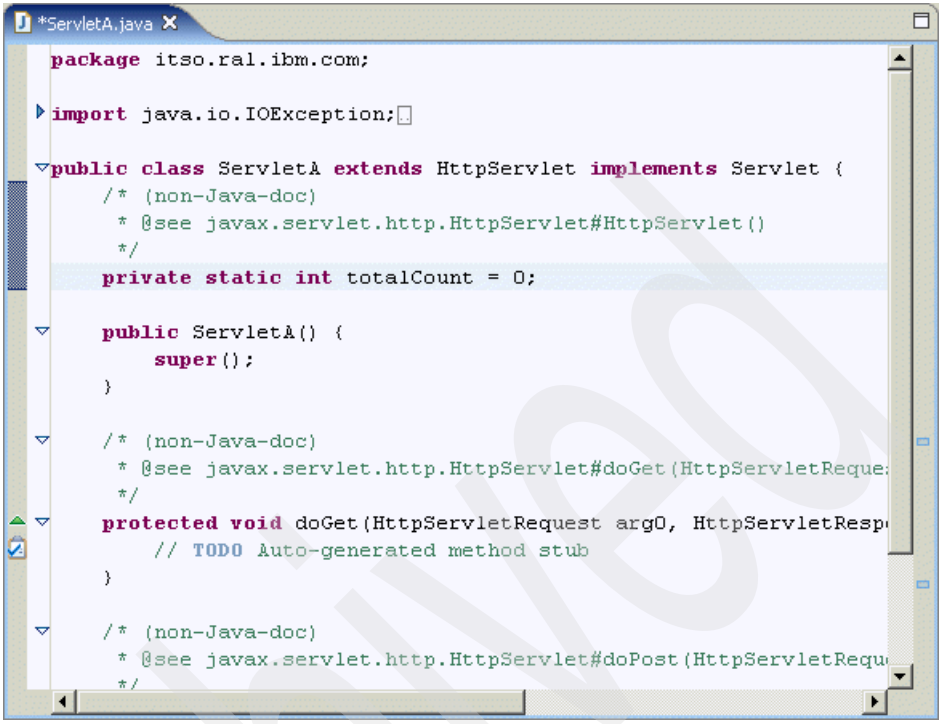


Figure 26-23 Addition of servlet attributes

Note: Files that are not saved have a asterisk (*) in front of their names in the title bar of the window. This assists in identifying resources that need to be saved.

4. Save your work using **File** → **Save**.

26.5.4 Synchronize with repository (step 3 - cvsuser1)

The first user now synchronizes with the repository after cvsuser2 has checked their changes.

1. Open the Web perspective using **Windows** → **Open Perspective** → **Other** → **Web**.
2. Expand the **Dynamic Web Project** tree in the Project Explorer view and select the **ITSOCVSGuide** project. Right-click and select **Team** → **Synchronize with Repository...**.
3. Click **Yes** to switch to the Synchronize view.

4. Expand out the **ITSOCVSGuide** → **JavaSource** tree to view the changes. The screen in Figure 26-24 should be presented to you.

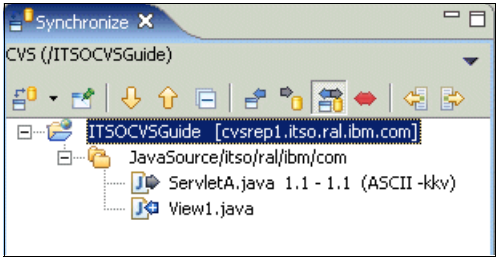


Figure 26-24 User cvsuser1 merging with CVS repository

> **Note:** The symbol in the diagram indicates that an existing resource differs from what is in the repository. The symbol indicates that a new resource is in the repository that does not exist on the local workspace.

5. To obtain updated resources from the CVS repository, right-click the project and select **Update**.
6. Verify that the changes do not cause problems with existing resources in the local workspace. In this case, there are none. Right-click and select **Commit...**.
7. In the Commit dialog, add the comment `Added static count variable to servlet.` and click **OK** (see Figure 26-25).

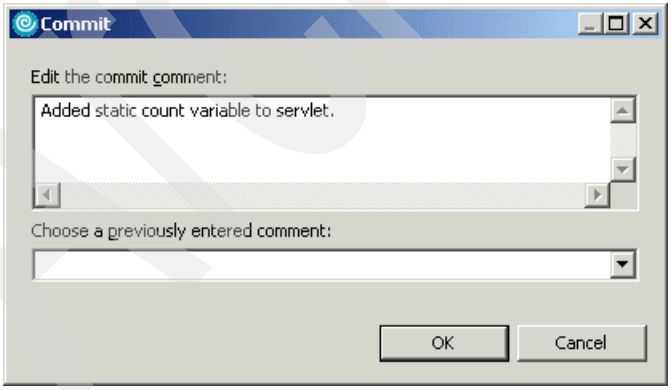


Figure 26-25 Adding comment for changes to servlet

The repository now has the latest changes to the code from both developers. The user cvsuser1 is in sync with the repository; however, cvsuser2, as yet, does not have the changes to the servlet.

26.5.5 Parallel development (step 4 - cvsuser1 and cvsuser2)

The previous steps have highlighted development and repository synchronization with two people working on two parts of a project. It highlights the need to synchronize between each phase in the development before further work is performed. This scenario highlights two developers working on the same area of the code, with one checking before the other. Each user's sequence of events is described in the sections below; however, to understand what is occurring refer to the timeline shown in Figure 26-26.

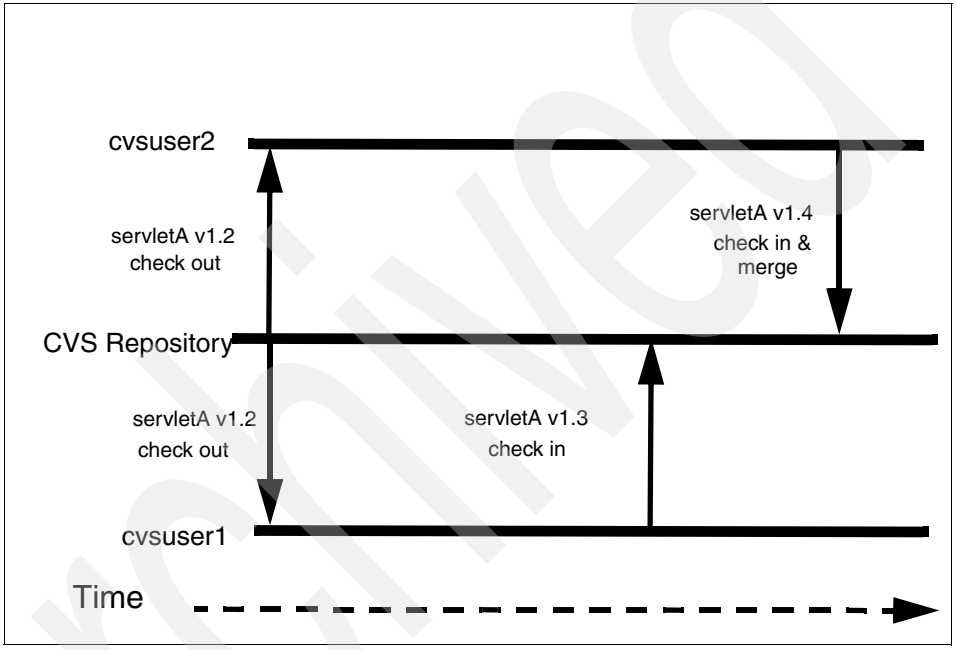


Figure 26-26 Parallel concurrent development of same resource by multiple developers

User cvsuser1 updates and commits changes

In this scenario, the user cvsuser1 modifies the dopost method to log information for an attribute. The following procedure demonstrates how to synchronize the source code and commit the changes to CVS.

1. Open the Web perspective using **Windows** → **Open Perspective** → **Other** → **Web**.
2. Open the tree under the **Dynamic Web Projects** → **ITSOCVSGuide** → **Java Resources** → **JavaSource** → **itso.ral.ibm.com** and double-click the file **ServletA.java** to open it.
3. Navigate to the doPost method by scrolling down the file and adding the code in Figure 26-27 on page 1336.

Archived

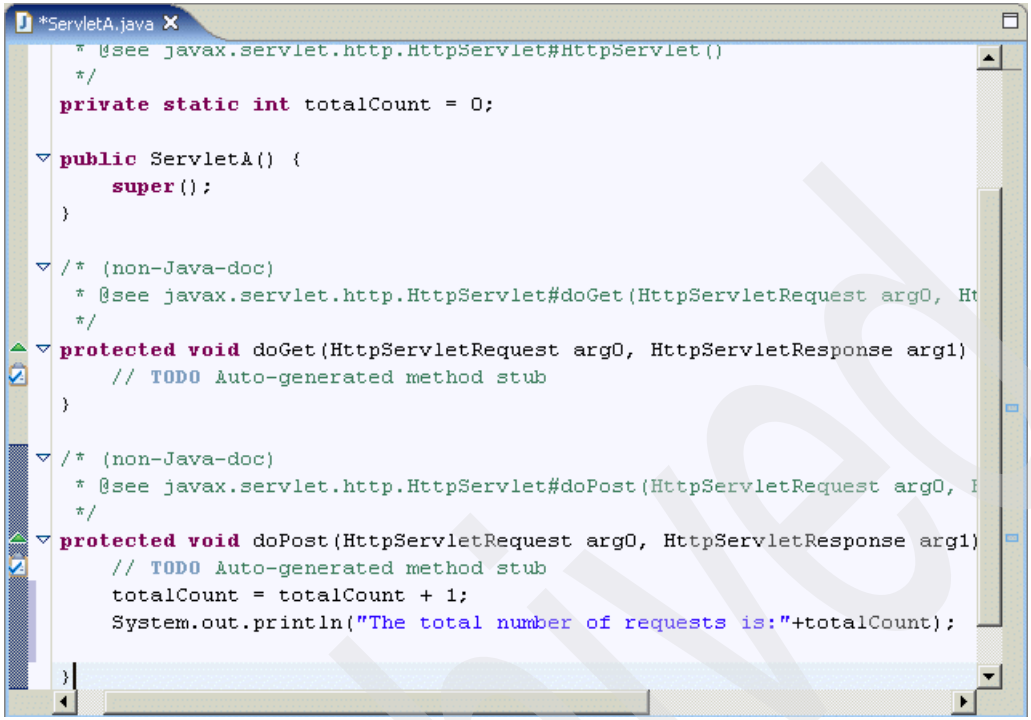


Figure 26-27 User cvsuser1 adding code to servlet in the local repository

4. Save the file by clicking **File** → **Save**.
5. Synchronize the project with the repository by right-clicking and selecting **Team** → **Synchronize Repository...,** and click **OK** to open synchronize view.
6. Fully expand out the tree in the Synchronize view. The servlet should be the only change, as shown in Figure 26-28.

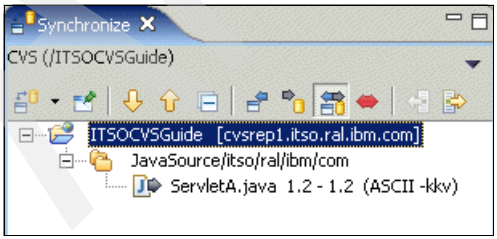


Figure 26-28 Changes in servlet from the repository

7. Right-click and select **Commit...**, add the comment `User1 updating the servlet`, and press **OK** to commit.

The developer cvsuser1 has now completed the task of adding code into the servlet. Changes can be picked up by other developers in the team.

User cvsuser2 updates and commits changes

The second developer updates their repository before beginning any new work to ensure that they have the latest copy of the code. This occurs before the first developer has checked in the changes to the servlet and while they are making their changes. The following steps are performed:

1. Open the Web perspective using **Windows** → **Open Perspective** → **Other** → **Web**.
2. Open the tree under **Dynamic Web Projects** → **ITSOCVSGuide** → **Java Resources** → **JavaSource itso.ral.ibm.com** and double-click the file **ServletA.java** to open it.
3. Navigate to the doPost method by scrolling down the file and adding the code in Figure 26-29. Save the code by selecting **File** → **Save**.

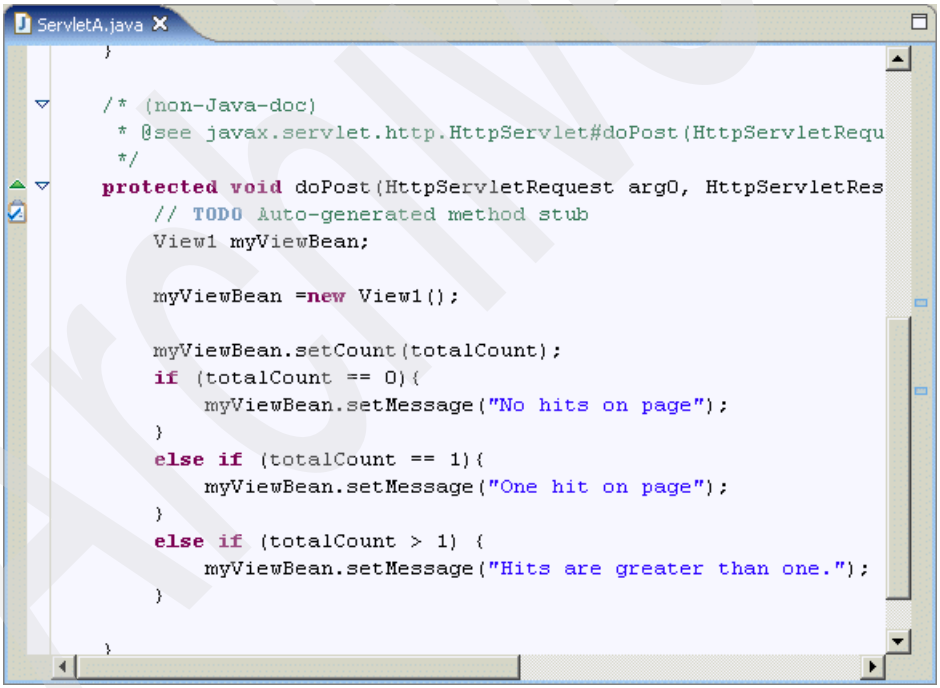


Figure 26-29 User cvsuser2 adds code into the servlet

4. Synchronize with the repository by clicking the project **ITSOCVSGuide** and clicking the right button and selecting **Team** → **Synchronize with Repository...**.

Archived

5. Click **Yes** to switch to the synchronize perspective and expand out the tree in the synchronize view to see change, as in Figure 26-30.

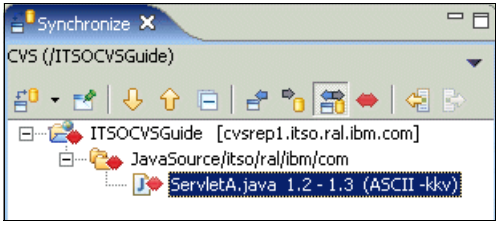


Figure 26-30 Conflict of file in CVS Repository for user cvsuser2

> **Note:** The symbol indicates that the file has conflicting changes that require merging.

6. Double-click the file **ServletA.java** to see the changes, as shown in Figure 26-31 on page 1339. On the right-hand side is the code in the repository (checked in by user cvsuser1), and on the left are the changes made by the current user cvsuser2.

   Merging in this case will require consolidation between the two developers as to the best solution. In our example, we assume that the changes in the repository need to be placed before changes performed by cvsuser2 (left-hand side).

Archived

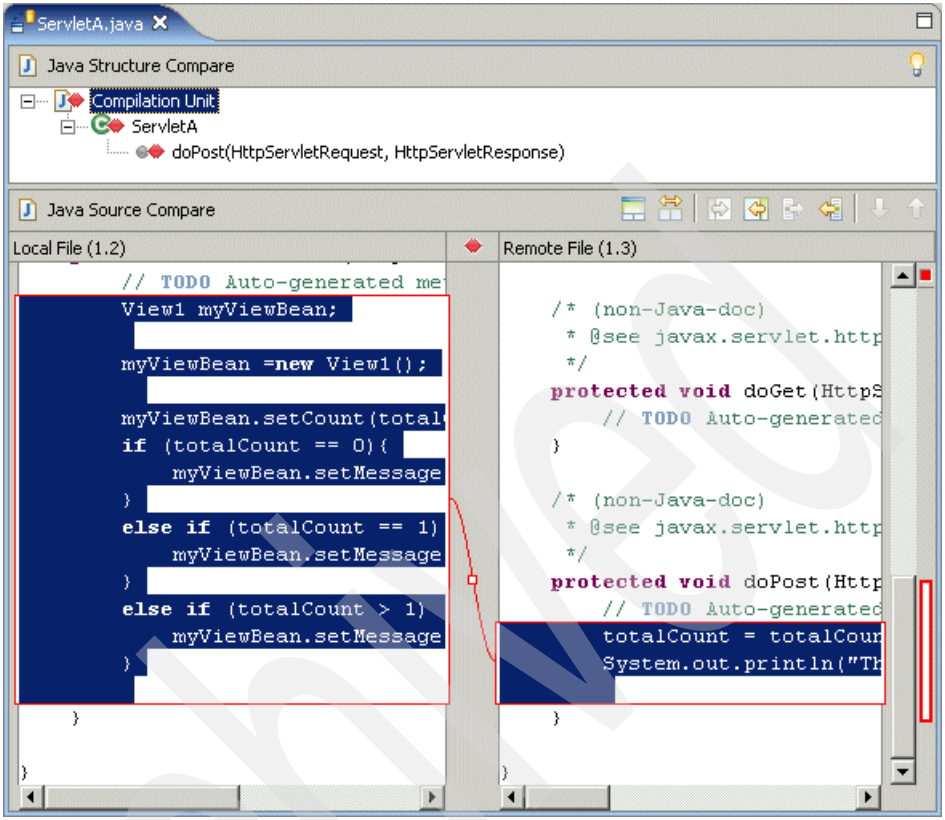


Figure 26-31 The changes between the local and remote repository

7. Click the icon (Copy current change from Right to Left). This will place the change at the end of the right-hand screen, as shown in Figure 26-32 on page 1340.

Archived

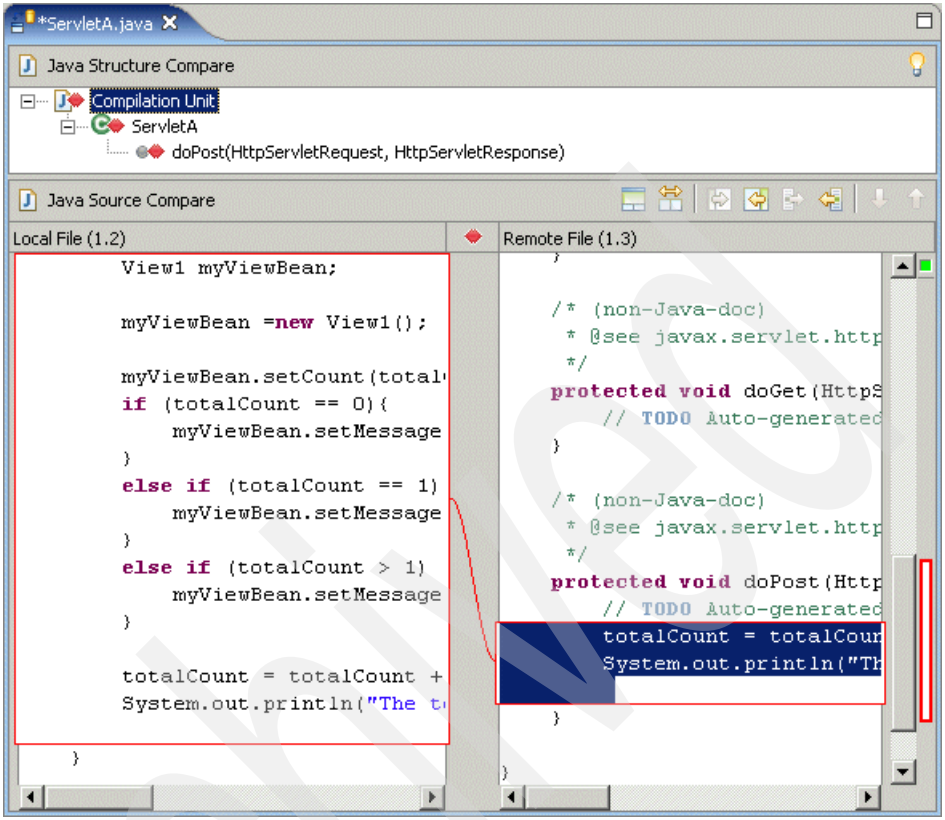


Figure 26-32 Merging changes from right to left

8. In the left pane, highlight the two lines of code added and cut and paste them to the correct location in the file, as shown in Figure 26-33 on page 1341.

Archived

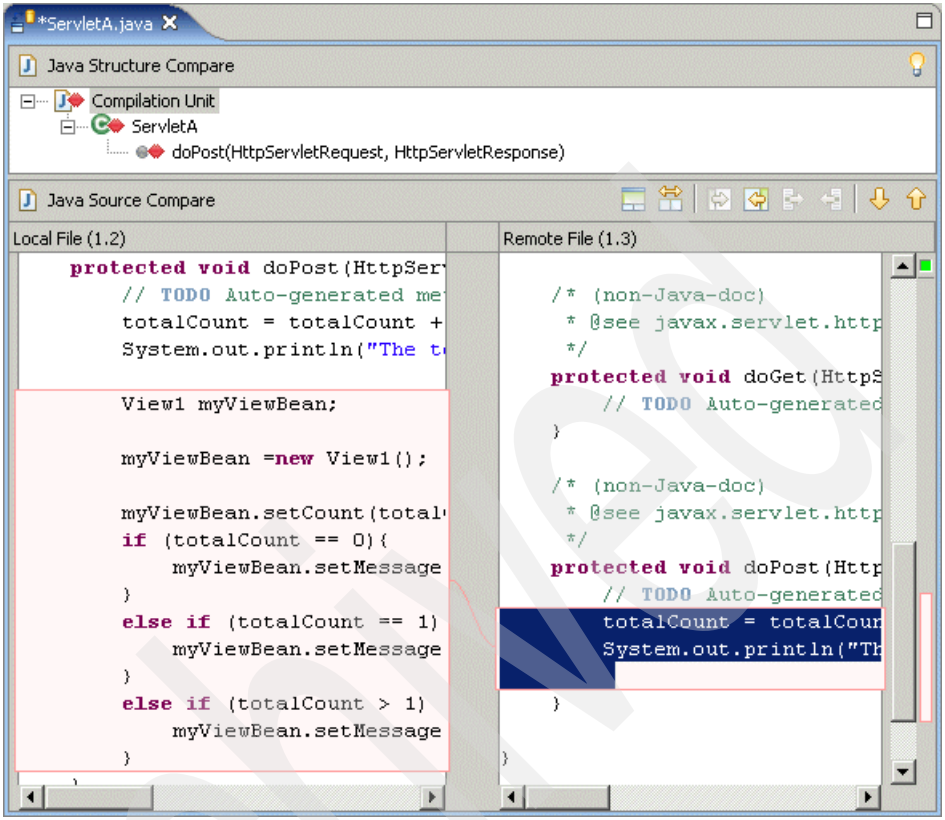


Figure 26-33 Move the added code to the correct merge point

9. Verify that the code is exactly as agreed by the developers, and if so save the new merged change using **File** → **Save**.
10. Resynchronize the file using steps 4, 5, and 6 above in the Web perspective.
11. Verify that the changes are correct, and then in the Synchronize view right-click and select **Mark as Merged**, and then right-click and select **Commit**.
12. In the Commit dialog that appears, type the comment `Changes and merge of Servlet`, as in Figure 26-34 on page 1342.

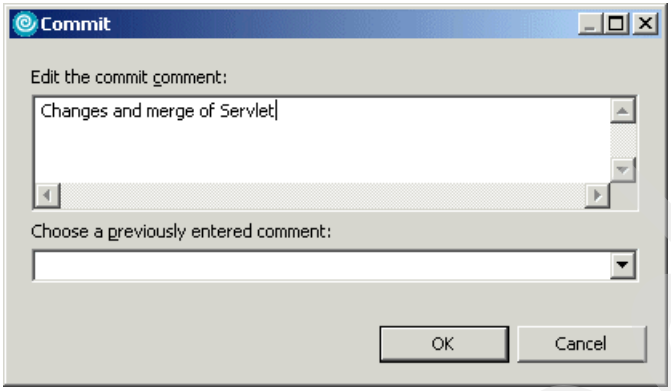


Figure 26-34 Comment for merged changes

This operation creates a version of the file, Version 1.4, which will contain the merged changes from users cvsuser1 and cvsuser2, in spite of the fact that both developers began working with Version 1.2.

26.5.6 Versioning (step 5- cvsuser1)

Now that all the changes are synchronized with the repository, we want to create a version to milestone our work.

1. Select the **ITSOCVSGuide** project in the Web perspective and Project explorer view and select **Team** → **Tag as Version...**. The Tag Resources dialog opens (seeFigure 26-35).

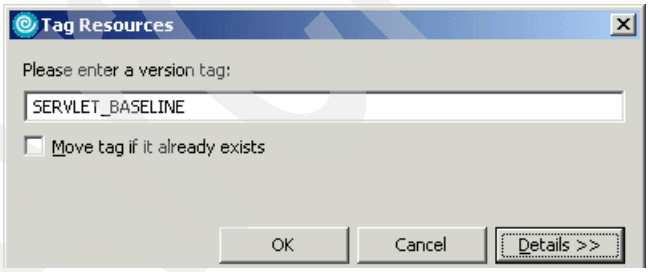


Figure 26-35 Tagging the project as a version

2. Verify that the tag has been performed by switching to the CVS Repository Exploring perspective and expand out the repository, as shown in Figure 26-36 on page 1343.

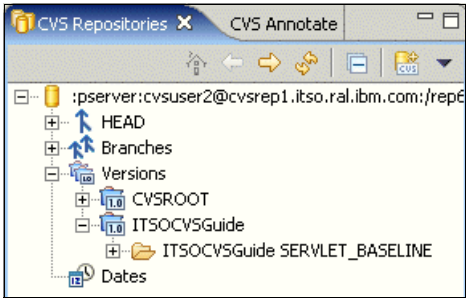


Figure 26-36 Project version

26.6 CVS resource history

The developer can view the resource history of a file of any shared project in their workspace. This is the list of all the revisions of a resource in the repository, shown in the CVS Resource History view. From this view you can compare two revisions, replace or revert the corresponding workspace file to the revision, or open an editor on a revision.

1. Open the Web perspective using **Window** → **Open Perspective** → **Other** → **Web**.
2. Expand in the Project Explorer **Dynamic Web Projects** → **ITSOCVSGuide** → **Java Resources** → **JavaSource** → **itso.ral.ibm.com** and highlight **ServletA.java**.
3. Right-click and select **Team** → **Show in Resource History** and a new view will appear, as in Figure 26-38 on page 1345.

   The CVS resource history will display the following information (Table 26-4 on page 1343).

Table 26-4 CVS resource history terminology

Resource History column	Description
Revision	The revision number of each version of the file in the repository. An asterisk (*) indicates that this is the current version in the workspace.
Tags	The tags associated with the revision. Selecting a revision line with a tag will display the tag in the lower pane of the view.
Date	The date and time when the revision was created in the repository.

Archived

Resource History column	Description
Author	The name of the used ID that created and checked in the revision into the repository.
Comments	The comment (if any) supplied for this revision at the time it was committed. Selecting a revision in the upper pane displays the comment in the lower right pane of the view.

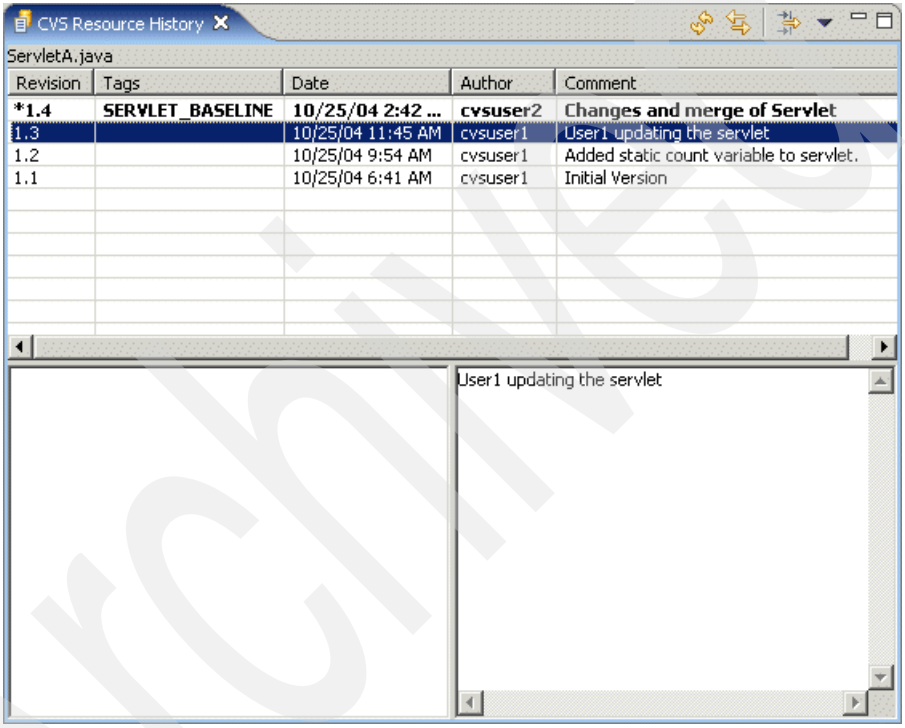


Figure 26-37 CVS Resource view for ServletA.java

26.7 Comparisons in CVS

Users on occasion require the ability to compare what changes have occurred with a version of their code against a version in the repository. There are two ways of comparing: One is to compare a file in the workspace with that in the repository; the other is to compare two files in the repository. To demonstrate these features a scenario has been provided as an example of both processes.

Archives

26.7.1 Comparing workspace file with repository

The developer has Version 1.4 of the ServletA.java resource and wants to compare the differences between their current version and 1.1 to understand changes made. They would perform the following:

1. Open the Web perspective using **Window** → **Open Perspective** → **Other** → **Web**.
2. Expand in the Project Explorer **Dynamic Web Projects** → **ITSOCVSGuide** → **Java Resources** → **JavaSource** → **itso.ral.ibm.com** and highlight **ServletA.java**.
3. Right-click and select **Compare with** → **Revision...**, and a screen similar to the CVS Resource History will display, as in Figure 26-38.

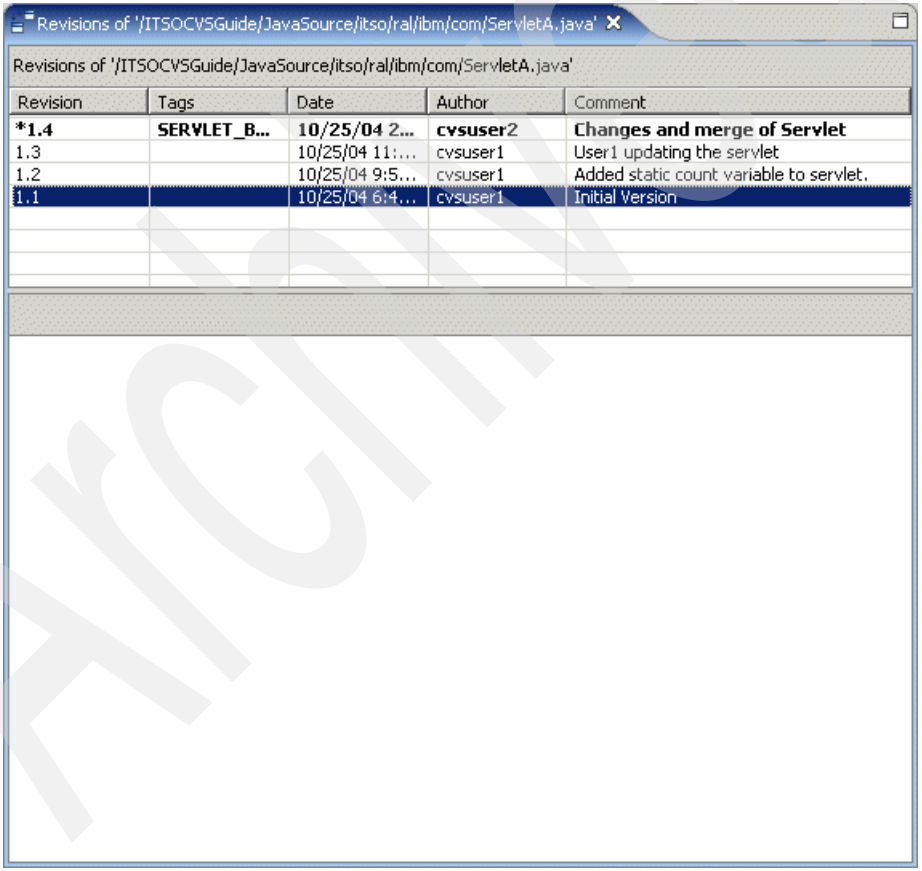


Figure 26-38 List of revisions for ServletA.java

Archive

4. Double-click the revision 1.1 version and a comparison will appear in the pane below, as in Figure 26-39.

   The changes are then displayed in a few ways. In the top right-hand corner the changes to the class are shown, which include attribute changes, and identifies the methods that have changed, while in the bottom two panes the actual code differences are highlighted. The left bottom pane has the revision in the workspace and the right bottom pane has the revision 1.2 from the repository.

   Note: The bars in the bottom pane on the right-hand side indicate the changes located in the file. By clicking them they will position the pane to highlight the changes and assist in quickly moving around large files with many changes.

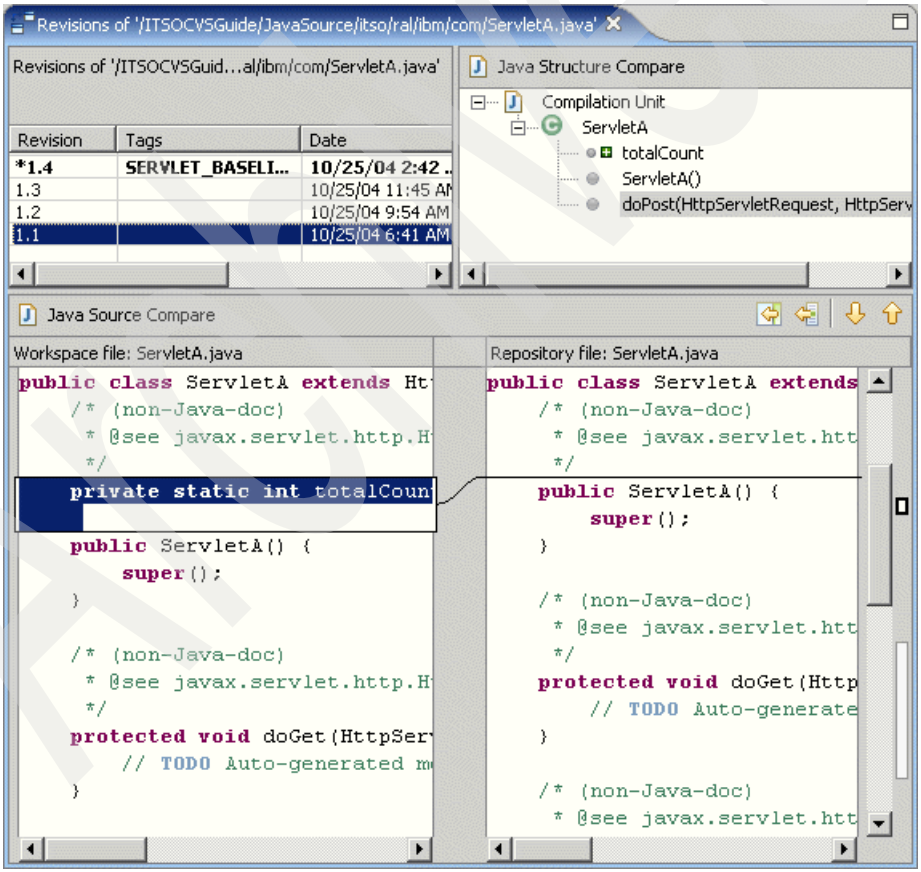


Figure 26-39 Comparison between Version 1.4 and Version 1.1 of ServletA.java

26.7.2 Comparing two revisions in repository

A developer wants to compare the differences between revision 1.1 and 1.3 in the repository of the file ServletA.java, and they have version 1.4 in their workspace and do not want to remove it. The procedure to follow is:

1. Open up the CVS Resource History using the procedure in “CVS resource history” on page 1343, which would display the view shown in Figure 26-37 on page 1344.
2. Click in the row of the first version you want to compare, say revision 1.1, and then, while pressing the Ctrl key, click in the row of the second version, which is 1.3, so that the screen looks as in Figure 26-40.

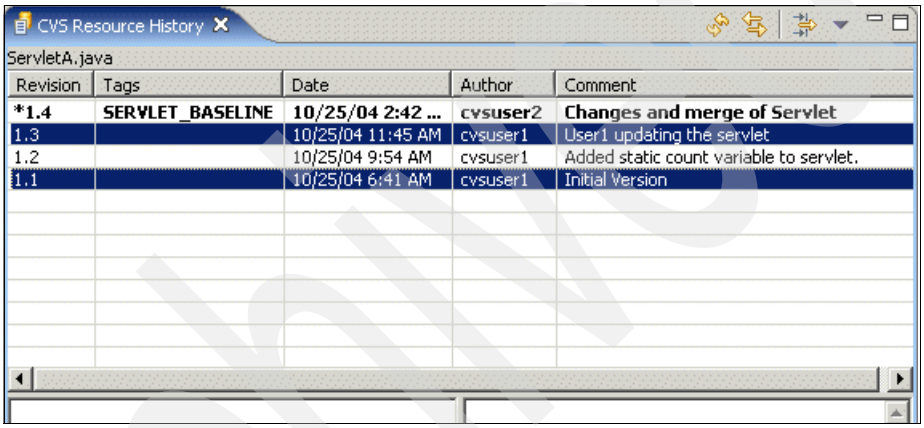


Figure 26-40 Highlight the two revisions to compare

3. Right-click, ensuring that the two revisions are highlighted, and click **Compare**, and the result will appear as in Figure 26-41 on page 1348. The higher version will always appear on the left-hand pane and the lower version in the right pane.

Archive

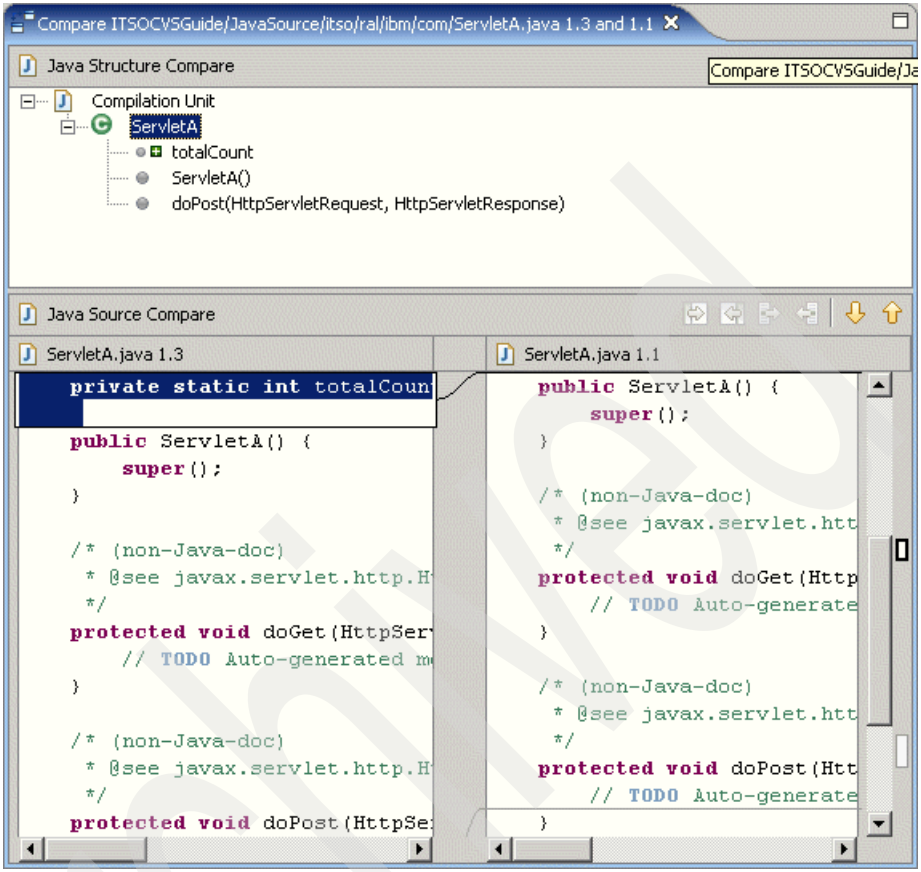


Figure 26-41 Comparisons of two revisions from the repository

26.8 Annotations in CVS

The annotation function of CVS has been included in IBM Rational Application Developer V6 to provide details of the changes that were performed on a particular revision. The annotation function displays to the request of what lines were changed in particular revisions, and the author responsible for the change (or the one to “blame”). This feature can assist developers with gaining an understanding of what has changed that may impact a functionality change in the code and identify who can assist them in their resolution of this change.

To demonstrate annotations we can go back to our example of looking at the resource ServletA.java and see what the process is and what it provides.

Archived

1. Open the Web perspective using **Window** → **Open Perspective** → **Other** → **Web**.
2. Expand in the Project Explorer **Dynamic Web Projects** → **ITSOCVSGuide** → **Java Resources** → **JavaSource** → **itso.ral.ibm.com** and highlight **ServletA.java**.
3. Right-click and select **Team** → **Show Annotation**. This will switch to the CVS Repository Exploring view and display the views as in Figure 26-42.

   The view on the left-hand side is the CVS Annotation view. The information that it displays is the user that made the change, followed by the version and the number of lines in brackets. Highlighting any of these will display the change that occurred in the top right-hand pane of the source, with the corresponding version information in the bottom right view.

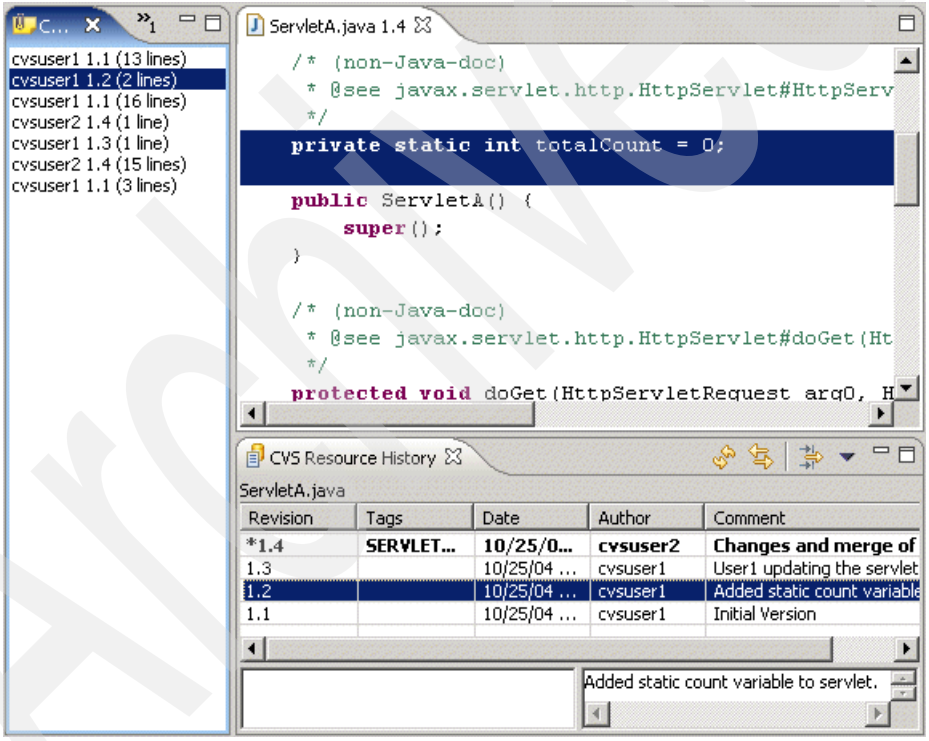


Figure 26-42 Annotation view for ServletA.java

The annotation view allows a developer to identify changes that may have occurred in a particular file and identify the root causes of issues that they may be having.

26.9 Branches in CVS

Branches are Software Configuration Management (SCM) techniques to allow current development of software based on a baseline that has been established.

In CVS there is the concept of the HEAD, which is a branch that refers to the current work that is being performed in a team environment. However, this is only useful in terms of one development team working with one release. The real-world situation is that development follows a life cycle that has development, maintenance, and enhancement based on a baseline. This is when branches can be useful and when CVS can allow you to create baselines and parallel streams of work to enhance software product development.

Typical development cycles would have a development cycle for new work, as well as a maintenance cycle in which code currently in use is enhanced and resolved of bugs. In these circumstances there would be two streams of development occurring that would be independent: The maintenance branch enhancing “operational” code and the development branch. At some point these would need to be merged together to provide a new baseline to be a production version. A representation is shown in Figure 26-43.

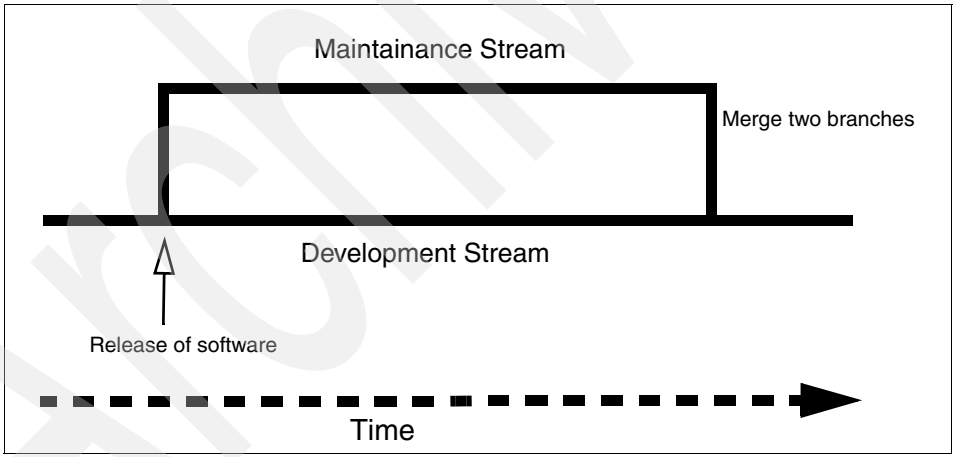


Figure 26-43 Branching of two development streams

26.9.1 Branching

Creating a branch is useful when you wish to maintain multiple versions of the software developed when they are in multiple stages of delivery.

The scenario that is identified is that a particular release has been deployed to a host machine; however, further work needs to continue to enhance the

application. In addition to this, existing software needs to be enhanced and maintained so that problems identified are fixed. Branching provides the mechanism to achieve this, and a process is outlined using the simple example as follows:

1. Open the Web perspective using **Window** → **Open Perspective** → **Other** → **Web**.
2. Expand in the Project Explorer **Dynamic Web Projects** → **ITSOCVSGuide** and highlight **ITSOCVSGuide**.
3. Create a tag for what is in your repository to use as the branch root by right-clicking and selecting **Team** → **Tag as Version...**, and typing the tag name `BRANCH_ROOT`, and clicking **OK**.
4. Highlight the project **ITSOCVSGuide**, right-click, and select **Team** → **Branch...**.
5. Type the name of the branch in the first dialog and the branch to base it from, which is `BRANCH_ROOT`. Leave the check box checked to start working on this branch, as shown in Figure 26-44 and press **OK**.

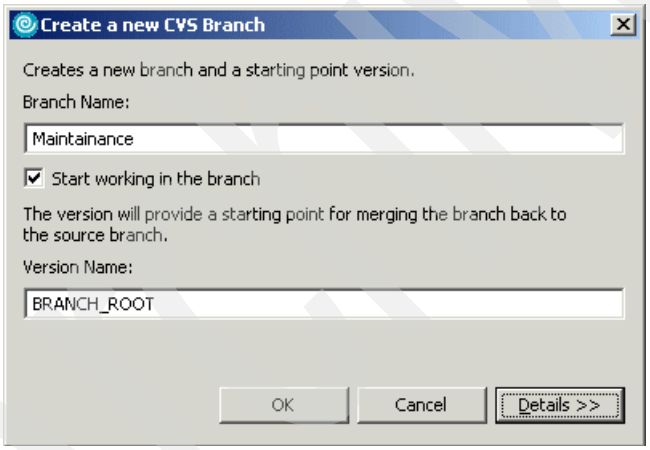


Figure 26-44 Creating a new CVS branch

> **Note:** The version name is important; you will need it when you want to merge the branches later. It identifies the point at which the branch was created.

6. Highlight the project **ITSOCVSGuide**, right-click, select **Properties**, and click the **CVS** tag. A view, as shown in Figure 26-45 on page 1352, will be displayed with the tag name displayed as `Maintainance (Branch)`, indicating that it has been set up correctly. Click **OK** when finished viewing.

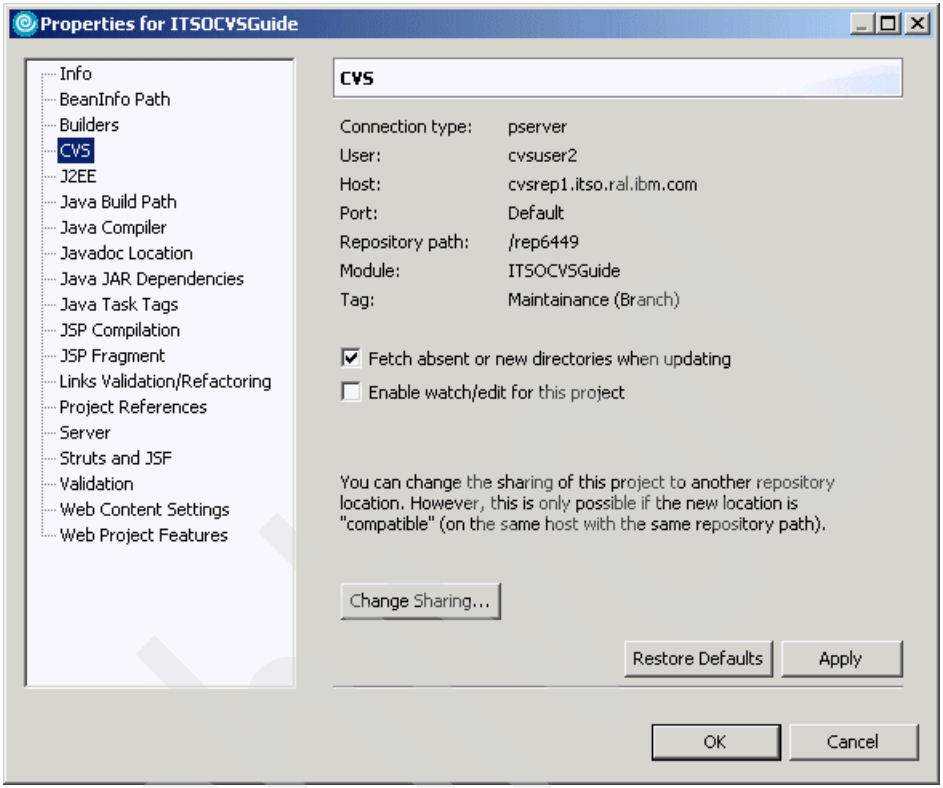


Figure 26-45 Branch information for a project in the local workspace

7. Open the CVS Repository Explorer window by clicking **Window** → **Open Perspective** → **Other** → **CVS Repository Explorer**.
8. Right-click the repository and click **Refresh View**.
9. Expand the tree and the branches sub tree to verify that the branch has been created in the repository, as shown in Figure 26-46.

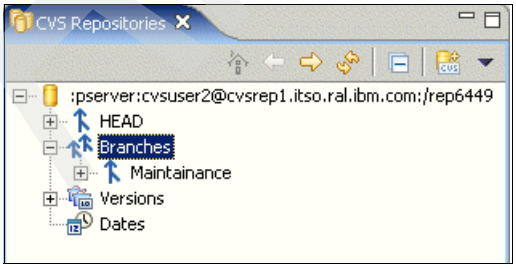


Figure 26-46 List of branches created

Updating Branch code

Assume now that there are changes required to be made to the servlet ServletA.java and a new viewbean View2.java, which is created with no methods or attributes. This will be used to demonstrate the merge process with the changes being made in the maintainance branch and saved into the repository.

1. Open the Web perspective using **Windows** → **Open Perspective** → **Other** → **Web**.
2. Expand in the Project Explorer **Dynamic Web Projects** → **ITSOCVSGuide** → **Java Resources** → **JavaSource** → **itso.ral.ibm.com** and double-click **ServletA.java** to open it.
3. Navigate to the doPost method and at the top of the class add the statement:

   ```
   System.out.println("Added in some code to demonstrate branching");
   ```

4. Select **File** → **Save** to save the information.
5. Highlight the package **itso.ral.ibm.com** and right-click **New** → **Class**
6. Type `View2` for the name of the class and click **Finish**.
7. Highlight the project **ITSOCVSGuide** and click **Team** → **Synchronize**, and click **Yes** to switch to the Synchronize view.
8. Select the project, right-click, select **Commit...**, and click **Yes** to add the resource into CVS.
9. When the Commit dialog window appears type `Branching example`, and click **OK**.
10. Open the CVS Repository Explorer by selecting **Windows** → **Open Perspective** → **Other** → **CVS Repository Explorer**, and expand out the tree, as shown in Figure 26-47 on page 1354.

Archived

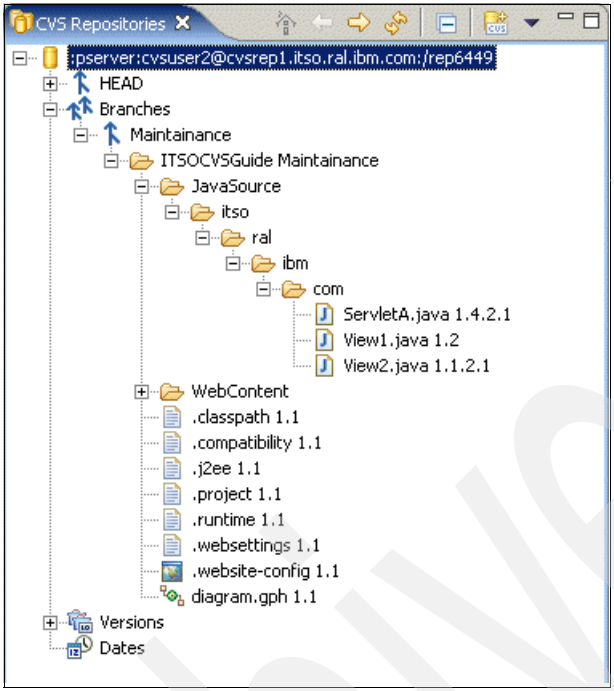


Figure 26-47 Code checked into the branch

The changes have now been committed into the branch called *Maintainance*, which will have contents that differ from the main branch. This will not be seen by developers working on the branch HEAD, which is the development stream in our scenario.

26.9.2 Merging

Merging of branches occurs when there is a point in time that requires the code from one branch to be incorporated into another branch for a major milestone. This could be a major integration step, release date, or changes are required from the branch to resolve some issues.

The scenario now is that development on the main CVS branch has completed development and is required to merge changes in the maintainance branch for release as a new version.

To merge the two branches, you have to know:

- The name of the branch or version that contains your changes.

Archived

- The version from which the branch was created. This is the version name that you supplied when branching.

In our case, the branch is *Maintenance*, and the version is *Branch_Root*.

Merging requires that the target or destination branch be loaded into the workspace before merging in a branch. Since in our scenario the changes will be merged to HEAD, this needs to be loaded in the workspace.

1. Open the Web perspective using **Windows** → **Open Perspective** → **Other** → **Web**.
2. Open the tree under the **Dynamic Web Projects** → **ITSOCVSGuide**, highlight the project **ITSOCVSGuide**, and right-click, selecting **Replace With...** → **Another Branch or Version...**.
3. When the dialog appears, select **HEAD** and click **OK**.
4. Highlight the project **ITSOCVSGuide** and right-click, selecting **Team** → **Merge**, which displays the dialog shown in Figure 26-48.

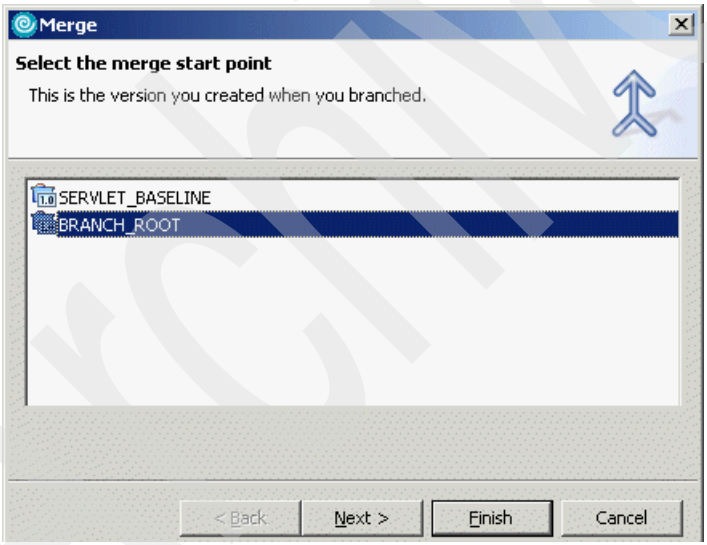


Figure 26-48 Selection of the merge start point

5. Select the branch name **BRANCH_ROOT** for the merge start point and click **Next**.
6. The merge branch dialog will be displayed requesting the branch to merge into the tag BRANCH_ROOT specified earlier. Select **Maintainance** and click **Finish**, as in Figure 26-49 on page 1356. Click **Yes** to switch to the synchronizing view.

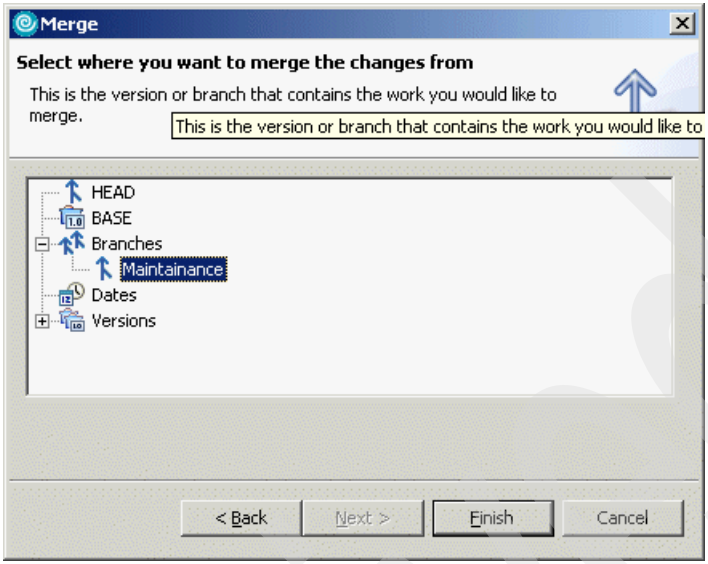


Figure 26-49 Identification of the merge target in CVS

7. Expand out the tree in the Synchronize view to display changes. Verify that there are no conflicts. If there are then the developer will need to resolve these conflicts. In our case, the merge is simple, as shown in Figure 26-50.

   Select the project **ITSOCVSGuide**, right-click, and select **Update**.

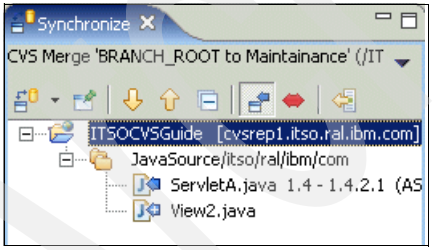


Figure 26-50 Files required to be merged

8. Open the Web perspective using **Windows** → **Open Perspective** → **Other** → **Web**.
9. Open the tree under the **Dynamic Web Projects** → **ITSOCVSGuide** and highlight the project **ITSOCVSGuide**, right-click, and select **Team** → **Synchronize with Repository**....
10. Click **Yes** to open the Synchronize view.

11. Expand out the Synchronize view to display the changed files ServletA.java and View2.java, as in Figure 26-51. Select the project, and right-click and select **Commit**.

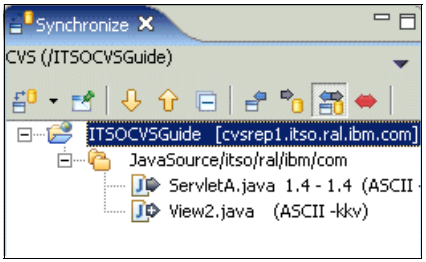


Figure 26-51 CVS updates to HEAD from the merge

12. When the commit dialog appears add the comment `Merged changes from maintainance branch` and click **OK**.

This scenario, although a simple one, highlights the technique required by users to work with branches. In a real scenario there would be conflicts found, and this would require resolution between developers. Be aware that branching and concurrent development is a complex process. IBM Rational Application Developer V6 provides the tools for merging; however, equally important are procedures on handling situations such as branching and merging of code.

26.9.3 Refreshing server-defined branches

If a branch exists on the server, you must refresh it in your workspace to be able to access it. The CVS Repositories view does not obtain a list of all branches from the server by default.

1. To define a branch manually to a repository, go to the CVS Repositories view by selecting **Window** → **Open Perspective** → **Other** → **CVS Repository Explorer**.
2. Select the repository and expand out the tree. Highlight the **Branches** node (see Figure 26-52 on page 1358), right-click, and select **Refresh Branches...**.

Archived

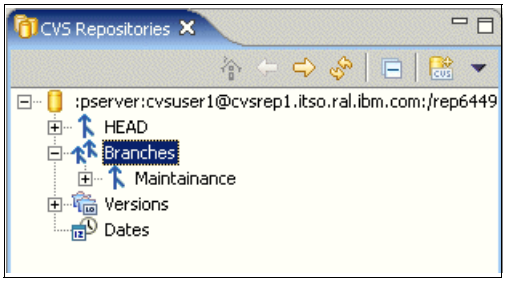


Figure 26-52 Selecting the Branches node in CVS Repository view

> **Note:** If there are no repository locations listed in the Repositories view, you have to add a location, as explained in “Add a CVS repository” on page 1323.

3. In the Refresh branches dialog specify the projects you wish to refresh (Figure 26-53 on page 1359) and click **Finish**.

Archived

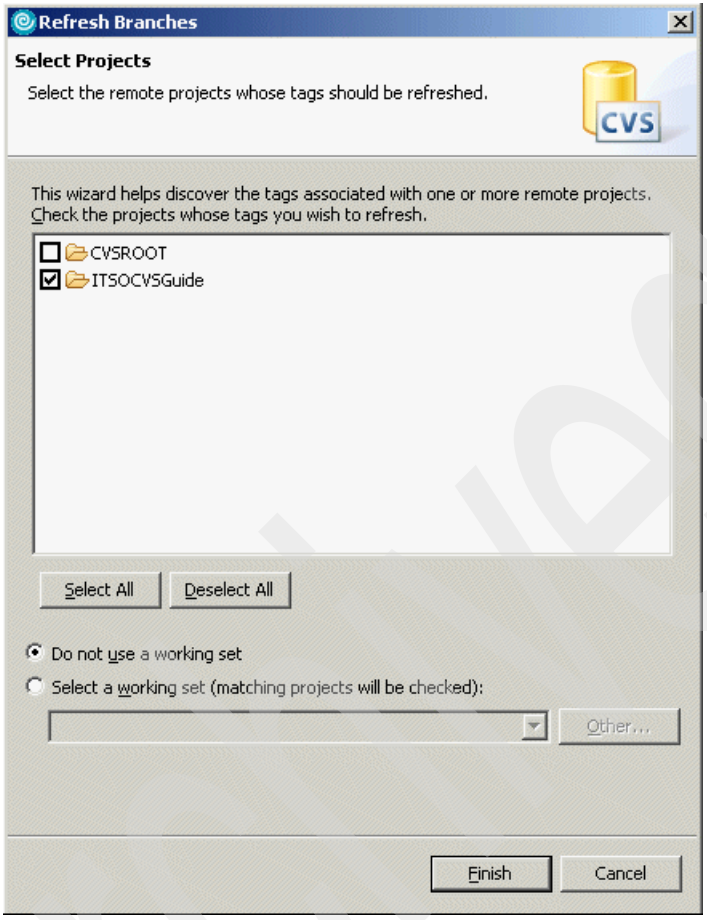


Figure 26-53 Refreshing branches

4. In the Repositories view, expand **Branches** and observe that it now contains the new SERVER_HEAD branch (see Figure 26-54 on page 1360).

   You can now check files out from this branch, if they exist.

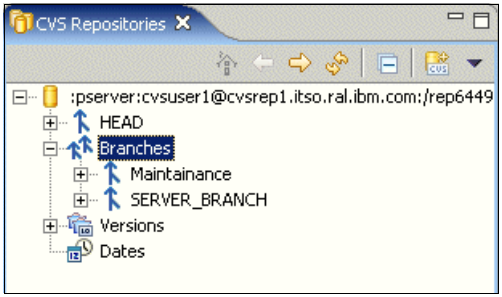


Figure 26-54 Refreshed branch created in server displayed in CVS Repository view

26.10 Work with patches

IBM Rational Application Developer V6 provides the facility for developers to be able to share work when they do not have write access to the repository. In this circumstance the developer that does have access to the repository can create a patch and can forward it to another developer who has and can apply the patch to the project and commit changes.

See the IBM Rational Application Developer V6 online help for a description of how to work with patches.

26.11 Disconnecting a project

Developers can disconnect a project from the repository.

1. Open the Web perspective using **Windows** → **Open Perspective** → **Other** → **Web**.
2. Open the tree under the **Dynamic Web Projects** → **ITSOCVSGuide**.
3. Highlight the **ITSOCVSGuide** project, right-click, and select **Team** → **Disconnect**.
4. A prompt will appear asking you to confirm the deletion of the CVS control information (see Figure 26-55 on page 1361).

   Select **Do not delete the CVS meta information (e.g, CVS sub directories)**, and click **OK**.

Archived

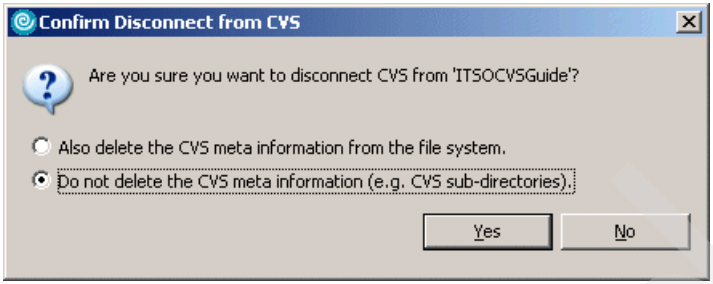


Figure 26-55 Disconnect confirmation

Important: By not deleting the CVS meta information we can reconnect the project with the CVS repository more easily. Removal of the CVS meta information may impact the synchronization of the files in the workspace and the repository.

CVS adds special directories named CVS to the project and its folders. These directories can be deleted or kept on disconnect. The feature to keep these folders and files in IBM Rational Application Developer V6 hidden is provided with the Eclipse framework.

Reconnect

You can reconnect a project to the repository (**Team** → **Share Project**). Reconnect is easier if the CVS folders are still in the project. If they were deleted, you are prompted to synchronize your code with the existing repository code.

26.12 Synchronize perspective

The synchronize perspective in IBM Rational Application Developer V6 has been used in describing concepts within this chapter but has not as yet been described. The purpose of this perspective is to provide to the user of the tool with an entry point to identify changes in the team repository versus what is on the local workspace, and assist in effectively using it. Features provided with the synchronize perspective include:

- Create custom synchronization of a subset of resources in the workspace.
- Schedule checkout synchronization.
- Provide a comparison of changes in the workspace (which has been demonstrated in “Comparisons in CVS” on page 1344).

26.12.1 Custom configuration of resource synchronization

The Synchronize view provides the ability to create custom synchronization sets for the purpose of synchronizing only identified resources that a developer may be working on. This allows the developer to focus on changes that are part of their scope of work and ensure they are aware of the changes that occur without focusing on the other aspects of the application. Problems can occur with this mode of operation as well, since with normal application development changes in one part of the application can impact other parts.

Important: Custom synchronization is most effective when an application is designed with defined interfaces where the partitioning of work is clear and defined. However, even in this scenario it needs to used with caution since it can introduce additional work in the development cycle for integration of the final product. Procedures need to be documented and enforced to ensure that integration is incorporated as part of the work pattern for this scenario.

The scenario to demonstrate custom synchronization requires that the ITSOCVSGuide project exists in the workspace. In addition, the developer will need to perform the following synchronization procedures:

- Full synchronization of the project ITSOCVSGuide
- Partial synchronization of the servlet ServletA.java

The procedure to perform this would be as follows:

1. Select **Window** → **Open Perspective** → **Other** → **Team Synchronizing**, which presents the window shown in Figure 26-56 on page 1363.

Archive

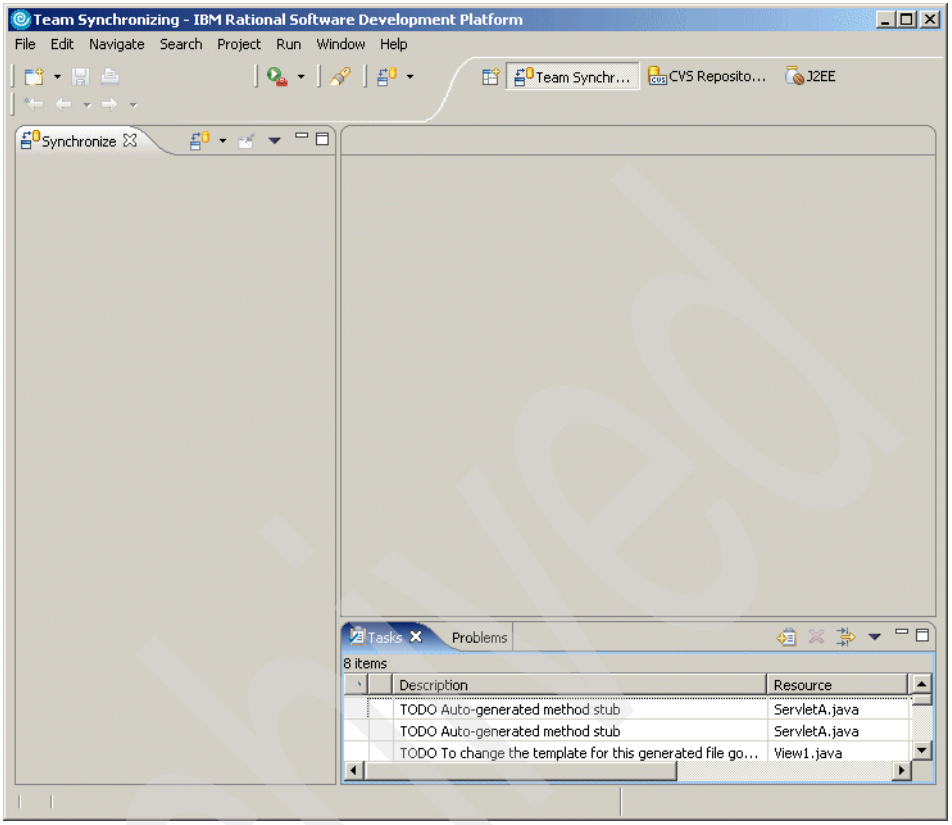


Figure 26-56 Team Synchronizing perspective

2. Click the Synchronize button at the top the Synchronize view (left pane) and click **Synchronize...** to add a new synchronization definition.
3. In the synchronize dialog that appears as in Figure 26-57 on page 1364, select **CVS** and click **Next**.

Archive

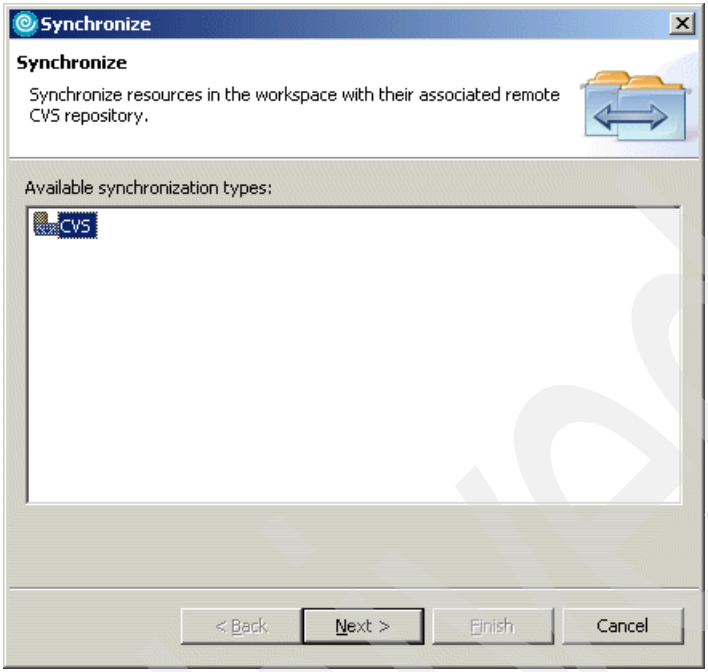


Figure 26-57 Synchronize dialog

4. Expand the project tree out to view the contents. Accept the defaults for the Synchronize CVS dialog, as shown in Figure 26-58 on page 1365, and click **Finish**.

   If there are no changes then a dialog box will appear saying `Synchronizing: No changes found`, and in the Synchronize view a message of `No changes in 'CVS (Workspace)'`.

Archived

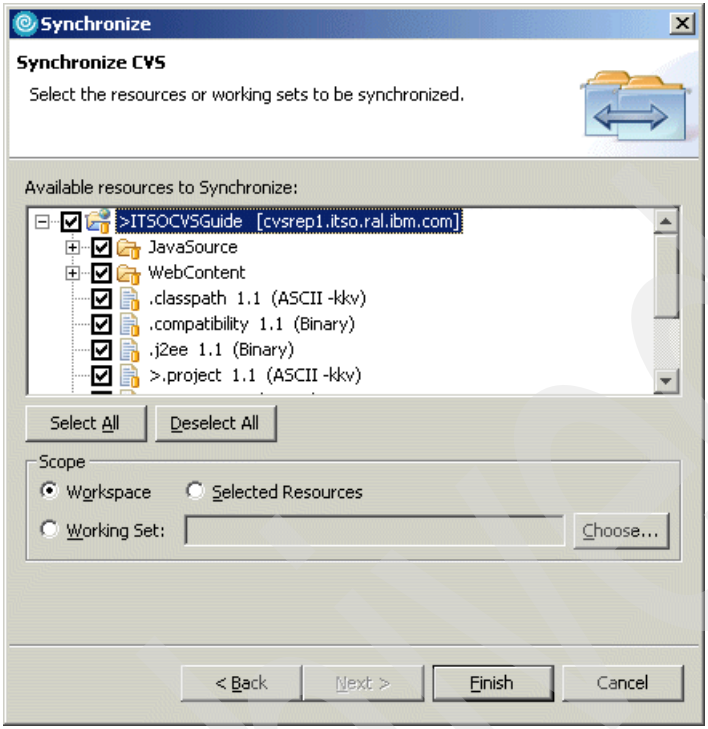


Figure 26-58 Default synchronization of the project ITSOCVSGuide

5. To preserve this synchronization, click the Pin Current Synchronization icon .
6. Add in a new synchronization by clicking the Synchronize icon at the top of the Synchronize view.
7. In the synchronize dialog that appears, as in Figure 26-57 on page 1364, select **CVS** and click **Next**.
8. Expand the project tree fully out under JavaSource to view the contents, click **Deselect All** to deselect all the resources, and click the check box for ServletA.java.

   Verify that Selected Resources is selected, as in Figure 26-59 on page 1366, and click **Finish**. If there are no changes, then a dialog box will appear saying `Synchronizing: No changes found`, and in the Synchronize view a message of `No changes in 'CVS (Workspace)'`.

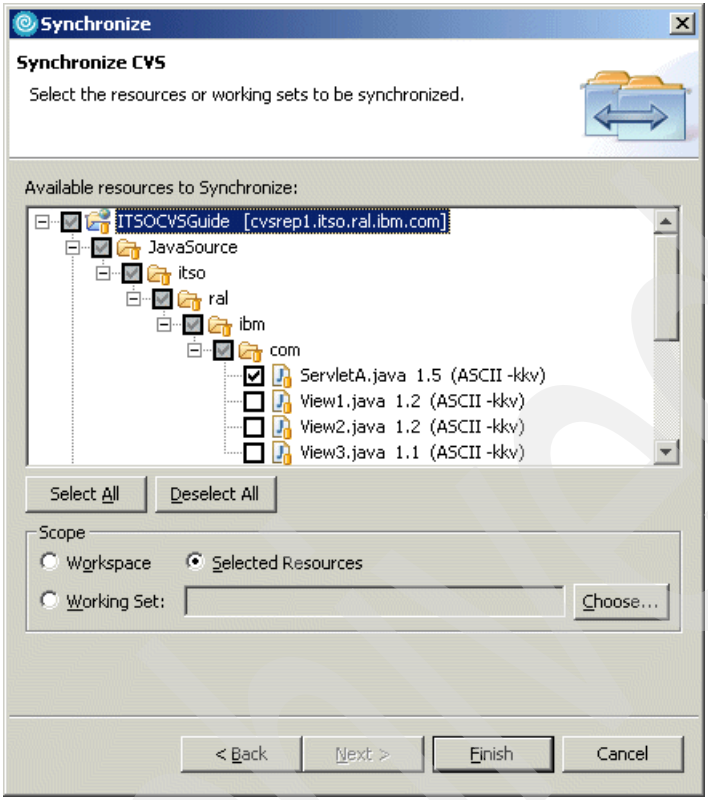


Figure 26-59 Selecting ServletA.java for synchronization

9. To preserve this synchronization, click the Pin Current Synchronization icon .

In the list of synchronizations two should appear, as shown in Figure 26-60.

Figure 26-60 List of synchronizations created

Worksets of synchronization can be defined when creating a synchronization. The workset supports three types of resources:

- Java

  Supports the creation of synchronization working sets consisting of pure java resources, such as java source and jar files.

- Help

  Supports the creation of synchronization working sets consisting of Help resources.

- Resource

  Supports the creation of synchronization working sets consisting of any file that can be saved into the team repository.

Working sets differ in that a name can be associated with the synchronization, allowing the developer to have a meaningful name related to the synchronization. This is not been shown in this book; however, it follows a similar procedure as described above and is left to the reader to attempt.

26.12.2 Schedule synchronization

A new feature that has been included in IBM Rational Application Developer V6 is the ability to schedule synchronization of the workspace. This feature follows on from “Custom configuration of resource synchronization” on page 1362, in which a user would like to schedule the synchronization that has been defined. Scheduling a synchronization can only be performed for synchronizations that have been pinned.

To demonstrate this feature, assume that the project ITSOCVSGuide is loaded in the workspace and a synchronization has been defined for this project and pinned. Scheduling of this project for synchronization is then performed using the following:

1. Open the synchronization view by clicking **Windows** → **Open Perspective** → **Other...** → **Team Synchronizing**.
2. In the Synchronize view click the drop-down arrow, as circled in Figure 26-61.

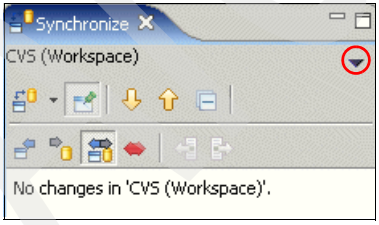


Figure 26-61 Synchronize view

3. A drop-down box appears. Click **Schedule...** as shown in Figure 26-62 on page 1368.

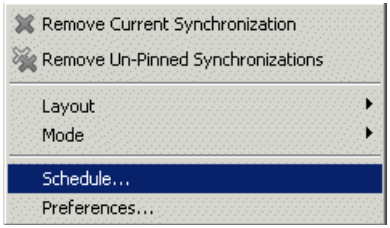


Figure 26-62 Selecting the Schedule option

4. The Configure Synchronize Schedule will be displayed. Select the radio button **Using the following schedule:** and the time period that you wish to synchronize, as shown in Figure 26-63. Click **OK**.

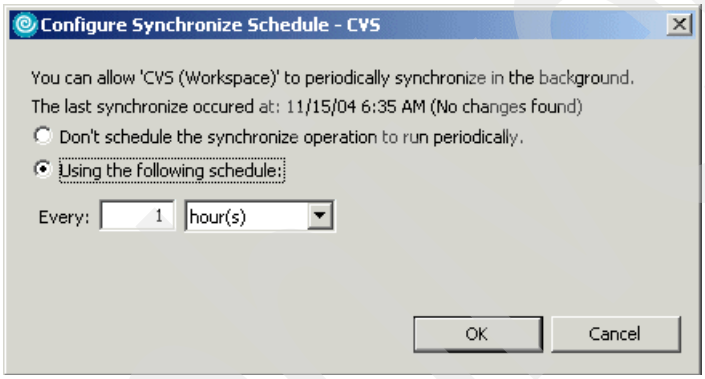


Figure 26-63 Setting synchronization schedule

By setting the synchronization schedule to an hour, the project ITSOCVSGuide will be synchronized every hour to ensure that the latest updates are available.

Abroad

Part 6

Appendixes

Archived

© Copyright IBM Corp. 2005. All rights reserved.

Archived

A

IBM product installation and configuration tips

The objective of this appendix is to highlight the key considerations and options for installation selected for IBM Rational Application Developer, IBM DB2 Universal Database, and IBM WebSphere Application Server for this redbook.

The appendix is organized into the following sections:

- IBM Rational Application Developer V6 installation
- IBM Rational Agent Controller V6 installation
- IBM Rational ClearCase LT installation
- IBM DB2 Universal Database V8.2 installation
- IBM WebSphere Application Server V6 installation
- WebSphere Application Server messaging configuration

© Copyright IBM Corp. 2005. All rights reserved.

IBM Rational Application Developer V6 installation

The purpose of this section is to highlight the key installation considerations, identify components installed while writing this redbook, and provide a general awareness to using the Rational Product Updater tool to install IBM Rational Application Developer V6 Interim Fix 0004.

This section includes the following tasks:

- Rational Application Developer installation
- WebSphere Portal V5.0 Test Environment installation
- WebSphere Portal V5.1 Test Environment installation
- Rational Application Developer Product Updater - Interim Fix 0004

Note: For detailed information on the IBM Rational Application Developer V6.0 installation, refer to the following product guides found on CD 1:

- *Installation Guide, IBM Rational Application Developer V6.0* (Open install.html in a Web browser.)
- *Release note, IBM Rational Application Developer V6.0* (Open readme.html in a Web browser.)
- *Migration Guide, IBM Rational Application Developer V6.0* (Open migrate.html in a Web browser.)
- *Installing IBM Rational ClearCase LT - Technote* (Open TechNote-Installing_CCLT.html in a Web browser.)

Rational Application Developer installation

The IBM Rational Application Developer V6.0 product includes detailed installation documentation. This section highlights the installation issues we found while writing the redbook, as well as the components we installed.

Installation considerations

Prior to installing IBM Rational Application Developer V6.0, beware of the following installation considerations:

- UNC network shares: Do not install Rational Application Developer from a UNC network share (for example, \\server\shareA). Instead, map the network drive to a drive letter (for example, net use x: \\server\shareA) so that the Rational Application Developer installer works properly.
- Standardize installation path for team development: Standardize the Rational Application Developer installation path for your development team. We found that many files within the projects have absolute paths based on the

installation path; thus when you import projects from a team repository such as CVS or ClearCase you will get many errors.

- Installer window in foreground/background: After clicking IBM Rational Application Developer V6.0, sometimes the welcome screen does not appear in the foreground. Simply select the new window from the task list to continue.

Rational Application Developer installation

While writing this redbook, we installed IBM Rational Application Developer V6.0 as follows:

1. Start the Rational Application Developer Installer by running **launchpad.exe** from CD 1.
2. The IBM Rational Application Developer V6.0 components have separate installations from the main Launchpad Base page, as seen in Figure A-1.

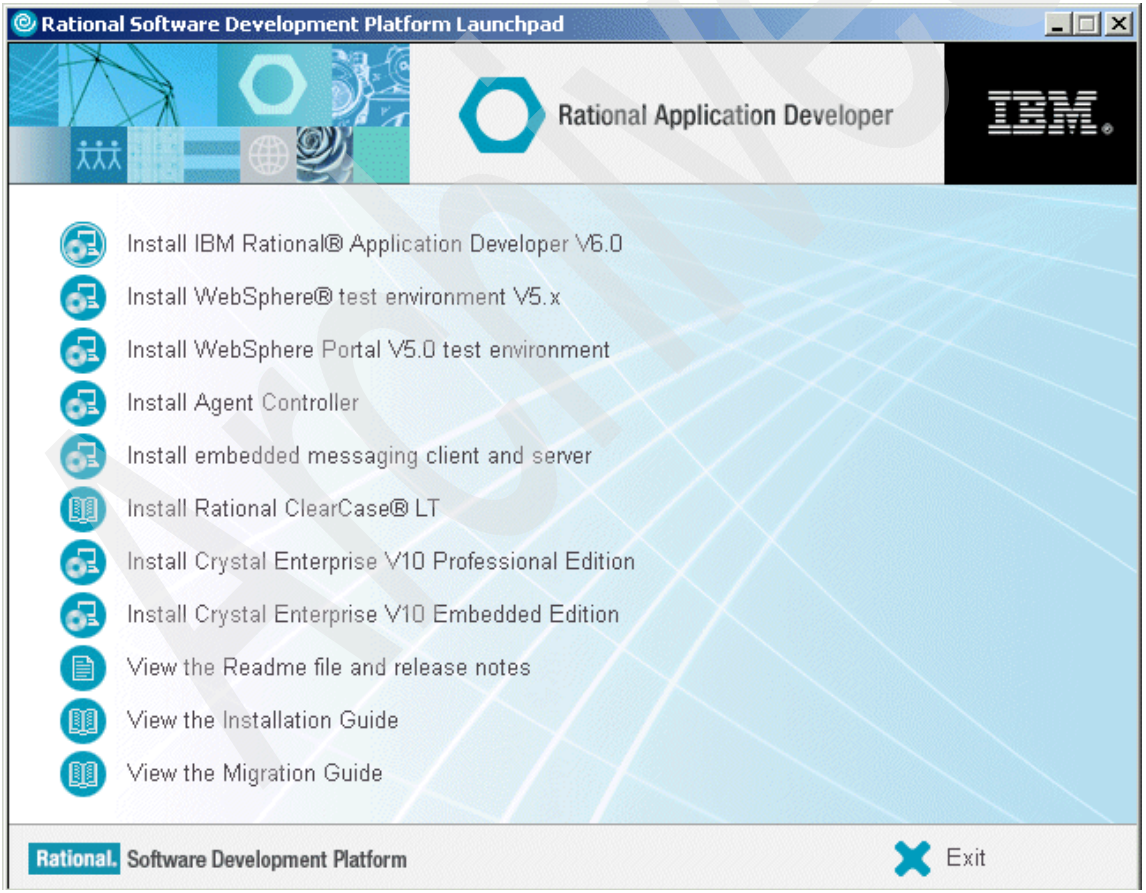


Figure A-1 IBM Rational Application Developer V6.0 installation components

Table A-1 IBM Rational Application Developer V6.0 description of install components

Component	Description
Install IBM Rational Application Developer V6.0	Core Rational Application Developer - sub components: ► Integrated Development Environment (required) ► IBM WebSphere Application Server V6.0 Integrated Test Environment ► Additional Features: – Language Pack – Enterprise Generation Language (EGL) – Portal tools – Examples for Eclipse Plug-in Development **Note:** While writing the redbook, we did not install (out of scope) the Language Pack or Examples for Eclipse Plug-in Development.
IBM WebSphere Test Environment V5.x	Select from the following sub components: ► WebSphere Application Server V5.1 ► WebSphere Application Server V5.0.2 ► WebSphere Application Server Express V5.1 ► WebSphere Application Server Express V5.0.2
IBM WebSphere Portal V5.0 Test Environment	IBM WebSphere Portal V5.0 Test Environment integrated with Rational Application Developer for local testing and debug.
Install Agent Controller	IBM Rational Agent Controller is needed for debug on WebSphere Application Server V5.x (capability built-in on V6), and profiling and testing on WebSphere Application Server V6 and V5.x.
Install the embedded messaging client & server	This feature is for embedded messaging for WebSphere Application Server V5.x. In WebSphere Application Server V6 messaging is built-in.
Install Rational ClearCase LT	**Note**: Due to a defect at the time of the Rational Application Developer product release, the ClearCase LT was made available as a Web download with a supporting patch. See *Installing IBM Rational ClearCase LT - Technote* (open TechNote-Installing_CCLT.html in a Web browser found on CD 1).
Install Crystal Enterprise V10 Professional Edition	
Install Crystal Enterprise V10 Embedded Edition	

Archive

3. When Welcome window appears, click **Next**.
4. When the License Agreement window appears, review the terms and if in agreement select **I accept the terms of the license agreement**. Click **Next**.
5. When the Install Directory window appears, we accepted the default C:\Program Files\IBM\Rational\SDP\6.0 and clicked **Next**.

> **Important:** It is very important that you understand the implication of changing the default installation path. For example, if you plan on team development, we strongly recommend that all developers use the same installation path (such as default); otherwise you will run into problems.
>
> We found that many files within the projects have absolute paths based on the installation path; thus when you import projects from a team repository such as CVS or ClearCase you will get many errors.

6. When the Select Features window appears, select the appropriate features for your environment. For example, we selected the features displayed in Figure A-2 for the redbook.

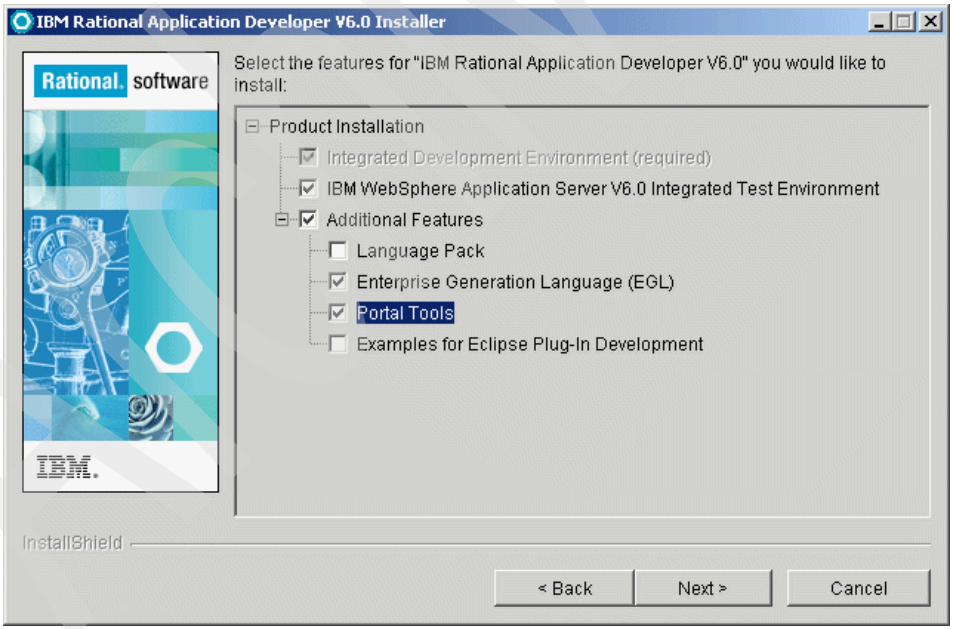


Figure A-2 Install IBM Rational Application Developer V6.0 Features

7. When the Installation Summary window appears, review your selections and click **Next** to begin copying files as seen in Figure A-3 on page 1376.

Note: The selected features take approximately 2.5 GB of disk space.

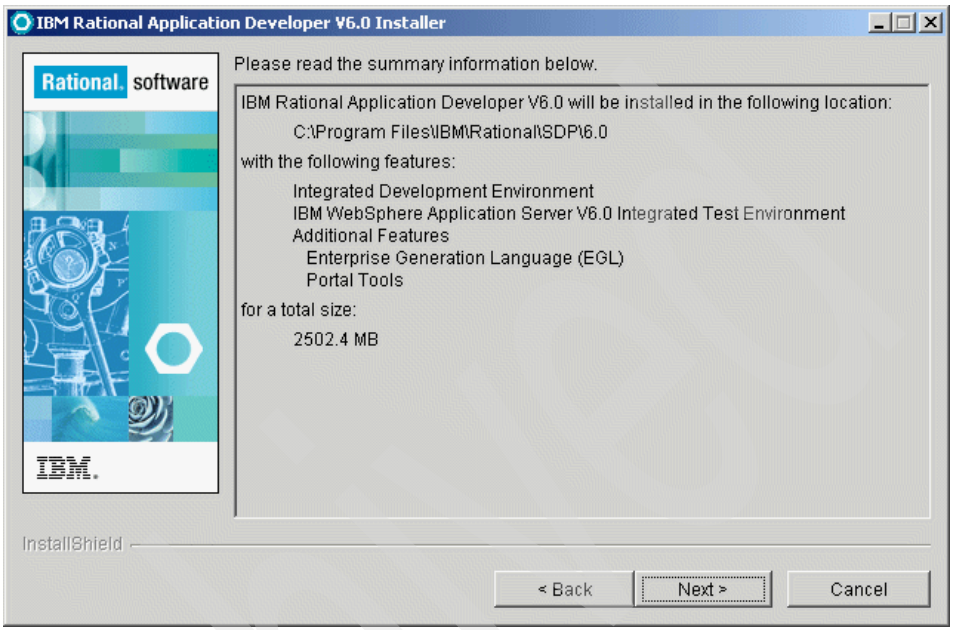


Figure A-3 IBM Rational Application Developer V6.0 install summary

The installation can take 30 minutes to 2 hours depending on the processing speed of your system and the features selected.

8. When you see the message `The Installation Wizard has successfully installed IBM Rational Application Developer V6.0`, click **Next**.
9. The installation is now complete. We unchecked the **Launch Agent Controller install** and then clicked **Finish**. We will install the IBM Rational Agent Controller separately for profiling and testing purposes.

WebSphere Portal V5.0 Test Environment installation

The development tools are installed as part of the base Rational Application Developer installation by selecting the Portal Tools feature. There are a couple of possible scenarios for the portal test environment.

- IBM WebSphere Portal V5.0 Test Environment (local)

  Install this feature from the main IBM Rational Application Developer V6.0 Installer dialog.

- IBM WebSphere Portal V5.0.2.2 (remote)

  This option requires IBM WebSphere Portal V5.0.2.2 (Express, Enable, Extend Editions); sold separately.

WebSphere Portal V5.1 Test Environment installation

The development tools are installed as part of the base Rational Application Developer installation by selecting the Portal Tools feature. There are a couple of possible scenarios for the portal test environment.

- IBM WebSphere Portal V5.1 (remote)

  This option requires IBM WebSphere Portal V5.1 (Express, Enable, Extend Editions); sold separately.

- IBM WebSphere Portal V5.1 Test Environment (local)

  The WebSphere Portal V5.1 Test Environment is included with IBM Rational Application Developer V6.0 distribution; however, it is installed via a separate installer found on the WebSphere Portal V5.1 Setup CD.

Tip: Eliminate installer prompting of CDs.

When installing the IBM WebSphere Portal V5.1 Test Environment from a directory or network drive (ensure drive is mapped), create the directory names as listed in Table A-2 to eliminate the need for the WebSphere Portal installer to prompt for CDs. For example, we created a directory structure on a network drive like the following:

```
/wp51/setup
/wp51/cd1-1
/wp51/cd1-2
/wp51/cd1-15
/wp51/cd2
/wp51/cd3
```

Table A-2 IBM WebSphere Portal V5.1 Test Environment CDs

IBM WebSphere Portal V5.1 Test Environment CDs included with RAD V6.0	Directory
IBM WebSphere Portal V5.1 - Portal Install (Setup)	setup
IBM WebSphere Portal V5.1 - WebSphere Business Integrator Server Foundation (1-1)	cd1-1
IBM WebSphere Portal V5.1 - WebSphere Business Integrator Server Foundation (1-2)	cd1-2
IBM WebSphere Portal V5.1 - WebSphere Business Integrator Server Foundation WebSphere Application Server V5.1 Fixpack 1 (1-15)	cd1-15

IBM WebSphere Portal V5.1 Test Environment CDs included with RAD V6.0	Directory
IBM WebSphere Portal V5.1 - Portal Server (2)	cd2
IBM WebSphere Portal V5.1 - Lotus Workplace Web Content Management™ (3)	cd3

To install the IBM WebSphere Portal V5.1 Test Environment, do the following:

1. Run **install.bat** from the WebSphere Portal Setup CD (or setup directory).
2. When prompted, select the desired language for the install wizard (for example, English) and click **OK**.
3. When the Welcome page appears, review the information and click **Next**.
4. When the Software License Agreement page appears, if in agreement, select **I accept the terms in the license agreement** and click **Next**.
5. When the Choose the setup type that best suits your needs page appears, select **Test Environment (**as seen in Figure A-4) and then click **Next**.

Archived

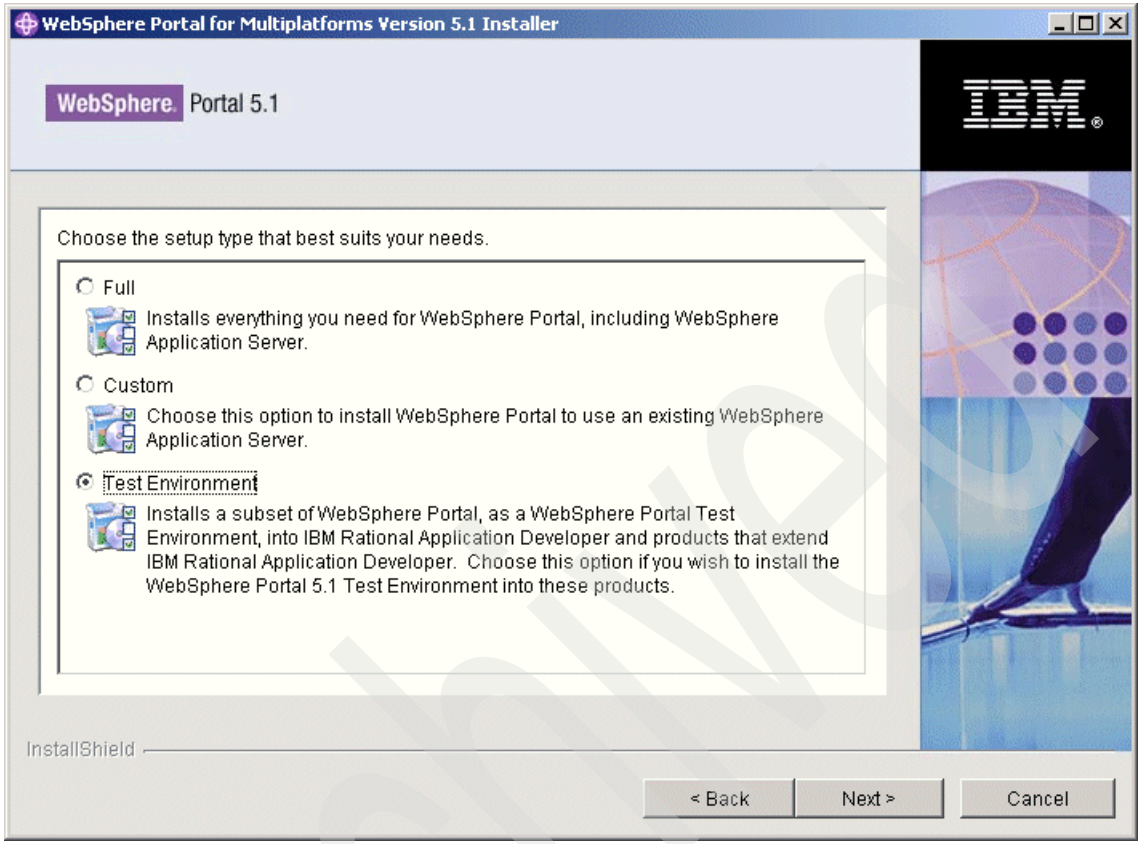


Figure A-4 Choose the setup type - Test Environment

6. When prompted, ensure instances of WebSphere Application Server and WebSphere Portal are not running. Click **Next**.
7. When prompted to enter the WebSphere Application Server installation directory used by the WebSphere Portal Test Environment, we accepted the default directory (C:\Program Files\Portal51UTE\AppServer). Click **Next**.

   Note: This requires approximately 1,650,000 KB of disk space.

8. When prompted to enter the WebSphere Portal installation directory, we accepted the default (C:\Program Files\Portal51UTE\PortalServer). Click **Next**.

   Note: This requires approximately 1,550,000 KB of disk space.

9. When prompted to enter the WebSphere Portal administrative user and password, we entered the following and then clicked **Next**:
   - WebSphere Portal administrative user: `wpsadmin`
   - WebSphere Portal administrative user password: `<password>`
10. When the WebSphere Portal install options Summary dialog appears, review the selections and click **Next**.

    The installation should now begin to copy files. The installation process takes several hours.
11. When the installation is complete click **Finish**.

> **Note:** The WebSphere Portal installation log can be found in the following directory:
>
> `C:\Program Files\Portal51UTE\PortalServer\log\wpinstalllog.txt`

Rational Application Developer Product Updater - Interim Fix 0004

The *Rational Product Updater* tool is used to apply fixes to IBM Rational Application Developer V6. We strongly recommend that you install the Interim Fix 004 for IBM Rational Application Developer for WebSphere Software V6.0. The Interim Fix 0004 can be installed directly over the Web using the Product Updater tool included with Rational Application Developer, or alternatively the Interim Fix 0004 can be downloaded and installed locally.

More detailed information on Interim Fix 0004 can be found at:

http://www.ibm.com/support/docview.wss?rs=2043&context=SSRTLW&dc=D400&uid=swg24008988&loc=en_US&cs=UTF-8&lang=enclass=

The following procedure describes how to install IBM Rational Application Developer V6.0 Interim Fix 0004 using the Product Updater tool:

1. Ensure that Rational Application Developer test servers are stopped and Application Developer is closed.
2. Start the Rational Product Updater by clicking **Start** → **Programs** → **IBM Rational** → **Rational Product Updater**.
3. Click **Find Updates** as seen in Figure A-5.

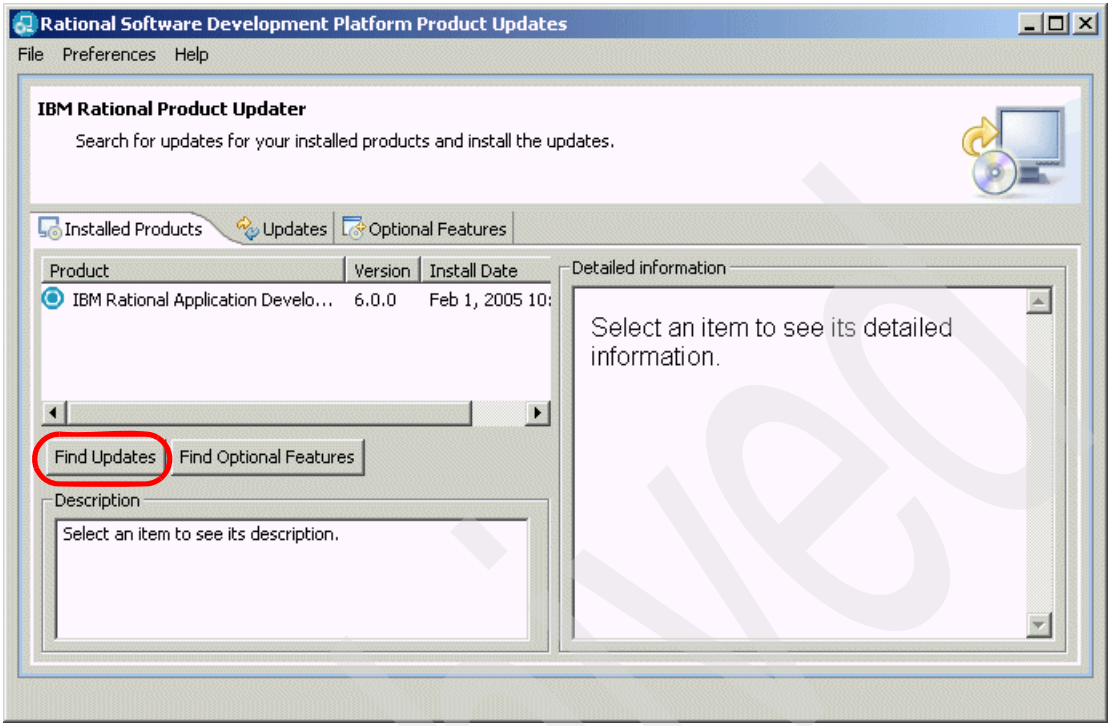


Figure A-5 Rational Product Updater

Note: The Rational Product Updater will detect that a newer version of the Product Updater is available. Download it and restart using the new version.

4. When prompted with a dialog that Interim Fix 0004 has been found, click **OK**.

Tip: If the Rational Product Updater requires an update, you are prompted to install it before you can continue. The Rational Product Updater installs the update, restarts, and retrieves a list of available updates.

5. The IBM Rational Product Updater should be populated with updated fix information. Ensure that **Interim Fix 0004** is checked. Click **Install Updates**.

Tip: Detailed information on the fix is displayed in the right-hand window by selecting the **Interim Fix**.

Archive

6. When the License Agreement dialog appears, if in agreement select **I accept the terms in the license agreements**, and then click **OK** to begin the installation.

   Depending on the speed of your computer processor, the amount of RAM, and the speed of your Internet connection, the update might take an extended period of time to download and install.

7. After the installation is complete, click the **Installed Products** tab to verify that the Interim Fixes were installed successfully.
8. Close the Rational Product Updater.

IBM Rational Agent Controller V6 installation

The IBM Rational Agent Controller is a daemon that allows client applications to launch and manage local or remote applications, and provides information about running applications to other applications. You must install Agent Controller separately before you can use the following tools:

- Profiling tools to profile your applications. Agent Controller must be installed on the same system as the application that you are profiling.
- Logging tools to import remote log files. Agent Controller must be installed and running on the remote system from which the log files are imported.
- Component testing tool to run test cases. Agent Controller must be installed on the systems on which you run the test cases.
- Run-time analysis tool for probe insertion, code coverage, and leak analysis.
- Tools for remote application testing on WebSphere Application Server version 5.0 or 5.1. (Agent Controller does not have to be installed for remote publishing of applications, or for local application publishing or testing.) Note that WebSphere Application Server Version 6.0 has this functionality built in, so Agent Controller is not required on Version 6.0 target servers.

The IBM Rational Agent Controller is available on the following platforms:

- Microsoft Windows 2000, XP, 2003
- IBM AIX®
- IBM OS/390
- IBM OS/400
- Linux on Intel
- Linux on S/390®
- SUN Solaris on Sparc
- HP/UX

Within this redbook, there are a couple of scenarios where selected components of the IBM Rational Agent Controller need to be installed.

- Chapter 20, “JUnit and component testing” on page 1081
- Chapter 21, “Debug local and remote applications” on page 1121
  - WebSphere Application Server V6.0

    If the remote system is running WebSphere Application Server V6.0 and you only intend to use remote debug, the IBM Rational Agent Controller is not required since the required functionality is built into WebSphere Application Server V6.0.
  - WebSphere Application Server V5.1 and V5.0

    If the remote system is running WebSphere Application Server V5.1 or V5.0 and you only intend to use remote debug, the IBM Rational Agent Controller is required. You will be prompted to provide the installation path for WebSphere Application Server V5.1 or V5.0 during the IBM Rational Agent Controller installation.
- Chapter 24, “Profile applications” on page 1237

To install the IIBM Rational Agent Controller for Windows, do the following:

1. Insert the IBM Rational Agent Controller CD included with IBM Rational Application Developer V6.0 in the CD-ROM drive of the system where WebSphere Application Server is installed.
2. Navigate to the win_ia32 directory and run setup.exe to start the installer.
3. When the Welcome window appears, click **Next**.
4. When prompted to make sure the Eclipse Platform is not running, click **Next**.
5. Review the terms of the license agreement, and if in agreement select **I accept the terms in the license agreement** and then click **Next**.
6. When prompted to enter the IBM Rational Agent Controller installation directory, we accepted the default C:\Program Files\IBM\AgentController and then clicked **Next**.
7. When prompted to select the components of the IBM Rational Agent Controller to install, we accepted the default (see Figure A-6) and clicked **Next**.

Archived

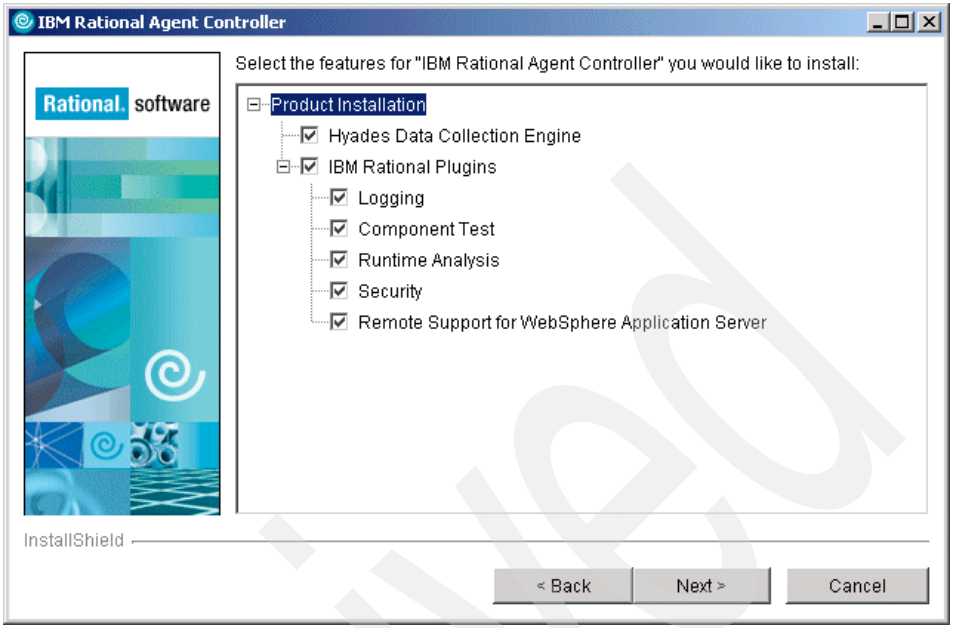


Figure A-6 IBM Rational Agent Controller component selection

8. When prompted to enter the path to the Java Runtime, we entered `c:\ibm\WebSphere\AppServer\bin\java\jre\bin\java.exe` and then clicked **Next**.

   If installing the IBM Rational Agent Controller on the node where Rational Application Developer is installed, enter the following path:

   `<rad_home>\runtimes\base_v6\java\jre\bin\java.exe`

   Where *<rad_home>* is the Rational Application Developer installation path.

9. When prompted to enter the WebSphere Application Server installation path, we left this page blank since we are installing the IBM Rational Agent Controller on a WebSphere Application Server V6.0 node. Click **Next**.
10. When the Host that can access the Hyades Data Collection Engine window appears, we accepted the default (any system) and clicked **Next**.
11. When the Specify security setting window appears, we accepted the default (Disable) and clicked **Next**.
12. When the Installation summary page appears, review the installation options and then click **Next** to begin copying files.
13. When the installation is complete, click **Finish**.

Archived

IBM Rational ClearCase LT installation

Rational Application Developer entitles you to a free license of Rational ClearCase LT. If you do not have the ClearCase LT product media, then you must first download and install the latest supported version of ClearCase LT. If you already have the ClearCase LT v2002.05 or ClearCase LT v2003.06 product media, you will also need the latest patches to support the integration with IBM Rational Application Developer V6.0.

Important: Information on obtaining a copy of IBM Rational ClearCase LT can be found in TechNote-Installing_CCLT.html of the IBM Rational Application Developer V6.0 CD1.

Installing the ClearCase LT Server component also installs the ClearCase LT Client component. We recommend that you install the ClearCase LT Server component before installing ClearCase LT Client on any additional machines.

Tips: When installing the ClearCase LT Server, you can be logged on either locally on your Windows machine or logged on to a Windows domain. If installing while logged on locally, you will only be able to connect to the server from your local machine. Other people in your development team will not be able to connect to your machine and use your ClearCase LT Server. The user account used when installing must be a member of the local Administrators group.

To use ClearCase LT in a team environment and let other team members use your ClearCase LT Server, you must be logged on to a Windows domain with a user account having Domain Administrator privileges while installing ClearCase LT Server. The domain must also have a group for the ClearCase users and all members of your development team must be members of this group. This group should also be the Primary Group for these users. You can use the Domain Users group for this.

It is highly recommended to use the Windows domain approach. Local setup can be useful for testing and demonstration purposes.

The installation instructions in this section are intended to help you install the client and server code for Rational ClearCase LT. For more detailed installation instructions refer to the *Rational ClearCase LT Installation Guide* product guide.

1. Run setup.exe from the root directory of the downloaded ClearCase LT installation image.
2. When the Rational Setup Wizard appears click **Next** to continue.

3. On the Product Selection page, select **Rational ClearCase LT** and click **Next**.
4. On the Deployment Method page, select **Desktop installation from CD image** and click **Next**.
5. On the Client/Server page, select both server and client software and click **Next**.
6. In the new welcome window that is displayed, click **Next**.
7. A warning message about potential issues with Windows change journals may be displayed next. If you have implemented change journals on the target system, check the advice on the referenced Technote and determine whether the fix is required. Click **Next** to continue.
8. Accept the licensing agreement on the next page and click **Next**.
9. On the Destination Folder page, verify the install path (we use the default of C:\Program Files\Rational). Click **Next**.
10. On the Custom Setup page, you may unselect the Web Interface, as it is not used in this exercise. Click **Next** to continue.
11. A Configure pop-up window is displayed for adjusting the Start menu and desktop shortcut settings. Click **Done**.
12. Click **Install** on the Ready to Install the program window to start the installation.
13. On the Setup Complete page, click **Finish**.
14. The ClearCase LT Getting Started Wizard is now started. The wizard guides you through the process of creating a ClearCase storage directory, a ClearCase VOB, and setting up a ClearCase project. On the first page of the wizard click **Next**.
15. On the Storage Directory page, you can select a directory to store your ClearCase VOB files. Keep the default and click **Next**.
16. On the Source VOB page, keep the default name for Source VOB and the Initial Component.
17. On the Initial Project page, keep the default project name InitialProject and select the **Parallel Stream Project option**. Click **Next**.
18. On the Summary page, click **Next**, and when the setup is done click **Close** on the final dialog.

Archived

Note: After you have installed the Rational ClearCase LT product, we recommend that you review the Rational ClearCase Support page on the IBM Software Support site and make sure that the latest fixes have been applied. The Web site can be found at:

`http://www.ibm.com/software/awdtools/clearcase/support`

IBM DB2 Universal Database V8.2 installation

There are several editions of IBM DB2 Universal Database; however, the core functionality is the same amongst the editions. The IBM Rational Application Developer V6.0 product packaging includes IBM DB2 Universal Database V8.2 Express Edition.

Several of the chapters found in this redbook require IBM DB2 Universal Database V8.2 to be installed. The purpose of this section is to highlight the key installation considerations for DB2 UDB specific to the redbook.

To start the DB2 UDB installation run setup.exe. We have highlighted the installation options we selected:

- Installation type: **Typical** (450 MB)
- Installation path: `C:\Program Files\IBM\SQLLIB`
- DB2 Administration Server user name: `db2admin`
- We accepted the defaults for the remaining installation settings.

Note: For detailed installation instructions, refer to the IBM DB2 Universal Database V8.2 product documentation.

IBM WebSphere Application Server V6 installation

Several of the chapters found in this redbook require IBM WebSphere Application Server V6.0 to be installed. The purpose of this section is to highlight the key installation considerations for WebSphere Application Server specific to this redbook.

The IBM WebSphere Application Server V6.0 installation is started by running launchpad.bat. The IBM WebSphere Application Server V6.0 components have separate installations from the main Launchpad Base page, as seen in Figure A-7 on page 1388.

We accepted the default WebSphere Application Server installation directory, C:\Program Files\IBM\WebSphere\AppServer.

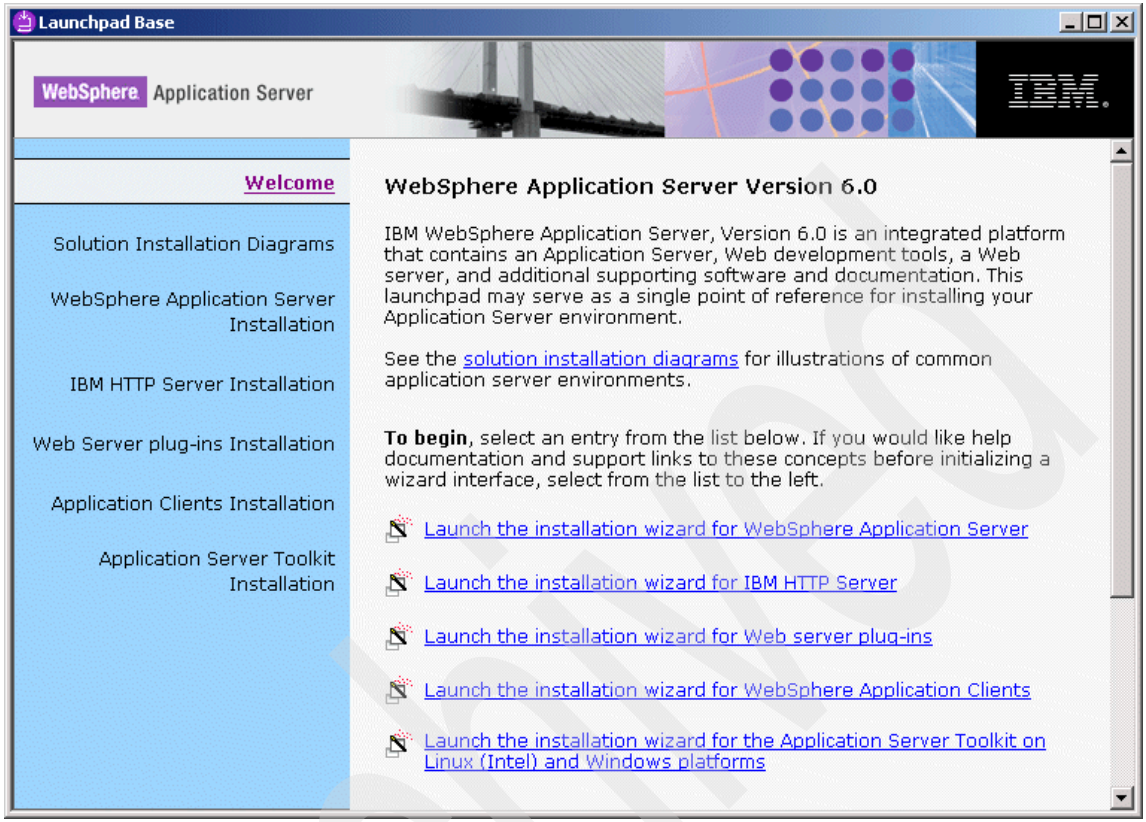


Figure A-7 IBM WebSphere Application Server V6.0 installation components

Table A-3 describes the IBM WebSphere Application Server V6.0 installation components and usage within this redbook.

Table A-3 IBM WebSphere Application Server V6.0 installation components

Installation component	Description	Use in redbook
WebSphere Application Server installation	Base WebSphere Application Server installation files.	Required * Target for deploying ITSO application sample
IBM HTTP Server installation	IBM HTTP Server installation files, which can be used with the WebSphere plug-in.	Optional

Installation component	Description	Use in redbook
WebSphere plug-ins	There are several supported WebSphere plug-ins depending on the Web server. If you install the IBM HTTP Server, you will need the IBM HTTP Server (apache) plug-in.	Optional
Application Clients installation	This included the IBM JRE V1.4.2, as well as libraries for J2EE clients.	Optional **Note**: Needed for J2EE Clients.
Application Server Toolkit installation	* Tool for assembly, deployment (EJB, Web Services) and debug J2EE applications. * No development support * WebSphere Rapid Deployment * Support for Enhanced EAR * Server Tools – support for remote server	Optional

For more detailed information on installing IBM WebSphere Application Server V6.0, refer to the following:

- IBM WebSphere Application Server V6.0 InfoCenter found at:

  `http://www.ibm.com/software/webservers/appserv/infocenter.html`
- *WebSphere V6 Planning and Design*, SG24-6446
- *WebSphere Application Server V6 Systems Management and Configuration*, SG24-6451

WebSphere Application Server messaging configuration

Note: This section is not required for the ITSO Bank sample application, as the ITSO Bank Enterprise Application does not use messaging, but is included for informational reasons.

The high-level configuration steps are as follows to configure messaging within WebSphere Application Server for the ITSO Bank application sample:

- Configure the service bus.
- Configure the bus members.
- Configure the destinations.
- Verify the messaging engine startup.

Archive

- Configure JMS connection queue factory.
- Configure the destination JMS queue.
- Configuration of a JMS activation specification.

Configure the service bus

To configure the service bus, do the following:

1. From the WebSphere Administrative Console, select **Service integration** → **Buses**.
2. Click **New**.
3. Enter the following and then click **OK**:
   - Name: `InterbankingBus`
   - Uncheck **Secure**.
4. Click **Save** and then when prompted click **Save** to Save to Master Configuration.

Configure the bus members

To configure the Bus members, do the following:

1. Select **Service integration** → **Buses**.
2. Click the **InterbankingBus** created in the previous section.
3. Under Additional Properties, click **Bus members**.
4. Click **Add**.
5. The default Bus member name is the <hostnameNode:server>. For example, our Bus name is was6jmg1Node01:server1. Accept the defaults for the remaining fields and click **Next**.
6. Click **Finish**.
7. Click **Save** and then when prompted click **Save** to Save to Master Configuration.

Configure the destinations

To configure the destinations for messaging, do the following:

1. Select **Service integration** → **Buses**.
2. Click **InterbankingBus**.
3. Under Additional Properties, click **Destinations**.
4. Click **New**.

Archived

5. When the Create new destination page appears, select **Queue** and then click **Next**.
6. Enter the Identifier. In our example we entered `interbankingQueue`. Click **Next**.
7. Select the Bus member created in "Configure the bus members" on page 1390 (for example, was6jmg1Node01:server1) and then click **Next**.
8. Click **Finish**.
9. Click **Save** and then when prompted click **Save** to Save to Master Configuration.
10. Log out of the WebSphere Administrative Console.
11. In order for the changes to take effect, you must restart the application server, in this case server1.
    a. Stop the server1 application server by clicking **Start** → **Programs** → **IBM WebSphere** → **Application Server V6** → **Profiles** → **default** → **Stop server**. Alternatively, use the **`stopServer.bat server1`** command.
    b. Start the server1 application server.

Verify the messaging engine startup

To verify the messaging engine startup, do the following:

1. Ensure the server1 application server has been restarted.
2. Start the WebSphere Administrative Console.
3. Select **Service integration** → **Buses**.
4. Click **InterbankingBus**.
5. Under Additional Properties, click **Bus members**.
6. Click **Node=was6jmg1Node01, Server=server1**.

   You should see the green arrow under Status to note the Messaging Engine has been started.

Configure JMS connection queue factory

To configure the JMS connection queue factory, do the following:

1. Select **Resources** → **JMS Provider** → **Default messaging**.
2. Click **JMS queue connection factory**.
3. Click **New**.

4. Enter the following and then click **OK**:

   Connection:

   – Name: `interbankingCF`
   – JNDI name: `jms/interbankingCF`

   Connection:

   – Bus name: Select **InterbankingBus**.

5. Click **Save** and then when prompted click **Save** to Save to Master Configuration.

Configure the destination JMS queue

To configure the destination JMS queue, do the following:

1. Select **Resources** → **JMS Provider** → **Default messaging**.
2. Click **JMS queue** under Destinations.
3. Click **New**.
4. Enter the following and then click **OK**:

   – Name: `InterbankingQueue`
   – JNDI name: `jms/interbankingQueue`
   – Bus name: Select **InterbankingBus**.
   – Queue name: Select **interbankingQueue**.

5. Click **Save** and then when prompted click **Save** to Save to Master Configuration.

Configuration of a JMS activation specification

The configuration for the JMS activation specification for the destination of the busification configuration is accomplished by the following simple steps:

1. Select **Resources** → **JMS Provider** → **Default messaging**.
2. Click **JMS activation specification** under Activation Specifications.
3. Click **New**.
4. Enter the following and then click **OK**:

   – Name: `interbankingAS`
   – JNDI name: `jms/interbankingAS`
   – Destination type: Select **Queue**.
   – Destination JNDI name: `jms/interbankingQueue`
   – Bus name: Select **InterbankingBus**.

Archived

5. Click **Save** and then when prompted click **Save** to Save to Master Configuration.

Archived

Archived

B

Additional material

The additional material is a Web download of the sample code for this redbook. This appendix describes how to download, unpack, describe the contents, and import the Project Interchange file. In some cases the chapters also require database setup; however, if needed, the instructions will be provided in the chapter in which they are needed.

This appendix is organized into the following sections:

- Locating the Web material.
- Unpack the 6449code.zip.
- Description of sample code.
- Import sample code from a Project Interchange file.

© Copyright IBM Corp. 2005. All rights reserved.

Locating the Web material

The Web material associated with this redbook is available in softcopy on the Internet from the IBM Redbooks Web server. Enter the following URL in a Web browser, and then download the 6449code.zip:

```
ftp://www.redbooks.ibm.com/redbooks/SG246449
```

Alternatively, you can go to the IBM Redbooks Web site at:

```
http://www.ibm.com/redbooks
```

Select the **Additional materials** and open the directory that corresponds with the redbook form number, SG246449.

The additional Web material that accompanies this redbook includes the following files:

File name	*Description*
6449code.zip	Zip file containing sample code

System requirements for downloading the Web material

The following system configuration is recommended:

Hard disk space:	20 MB minimum
Operating System:	Windows or Linux
Processor:	1 GHz
Memory:	1 GB

Unpack the 6449code.zip

After you have downloaded the 6449code.zip, unpack the zip file to your local file system using WinZip, PKZip, or similar software. For example, we have unpacked the 6449code.zip to the c:\6449code directory. Throughout the samples we will reference the sample code as if you have already unpacked the zip (for example, c:\6449code).

Description of sample code

Table B-1 on page 1397 describes the contents of the 6449code.zip file after being unpacked.

Archived

Table B-1 Sample code description

Unpack directory	Description
c:\6449code	Root directory after unpack of 6449code.zip
c:\6449code\java	Chapter 7, “Develop Java applications” on page 221 ► Java sample code packaged in BankJava.zip Project Interchange file.
c:\6449code\database	Chapter 8, “Develop Java database applications” on page 333 ► Java sample code to interact with databases packaged in BankDb.zip Project Interchange file. ► Database scripts used to create database tables (Table.ddl) and load sample data (loadData.sql) for both Cloudscape and DB2 UDB. These scripts are shared by many redbook chapters.
c:\6449code\gui	Chapter 9, “Develop GUI applications” on page 415 ► Java GUI sample code packaged in BankGUI.zip Project Interchange file.
c:\6449code\xml	Chapter 10, “Develop XML applications” on page 443 ► XML sample code packaged in BankXMLWeb.zip Project Interchange file.
c:\6449code\web	Chapter 11, “Develop Web applications using JSPs and servlets” on page 499 ► Web application sample code packaged in BankWeb.zip Project Interchange file. ► Source code, found in the source subdirectory.
c:\6449code\struts	Chapter 12, “Develop Web applications using Struts” on page 615 ► Web application using Struts sample code packaged in BankStrutsWeb.zip Project Interchange file.
c:\6449code\jsf	Chapter 13, “Develop Web applications using JSF and SDO” on page 673 ► Web application using JSF and SDO sample code packaged in BankJSF.zip Project Interchange file.
c:\6449code\egl	Chapter 14, “Develop Web applications using EGL” on page 751 ► Web application using EGL sample code packaged in BankEGL.zip Project Interchange file.
c:\6449code\ejb	Chapter 15, “Develop Web applications using EJBs” on page 827 ► EJB application sample code packaged in BankEJB.zip Project Interchange file.

Archived

Unpack directory	Description
c:\6449code\j2eeclt	Chapter 16, “Develop J2EE application clients” on page 925 ► The J2EE application client sample code packaged in BankAppClient.zip Project Interchange file. The BankAppClient_with_BankEJB.zip contains the EJB projects BankAppClient is dependent (import all complete projects needed).
c:\6449code\webservices	Chapter 17, “Develop Web Services applications” on page 951 ► Web Services application sample code packaged in BankWebServices.zip Project Interchange file.
c:\6449code\portal	Chapter 18, “Develop portal applications” on page 985 ► Portal application sample code packaged in Portal.zip Project Interchange file.

Import sample code from a Project Interchange file

This section describes how to import the redbook sample code Project Interchange zip files into Rational Application Developer. This section applies for each of the chapters containing sample code that have been packaged as a Project Interchange zip file.

To import a Project Interchange file, do the following:

1. Start Rational Application Developer.
2. From the Workbench, select **File** → **Import**.
3. From the Import dialog, select **Project Interchange** (as seen in Figure B-1 on page 1399), and then click **Next**.

Archived

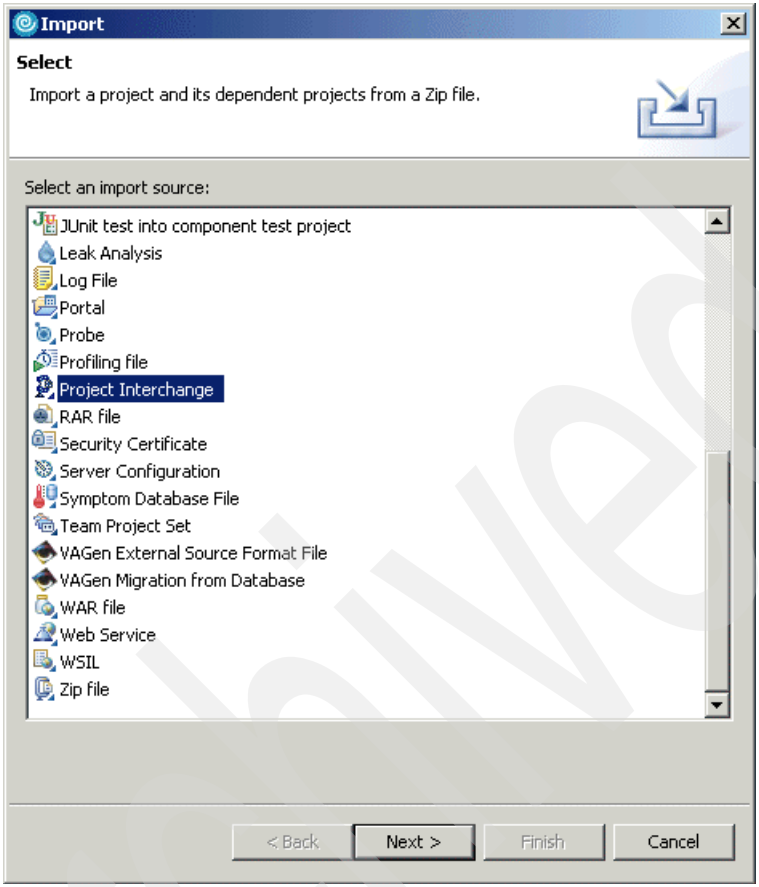


Figure B-1 Import a Project Interchange file

4. When prompted for the Project Interchange path and file name, and target workspace location, we entered the following:
   - From zip file: `c:\6449code\java\BankJava.zip`

     Enter the path and zip file name (for example, c:\6449code\java\BankJava.zip).
   - Project location root: `c:\workspace`

     Enter the location of the desired workspace (for example, our workspace is found in c:\workspace).
5. After entering the zip file, check the project and then click **Finish**.

   For example, we checked **BankJava** and clicked **Finish**, as seen in Figure B-2 on page 1400.

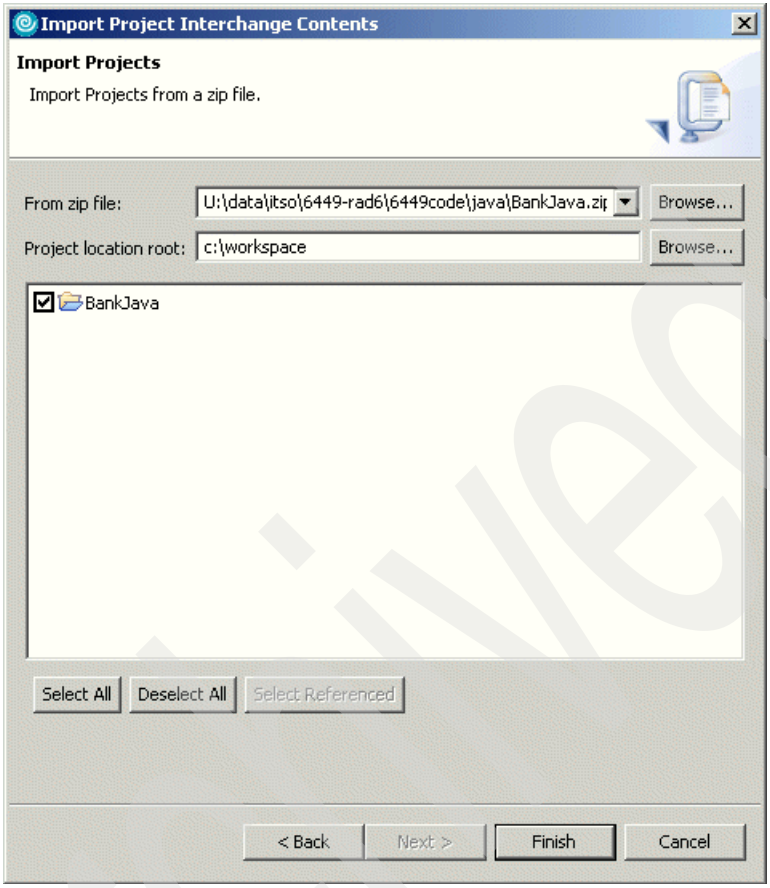


Figure B-2 Import Interchange Projects location

Archived

Related publications

The publications listed in this section are considered particularly suitable for a more detailed discussion of the topics covered in this redbook.

IBM Redbooks

For information on ordering these publications, see “How to get IBM Redbooks” on page 1404. Note that some of the documents referenced here may be available in softcopy only.

- *EJB 2.0 Development with WebSphere Studio Application Developer*, SG24-6819
- *WebSphere Studio V5.1.2 JavaServer Faces and Service Data Objects*, SG24-6361
- *IBM WebSphere Portal V5.1 Portlet Application Development*, SG24-6681
- *IBM WebSphere Portal V5 A Guide for Portlet Application Development*, SG24-6076
- *WebSphere Application Server V6: Web Services Development and Deployment*, SG24-6461
- *WebSphere V6 Planning and Design*, SG24-6446
- *WebSphere Application Server V6 Scalability and Performance*, SG24-6392
- *WebSphere Application Server V6 Security*, SG24-6316
- *WebSphere Application Server V6 Systems Management and Configuration*, SG24-6451
- *Eclipse Development using the Graphical Editing Framework and the Eclipse Modeling Framework*, SG24-6302
- *Transitioning: Informix 4GL to Enterprise Generation Language (EGL)*, SG24-6673
- *WebSphere Studio Application Developer Version 5 Programming Guide*, SG24-6957

Archived

© Copyright IBM Corp. 2005. All rights reserved.

Other publications

These publications are also relevant as further information sources:

- Crupi, Malks and Alur, *Core J2EE Patterns: Best Practices and Design Strategies*, Prentice Hall, 2003, ISBN 0131422464
- Marinescu, *EJB Design Patterns: Advanced Patterns, Processes and Idioms*, Wiley, 2002, ISBN 0471208310
- Shavor, D'Anjou, Fairbrother, Kehn, Kellerman and McCarthy, *The Java Developer's Guide to Eclipse*, Addison-Wesley, 2003, ISBN 0321159640
- Eric Gamma, et al., *Design Patterns, Elements of Reusable Object-Oriented Software*, Addison-Wesley Professional, 1995, ISBN 0-201-63361-2
- *IBM Informix 4GL to EGL Conversion Utility User's Guide*, G251-2485

Online resources

These Web sites and URLs are also relevant as further information sources:

- Sun Microsystem's Java site, with specifications, tutorials, and best practices

  `http://java.sun.com/`
- The Eclipse Project site, with information on the underlying platform of Rational Application Developer

  `http://www.eclipse.org/`
- Eclipse Hyades project

  `http://www.eclipse.org/hyades`
- The WorldWide Web Consortium

  `http://www.w3c.org/`
- Apache Jakarta Project, for information on Tomcat

  `http://jakarta.apache.org/`
- Apache Struts site

  `http://struts.apache.org/`
- The Java Community Process site, for Java specifications

  `http://www.jcp.org/`
- OASIS, for UDDI

  `http://www.oasis-open.org/`

Archived

- Web Services Interoperability Organization

  http://www.ws-i.org/

- The ServerSide.com is an enterprise Java site with articles, books, news, and discussions

  http://www.theserverside.com/

- *Writing Robust Java Code* white paper by Scott Ambler

  http://www.ambysoft.com/javaCodingStandards.pdf

- Apache Ant Project home page

  http://ant.apache.org/

- Apache Ant documentation

  http://ant.apache.org/manual/index.html

- CVS home

  http://www.cvshome.org

- CVSNT

  http://www.cvsnt.org

- CVSNT installation tips

  http://www.cvsnt.org/wiki/InstallationTips

- IBM developerWorks EGL home page

  http://www.ibm.com/developerworks/rational/products/egl/

- *Generating Java using EGL and JSF with WebSphere Studio Site Developer V5.1.2*, white paper

  http://www.ibm.com/developerworks/websphere/library/techarticles/0408_barosa/0408_barosa.html

- VisiBone HEX HTML Color Codes

  http://html-color-codes.com/

- The complete J2SE specification

  http://java.sun.com/j2se/

- Information on JDBC

  http://java.sun.com/j2se/1.4.2/docs/guide/jdbc/

- Information on the AWT

  http://java.sun.com/j2se/1.4.2/docs/guide/awt/

- Information on Swing

  http://java.sun.com/j2se/1.4.2/docs/guide/swing/

Archived

- Information on the SWT

  http://www.eclipse.org/swt/
- XML - the World Wide Web Consortium (W3C)

  http://www.w3c.org/XML/

How to get IBM Redbooks

You can search for, view, or download Redbooks, Redpapers, Hints and Tips, draft publications and Additional materials, as well as order hardcopy Redbooks or CD-ROMs, at this Web site:

http://www.ibm.com/redbooks

Help from IBM

IBM Support and downloads

http://www.ibm.com/support

IBM Global Services

http://www.ibm.com/services

Archived

Index

Symbols
.eglbld 766

Numerics
6449code.zip 1396

A
abstract facade
  implement 560
Abstract Window Toolkit 36, 416
access control
  list 833
access CVS repository 1310
accessor methods 257
accessors 254, 557
AccountDetails 514
ACID 829
ACL 833
Active Script 1124
activity 1259
add a project to a server 1064
adding emulator support 994
additional material 1395
additional supported test servers 23
aliases 357
analyze test results 1113
animated GIF 511
AnimatedGif Designer 511
annotated programming 15
Annotation-based programming 1210
Ant 1155–1156
  build
    path 1157
    project 1156
    property 1157
    target 1156
    task 1156
  build J2EE application 1178
    create build script 1180
    deployment packaging 1179
    prepare for sample 1179
    run Ant build 1183
build simple Java application 1167
  build targets 1171
  classpath problem 1178
  clean 1173
  compile 1172
  create build file 1169
  forced build 1177
  global properties 1171
  init 1172
  project definition 1171
  rerun Ant 1177
  run Ant 1173
documentation 1157
headless build 1186
introduction 1156
  build files 1156
  new features 1157
  tasks 1157
J2EE
  applications 1178
  build script 1180
new features
  Code Assist 1158
  Code snippets 1159
  define format of an Ant script 1164
  format an Ant script 1163
  Problem view 1166
run 1173
Run Ant wizard
  Build 1175
  Classpath 1176
  Common 1176
  Environment 1176
  JRE 1176
  Main 1175
  Properties 1176
  Refresh 1175
  Target 1175
run headless build 1187
run outside Application Developer 1185
runAnt.bat 1186
script 1171
tasks 1157
  delete 1157

© Copyright IBM Corp. 2005. All rights reserved.

echo 1157
jar 1157
javac 1157
mkdir 1157
tstamp 1157
Web site 1157
Ant build files 1156
Apache Jakarta
Ant 1156
appearance
preferences 100
appearance of Java elements 100
applet 501
application classloader 1195
application client module 500, 1191
application code analysis 15
application deployment 1190
application deployment descriptor
access 630
application flow 842
application modeling with UML 15
application profiling 1237
Application Server
Network Deployment 830
Application server profile 1048
application.xml 501
ApplicationResource.properties 652
ArrayList 269
assertEquals 1096
assertions 1087
assertTrue 1096
association 267
association relationship 267, 874
atomicity 829
attributes 254
automated component test
run test 1110
automated component testing 1102
automatic build 85
Automatic fail-over support 830
AWT
See Abstract Window Toolkit

B

backend folder 895
banking model 514
baseline 1259
make 1282
Basic 753
bean 501
bean-managed persistence 836
BMP 836
bootstrap 1192
bottom up 893
BPEL
See Business Process Execution Language
breakpoint 1124, 1132
condition 1134
conditional 1134
JSP 1135
properties 1133
set 1132
Breakpoints view 148
breakpoints 1142
breakpoints view 1142
Browse diagram 197
build 85, 1155
Ant
build file 1156
compiler 1171
targets 1171
build applications 1155
build verification test 1082
build.xml 1167
J2EE application 1180
Java application 1170
bus members 1390
Business Process Execution Language 6
BVT
See build verification test

C

C 32, 753
C++ 32
calibration 1084
Call Hierarchy view 223
capabilities 86
Cascading Style Sheets 42
change
variable value 1143
check in and check out 1259
check out 1289
cHTML 994
Class Diagram 194, 205, 233, 243, 370
classes directory 502
classloaders 1195

Archived

WebSphere classloaders
RCP directory 1194
RE directory 1194
RP directory 1194
classpath
add JAR 297
Ant
classpath 1178
ClearCase 1258
add project 1274
check out 1289
connect 1269, 1284
deliver 1279
help 1261
import 1287
preferences 1261
project 1265
rebase 1290
scenario 1263, 1277
setup 1264
ClearCase LT 1259
Cloudscape CView 344
Cloudscape JDBC driver 391
CMP 836
CMP attributes 863
CMR 838
See container-managed relationships
COBOL 753
code
assist 121
formatter preferences 101
Code Assist 118, 320
code coverage analysis 1240
code formatter 105
blank lines 110
braces 108
comments 114
control statements 112
indentations 108
line wrapping 113
new lines 111
white space 109
Code Review 124, 226
add filter 126
add rule 126
Complete 125
excluding matches 130
exclusion of reources 129
Globalization 125
J2EE Best Practices 125
J2SE Best Practices 125
Quick 124
set options 125
Code Review Details view 229
Code Review view 226
code style 102
code style and formatting 101
coexistence 26
collection type 269
COM.ibm.db2.jdbc.app.DB2Driver 392
COM.ibm.db2.jdbc.net.DB2Driver 392
command
design pattern 515
Common Object Request Broker Architecture 34
compact Hyper Text Markup Language 994
compare a file 92
compare with 93
compatibility 27
compilation target 1172
compiler options 115
preferences 115
component 1259
interface 838
component interfaces 881
component test 1082
automated 1102
component test benefits 1085
Component Test project 177
component testing 17, 1081
component-under-test 1102
composer 844
composition relationship 876
concurrency 832
Concurrent Version System 1299
condition
breakpoint 1134
configuration
WebSphere Application Server
messaging 1389
conflict 1300
connect to ClearCase 1269
connection
JDBC
connection 334
pooling 335
Connector project 176
consistency 829, 1086
Console view 146

Archived

constructors 262, 326, 882
container
EJB 832
container-managed
persistence 836
relationships 838
container-managed persistence 892
Container-managed relationships 60
content area 533
context root 519
controller
MVC 504, 514
converter 844
cookies 41
CORBA
See Common Object Request Broker Architecture
Core Java APIs 33
create
Class Diagram 551
database 344
database connection 347
database schema 357
database tables from scripts 349
DTD file 450
fields 554
Java stored procedures 398
model classes 547, 550
packages 551
page template 526
simple Ant build file 1169
SQL statements 376
static Web resources 544
table 357–358
test case
create 1093
visual class 420
Web Project 517
XML schema 463
XSL file 483
Crystal Reports integration 16
CSS
See Cascading Style Sheets
CSS Designer 511
CSS File wizard 513
custom finders 890
Custom profile 1048
customer acceptance test 1083
CVS 1299
client configuration
access CVS repository 1310
enable CVS Team capability 1309
configuration 1312
development scenario 1321
features 1300
introduction 1300
new features for team development 1301
preferences 1312
Repositories view 1324
repository
add 1323
Resource History view 1343
scenario 1321
Web site 1301
CVS Annotate view 144
CVS client configuration 1309
CVS for NT 1301
CVS preferences
Rational Application Developer 1312
CVS specific settings 1317
file content 1313
ignored resources 1315
label decorations 1312
CVS Repositories view 143
CVS Repository Exploring perspective 143
CVS Resource History view 144
CVSNT
create CVS users 1308
create Windows users and groups 1306
server implementation 1301
server installation 1302
server repository configuration 1303
verfiy installation 1307

D

DADX 957
Data Definition view 145, 340
Data perspective 145, 339
data source 335, 393
configure 628, 630
deployment descriptor 630
objects 335
database
create
Cloudscape CView 343
create connection 347
create database

Archived

DB2 command window 346
create database tables
Application Developer 350
Cloudscape CView 351
DB2 command window 351
defintion 355
objects 354
populate data in tables
Application Developer 352
Cloudscape CView 353
DB2 command window 354
schemas 357
table 358
Database Explorer view 146, 341
databaseName property 631
DB Output view 146, 342
DB2 UDB JDBC drivers 392
DB2 Universal Database 1387
class path environment variable 783
db2java.zip 392
DB2UNIVERSAL_JDBC_DRIVER_PATH 783
DDL file
copy objects to workspace 362
deploy to database 362
generate 358, 361
generate for database objects 365
debug 1122
Web application on local server 1132
breakpoints view 1142
debug functions 1141
debug JSP 1145
debug view with stack frames 1141
evaluate an expression 1144
inspect variable 1143
run application in debug 1136
set breakpoint in JSP 1135
set breakpoint in servlet 1132
start application for debug 1136
watch variables 1142
Web application on remote server 1145
attach to remote server 1148
configure debug in WAS 1147
debug application 1151
deploy WAR 1146
export project to WAR file 1145
install Agent Controller 1147
XSLT 493
debug features
breakpoints 1124
drop to frame 1126
step filter 1126
view management 1127
XSLT debugger 1128
Debug perspective 146
debug tooling
supported environments
WebSphere Application Server 1124
WebSphere Portal 1124
supported languages 1124
compiled languages 1124
EGL 1124
Java 1124
JavaScript 1124
mixed language 1124
SQL stored procedures 1124
SQLJ 1124
XSLT 1124
Debug view 147, 1141
debugging
icons 1141
remote 1145
declaration 259
Declaration view 225
declarative programming language 752
default CMP data source 858
default workspace 82
deliver 1279
stream 1259
delta versioning 1300
demarcation 833
deployment 1189
common considerations 1190
deploy the enterprise application
configure data source in WAS 1225
deploy the EAR 1229
descriptors 1196
J2EE application components
applets 1191
application clients 1191
EJBs 1191
Web applications 1191
J2EE deployment modules
application client module 1191
EJB module 1191
resource adapter module 1191
Web application module 1191
J2EE packaging
EAR 1191

WAR 1191
Java and WebSphere class loader 1191
package an application 1218
customize deployment descriptors 1220
export the EAR 1222
generate deploy code 1222
generate EJB to RDB mapping 1218
recommendations 1218
prepare for sample
deployment scenario 1213
import sample code 1214
install prerequisite software 1213
verify the enterprise application 1230
WebSphere deployment architecture 1199
WebSphere enhanced EAR 1201
WebSphere Rapid Deployment 1210
deployment descriptor 1197
application.xml 501
enterprise application 501
Web application 502
Web module 502–503
web.xml 502
Deployment Manager profile 1048
derived beans 870
desktop applications 32
destination JMS queue 1392
destinations 1390
development stream 1259, 1270, 1285
Diagram Navigator view 230
Display view 1143
distribution 829
document access definition extension 957
Document Type Definition 445
doGet 579
doPost 579
DriverManager 335, 390
drop to frame 1126
DTD 445
DTD editor 448
features 452
DTD file
create 449
validate 456
DTO 842
durability 829
Dynamic Web Project 679
Dynamic Web project 175
dynamic Web resources 549

E

EAR 1179, 1191
EAR file 500
Eclipse and IBM Rational Software Development Platform 14
Eclipse Hyades 22, 1081, 1086
Web application testing 1112
analyzing test results 1113
editing a test 1112
generate an executable test 1113
recording a test 1112
run a test 1113
Eclipse Java Development Tools 21
Eclipse Modeling Framework (EMF) 22
Eclipse Platform 20
Eclipse Plug-in Development Environment 21
Eclipse Project 19
Eclipse Platform 20
Java Development Tools 21
Plug-in Development Environment 21
Eclipse Software Developer Kit (SDK) 21
editing a test 1112
Editing JSP 595
editors 137
EGL 751
debug 765
introduction 752
migration
EGL migration to V6.0 764
Informix 4GL to EGL Conversion Utility 764
VisualAge Generator to EGL migration 764
overview
application architecture 756
feature enhancements 759
history 754
Rational brand software 758
target audience 754
value proposition 755
perspective and views 762
EGL Debug Validation Errors 762
EGL Generation Results 762
EGL Parts Reference 762
EGL SQL Errors 762
EGL Validation Results 762
preferences 761
programming paradigms 752
projects
EGL Project 763
EGL Web Project 763

Archived

tooling in Rational Application Developer 761
Web application components
data 766
Deployment Descriptor 765
EGL Web Project 765
EGLEAR 765
EGLSource folder 766
filename.eglbld 766
JavaSource folder 766
libraries 766
pagehandlers 766
WebContent folder 766
where to find more information 761
wizards
Data Table 764
EGL Build File 763
EGL Data Parts 764
EGL Data Parts and Pages 764
EGL Package 763
EGL Source File 763
EGL Source Folder (EGLSource) 763
Faces JSP File 763
Form Group 764
Library 763
Program 763
EGL Data Parts and Pages wizard 784
EGL Project 763
EGL Web application 751
add EGL components to Faces JSPs 806
create a connection between Faces JSPs 804
create a Faces JSP 803
create a Web Diagram 802
create a Web page 802
create an EGL Web Project 773
create Faces JSP 802
create page template 794, 799
create records and libraries 785
develop the application 783
EGL Data Parts wizard 789
exporting an EGL project
add runtime libraries 822
export WAR - EAR with source 823
reduce file size 821
generate Java code from EGL 794
import and run EGL sample Web application 816
prepare
configure data source 780
configure EGL preferences for SQL data-base connection 779
enable EGL development capability 771
install EGL component 768
EGL Web Project 763
EGL wizards 763
EJB 827
application 842
bean class 837
client 837
component 834
component interface 838
container 832
home interface 838
inheritance 839
JARs 1179
new features 828
overview 828
project 844
create 846
QL 840
custom finder 890
Query Language 839
See Enterprise Java Bean
server 831
specification 832
types 836
universal test client 915
EJB application
develop the application
add business logic 880
association relationship 874
composition relationship 876
create custom finders 890
create entity beans 859
create entity relationships 872
customize entity beans 880
implement session facade 901
object-relational mapping 892
prepare for development 844
configure data source 856
configure EJB project 849
create EJB project 844–845
import Web application project 853
setup sample database 854
testing with UTC 915
EJB JAR 171
EJB local and remote interfaces 61
EJB module 500, 1191
EJB project 176

Proofed

EJB query language 60, 839
EJB Timer Service 61
enabling transcoding for development 994
encapsulation 842
Enhanced EAR Editor 1201
Enhanced EAR tooling 1070
enterprise application 500
Enterprise Application Archive 1191
Enterprise Application project 175
Enterprise archive
  See EAR
Enterprise Generation Language 751
Enterprise Java Server 831
Enterprise JavaBean
  types
    entity 59
    message-driven 59
    session 58
Enterprise JavaBeans 57, 827
  architecture 831
  components
    client views 834
    EJB types 836
  EJB container 832
    concurrency 833
    life cycle 834
    messaging 834
    naming 833
    persistence 833
    security 833
    transaction 833
  EJB types
    entity beans 836
    message-driven beans 837
    session bean 836
  overview
    distributed 829
    persistent 829
    portable 830
    scalable 830
    secure 828
    transactional 829
entity beans 836
  create 859
  customize 880
Entity EJBs 59
entity relationships 872
E-R Modeling 195
evaluate an expression 1143
exception classes 572
execution time analysis 1240
  views 1240
export
  Java code to JAR file 299
export EAR
  export the EAR file 1223
  filtering the ontent of the EAR 1222
Expression Language 48
Expressions view 1143
extends 267
Extensible Markup Language 38, 444
Extensible Style Language 446
extensions classloader 1194–1195

F

facade 516
Faces Action 708
Faces JSP File wizard 513
fail 1096
features 12
  summary 13–14
    specification versions 14
fields 554
file
  associations 90
File Creation wizard 512
filter errors 289
filters 48
find errors in Problems view 288
folders 171
foreign key 358
form bean 619
forms 548
Fortran 753
Free Layout 544
Free Table Layout 533
function verification test 1083
FVT
  See functional verification test

G

generate EJB to RDB mapping 895
generate getter and setter 324
generating an executable test 1113
Generic Log Adapter perspective 148
getter 254, 257
graphical user interfaces 35

Archived

Abstract Window Toolkit 36
Java components 37
Standard Widget Toolkit 36
Swing 36
GROUP BY 382
GUI Java application
add event handling 434
prepare
Add Cloudscape JDBC driver 417
create Java Project 417
import model classes 419
setup database 418
testing 433
verify sample application 435
Visual Editor 423

H

HEAD branch 1327
headless
Ant
build 1185
headless build 1186
history 92
hit count 1134
home interface 838, 880–881
host variable 382
HTML 42, 501, 994
HTML error tag 657
HTML File wizard 512
HTTP 40
See Hypertext Transfer Protocol
status codes 40
HyperText Markup Language 42, 994
Hypertext Transfer Protocol 40, 43

I

IBM DB2 Universal Database V8.2
installation 1387
IBM Eclipse SDK V3.0 21
IBM Enterprise Generation Language 753
IBM Rational Agent Controller 9, 1102, 1382
IBM Rational Agent Controller V6
installation 1382
IBM Rational ClearCase 1258
IBM Rational Software Development Platform 4
products 5
Rational Application Developer 6
Rational Function Tester 6
Rational Performance Tester 6
Rational Software Architect 5
Rational Software Modeler 5
Rational Web Developer 6
WebSphere Business Integrator Modeler 6
IBM WebSphere Application Server V6.0 1387
IBM WebSphere Portal 56, 989
icons for debugging 1141
IDE
See integrated development environment
IDEF1X Diagram 375
IDEF1X notation diagram 196
IE notation diagram 196
image 501
Image File wizard 513
implements relationship 267
import
from ClearCase 1287
generation 321–322
Java JAR 301
imports
resolve (Ctrl+Shift+O) 255
resolve (Ctrl+Shift-O) 256
indexes 357
Information Engineering (IE) Diagram 374
Informix 4GL migration 764
inheritance 839
init 579
Initialization target 1172
inspect variable 1143
Installation
CVS for NT 1302
installation
IBM DB2 UDB V8.2 Express Edition 1387
IBM Rational Agent Controller 1382
IBM Rational Application Developer 1372
Rational Application Developer
Interim Fix 0004 1380
Rational ClearCase LT 1385
WebSphere Application Server V6 1387
WebSphere Portal V5.0 Test Environment 1376
WebSphere Portal V5.1 Test Environment 1377
Installed JREs
preferences 119
integrated development environment 132
integrated test servers 23
integration

Archived

stream 1259, 1279
interface 258, 834
Interim Fix 0004 1380
Internet
preferences 95
Internet preferences 95
Proxy settings 95
Web Browser settings 96
Introduction
application development challenges 7
Java database programming 334
Rational Software Development Platform 4
version 6 terminology 7
InvalidateSession 514
ISD 958
isolation 829
itso.ant.hello 1168

J

J2EE Application Client JAR 171
J2EE Application Client project 175
J2EE Application Clients 62
J2EE Connector 18
J2EE Deployment API 19
J2EE Management API 19
J2EE modules and projects 173
J2EE perspective 149
J2EE Request Profiling Agent 1244
J2EE Visualization 219
JAAC
See Java Authorization Service Provider Contract for Containers
JAAS
See Java Authentication and Authorization Service
JAF
See Java Activation Framework
JAR 501
Java
development
preferences 98
Editor
preferences 117
Runtime Environment 119
Scrapbook 293
source folder 523
test case 1110
utility JAR 1179
Java 2 Platform Enterprise Edition 170
Java accessor methods 254
Java Activation Framework 18
Java API for XML Processing 18
Java API for XML Registries 19
Java API for XML RPC 18
Java application
export code to JAR 299
locate compile errors 287
filter errors in Problems view 289
Problems view 288
run external 301
run sample 269, 286
working example 231
Java attributes 254
Java Authentication and Authorization Service 19
Java Authorization Service Provider Contract for Containers 19
Java Beans 231
Java Browsing perspective 153
Java class
create via Diagram Navigator 250
Java Class wizard 253
Java class loader 1192
extensions class loader 1192
hierarchy 1193
system class loader 1192
Java classes 249
Java classpath variables 98
Java component test 1103
Java database application
access database via data source 393
access database via DriverManager 390
load JDBC driver 390
prepare
setup BANK database 338
prepare for sample 337
Java database programming 334
Java development preferences 98
appearance of Java elements 100
code style and formatting 101
Java classpath variables 98
Java Editor 311
Java Editor settings 117
Java Field wizard 255
Java interface 248
Java language 32
Java Management Extensions 19
Java Message Service 18, 71

Archived

Java methods 262
Java Native Interface 36
Java package
create via Diagram Navigator 247
create via Java Package wizard 247
Java packages 246
Java perspective 151, 222
Java Portlet specification 56
Java Profiling Agent 1244
Java project 177, 235
Java Remote Method Invocation 34
Java Scrapbook 293
Java Search 318
Java Servlet 18, 499
Java stored procedures 394
access via DriverManager 408
access via JavaBean 409
build 405
deploy 405
enable capability 398
Store Procedure wizard 398
Java Transaction API 18
Java Type Hierarchy perspective 155
Java Virtual Machine 33
Javadoc 231, 303
generate 304
Ant script 310
Export wizard 306
preferences 121
JavaMail 18
Javascript Editor 511
Javascript File wizard 513
JavaServer Faces 19, 52
application architecture 675
benefits 674
features 674
overview 674
JavaServer Page 18
JavaServer Pages 46, 499
JAXP
See Java API for XML Processing
JAXR
See Java API for XML Registries
JAX-RPC
See Java API for XML RPC
JDBC 34, 334
2.0 Standard Extension API 335
type 2 driver 392
type 3 driver 392
JDBC provider
configure 631
JDT
See Java Development Tools
JMS
See Java Message Service
JMS activation specification 1392
JMS connection queue factory 1391
JMX
See Java Management Extensions
JNDI
data source 335
namespace 833
See Java Naming and Directory Interface
JNI 1195
See Java Native Interface
join
project 1284
JRE 119
JSF
See JavaServer Faces
JSF and SDO Web application
add connection for action 712
add navigation rules 713
configure data source via enhanced EAR 681
create connection between JSF pages 705
create Dynamic Web Project 679
create Faces Action 708
create Faces JSP 700, 702
create page template 684
customize page template 695
content area 699
logo image and title 695
style (fonts, size) 698
edit JSF page
add relational record 730
add relational record list to page 738
add relational record to page 733
add reusable JavaBean 742
add row action 741
add SDO to JSF page 720
add simple validation 718
add static navigation to page 719
add UI components 715
add validation 727
add variables 716
display relational record 735
format field for currency 740
link button 733

Archived

link button to action 735
run sample application 746
setup database 681
JSP 46
breakpoint 1135
Tag libraries 47
JSP File wizard 512
JSP Page Designer 619
JSPs 595
JSR 168 993
JTA
See Java Transaction API
JUnit 231, 1081, 1087
assert method 1089, 1096
automated component test 1102
class
TestCase 1088
TestRunner 1088
testRunner 1088
TestSuite 1088
create test case 1089
fail method 1096
fundamentals
instantiate an object 1087
invoke methods 1087
verify assertions 1087
methods 1096
run test case 1098
setUp and tearDown 1093
view 1099
junit.framework.Assert 1096
junit.jar 1091
JVM
See Java Virtual Machine

K

key
class 838
field 863
wrapper class 864
Key themes of version 6 8
broaden appeal 8
extended integration 8
maintain standards 8
raise productivity 8
team unifying platform 8
keyword
expansion 1318

L

Layout Mode 544
lib directory 502, 524
licensing 23
life cycle 832, 834
links 544
Linux 1155
list 547
ListAccounts 514
listeners 49
local
component interface 880
history 92
local history 92
compare with 93
replace with 93
locate compile errors 287
location independence 835
Log Console view 1176
log files 83

M

Maintenance 1086
make 1155–1156
MANIFEST.MF 524
mapping
strategies 893
mark occurrences 118
marker
breakpoint
marker 1133
MDB
See Message Driven Bean
Meet in the middle 893
memory analysis 1239
memory analysis views 1239
merging 1354
Merging from a stream 1354
message-driven bean 837
Message-driven EJBs 59, 72
messaging 832, 834
messaging engine startup 1391
messaging resources 1079
messaging systems 70
META-INF 523
method
accessor 257
declaration 258

Archived

implement 325
override 325
Microsoft Windows 2000 Professional 9
Microsoft Windows 2000 Server 9
Microsoft Windows 2003 Enterprise Edition 9
Microsoft Windows 2003 Standard Edition 9
Microsoft Windows XP Professional 9
migration 26
model
MVC 514
model classes 550
model-view-controller 51, 503
controller 504
dependencies between layers 505
JSP and servlet Web application 514
model 503
Struts Web application 616
view 504
multi-dimensional association 269
MVC 514
controller 504
pattern 505
See model-view-controller
Struts 617

N

namespace 833
naming 832–833
Navigator view 146, 343
no natural unique identifier 870

O

object
caching 830
pooling 830
Object-relational mapping
bottom up 893
meet-in-the-middle 893
top down 893
object-relational mapping 892
ODBC 334
online help 132
ORDER BY 382
Outline view 148, 230, 311
override methods 325

P

Package Explorer view 223
packages
create 551
Page Designer 509
page template 510, 526
create dynamic JSP 530
create static 527
customize 531
Page Template File wizard 513
page templates versus style sheets 527
parallel development 1335
Pascal 753
pattern
command 515
MVC 514
PDE
See Plug-in Development Environment
performance test 1083
PerformTransaction 515
persistence 829, 832–833
perspective layout 137
perspectives 136
customizing 140
CVS Repository Exploring perspective 143
Data perspective 145
Debug perspective 146
Generic Log Adapter perspective 148
J2EE perspective 149
Java Browsing perspective 153
Java perspective 151, 222
Java Type Hierarchy perspective 155
Plug-in Development perspective 157
preferences 94
Profiling and Logging perspective 158
Resource perspective 159
specify default 140
switching 138
Team Synchronizing perspective 160
Test perspective 161
Web perspective 162
pluggable JRE 296
Plug-in Development perspective 157
populate
database tables with data 352
portabiliy 830
portal application development 14
Portal applications 55
portal applications 985

Archive

develop 992
development strategy
choosing markup languages 994
IBM or JSR 168 993
JavaServer Faces 995
Struts 995
introduction 986
run project in test environment 1037
samples and tutorials 992
tools
coexistence and migration 1023
deploy projects 1001
portal administration 1022
Portal Designer 998
Portal Import wizard 996
Portal Project wizard 997
Portal Server Configuration 1016
Portlet Project wizard 1005
remote test server 1017
skin and theme design 999
WebSphere Portal test environment 1018
portal concepts and definitions
portal page 986
portlet 987
portlet application 987
portlet events 988
portlet modes 988
portlet states 988
Portal Designer 998
Portal Import Wizard 996
portal page 986
add portlets 1033
customize 1027
Portal Project
create 1026
Portal Project wizard 997
portal samples and tutorials 992
portal test environments
WebSphere Portal V5.0 990
WebSphere Portal V5.1 990
portal tools 990
portlet 987
portlet application 987
portlet applications 985
portlet events
action events 988
message events 989
Window events 989
portlet modes
configure 988
edit 988
help 988
Portlet Project Wizard 1005
portlet states 988
maximized 988
minimized 988
normal 988
portlet modes 988
preferences 84
capabilities 86
ClearCase 1261
CVS 1312
file associations 90
local history 93
perspectives 94
startup and shutdown 83
primary key 358
probekit analysis 1241
Problems view 148, 225
procedure programming languages 753
Process Advisor 191
Process Browser 192
Process Preferences 193
product configuration 1371
product features 12
product installation 1371
production environment 1084
profiles
WebSphere Application Server 1201
profiling
agent types
J2EE Request Profiling Agent 1244
Java Profiling Agent 1244
architecture 1242
agent 1243
Agent Controller 1243
application process 1243
deployment hosts 1243
development hosts 1244
test client 1243
features
code coverage 1240
execution time analysis 1240
memory analysis 1239
probekit analysis 1241
thread analysis 1239
prepare for profiling sample
enable Profiling and Logging capability

1247
prerequiste hardware and software 1247
publish and run sample application 1249
profile the sample application
analysis of code coverage information 1253
collect profile information 1253
start server in profile mode 1249
Profiling and Logging perspective 1244
profiling sets 1245
Profiling and Logging capability 1244
Profiling and Logging perspective 158, 1244
profiling applications 1237
profiling sets 1245
Profiling tools 17
programming languages
declarative 752
programming paradigms
4GL 753
object oriented 753
procedural 753
programming technologies 31
desktop applications 32
dynamic Web applications 43
enterprise JavaBeans 57
J2EE Application Clients 62
messaging systems 70
static Web sites 39
Web Services 66
project
ClearCase control 1274
create
Java 235
create a new 178
directory structure 522
disconnect from CVS 1360
EJB 844
join 1284
properties 178, 180
version 1342
Project Explorer view 149
Project Interchange file 1398
projects 171
promote 863
Proxy Settings 95

Q

Quick Assists 320
Quick Fix 320

R

ramp-up time 1086
Rapid Application Development 311
features
Code Assist 321
generate getter and setter 324
import generation 322
Java Search 318
navigate through the code 311
Quick Assist 320
Smart Insert 316
source folding 314
Type Hierarchy 315
Word Skipping 317
working sets 319
RAR 171
Rational
ClearCase
see ClearCase
Web site 1259
Rational Application Developer 6
CDs 10
ClearCase LT integration 1257
configure CVS 1312
CVS preferences 1312
database features 336
debug
attach to remote server 1148
debug tooling 1121
summary of new features 1122
Eclipse Hyades test framework 1086
editors 137
EGL tooling 761
folders 172
installation 24, 1372
JavaServer Faces support 677
JUnit 1087
licensing 23
local vs remote test environment 1046
log files 83
migration and coexistence 26
new EJB features 828
new server 1057
online help 132
perspectives 136
preferences 84
product packaging 10
projects 171
Component test 177

Archived

Connector project 176
Dynamic Web project 175
EJB project 176
Enterprise Application project 175
J2EE Application Client project 175
Java project 177
Server project 177
Simple project 177
Static Web project 176
summary 173
Rational ClearCase features 1261
samples 181
server configuration 1043
startup parameters 81
Struts 619
Stuts support
Project Explorer view 619
Struts Component wizard 619
Struts Configuration Editor 619
Web Project Struts enabled 619
supported test servers 1045
test server introduction 1044
uninstall 25
views 137
Web Services tools 957
XML tools 447
Rational ClearCase
ClearCase preferences 1261
new features 1261
terminology
activity 1259
baseline 1259
check in 1259
check out 1259
component 1259
deliver stream 1259
development stream 1259
integration stream 1259
rebase 1259
verioned object base 1259
view 1259
Rational ClearCase LT
development scenario 1277
installation 1260, 1385
integration with Rational Application Developer 1260
scenario overview 1263
setup for new project
add project to ClearCase source control 1274
create a Web project 1274
create new ClearCase project 1265
enable Team capability 1264
join a ClearCase project 1268
Rational Developer 7
Rational Functional Tester 6
Rational Performance Tester 6
Rational Product Updater 25, 168, 1372, 1380
Rational Software Architect 5
Rational Software Modeler 5
Rational Unified Process 189–190
disciplines 191
lifecycle phases 191
Process Advisor 191
Process Browser 192
Rational Unified Process integration 15
Rational Web Developer 6
RCP directory 1194
RE directory 1194
rebase 1259, 1290
recording a test 1112
Red Hat Enterprise Linux Workstation V3 9
Redbooks Web site 1404
Contact us xxvii
refactor 328
change method signature 329
encapsulate field 330
extract constant 330
extract interface 329
extract local variable 330
extract method 330
inline 330
move 329
pull up 330
push down 329
redo 330
rename 329
undo 330
refactoring
example 330
Relational database to XML mapping 448
relationship
methods 879
relationships 267, 838
create 872
remote
client view 903
debugging 1145

remove project from a server
via Rational Application Developer 1066
via WebSphere Administrative Console 1067
request sequence 618
resource adapter 1195
Resource adapter module 1191
Resource perspective 159
resoure adapter achive
See RAR
resume 1141
RMI 832
See Remote Method Invocation
role-based development model 131
RP directory 1194
run
Java application outside Application Developer 301
Java applications 290
runAnt.bat 1186
running a test 1113
RUP
See Rational Unified Process

S

SAAJ
See SOAP with Attachments API for Java
sample
Java database application
prepare
import BankDB.zip 337
sample code 1395
6449code.zip 1396
description by chapter 1396
locate 1396
Project Interchange files 1398
scalability 830
schema 357
Scrapbook 293
SDO
See Service Data Objects
security 832
Sequence Diagram 195, 213
server
debugging 1136
Server project 177
server resources 1078
Servers view 148
service broker 952–953
service bus 1390
service client 953
Service Data Objects 16, 19, 52, 678
service integration 1390
service provider 952
service requester 952–953
service-oriented architecture 951–952
service broker 953
service provider 952
service requester 953
servlet 577
add to Web Project 576–577
create 577
implement command interface 586
servlet container 501
servlets 44, 576
session bean 836, 901
business methods 904
create 902
session EJBs 58
session facade 901
set breakpoint 1133
setter 254, 257
setUp 1093
Simple Object Access Protocol 954
Simple project 177
site appearance 524
site navigation 517, 524
Smart Insert 316
snippet 295
SOA 951
SOAP with Attachments API for Java 18
software
configuration management 1258
sound 501
source folding 314
specification
Enterprise JavaBeans (EJB) 18
IBM Java Runtime Environment 18
J2EE Connector 18
J2EE Deployment API 19
J2EE Management API 19
Java Activation Framework 18
Java API for XML Processing 18
Java API for XML Registries 19
Java API for XML RPC 18
Java Authentication and Authorization Service 19
Java Authorization Service Provider Contract for

Archived

Containers 19
Java Management Extensions 19
Java Message Service 18
Java Servlet 18
Java Transaction API 18
JavaMail 18
JavaServer Faces 19
JavaServer Page (JSP) 18
Service Data Objects 19
SOAP with Attachments API for Java 18
Struts 19
Web Services 18
specification versions 14, 18
SQL
statement
execute 383
SQL commands 334
SQL Query Builder 384
example 384
SQL statement 376
SQL Statement wizard 376
define conditions for WHERE clause 380
define table joins 379
execute SQL statement 383
groups and order 382
parse the statement 382
select tables and columns 377
use a variable 382
view SQL statement 382
staging environment 1084
Standard Widget Toolkit 36, 416
standardization 830
start server in profile mode 1249
startup parameters 81
stateless 901
static and dynamic 500
Static Method Sequence Diagram 195, 203
static pages
create a list 547
create tables 544, 546
forms 548
links 544
text 544
Static Web project 176
static web sites 39
step debug 1126
step filter 1126
step into 1141
step over 1141–1142
stored procedure 394
Stored Procedure wizard 398
structured types 357
Struts 19
configuration file editor 659
controller 616
create components
realize a JSP 648
realize Struts action 645
realize Struts form bean 641
Struts Action 634
Struts Form Bean 634
Struts Web Connection 636
Struts Web Connections 650
Web Diagram 633
Web Page 635
introduction 616
model 616
MVC 617
tag library 655
view 616
Struts Action 634
Struts Component wizards 619
Struts Configuration Editor 619
Struts Form Bean 634
Struts validation framework 653
Struts Web application
import and run sample 665
import BankStrutsWeb.zip 665
prepare sample database 666
run sample application 666
prepare for sample 620
Dynamic Web Project Struts enabled 622
Struts Web Connection 636
Struts-bean tags 655
Struts-html tags 655
Struts-Logic tags 656
Struts-Nested tags 656
Struts-Template tags 656
Struts-Tiles Tags 656
Stuts tag library 655
style sheets 511
customize 535
sualization 219
SubType 224
summary
features
annotated programming 15
application code analysis 15

application modeling with UML 15
component testing 17
Crystal Reports integration 16
Eclipse 14
Enterprise Generation Language 17
JavaServer Faces 16
profiling tools 17
Rapid Web Development 15
Rational Unified Process 15
Service Data Objects 16
test server environments 14
Web Services 14
SuperType 224
supported
databases
Cloudscape 9
DB2 Universal Database 9
Informix Dynamic Server 10
Microsoft SQL Server 10
Oracle 10
Sybase Adaptive Server Enterprise 10
platforms
Microsoft Windows 2000 Professional 9
Microsoft Windows 2000 Server 9
Microsoft Windows Server 2003 9
Microsoft Windows XP Professional 9
Red Hat Enterprise Linux Workstation V3 9
SuSE Linux Enterprise Server V9 9
SuSE Linux Enterprise Server V9 9
suspend 1141
SVT
See system verification test
Swing 36
SWT
See Standard Widget Toolkit
system verification test 1083

T

tables 357, 546
tag libraries 47
Tasks view 146
TCP/IP Monitor 959, 1079
Team Synchronizing perspective 160
tearDown 1093
template
page templates 535
templates 121
terminate 1141
terminology 7
test
component test 1081
introduction 1082
JUnit 1081, 1087
test calibration 1084
test case
create 1089
test environments 1084
test execution 1085
Test perspective 161
test phases
build verification test 1082
component test 1082
customer acceptance test 1083
function verification test 1083
performance test 1083
system verification test 1083
unit tests 1082
test results recording 1085
test server environments 14
TestCase 1088
TestSuite 1088, 1097
text 544
theme 524
thread analysis 1239
thread analysis views 1240
Tomcat 23
top down 893
Topic Diagram 195
Topic Diagrams 199
transaction 832
demarcation 833
transactions
EJB 829
transfer object 842
triggers 357
Type Hierarchy view 224

U

UDDI registry 982
UML 194
more information 220
UML Visualization 370
browse diagram 196
Class Diagram 196
Sequence Diagram 196
Static Method Sequence Diagram 196

Archived

Topic Diagram 196
UML visualization 195
Unified Change Management 1258
Unified Modeling Language
See UML
uninstall 25
unique identifier 870
unit test 1082
benefits 1085
case 1085
Universal Description, Discovery and Integration 955
universal test client
EJB 915
Universal Test Client (UTC) 915
UNIVERSAL_JDBC_DRIVER_PATH 783
UNIX 1155
URL 391
utility classes 501
Utility Java projects 302

V

value object 842
variable
change value 1143
Variables view 147
version
project 1342
version 6 terminology 7
versioned object base 1259
versioning 1342
video 501
view 1259
CVS Repositories 1324
CVS Resource History 1343
Display 1143
Expressions 1143
JUnit 1099
MVC 515
view management 1127
views 137, 357
Breakpoints view 148, 1142
Call Hierarchy view 223
Code Review Details view 229
Code Review view 226
Console view 146
CVS Annotate view 144
CVS Repositories view 143
CVS Resource History view 144
Data Definition view 146
Database Explorer view 146
DB Output view 146
Debug view 147
Declaration view 225
Diagram Navigator view 230
Navigator view 146
Outline view 230
Outlines view 148
Package Explorer view 223
Problems view 148, 225
Sensor Results view 149
Servers view 148
Tasks view 146
Type Hierarch view 224
Variables view 148
visual class 420
visual development 194
Visual Editor 415–416
add JavaBeans to visual class 428
binding 438
change component properties 428
code synchronization 427
create visual class 420
customize appearance 424
launch 419
layout 423
open existing class 422
overview 423
resize JavaBean component 427
VisualAge Generator to EGL migration 764
VOB 1259

W

W3C 444
WAR 170, 1179, 1191
WAR classloader 1195
watch variables 1142
watching variables 1142
Web
content folder 523
project
create 507
Web application 501
debug on local server 1132
debug on remote server 1145
Web Application Archive 1191

Archived

Web application module 1191
Web application test 1112
Web application testing
sample
analyze test results 1118
deployment definition 1116
edit the test 1115
generate an executable test 1115
record a test 1113
run test 1117
Web applications 43, 499
concepts and technologies 500
introduction 500
using EGL 751
using EJBs 827
using JSF and SDO 673
using JSPs and servlets 499
prepare for sample 513
using Struts 615
Web archive
See WAR
Web Browser Settings 96
Web development tooling
AnimatedGif Designer 511
CSS Designer 511
file creation wizard 512
Javascript Editor 511
Page Designer 509
page templates 510
Web perspective and views 506
Web Project 507
Web Site Designer 508
WebArt Designer 511
Web Diagram 197
Web module 500
Web perspective 162, 506
Web Project 507
directory structure 517
Web Service wizard 964
Web Services 14, 18, 66, 951
client development 958
create from an EJB 980
create Web Service from JavaBean 964
EJB from WSDL 958
enable development capability 960
introduction
related standards 955
service-oriented architecture 952
SOA implementation 953
JavaBean from WSDL 958
monitor using TCP/IP Monitor 976
prepare for development 959
publish using UDDI 982
security 980
test the client proxy 971, 973
test tools 959
TCP/IP Monitor 959
test environment 959
Universal Test Client 959
Web Services Explorer 959
Web Services Description Language 954
Web Services Explorer 971
Web Services tools 957
Web Site Designer 508, 525
launch 525
web.xml 502
WebArt Designer 511
WEB-INF 502, 524
WebLogic 23
WebSphere Administrative Console 1390
WebSphere Application Server
(base) Edition 1189
Base Edition 1189
configure data source 1225
deployment architecture 1199
enable debug 1147
enhanced EAR 1201
Express Edition 1189
installation 1050
messaging 1389
Network Deployment Edition 1189
profile creation 1051
profiles 1201
v6.0 Profiles 1047
WebSphere Business Integrator 6
WebSphere Business Integrator Modeler 6
WebSphere class loader 1193
application class loader 1195
extensions to class loader 1194
handling JNI code 1195
hierarchy 1194
Web module class loader 1195
WebSphere enhanced EAR 1201
WebSphere Portal 985, 989
WebSphere Portal V5.0 Test Environment 1376
WebSphere Portal V5.1 Test Environment 1377
WebSphere Profile wizard 1051
WebSphere Profiles 1047

application server profile 1048
custom profile 1049
deployment manager profile 1048
WebSphere Rapid Deployment 1210
annotation-based programming 1210
modes 1212
tools 1210
WebSphere Studio 7
Wireless Markup Language 994
WML 994
Word Skipping 317
Workbench
basics 76, 78
Working sets 319
workload optimization 830
ws.ext.dirs 1194–1195
WYSIWYG 419, 509

X

X/Open SQL 334
Xdoclet 1210
XML 38, 444
Metadata Interchange 335
namespaces 446
overview 444
processor 444
schema 445, 458
create new 463
generate from DTD file 458
generate from relational table 461
generate from XML file 461
graph 370
validate 489
transform 491
where to find information 497
XML and relational data 448
XML editor 448
XML Path Language 446
XML schema editor 448
XML to XML mapping editor 448
XML tools
DTD editor 448
XML editor 448
XML schema editor 448
XPath Expression wizard 448
XSL editor 448
XPath 446–447
XPath expression wizard 448
XSL 446, 483
edit 485
XSL debugging and transformation 448
XSL editor 448
XSL Transformations 446
XSL-FO 446
XSLT 446
debug 493
XSLT debugger 1128

Z

zSeries 759

Archived

Archived

Redbooks

Rational Application Developer V6 Programming Guide

Archived

Rational Application Developer V6 Programming Guide

Redbooks

Develop Java, Web, XML, database, EJB, Struts, JSF, SDO, EGL, Web Services, and portal applications

Test, debug, and profile with built-in and remote servers

Deploy applications to WebSphere Application Server and WebSphere Portal

IBM Rational Application Developer V6.0 is the full function Eclipse 3.0 based development platform for developing Java 2 Platform Standard Edition (J2SE) and Java 2 Platform Enterprise Edition (J2EE) applications with a focus on applications to be deployed to IBM WebSphere Application Server and IBM WebSphere Portal. Rational Application Developer provides integrated development tools for all development roles, including Web developers, Java developers, business analysts, architects, and enterprise programmers.

This IBM Redbook is a programming guide that highlights the features and tooling included with IBM Rational Application Developer V6.0. Many of the chapters provide working examples that demonstrate how to use the tooling to develop applications, as well as achieve the benefits of visual and rapid Web development.

This redbook consists of six parts:

- Introduction to Rational Application Developer
- Develop applications
- Test and debug applications
- Deploy and profile applications
- Team development
- Appendixes

INTERNATIONAL TECHNICAL SUPPORT ORGANIZATION

BUILDING TECHNICAL INFORMATION BASED ON PRACTICAL EXPERIENCE

IBM Redbooks are developed by the IBM International Technical Support Organization. Experts from IBM, Customers and Partners from around the world create timely technical information based on realistic scenarios. Specific recommendations are provided to help you implement IT solutions more effectively in your environment.

For more information:
ibm.com/redbooks

SG24-6449-00 ISBN 0738491209