Tivoli software

IBM

Developing Workflows and Automation Packages for IBM Tivoli Intelligent Orchestrator V3.1

- Reduces infrastructure and deployment costs through automated best practices
- Enables your solution for on demand provisioning
- Gives practical aspects for workflows and automation packages design

Edson Manoel
Mark Hicks
Morten Moeller
Indran Naick
Mark Poulson
Joerg Surmann

Redbooks

ibm.com/redbooks

Archived

Archived

International Technical Support Organization

Developing Workflows and Automation Packages for IBM Tivoli Intelligent Orchestrator V3.1

December 2006

Archived

SG24-6057-01

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

Archived

Second Edition (December 2006)

This edition applies to Version 3.1.3 of IBM Tivoli Intelligent Orchestrator and IBM Tivoli Provisioning Manager.

© Copyright International Business Machines Corporation 2004, 2006. 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

Figures xiii

Tables xix

Notices xxi
Trademarks xxii

Summary of changes xxiii
November 2006, Second Edition xxiii

Preface xxv
The team that wrote this redbook xxvi
Become a published author xxvii
Comments welcome xxviii

Chapter 1. Introduction 1
1.1 Overview of an on demand operating environment 2
1.2 Automation component blueprint 5
1.2.1 Provisioning 8
1.2.2 Policy-based orchestration 9
1.3 IBM Tivoli Intelligent Orchestrator 10
1.3.1 Product overview 13
1.4 Enabling software for on demand provisioning 15
1.4.1 The benefits of enabling software for on demand 16
1.5 The IBM Orchestration and Provisioning Automation Library 18

Chapter 2. IBM Tivoli Intelligent Orchestrator concepts 21
2.1 Concepts and architecture 22
2.1.1 Web clusters 22
2.1.2 Concepts and terminology 24
2.1.3 Architecture 28
2.2 Enabling an application/device for provisioning or orchestration 31
2.2.1 Workflows for provisioning and orchestration 31
2.3 Workflows basics 33
2.3.1 What is a workflow 33
2.3.2 What can you do with workflows 34
2.3.3 How are workflows controlled 35
2.3.4 Workflow transitions and variables 35
2.3.5 Building workflows 39

© Copyright IBM Corp. 2004, 2006. All rights reserved.

2.4 Automation packages basics . . . 40
2.4.1 What is an automation package . . . 40
2.4.2 Automation package .tcdriver file . . . 42
2.4.3 Managing automation packages . . . 42

Chapter 3. Architectural design . . . 45
3.1 Functional overview . . . 46
3.1.1 Workflow overview . . . 46
3.1.2 Overview of automation packages . . . 48
3.2 Designing the automated provisioning solution . . . 49
3.2.1 The Orchestration Solution Package . . . 50
3.2.2 Architecting Orchestration Solution Packages . . . 53
3.2.3 Define Orchestration Solution Package content and scope . . . 53
3.2.4 Determine Automation Package functionality . . . 55
3.2.5 Verify automation feasibility . . . 56
3.2.6 Determine device types to be supported . . . 56
3.2.7 Define high-level operations . . . 56
3.2.8 Define low-level operations . . . 57
3.2.9 Break down operations into workflows . . . 58
3.2.10 Define external prerequisites, dependencies, and interfaces . . . 58
3.2.11 Define location and naming conventions for external files . . . 60
3.2.12 Identify transitions . . . 60
3.2.13 Create specifications for new code to be developed . . . 61
3.2.14 Develop recovery policies . . . 61
3.2.15 Design templates for DCM object and related files . . . 62
3.2.16 Design implementation procedures . . . 62
3.2.17 Ensure proper documentation . . . 62
3.2.18 Architecting best practices . . . 63
3.3 Practical aspects of designing automation packages and workflows . . . 64
3.3.1 Introducing the environment . . . 64
3.3.2 Building the development environment . . . 69
3.3.3 Design considerations for automation packages and workflows . . . 69
3.3.4 Reuseability of workflow transitions . . . 75
3.3.5 Authentication and authorization internals . . . 76
3.3.6 Naming conventions . . . 88
3.3.7 Documentation guidelines . . . 91

Chapter 4. Workflow development environments . . . 103
4.1 Preparing the TIO/TPM environment . . . 106
4.1.1 Security and authentication considerations . . . 106
4.1.2 TIO/TPM Database performance considerations . . . 113
4.2 The development environments . . . 114
4.2.1 The Workflow Composer . . . 115

Archived

4.2.2 The Automation Package Development Environment 116
4.2.3 Installing APDE 118
4.2.4 Configuring the Eclipse environment 124
4.2.5 Showing APDE views 132
4.2.6 Defining an APDE project 133
4.3 The text editor based development environment 136

Chapter 5. Workflow development quickstart. 137
5.1 Development Environment 138
5.1.1 Workflow composer 138
5.1.2 Editor 138
5.1.3 Automation Package Development Environment (APDE) 138
5.2 Understanding Orchestrator and Provisioning Automation Library 139
5.3 Workflow development 139
5.4 Automation Package Baseline Workflow Requirements 140
5.4.1 Software solutions 140
5.4.2 Hardware products or devices 141
5.5 Defining and classifying workflows (logical operations) 143
5.5.1 Use logical device operations where possible 144
5.6 Workflow headers and copyright 144
5.7 Workflow comments 146
5.8 Variable naming conventions and handling 146
5.8.1 Variable encryption 146
5.8.2 Variable naming conventions 147
5.8.3 Variable handling 148
5.8.4 Error handling within workflows 148
5.9 DCMQuery 150
5.9.1 Using DCMQuery to retrieving DCM information 151
5.9.2 Using DCMInsert to insert data into the DCM 152
5.9.3 Using DCMUpdate to update data in the DCM 153
5.9.4 Using DCMDelete to delete data from the DCM 154
5.10 Use Jython for string concatenation and evaluation 154
5.11 Using scriptlets in workflows 155
5.12 Using Java in workflows 156
5.13 Supporting documentation for automation packages 156
5.14 Automation package documentation (/doc) 156
5.15 License (/license) 156
5.16 Repository (/repository) 157
5.17 Script and binary files run on the Orchestrator Server (/bin) 157
5.18 Java plug-ins (/java-plugin) 157
5.19 The lib directory (/lib) 158
5.20 Putting it all together: creating the automation package 159
5.21 Naming the automation package 159

Archived

5.22 Components of the automation package 160
5.23 Manually creating the automation package 161
5.24 A sample README file 161
5.25 The XML Manifest file (\TC-INF directory) 166

Chapter 6. Developing workflows 175
6.1 Workflows overview from a developer's perspective. 176
6.1.1 Workflows in the context of TIO/TPM 176
6.1.2 Transitions. 180
6.1.3 Workflow development challenges 181
6.1.4 Workflow prerequisites 183
6.2 Preparing to write a workflow 185
6.2.1 Establish workflow scope and review workflow objects 185
6.2.2 Detail the workflow operations and their order 186
6.2.3 Create a new workflow or create one from a template 188
6.2.4 Add new transitions to the workflow 189
6.2.5 Map the variables 190
6.2.6 Validate and update the recovery actions 191
6.2.7 Validate and test the new workflow. 191
6.3 Workflow elements 191
6.3.1 Variables 196
6.3.2 Comments. 198
6.3.3 String manipulation 199
6.3.4 Number manipulation 200
6.3.5 Manipulating DCM objects 202
6.3.6 Testing, conditional processing, and looping 207
6.3.7 Using Java in workflows 208
6.3.8 Logging 208
6.3.9 Handling workflow exceptions. 209
6.3.10 Embedding scripts. 211
6.4 Creating your first simple workflows 212

Chapter 7. Extending your workflows with Java 213
7.1 Workflows and Java programs 214
7.2 Developing Java programs 215
7.2.1 Install the Java Software Development Kit 215
7.2.2 Develop a Java program using Eclipse. 216
7.2.3 Manually creating the package file 231

Chapter 8. Automation package content and packaging. 233
8.1 Introduction 234
8.2 Automation package anatomy. 234
8.2.1 Directories. 234
8.3 The manifest file 237

Archived

8.3.1 The <driver-name> and <version> tags 238
8.3.2 The <dependencies> tag 238
8.3.3 The <property> tag 240
8.3.4 The <actions> tag 240
8.3.5 The <items> tag 240
8.3.6 The <device-model> tag 241
8.3.7 The <post-install-workflow> tag 242
8.3.8 The <software-products> tag 242
8.4 Packaging and deploying an automation package 243
8.4.1 Setting up the environment for packaging. 244
8.4.2 Exporting the workflows and simple commands 245
8.4.3 Creating the Java plug-ins XML and JAR files 246
8.4.4 Creating all the supporting scripts. 246
8.4.5 Creating the automation package documentation: readme file 247
8.4.6 Creating the automation package manifest file 247
8.4.7 Pre-package verification task 249
8.4.8 Packaging the automation package into a .tcdriver file. 250
8.4.9 Deploying the automation package. 252
8.4.10 Verifying deployment. 253
8.5 Generating the automation package using APDE. 256

Chapter 9. Case study: basic workflows. 259
9.1 Get_DeploymentEngine_DeviceID 260
9.1.1 Objective 260
9.1.2 Development steps 260
9.2 LAB1 - Execute_Local_Command 261
9.2.1 Objectives 261
9.2.2 Development steps (method A). 261
9.2.3 Development steps (method B). 262
9.3 LAB2 - Getting information from the DCM using DCMQuery 263
9.3.1 Objective 263
9.3.2 Development steps (method A). 263
9.3.3 Development Steps (method B): Designed specifically for the TIO Server 264
9.4 Jython string manipulation. 266
9.4.1 Objective 266
9.4.2 Development steps 266
9.5 Implementing Logical Device Operations 268
9.5.1 Objective 268
9.6 Development steps 268
9.6.1 Test 268
9.7 Using DCMInsert 270
9.7.1 Objective 270

Archived

9.7.2 Development steps 270
9.8 Working with SAPs 273
9.8.1 Objective 273
9.8.2 Development steps 273
9.9 Scriptlets and exceptions 275
9.9.1 Objective 275
9.9.2 Development steps 276
9.9.3 Challenges 276

Chapter 10. Case study: Trade3 composite application 281
10.1 Overview of case study 282
10.2 The Trade application 282
10.2.1 Main architecture of the Trade3 application 283
10.2.2 Trade3 installation instructions 284
10.3 Scoping the job of provisioning Trade 286
10.3.1 Assumptions 287
10.3.2 Prerequisites and limitations 288
10.3.3 Functionality 289
10.3.4 Content 290
10.3.5 Requirements and capabilities 298
10.3.6 Prerequisites and their configuration. 300
10.3.7 Delivery of executables 301
10.3.8 Customization 302
10.3.9 Transition from test to production 303
10.4 Creating the Test environment 304
10.4.1 Windows 2000 Server SP4 305
10.4.2 Trade Simulated DB2 Server 306
10.4.3 Trade Simulated DB2 Client 310
10.4.4 Trade Simulated WebSphere Application Server 313
10.4.5 Trade Simulated HTTP Server 317
10.4.6 Building the Trade_Simulated_Infrastructure Automation Package320
10.5 Developing the Trade automation package. 322
10.5.1 Creating the trade Service Access Point. 323
10.5.2 Finding installation directories. 324
10.5.3 Controlling simulation 327
10.6 Creating the Trade database. 328
10.6.1 The Trade DBServer Module Software Product 330
10.6.2 The Trade_DBServer_Module_Installable_Driver 345
10.6.3 The Trade_DBServer_Module_Installation_Driver 353
10.6.4 The Trade_DBServer_Module_Instance_Driver 356
10.6.5 Trade DBServer Module Helper workflows 370
10.7 Cataloging the Trade database with the DB2 Client 387
10.7.1 The Trade DBClient Module Software Product 387

Archived

10.7.2 Trade_DBClient_Module_Installable_Driver 401
10.7.3 Trade_DBClient_Module_Installation_Driver 405
10.7.4 Trade DBClient Module Helper workflows 408
10.8 Installing the Trade application 414
10.8.1 The Trade Application Module Software Product 415
10.8.2 The Trade_Application_Module_Installable_Driver 428
10.8.3 The Trade_Application_Module_Instance_Driver 434
10.8.4 The Trade_Application_Module_Installation_Driver 441
10.8.5 Trade Application Module Helper workflows 443
10.9 Front end of the Trade application 457
10.9.1 The Trade Web Module Software Product 458
10.9.2 The Trade_Web_Module_Installable_Driver 470
10.9.3 The Trade_Web_Module_Instance_Driver 476
10.9.4 The Trade_Web_Module_Installation_Driver 480
10.9.5 Trade Web Module Helper Workflows 483
10.10 Utility workflows 494
10.10.1 ITSO_Find_Supporting_Installation_for_Module_Requirements 494
10.10.2 ITSO_Synchronize_Credentials 497
10.10.3 ITSO_Add_Matching_DeploymentEngine_Client_Password_Credential 499
10.10.4 ITSO_HostingEnvironment_Host 500
10.10.5 ITSO_Delete_Credentials 504
10.10.6 ITSO_Create_User 505
10.10.7 ITSO_Destroy_User 508
10.10.8 ITSO_Copy_And_Unpack_Archive_From_Repository 509
10.10.9 ITSO_Copy_File_From_Repository 510
10.11 Documenting your automation package 512
10.12 Building the Trade automation package 513

Appendix A. Trade3 original installation instructions 523
Trade3 installation instructions 524
Step 1 - Prepare for installation 524
Step 2 - SOAP enablement 524
Step 3 - DB2 525
Step 3 - Oracle 525
Step 3 - Install, configure, and start Trade3 application and resources 526
Step 4 - Populate the Trade3 database and go trade 526

Appendix B. The TIO/TPM V3 software model 527
Software model 528
Software definitions 529
Installable files 530

Archived

Associating installable files with software definitions 531
Software dependencies. 533
Software configuration templates 537
Creation of software definitions. 547
Software provisioning. 547
Developing software automation packages 548
Extending the automation package example 561

Appendix C. Popular Tivoli Java classes and methods. 565
Java methods returning values 566
Java classes not returning values 576

Appendix D. Jython reference 583
Jython string methods 584
List methods. 589
Numeric methods 591

Appendix E. Developing Java plug-ins 595
Java plug-in development 596
Install the Java Software Development Kit 597
Copy the TIO/TPM packages to the development directory 598
Write the Java source code. 599
Compile the source code 605
Compress the class file. 606
Add the jar file to the deployment engine classpath 606
Create and load the Java plug-in XML file 607

Appendix F. Systems Management using the IT Infrastructure Library. 613
Overview 614
Configuration Management. 616
Business perspective 617
Technical perspective 618
Example of using the IT Infrastructure Library 620

Appendix G. Additional material 625
Locating the Web material 625
Using the Web material 626
System requirements for downloading the Web material 626
How to use the Web material 626

Glossary 629

Abbreviations and acronyms 635

Related publications 637

Archived

IBM Redbooks 637
Other publications 638
Online resources 638
How to get IBM Redbooks 639
Help from IBM 640

Index 641

Archived

Archived

Figures

1-1 On demand operating environment overview 3
1-2 On demand environment key components 4
1-3 IBM Automation blueprint. 6
1-4 Provisioning and Orchestrator products on the Automation blueprint . . . 8
1-5 Just-in-case provisioning 11
1-6 Orchestrated provisioning 12
1-7 Product overlap 15
2-1 Basic Web cluster 23
2-2 Data center relationship model 26
2-3 Customer relationship model 27
2-4 High-level component architecture 28
2-5 Deployment Engine 30
2-6 Workflows for provisioning and orchestration 32
2-7 Workflow component relationships 36
2-8 Workflow component relationships summary 39
3-1 Configuration pane. 46
3-2 Groups of device drivers 48
3-3 Orchestration Solution Package overview 52
3-4 Logical View of an application tier managed by TIO/TPM 66
3-5 Inventory tab 66
3-6 Overview of an existing data center. 67
3-7 Physical View of an application tier managed by TIO/TPM 68
3-8 Flowchart for step 1 of designing a workflow 71
3-9 Flowchart for step 2 of designing a workflow 72
3-10 Flowchart for step 3 of designing a workflow 74
3-11 Adding a new Service Access Point to a server 78
3-12 Password credentials for Telnet and FTP protocols 79
3-13 RSA credentials for SSH and SCP protocols. 80
3-14 SNMP credentials 81
3-15 SAP for application specific credentials 82
3-16 User ID and password defined in a Software Resource Template 84
3-17 User ID and password defined as software product variables. 84
3-18 Automation package documentation 93
3-19 Internal documentation of a workflow 93
3-20 Object documentation 95
3-21 Workflow documentation 95
3-22 Java plug-in documentation 96
3-23 Java program documentation 96

© Copyright IBM Corp. 2004, 2006. All rights reserved.

3-24 Documentation Data Center Objects assignment 97
3-25 Variables of the workflow ITSO WebAppInstall 98
3-26 Documentation of variables 98
3-27 Headline of automation package documentation. 99
3-28 Automation package documentation 99
3-29 Object documentation 100
3-30 Assigned object of data center. 100
3-31 Workflow specific documentation. 100
3-32 Documentation of the workflow's variables 101
3-33 Documentation of the workflow's variables 101
4-1 APDE architecture 104
4-2 TIO/TPM users, roles, and permissions. 107
4-3 APDEDeveloper security role properties 108
4-4 Assigning security roles to users 109
4-5 Resources in the QA_Customer Access Group. 110
4-6 Access permissions 111
4-7 Access Group with associated access permissions and users 112
4-8 Resource authorizations using Access Groups and Permissions 113
4-9 Navigate to Configuration 115
4-10 Workflow search 115
4-11 Workflow browsing in Workflow Composer 116
4-12 APDE architecture 118
4-13 Select workspace. 124
4-14 APDE Workbench defaults 125
4-15 Defining the TIO/TPM server to APDE 126
4-16 Configuring APDE encryption 127
4-17 Configuring database access for the APDE. 129
4-18 Workflow editor preferences 131
4-19 Customizing file encryption 132
4-20 Create new APDE project 133
4-21 Naming a new APDE project 134
4-22 Setting APDE project dependencies 135
6-1 Add_Server workflow example 178
6-2 Orchestration Package development scope 182
6-3 Add workflow 188
6-4 Dragging a simple command onto the Transition Area 190
6-5 Workflow elements. 191
7-1 Java SDK Installation Wizard. 216
7-2 Create a new Java Project. 217
7-3 Naming and customization. 218
7-4 Specifying source and build locations 219
7-5 Setting build path for you Java project. 220
7-6 Initial Java project content 221

Archived

7-7 Creating Java package com.itso.kanaha.util 221
7-8 Creating java class ItsoPathHelper 222
7-9 Exporting the package jar file. 224
7-10 Exporting the package jar file. 225
7-11 Specifying Java package output file. 226
7-12 Specifying Java project manifest file 227
7-13 Eclipse Java project content after successful export 228
7-14 Converting pathnames using Java programs 231
8-1 Apache Driver view 254
8-2 Launch the automation package builder 256
8-3 Select build options 257
8-4 Build results 258
9-1 Get_DeploymentEngine_DeviceID workflow 261
9-2 Execute_Local_IPconfig workflow 262
9-3 Execute_Local_IPconfig workflow with parameter input 263
9-4 Workflow with DCMQuery 264
9-5 DCMQuery workflow for TIO Server 265
9-6 Jython string manipulation workflow 267
9-7 Workflow implementing a Logical Device Operation 269
9-8 DCMInsert workflow. 271
9-9 Create SAP workflow. 274
9-10 Workflow with scriptlet - part 1 - initialize. 277
9-11 Workflow with scriptlet - part 2 - the scriptlet 277
9-12 Workflow with scriptlet - part 3- error handling and cleanup 278
10-1 Trade - standard installation on a single system 282
10-2 Trade J2EE components - Model-View-Controller architecture. 283
10-3 TIO/TPM objects delivered with the Trade Automation Package 288
10-4 Trade infrastructure 291
10-5 Software components included in the Trade Automation Package . . . 294
10-6 Trade Software Module and infrastructure relationships 299
10-7 Windows 2000 Server SP4 operating system properties. 305
10-8 Trade Simulated DB2 Server Software Product properties 306
10-9 Trade Simulated DB2 Server Installable Software Package 307
10-10 Trade Simulated DB2 Client Software Product properties. 310
10-11 Trade Simulated DB2 Client Installable Software Package. 311
10-12 Trade Simulated WAS Server Software Product properties 314
10-13 Trade Simulated WebSphere Application Server Installable Software Package 315
10-14 Trade Simulated HTTP Server Software Product properties. 318
10-15 Trade Simulated HTTP Server Installable Software Package 319
10-16 Trade DBServer Module properties 330
10-17 Trade DBServer Module capabilities and requirements 331
10-18 Trade DBServer Module Installable software package properties. . . . 333

10-19 Trade DBServer Module Installable software package variables 334
10-20 Trade DBServer Module Software Resource Templates 340
10-21 Trade DBServer Module deployment model 341
10-22 The Trade_DBServer_Module_Installable_Driver 346
10-23 Trade_DBServer_Module_Installable_Driver Workflows 347
10-24 Trade_DBServer_Module_Installation_Driver Overview 354
10-25 Trade_DBServer_Module_Installation_Driver Workflows 354
10-26 Trade_DBServer_Module_Instance_Driver Overview 357
10-27 Trade_DBServer_Module_Instance_Driver Workflows 357
10-28 Trade DBClient Module properties 388
10-29 Trade DBClient Module capabilities and requirements 389
10-30 Trade DBClient Module Installable software package properties 390
10-31 Trade DBClient Module Installable software package variables 391
10-32 Trade_DBClient_Module_Installable_Driver 391
10-33 Trade DBClient Module Software Resource Templates 396
10-34 Trade DBClient Module deployment model 397
10-35 Trade_DBClient_Module_Installable_Driver Overview 401
10-36 Trade_DBClient_Module_Installable_Driver 401
10-37 Trade_DBClient_Module_Instation_Driver Overview 405
10-38 Trade_DBClient_Module_Installation_Driver 405
10-39 Trade Application Module properties 416
10-40 Trade Application Module capabilities and requirements 417
10-41 Trade Application Module Installable software package properties . . . 418
10-42 Trade Application Module Installable software package variables . . . 419
10-43 Trade Application Module Software Resource Templates 423
10-44 Trade Application Module deployment model 424
10-45 Trade_Application_Module_Installable_Driver overview 428
10-46 Trade_Application_Module_Installable_Driver 429
10-47 Trade_Application_Module_Instance_Driver overview 434
10-48 Trade_Application_Module_Instance_Driver 434
10-49 Trade_Application_Module_Installation_Driver overview 441
10-50 Trade_Application_Module_Installation_Driver 441
10-51 Trade Web front end 458
10-52 Trade Web Module properties 459
10-53 Trade Web Module capabilities and requirements 460
10-54 Trade Web Module Installable software package properties 461
10-55 Trade Web Module Installable software package variables 462
10-56 Trade Web Module Software Resource Templates 466
10-57 Trade Web Module deployment model 467
10-58 Trade_Web_Module_Installable_Driver overview 471
10-59 Trade_Web_Module_Installable_Driver 472
10-60 Trade_Web_Module_Instance_Driver overview 476
10-61 Trade_Web_Module_Instance_Driver 477

10-62 Trade_Web_Module_Installation_Driver overview 481
10-63 Trade_Web_Module_Installation_Driver 481
B-1 Software definition overview 528
B-2 Software definition with installable files 532
B-3 Software definition with multiple installation mechanisms 536
B-4 Software resources derived from the software definition 538
B-5 Software resources 540
B-6 Software stacks 543
E-1 Java SDK Installation Wizard. 598
E-2 TIO/TPM Java API documentation home page 603
E-3 TIO/TPM API Interface CommandDriver method documentation 604
E-4 Importing a Java plug-in. 611
E-5 Imported Java plug-in String concatenation (multi) 612
F-1 Overview of the IT Infrastructure Library (ITIL) 614
F-2 Systems management on a High Level View (ITIL). 616
F-3 Architectural Workflow Cluster.AddServer. 621

Archived

Archived

Tables

1-1 Product overview table 14
2-1 tc-driver-manager command line 43
3-1 Components of the application environment 64
3-2 Distinguishing the processes 72
3-3 Modularized the process of re-initiation 73
3-4 Transitions of Workflow TMA_Assign_Policy_Region 76
4-1 Development environment functional comparison 104
5-1 Software related logical operations 141
5-2 Hardware related logical operations 142
5-3 Automation package components 162
6-1 DCM objects and Java code analogies 177
6-2 RPM software install workflow operations 186
6-3 Cluster.RemoveServer.Template workflow operations 187
6-4 Jython numeric operations 201
10-1 Trade provisioning activities 290
10-2 The Trade software model resource templates 293
10-3 Trade provisioning activities 295
10-4 Trade Software Module capabilities and requirements 299
10-5 Trade DBServer Module related devise drivers and workflows 337
10-6 Trade DBServer Module related SRT parameters 338
10-7 Trade DBClient Module related device drivers and workflows 394
10-8 Trade DBClient Module related SRT parameters 395
10-9 Trade Application Module related device drivers and workflows 420
10-10 Trade Application Module related SRT parameters 422
10-11 Trade Web Module related devise drivers and workflows 464
10-12 Trade Web Module related Software Resource Template parameters 465
B-1 Software definitions for software stacks 543
B-2 Example of a software stack with software definitions and an iterator 544
B-3 Example of a software stack with images only 545
B-4 Example of a software stack with images and an iterator 546
C-1 Polular Java classes and methods returning variables 566
C-2 Popular Java classes and methods referenced from workflows 576
D-1 Jython string methods 584
D-2 List methods 589
D-3 Numeric methods 591
F-1 IT Infrastructure Library disciplines used by Cluster.AddServer 622

© Copyright IBM Corp. 2004, 2006. All rights reserved.

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 illustrate 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.

© Copyright IBM Corp. 2004, 2006. All rights reserved.

Trademarks

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

AIX®	IBM®	System p™
DB2®	ibm.com®	Tivoli Enterprise™
DB2 Universal Database™	Lotus®	Tivoli Enterprise Console®
developerWorks®	Redbooks™	Tivoli®
e-business on demand®	Redbooks (logo) ™	WebSphere®

The following terms are trademarks of other companies:

SAP, and SAP logos are trademarks or registered trademarks of SAP AG in Germany and in several other countries.

IT Infrastructure Library, IT Infrastructure Library is a registered trademark of the Central Computer and Telecommunications Agency which is now part of the Office of Government Commerce.

ITIL, is a registered trademark, and a registered community trademark of the Office of Government Commerce, and is registered in the U.S. Patent and Trademark Office.

EJB, IPX, Java, JavaScript, JDBC, JDK, JRE, J2EE, J2ME, J2SE, Solaris, Sun, and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

Expression, Microsoft, Visual Basic, Windows Server, Windows, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

Intel, Intel logo, Intel Inside logo, and Intel Centrino logo are trademarks or registered trademarks of Intel Corporation or its subsidiaries 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, or service names may be trademarks or service marks of others.

Archived

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 SG24-6057-01
for Developing Workflows and Automation Packages for IBM Tivoli Intelligent Orchestrator V3.1
as created or updated on December 6, 2006.

November 2006, Second Edition

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

New information

- Installation, customization and use of the Eclipse based Automation Package Development Environment.
- Detailled description of the new software resource model introduced in TIO/TPM V3.
- Detailed description of the workflow language constructs and selected Jython methods
- Easy-to-follow example of extending the functionality of the workflow language using Java™.
- Comprehensive example of provisioning the composite Trade application in a three-tier environment.

Changed information

- Throughout the book, the material have been updated to reflect the capabilities provided by Tivoli® Intelligent Orchestrator and Tivoli Provisioning Manager V3.1.
- In TIO/TPM V3, the use of Java Plug-Ins have been replaced with support for native Java modules. For compatibility reasons, this topic has been kept in this book; however, it has been moved to an appendix.

Archived

© Copyright IBM Corp. 2004, 2006. All rights reserved.

Archived

Preface

IBM Tivoli Intelligent Orchestrator (TIO) and IBM Tivoli Provisioning Manager (TPM) are two of the most advanced solutions available today for automatic provisioning and orchestration of a large number of servers in a datacenter. Using low-cost servers in clustered configurations, many datacenters have found themselves in a situation where commissioning and de-commissioning of servers takes up more and more manpower, and the average utilization of the servers deployed to meet peak demand is not acceptable.

IBM Tivoli Intelligent Orchestrator and IBM Tivoli Provisioning Manager help automate the provisioning and orchestration tasks by invoking pre-prepared workflows that perform most of the provisioning and orchestration tasks automatically. This frees up resources to focus on more productive issues, and helps allocate CPU capacity to where it is needed and when it is needed.

In IBM Tivoli Intelligent Orchestrator terms, workflows represent a pre-prepared series of tasks that must be followed in a data center in order to carry out a provisioning operation. These tasks have been scripted in Java or shell scripts in order for them to be executed automatically. As such, workflows represent the best practices of an organization, a software vendor, or a hardware vendor, for performing specific operations against the business-objects, program-packages, or hardware devices. Workflows implementing the same tasks against similar objects implemented on different platforms may be grouped into Automation Packages, which allows for easy implementation at the customer site.

Workflow and Automation Package management are the single most important success factor in any IBM Tivoli Intelligent Orchestrator and IBM Tivoli Provisioning Manager implementation. Understanding and mastering this issue is critical for Independent Software Vendors (ISV) who wish to allow their products to be an integrated part of any IBM Tivoli Intelligent Orchestrator and IBM Tivoli Provisioning Manager environment.

The primary goal of this IBM® Redbook is to support ISVs, Business Partners, Clients, and IBMers in developing and implementing Workflows and Automation Packages to be used for automation purposes by IBM Tivoli Intelligent Orchestrator and IBM Tivoli Provisioning Manager. This IBM Redbook is focused on effectively planning, developing, testing, and implementing product, device, or customer specific best practices into the IBM Tivoli Intelligent Orchestrator and IBM Tivoli Provisioning Manager engines, in order to automate management processes related to specific system and device implementations.

© Copyright IBM Corp. 2004, 2006. All rights reserved.

The target audience of this IBM Redbook are the technical professionals responsible for designing, developing, and implementing IBM Tivoli Intelligent Orchestrator and IBM Tivoli Provisioning Manager Workflows and Automation Packages, and the material must guide the readers through all the phases included.

The team that wrote this redbook

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

Edson Manoel is a Software Engineer at IBM Corporation - International Technical Support Organization, Austin Center, working as an IT Specialist in the Systems Management area. Prior to joining the ITSO, Edson worked in the IBM Software Group as a Tivoli Technology Ambassador and in the IBM Brazil Professional Services Organization as a Certified IT Specialist. He was involved in numerous projects, designing and implementing systems management solutions for IBM customers and Business Partners. Edson holds a BSc degree in Applied Mathematics from Universidade de Sao Paulo, Brazil.

Mark Hicks is an e-business Architect with e-Architect Consulting Services and Support, ISV and Developer Relations, IBM Software Group. He has 25 years of experience designing and developing business applications using a variety of IBM technologies and holds certifications in Lotus®, DB2®, WebSphere®, Java, and Linux®. He has a BA in History from University of Connecticut and an MBA from Louisiana State University in Shreveport. He resides in Austin, Texas.

Morten Moeller is an IBM Certified IT Specialist working as a Project Leader at the International Technical Support Organization, Austin Center. He applies his extensive field experience to his work at the ITSO, where he writes extensively about all areas of Systems Management. Before joining the ITSO, Morten worked as a Distributed Systems Management Specialist in the Professional Services Organization of IBM Denmark, where he was involved in numerous projects designing and implementing systems management solutions for major customers of IBM Denmark.

Archived

Indran Naick is an e-business Architect for IBM Developer Relations Technical Consulting in Austin, Texas, which provides education, enablement, and consulting to IBM business partners. Indran has over 14 years of industry experience. He joined IBM in South Africa in 1990. Prior to being transferred to Austin, Texas, he served as a Software Solutions Architect consulting for a number of financial and government institutions. He has authored a number of publications and is a graduate of the University of the Witwatersrand in South Africa. He can be reached at indrann@us.ibm.com.

Mark Poulson is an e-business On Demand Tivoli Services Consultant in the United States. He has more than nine years of experience in architecting, implementing, and operating IT technology in the Enterprise Systems Management field. His areas of expertise include extensive experience with the Automation line of software products from IBM Tivoli. He is currently located in Minneapolis, Minnesota.

Joerg Surmann is a Systems Management Architect at IBM Software Group, working as a Tivoli Services Consultant specializing in the e-business on demand® arena. He has more than 13 years of experience in architecting, implementing, and operating leading IT technology and is also experienced in consulting customers in this area. Joerg holds a Bachelor degree from the University of Applied Sciences of Furtwangen, Germany, with a major in Computer Science. He is currently located near Munich, Germany.

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

Dale Ullrich, Sara Carlstead Brumfield, and Kevin Major
IBM Tivoli Intelligent Orchestrator Level 2 support,
IBM Software Group, Austin.

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 clients.

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 Internet note to:

  redbook@us.ibm.com

- Mail your comments to:

  IBM Corporation, International Technical Support Organization
  Dept. HYTD Mail Station P099
  2455 South Road
  Poughkeepsie, NY 12601-5400

Archived

1

Introduction

This chapter discusses the following:

- 1.1, “Overview of an on demand operating environment” on page 2
- 1.2, “Automation component blueprint” on page 5
- 1.3, “IBM Tivoli Intelligent Orchestrator” on page 10
- 1.4, “Enabling software for on demand provisioning” on page 15
- 1.5, “The IBM Orchestration and Provisioning Automation Library” on page 18

© Copyright IBM Corp. 2004, 2006. All rights reserved.

1.1 Overview of an on demand operating environment

What is an on demand operating environment? It is not a specific set of hardware and software. Rather, it is an environment that supports the needs of the business, allowing it to become and remain responsive, variable, focused, and resilient.

An on demand operating environment unlocks the value within the IT infrastructure to be applied to solving business problems. It is an integrated platform, based on open standards, that enables rapid deployment and integration of business applications and processes. Combined with an environment that allows true virtualization and automation of the infrastructure, it enables delivery of IT capability on demand.

An on demand operating environment must be:

- Flexible
- Self-managing
- Scalable
- Economical
- Resilient
- Based on open standards

The building of an on demand operating environment can be divided into two primary categories: integration and IT simplification. IT simplification is achieved through Virtualization and Automation. So, to build an on demand operating environment, we define these three areas as:

- Integration

  Provides the facilities to gain a unified view of processes, people, information, and systems.

- Virtualization

  Simplifies deployment and improves use of computing resources by hiding the details of the underlying hardware and system software, allowing for consolidation and the ability to adapt to changing demand.

- Automation

  Overcomes the complexity of systems management to enable better use of assets, improved availability and resiliency, and reduce costs based on business policy and objectives.

Figure 1-1 on page 3 gives an overview of an on demand operating environment.

Archived

Figure 1-1 On demand operating environment overview

The value of the operating environment is in the ability to dynamically link business processes and policies with the allocation of IT resources using offerings across all of these categories. In the operating environment, resources are allocated and managed without intervention, enabling resources to be used efficiently based on business requirements. Having flexible, dynamic business processes increases the ability to grow and manage change within the business.

Archived

Figure 1-2 provides an overview of the key components of an on demand operating environment.

Figure 1-2 On demand environment key components

As seen in Figure 1-2, Policy-based Orchestration and Provisioning are key elements of the Automation component on an IBM on demand operating environment. IBM has products to support such elements: IBM Tivoli Intelligent Orchestrator and IBM Tivoli Provisioning Manager.

In the following sections, we go into more details about the Automation component and how IBM Tivoli Intelligent Orchestrator and IBM Tivoli Provisioning Manager support the Automation component.

1.2 Automation component blueprint

Software developers have fully exploited a four to six order of magnitude increase in computational power—producing ever more sophisticated software applications and environments. There has been exponential growth in the number and variety of systems and components. The value of database technology and the Internet has fueled significant growth in storage subsystems, which are now capable of holding petabytes of structured and unstructured information. Networks have interconnected our distributed, heterogeneous systems. And today, those increasingly valuable, complex systems require more and more skilled IT professionals to install, configure, operate, tune, and maintain. To ensure that we are not consumed by the complexity we have created, we have to strive to constantly simplify and abstract the environment that we have created. For this reason, automation is key.

Automation is the capability to dynamically deploy, monitor, manage, and protect an IT infrastructure to meet business needs with little or no human intervention.

Archived

To help customers plan their own automation implementations, IBM has created an Automation blueprint to assist customers in breaking down the tasks of implementing automation into specific capabilities that they can focus on as their business needs require (see Figure 1-3).

Figure 1-3 IBM Automation blueprint

At the bottom of the blueprint is the foundation: the software and system resources with native automation capabilities required for higher-level automation functions. IBM has a full portfolio of hardware and software with built-in autonomic capabilities to allow for the most advanced levels of automation. Many of these resources can be virtualized to the other capabilities. The key point is that in order to achieve the highest levels of on demand automation, resources must be virtualized so that they can be dynamically provisioned as business policies require.

The second layer from the bottom shows the key automation capabilities:

- *Availability* helps ensure that systems are available.
- *Security* keeps systems protected from threats and provides the functions for a great user experience in accessing applications and data they need while keeping out unwelcome users.

- *Optimization* provides tools to make the most of the resources that are in place so that they are running at peak performance and efficiency and provide the maximum return on investment.
- *Provisioning* focuses on the self-configuring, dynamic allocation of individual elements of the IT infrastructure so that identities, storage, or servers are provisioned as business needs dictate.

The next layer, Policy-Based Orchestration, helps customers automatically control all of the capabilities of the four areas we just discussed so that the entire IT infrastructure responds dynamically to changing conditions according to defined business policies. This orchestration builds on the best practices of the customer's collective IT experience and helps to ensure that complex deployments are achieved with speed and quality.

Finally, Business Services Management capabilities provide the tools needed to manage service levels, meter system usage and bill customers for that usage, as well as model, integrate, connect, monitor, and manage business processes end-to-end for complete linkage of IT and business processes.

Archived

Figure 1-4 shows the areas in the Automation blueprint that IBM Tivoli Intelligent Orchestrator and IBM Tivoli Provisioning Manager support. Chapter 2, “IBM Tivoli Intelligent Orchestrator concepts” on page 21 goes into detail for each product’s architecture and functionality and how they interact.

Figure 1-4 Provisioning and Orchestrator products on the Automation blueprint

1.2.1 Provisioning

Provisioning is the end-to-end capability to automatically deploy and dynamically optimize resources in response to business objectives in heterogeneous environments. Provisioning helps to respond to changing business conditions by enabling the ability to dynamically allocate resources to the processes that most need them, as driven by business policies. Provisioning of individual elements, such as identities, storage, servers, applications, operating systems, and middleware, is a critical step to being able to then orchestrate the entire environment to respond to business needs *on demand*.

Benefits of on demand provisioning

Provisioning is done today using manual or some rudimentary scripting language. On demand provisioning differs from what we experience today in the following ways:

- Servers, software, and network resources are added, deleted, moved, and configured to match workloads, as opposed to the just-in-case provisioning that is done today.
- Resources configuration and installation are automated, as opposed to the need to have IT operators focus on maintenance.
- All IT processes are executed in a consistent, customized, and error-free manner, as opposed to labor intensive processes that are prone to human error.
- Servers can be pooled to increase utilization, as opposed to over provisioning, which results in higher capital costs and lower utilization.

1.2.2 Policy-based orchestration

Policy-based orchestration is about providing an end-to-end IT service that is dynamically linked to business policies, allowing the ability to adapt to changing business conditions. Having each individual element of an IT system respond to change is a great start, but in the end, to truly be an on demand business requires orchestration of the automation of multiple elements of the systems so that the entire IT infrastructure is responding as it should to changes in business policies or conditions. For example, if a customer's order entry application suddenly experiences a surge in load, just allocating more CPU may not be enough; it may also need additional storage, more network capacity, and even additional servers and new users to handle the increased activity. All of these changes must be orchestrated so that the dynamic allocation of multiple resource elements occurs seamlessly.

Policy-based orchestration is fundamental to the IBM on demand automation strategy. It enables customers to utilize resources where they support business results most efficiently, by proactively sensing and responding to peaks in demand and allocating IT resources to the most important processes based on business policy:

- Orchestration allows the manipulation of the IT environment in real time, according to defined business policies, to achieve desired business goals.
- Orchestration senses changes in demand for resources and automatically triggers action to re-allocate resources (such as hardware, software, and applications) accordingly throughout the entire system.

- Orchestration introduces a level of automation that can evolve to act as the intelligence across security, availability, provisioning, and optimization, ensuring that the entire IT environment is aligned to business goals.
- Key competitive advantages that orchestration provides include:
  - A software product that adapts to existing hardware, software, architecture and processes
  - Packaged Intelligence using well-defined policies, workflows, and scripts that capture best practices for execution of repeatable tasks
  - Autonomic technology that senses and responds to changes
  - Flexibility to adapt at a customer's own pace

1.3 IBM Tivoli Intelligent Orchestrator

As discussed previously, orchestration enables data centers to move from just-in-case provisioning (providing enough resources to fulfill peaks in IT infrastructure demand, which typically results in low resource utilization) to just-in-time provisioning: automating the infrastructure and executing configuration changes in a repeatable manner, eliminating human execution errors.

Figure 1-5 on page 11 shows a typical example of a data center running three applications, in which one application needs additional resources to attend to user demand, while the other two have enough or even extra resources allocated to them. Traditional, manual provisioning practices do not make it practical to move resources from one application to another to meet short-term peaks in demand. Instead, we engage in what we call the just-in-case provisioning cycle, as seen in Figure 1-5 on page 11.

Archived

Figure 1-5 Just-in-case provisioning

IBM has changed the provisioning paradigm from just-in-case to just-in-time on demand provisioning with IBM Tivoli Intelligent Orchestrator by managing resources information to enhance automation. IBM Tivoli Intelligent Orchestrator dynamically moves computing resources to support the applications with the greater immediate impact on the business just-in-time to meet user demand. Therefore, an organization can reach higher levels of automation and rapidly respond to fluctuating business demands on the IT infrastructure, in line with the overall business goals of the organization.

IBM Tivoli Intelligent Orchestrator automates the traditional, manual provisioning process, performance measurement, capacity planning, and infrastructure deployment. IBM Tivoli Intelligent Orchestrator operates in a closed loop that performs automatic resource requirements prediction, based on predefined service level objectives and agreements, and automates infrastructure deployment. This just-in-time cycle ensures that each application has the resource it needs, when it needs it, without static over-provisioning.

An overview of orchestrated provisioning is given in Figure 1-6.

Figure 1-6 Orchestrated provisioning

While any application can be allocated to any server at any time, a server could be built specifically for an application when it needs it and is subsequently de-allocated when it is no longer needed by the application.

At the foundation of this just-in-time provision cycle is a new way of viewing the IT infrastructure: Rather than traditional single-purpose servers dedicated to one application, IBM Tivoli Intelligent Orchestrator logically aggregates the computing resource it manages into pools available to all applications in the data center while maintaining a secure environment for each.

The paradigm shift in data center operation introduced by IBM Tivoli Intelligent Orchestrator changes the way available IT resources are managed and orchestrated on a daily basis, and helps transition the mind set of the entire IT organization to become more aligned with the primary business of the enterprise. By applying business rules to IT center operations, all levels of the IT organization become more aware of the issues related to, and become more tightly integrated with, the primary lines of business. This enables the IT organization to fulfill its mission of supporting the business rather than being the

one that prevents new initiatives that—rightfully or not—it is regarded as by some today.

However, such a drastic change, which influences all established processes and procedures in the IT organization, requires careful planning. Because business priorities will affect IT operations more than is the case today, the IT organization has to be prepared to give up some areas of responsibility and turn these back to the lines-of-business. In addition, automating many of the processes that are currently handled manually may impose new restrictions and decrease the flexibility with which the data center operates today. This will be necessary in order to reach as high a degree of automation as possible, which in turn will help increase the quality, productivity, resource usage, and agility of the IT organization, thereby allowing for increasing service levels, customer satisfaction, and profits.

In short, by implementing the operational model supported by IBM Tivoli Intelligent Orchestrator, the IT organization will be able to *do more with less*.

This sounds intriguing, and transforming the operations of a complex data center to the ideas of the on demand operational model requires strategy, cautious planning, time, and hard work.

1.3.1 Product overview

The IBM Tivoli Intelligent Orchestrator and IBM Tivoli Provisioning Manager are automated resource management solutions for data centers. The system is a software solution that creates a direct, real-time correlation between application specific service level objectives and the computing resources required to meet these objectives. IBM Tivoli Intelligent Orchestrator proactively configures resources among applications in a multi-application environment to balance user traffic demands, excess capacity, and the obligations of service level objectives. The system predicts capacity fluctuations and facilitates dynamic infrastructure reallocation. The IBM Tivoli Intelligent Orchestrator automates three key data center processes:

- Infrastructure provisioning
- Capacity Management
- Service Level Management

Although this accurately depicts the products, it is difficult to discern from this explanation what the products really encompass. Table 1-1 gives a high-level description of the two products and their relationship.

Table 1-1 Product overview table

IBM Tivoli Intelligent Orchestrator	IBM Tivoli Provisioning Manager
Orchestrated automation.	Coordinated provisioning.
Determines why, when, and where.	Executes what and how.
Senses why to take action, anticipates when to start provisioning, and prioritizes where to put resources.	Coordinates what data center resources are provisioned and executes "personalized" how.
Benefits: ► Reduced hardware and software capital costs. ► Improved alignment between business priorities and IT resources. ► Increased resource utilization. ► Improved speed of deployment. ► Heightened service level delivery.	**Benefits:** ► Controlled labor costs by allowing execution of IT tasks in a repetitive, error-free manner. ► Automated best practices of the company. ► Investment protection by working with your existing hardware, software, and network devices. ► Improved server-to-administrator ratio.

Note: If you acquire IBM Tivoli Intelligent Orchestrator, you get the functionality of IBM Tivoli Provisioning Manager, but *not* vice-versa. Figure 1-7 illustrates this point. The IBM Tivoli Provisioning Manager is a stand-alone product that can be purchased separately, based on your data center business needs.

Archived

Figure 1-7 Product overlap

Figure 1-7 shows that IBM Tivoli Intelligent Orchestrator contains all of the functionality and performs all of the tasks that IBM Tivoli Provisioning Manager provides. We cover the IBM Tivoli Intelligent Orchestrator product in detail in this IBM Redbook and, whenever possible, point out the differences between products.

1.4 Enabling software for on demand provisioning

Instrumenting an application for on demand provisioning requires the simplification of interactions with your application. In essence, it requires the capturing administrator to know how to use reusable elements, thereby being able to automate complex processes. This will allow service personnel to complete engagements rapidly and without human error.

The orchestration and provisioning platforms provided by IBM Tivoli Intelligent Orchestrator and IBM Tivoli Provisioning Manager allow you to store, reuse, and present these elements in a consistent manner. When you build these elements, you are building them for IBM Tivoli Provisioning Manager and therefore for IBM Tivoli Intelligent Orchestrator as well.

For these products, these reusable elements are called workflows. Workflows capture IT expert know-how, and they represent the steps that must be followed in a data center environment in order to carry out a provisioning operation. The individual steps that make up a workflow are called transitions. Workflows can be made to dynamically execute a task and they could be initiated by a changing business requirement.

For example, a workflow could be used to deploy an application to a server, allocate server capacity due to decreased response time, or configure a network

device for new deployment. The types of workflows required will be based on common operating procedures within the supported environments.

1.4.1 The benefits of enabling software for on demand

There are many benefits to enabling software for on demand or writing workflows. These benefits will be outlined here.

Customer experience

Customers solutions usually include a large number of software applications that might come from many vendors. Software technicians usually have to learn the installation and administrative features of each application. Building workflows allows an enterprise to offer a common interface to the multiple applications that make up the solution. This results in:

- Lower support costs by providing a common interface to help desk and other personnel
- Less training and retraining as the operating environment changes, so that the ISV need only modify and adapt the workflows
- Less opportunity for human error in the installation and configuration

Reduce Total Cost of Ownership (TCO)

As mentioned above, making an application easier to use will definitely result in reducing the TCO. Other factors that will result in a lower TCO include:

- Better asset utilization with the ability to provision closer to the time resources are needed
- Reduced costs in time to provision environments
- Increased efficiency
- Reduce infrastructure and deployment costs through automated best practices
- Higher service levels are ensured
- Dynamically respond to market and capacity demands
- Decrease IT staff costs by reducing skill level requirements
- Reduced security exposure through an effective patch management process

Software value

As the computing industry matures, most software packages will support a similar set of features. Competitors mimic each others features and improve on them and, over time, it becomes increasingly difficult to differentiate purely on feature function. Adding integration, automation, or provisioning interfaces

Archived

provides a significant amount of differentiation in the marketplace. Adding these interfaces allow you to demonstrate some tangible cost and productivity benefits.

IBM Tivoli market share

IBM Tivoli has a significant share of the enterprise market. They have a complete management portfolio, making them one of the top providers of systems management solutions. They also have a vast amount of experience. There are just a few vendors that can play in this space. It is therefore conceivable that an application sold into the enterprise space will need to be provisioned and orchestrated by IBM Tivoli Intelligent Orchestrator and IBM Tivoli Provisioning Manager.

Global visibility

IBM has publicly stated its on demand strategy and many of the promised features are being built into the IBM product line. As such, the IBM sales force will be looking for vertical product solutions that have on demand functionality to sell to clients.

ISVs could also take advantage of the on demand message by demonstrating to their clients that their products can be manipulated by IBM Tivoli Intelligent Orchestrator and IBM Tivoli Provisioning Manager workflows.

Systems Integrators (SI)

Many software vendors do not have their own services organizations. Instead, they look to Systems Integrators to provide the necessary services to clients. Systems Integrators' profit is based on productive and successful client engagements.

Systems Integrators would benefit for many of the same reasons that a large enterprise client would benefit. Their personnel would more likely recommend a more usable and manageable system over a more complex environment.

Using workflows, these business partners could also decrease their installation and partner help desk costs

A tool for testing

Many software vendors spend a significant amount of time testing their software. These tests range from installation to granular functional testing. The repetitive nature of these tests make them ideal workflows. Workflows may be created to quickly set up and break down test environments, resulting in an improved time to market and lowering the labor costs

Archived

1.5 The IBM Orchestration and Provisioning Automation Library

To enable software vendors to instrument their applications for on demand capability, IBM provides a number of resources. All of these resources are part of the *IBM Orchestration and Provisioning Automation Library (OPAL)* program. the OPAL program makes it easier to develop industry-leading automation components or workflows, to deliver superior user service, maximize performance of IT resources, and reduce development costs.

The IBM Orchestration and Provisioning Automation Library is available at the following Web site:

http://www.developer.ibm.com/tivoli/workflow.html

Software vendors who participate in the program have access to:

- The workflow developer resource kit, which is a focal point for workflow developers, and provides up-to-date access to developer guides, videos, product downloads, and other useful resources.
- Telephonic consultation that defines a series of next steps.
- Some ISVs will even be entitled to on-site assistance during development.
- Product demonstration, educational material, and classroom technical workshops have been developed. Technical workshops are being held at IBM's worldwide innovation centers. These technical workshops include the following:
  - IBM Tivoli Intelligent Orchestrator Implementation
  - IBM Tivoli Intelligent Orchestrator: Writing workflows

  Additional information about these workshops and other education material can be found at:

  http://www.developer.ibm.com/tivoli/workflow/education.html

Once a application is enabled for on demand provisioning, a series of workflows are developed and properly packaged into an *automation package*. Application vendors can then validate their workflow-enabled applications as Ready for IBM Tivoli Software. These workflows and automation packages are promoted in the *IBM On Demand Automation Catalog*, which is available at the following Web site:

http://www.ibm.com/software/ondemandcatalog/automation

The automation catalog increases the visibility of vendor applications to clients, Business Partners, and global IBM sales teams.

Ready for IBM Tivoli Software validation is designed to provide business value by allowing vendors to demonstrate their integration with IBM Tivoli software. Tivoli software is used in more than 80% of the top financial, retail, telecommunications, and health care companies, and integrating your product can help you gain access to a marketplace of customers seeking compatible solutions.

Products can receive the Ready for IBM Tivoli Software mark by meeting certain IBM-specified integration standards. Ready for IBM Tivoli Software validation is available for a wide range of Tivoli software in the product families of security, storage, performance and availability, and configuration and operations.

Archived

Archived

2

IBM Tivoli Intelligent Orchestrator concepts

This chapter discusses the following:

- 2.1, “Concepts and architecture” on page 22
- 2.2, “Enabling an application/device for provisioning or orchestration” on page 31
- 2.3, “Workflows basics” on page 33
- 2.4, “Automation packages basics” on page 40

© Copyright IBM Corp. 2004, 2006. All rights reserved.

2.1 Concepts and architecture

The product overview table provided in 1.3, “IBM Tivoli Intelligent Orchestrator” on page 10, does not answer the important question for those who wish to implement IBM Tivoli Intelligent Orchestrator: How does TIO/TPM do all of this? In order to answer it, we take a closer look at TIO/TPM’s architecture. However, we must first develop a common vocabulary and have a basic understanding of Web clusters. It will become clear why Web clusters are important as you read more.

First, we look at Web clusters and how they are typically deployed in a data center. Although the IBM Tivoli Intelligent Orchestrator is capable of being deployed and managing various environments, it is particularly well suited for the Web cluster environment.

2.1.1 Web clusters

At a high level, a Web cluster is made up of three main components:

1. Cluster manager: The cluster manager, more commonly know as a *load balancer*, distributes workload to the participating members of the cluster or replicas. Cluster users know the cluster via a virtual IP address, which represents the cluster to the outside world. This effectively hides the actual implementation of the cluster. The IP addresses of cluster replicas (described below) can, and often are, hidden from the user. The cluster manager/load balancer can distribute traffic to replicas based on various characteristics. For example, a common distribution mechanism is round robin, in which the load balancer cycles through each member of the cluster as it distributes load. A cluster manager can and often does represent many Web clusters.

   Note: There are many balancing techniques with today’s load balancers, as well as advanced configurations involving stickiness and content-based routing that are not discussed in the chapter. A detailed understanding of load balancers is required in order to appropriately manage a cluster.

2. Replicas: These are the servers that actually service the requests. Workload is distributed to these systems via the load balancer. It is important to understand that all replicas in a given cluster provide identical services. The parallel nature of replicated servers or services often leads to both improved scalability and increased availability.

Archived

Note: In IBM Tivoli Intelligent Orchestrator, there is a more stringent requirement for the replicas of the cluster: They should be homogenous (hardware and software). On the surface, this may seem like a restriction, but clusters are often built this way.

3. Reserve systems: This component is implied in a Web cluster. Web clusters use a horizontal scaling technique to grow and possibly shrink. Horizontal scaling is a common technique in the Web space and is simply the adding or subtracting of the same service as the workload changes. There are many techniques available for horizontal scaling, the most common being over-provisioning the replicas. (Of course, this assumes that the load always increases.) This is often acceptable in small, single-cluster implementations. In a multi-cluster environment, this over-provisioning can become costly. IBM Tivoli Intelligent Orchestrator introduces a new, powerful way to optimally manage the Web cluster. Figure 2-1 illustrates this basic concept.

Figure 2-1 Basic Web cluster

In order to manage a collection of clusters, some basic tasks must be performed:

1. Information gathering: To make good decisions about a cluster's needs, we need to know the load characteristics of the cluster (such as the number of requests, resource consumption on the replicas, and so on). The real-time acquisition of data is paramount to understanding the needs of a cluster. This information can be obtained directly from both the load balancers and the replicas or via a proxy.
2. Decision-making: Determining where the available resources (reserve systems) should go is more than just interpreting gathered performance metrics from the cluster. A balance must be achieved between available resources and application service level objectives. Application priority (its importance to the business) must also play a part in the decision-making process. Finally, there must be a way to control the allocation of assets so that the clusters do not thrash (rapidly add and subtract resources).
3. Automated provisioning: The introduction and removal of systems in a cluster follows the organization's provisioning processes. Ideally, this process should be able to run completely unattended as a controlled, verifiable, and automated operation or workflow.
4. Cluster visualizing/configuring: Finally, we need to be able to see, update, and enhance the clusters as a whole. It is also important to access information interactively as well as programmatically.

2.1.2 Concepts and terminology

Now that we understand Web clusters, now we will develop our basic IBM Tivoli Intelligent Orchestrator vocabulary.

Throughout this IBM Redbook, we use IBM Tivoli Intelligent Orchestrator terminology. These key terms are introduced to give a context and common vocabulary:

Workflow A series of steps that are sequentially executed to accomplish a particular task. A step in a workflow is called a *transition*. Each workflow has a single compensating workflow that is executed if any transition fails. Chapter 6, "Developing workflows" on page 175 goes into detail on workflows components and development.

Automation package Also referred to as *TC driver*, *device driver,* or simply *driver*, an automation package is a collection (or a container) of commands, shell scripts, workflows, *logical operations*, and Java programs and plug-ins that applies to the operation of one specific type of software component or a physical device. It contains a grouping of

tasks that corresponds to a physical or logical device. These tasks typically implement logical operations. A device could be a specific piece of hardware, an operating system, a service, or a cluster. Chapter 8, “Automation package content and packaging” on page 233 provides instructions on how to create and package your workflows into automation packages.

Logical operation A task that is abstracted from its implementation. Logical operations are implemented via Enterprise Java Beans (EJB™). They provide a common interface and can perform logic. An example is the data center task of adding an IP address. It is a logical operation in that it makes no assumptions about the implementation. (Note that adding an IP address to Linux is very different from adding an IP address to Windows®.)

Transition A step in a workflow. This could be another workflow, a logical operation, a Java plug-in, or a Java program.

Data Center Model A representation of all of the physical and logical assets under IBM Tivoli Intelligent Orchestrator management.

Archived

Figure 2-2 illustrates the relationships between workflows, the Data Center Model, transitions, device models, and more.

Figure 2-2 Data center relationship model

Customer A customer owns applications. Customers can be unique corporations or departments within a single corporation.

Application A grouping of one or more clusters. Service level priority (Silver, Gold, or Platinum) is assigned at this level.

Application cluster A grouping or container for like resources or servers that support an application. Automated resource allocation and deallocation occurs at the cluster level.

Resource pool A container of available (deallocated) servers that support one or more application clusters. Also referred to as a spare pool.

Servers Data Center Model objects that represent physical servers. They belong to or are assigned to pools and clusters.

Archived

Figure 2-3 illustrates the relationship between customers, applications, clusters, pools, and servers.

Figure 2-3 Customer relationship model

Software stack Either an image stack or product stack that contains an ordered list of Software Products, software stacks, or both. An image stack is created via a boot server, while a product stack is not.

Software product The attributes and the methods for deploying a piece of software on an asset. A Software Product can be user-written or commercial off-the-shelf (COTS).

Service access point (SAP®)
A definition of the protocol and credentials used by or associated with an asset. An asset can have more than one SAP.

Service level objective (SLO)
A definition of the desired behavioral characteristics of an application and clusters.

Switch fabric The container for “related” infrastructure components (switches, routers, and so on). Your Data Center Model can have more than one switch fabric.

Archived

2.1.3 Architecture

This section introduces components of the IBM Tivoli Intelligent Orchestrator architecture. For a more-detailed discussion, refer to *IBM Tivoli Intelligent Orchestrator Overview Guide*, SC32-1419. Figure 2-4 depicts the IBM Tivoli Intelligent Orchestrator's high-level component architecture.

Figure 2-4 High-level component architecture

IBM Tivoli Intelligent Orchestrator's architecture includes the following main components:

- Data Acquisition Engine
- Application Controller
- Global Resource Manager
- Deployment Engine
- Management Interface

Data Acquisition Engine

The Data Acquisition Engine is responsible for acquiring and preprocessing performance metric data from each managed application environment. Data can

Archived

be captured and filtered from the operating system and infrastructure layers. This component can also distribute signals to other components of the IBM Tivoli Intelligent Orchestrator. This component participates in the information-gathering task.

Application Controller

An instance of the Application Controller component is created for each application environment under management. Based on the application's workload model and predictions, as well as on real-time performance data (acquired from the Data Acquisition Engine), this component determines the resource requirements of the application. This component provides information to the decision-making task.

Global Resource Manager

This component receives requirements for servers or network devices from all Application Controllers and manages the overall optimization of data center assets. It has two primary responsibilities:

1. It makes optimal resource allocation decisions.
2. It ensures a stable control over the application infrastructure.

This component participates in the decision-making task. Considering the different server requirements for each application environment, it determines where the servers are to be allocated.

Deployment Engine

This component is responsible for the creation, storage, and execution of repeatable workflows that automate the configuration and allocation of assets. A workflow can represent either an entire re-configuration process affecting multiple servers or a single step in a larger re-configuration process. This component participates in the automated-provisioning task.

The Deployment Engine processes requests by sequentially executing workflow transitions or steps. Workflows are coordinated through the Deployment Engine's controller, which contains the following components (shown in Figure 2-5 on page 30):

- Workflow assembly component: This component receives a request or command and coordinates the translation of the deployment request into an executable workflow. This mechanism searches the workflow database to determine whether the deployment request, or parts of it, can be represented by workflow.
- Workflow execution component: This component receives the workflow for execution and determines which asset is pertinent to each transition. Commands are created for the corresponding steps and formatted to be

recognizable to the asset. Asset-specific commands are then forwarded for implementation on that asset (such as Linux Add IP Address). A workflow execution controller controls workflow execution and provides multiple working threads to enable simultaneous execution of workflows on multiple machines.

Figure 2-5 Deployment Engine

Management Interface

This component provides an overview of the state of all physical and logical assets in the data center infrastructure, offering information about the servers and their allocation, and generating configurations and allocations. It can also be used to create application environments. It includes two user interfaces: a Web-based interface and a command-line interface. This component participates in the cluster visualizing and configuring task.

The Management Interface component includes two distinct user interfaces that can be used to manage and configure the application resources:

- Web-based interface

  This offers an intuitive way to display information about applications and components. The Web-based user interface offers real-time access to the IBM Tivoli Intelligent Orchestrator, as well as a way to display information about the deployed application and components. It enables you to monitor and control the managed resources and application.

- Command-line interface

  This can be used by operators who prefer to access the system's internal properties and operations using the command line. It is in the format of Web services SOAP calls.

2.2 Enabling an application/device for provisioning or orchestration

In this IBM Redbook, we will focus specifically on the components that need to be developed to instrument an application or hardware device for provisioning and orchestration.

If we take an abstract view of an application or device, as shown in Figure 2-6 on page 32, most have either a command-line or graphical user interface. These interfaces may be used to administer, configure, or use the device or application. For example, a router or a Web application usually have both interfaces. Some applications or devices may even have an application programming interface if low level interaction is required.

Instrumenting an application or device involves using the native interfaces, whenever possible, to ease the manipulation of such an application or device. In IBM Tivoli Intelligent Orchestrator and IBM Tivoli Provisioning Manager terminology, the set of steps required to achieve a particular task is called a workflow, and the grouping of these workflows together with the required artifacts is called an automation package. More complete descriptions are given in the sections that follow.

2.2.1 Workflows for provisioning and orchestration

Previously, we defined provisioning, or the work done by the IBM Tivoli Intelligent Orchestrator or IBM Tivoli Provisioning Manager, as coordinating what data center resources are provisioned and executing customized processes, so workflows developed by software or hardware vendors will typically be used for some sort of provisioning activity. These workflows will usually perform tasks very similar to the steps described in the application or devices installation and configuration guide and manuals. Such manuals are very often a good place to plan which workflows should be developed and specify the tasks required to provision that particular application or device.

Orchestration was defined as sensing when to take action and anticipating when to start provisioning, and prioritizing where to put resources. In this context, orchestration involves sensing and provisioning. The sensing will typically be done in coordination with some other product or process within a datacenter.

Provisioning is then done based on what is sensed. Workflows developed for this environment will usually be built at the datacenter by the user, possibly with the aid of the ISV. These workflows will usually be built using the workflows templates developed by the ISV for provisioning the application or device. In case the provisioning action cannot be accomplished, it is always a good idea for the workflows developer to create recovery actions in the workflows. These recovery actions steps in the workflows will probably attempt to perform the tasks described in the application or device troubleshooting and problem determination manuals. Such manuals are the best place to plan the recovery actions that need to be coded in the workflows. One could use these guides to decide how granular the provisioning workflows should be.

Figure 2-6 Workflows for provisioning and orchestration

For a more-detailed discussion about the IBM Tivoli Intelligent Orchestrator and IBM Tivoli Provisioning Manager products and their capabilities, refer to *Provisioning On Demand: Introducing IBM Tivoli Intelligent Orchestrator*, SG24-8888.

Archived

2.3 Workflows basics

As we have seen so far, workflows provide detailed specification of the activities to be performed during a provisioning operation. IBM Tivoli Intelligent Orchestrator and IBM Tivoli Provisioning Manager provide a number of best-practice workflows that are extensible for addressing existing processes.

This section introduces workflows and covers the following topics:

- What workflows are and how to use them in a data center environment
- How they are run
- The components that make up a workflow

This section will provide an overview and explain the principal steps required to produce workflows. Subsequent chapters will provide greater detail for each of the building workflow steps.

> **Note:** In defining workflows for use in enterprise data centers, you are effectively defining some of the operational processes that will be observed. It is therefore important to ensure that workflows follow any company policies and best practices in place.

At the end of this section, we provide a summary of how workflow components fit together.

2.3.1 What is a workflow

In IBM Tivoli Intelligent Orchestrator terms, a workflow is a representation of the set of steps that must be followed in a data center environment in order to carry out a provisioning operation. The individual steps that make up a workflow are called *transitions*.

In addition to transitions, workflows can include control logic to perform loops (while-do, foreach-do), conditional processing (if-else), number and string manipulation through the built-in Jython support, exception handling, as well as execution of imbedded scripts (so called scriptlets).

There are four different types of transitions:

1. Logical operation

   A logical operation can be defined as a task that is abstracted from its implementation. An example would be the data center task of adding an IP address. It is a logical operation in that it makes no assumptions about the implementation. For example, adding an IP address to Linux is very different from adding an IP address to Windows.

2. Java program

   Java scripts that provide special functionality may be called directly from a workflow, as long as they are present in the class path. This allows the workflow developer to provide extended capabilities not available through normal scripting to the provisioning workflows, for example, to communicate with a Web Services enabled application.

3. Java plug-in

   A Java plug-in is the Java class that contains the interface or protocol code that interacts with the devices in the data center. Very often, you will find that Java plug-ins are wrapped by workflows, describing the plug-in's input and output requirements via its input and output variables, in order to provide an extra layer of virtualization that will help customization and migration.

   Important: As of IBM Tivoli Intelligent Orchestrator and IBM Tivoli Provisioning Manager Version 3, the support for user-written Java plug-ins are not being enhanced; rather, it is replaced with support for calling Java programs directly from your workflows, but support for Java plug-ins is continued to provide backward compatibility.

4. Workflow

   Transitions can also be other nested workflows. In this way, workflows make it possible to create a library of processes that can meet any data center process requirement.

When a workflow is run, the transitional steps are carried out one after another in list order until the final step is finished and the workflow has been completed.

2.3.2 What can you do with workflows

Because workflows are used to carry out provisioning operations in a data center, the transitions enable a wide variety of activities to be performed on systems and network devices.

The set of activities for a particular provisioning operation may affect multiple systems and network devices in the data center.

Some examples of the types of operation that can be performed with workflows are:

- Allocate servers to clusters.
- Remove servers from clusters.
- Install software and patches.
- Un-install software.

- Configure servers.
- Configure network devices.
- Run commands.
- Perform bare-metal builds.

For example, when using IBM Tivoli Intelligent Orchestrator, you can run a workflow when an application is overloaded in order to add a new server to the application cluster. This add server operation may involve several steps, such as assigning an IP address to the server, configuring network devices and load balancer in the data center, and installing software packages and required software fixes on the server. Depending on the ISV solution, workflows developed by the ISV can be used on one or more of the sample steps described here.

2.3.3 How are workflows controlled

As mentioned in 2.1.3, “Architecture” on page 28, workflows are controlled by the Deployment Engine. It is through the running of workflows and their components transitions (some of which can also be run in isolation) that operations are carried out in the data center

Essentially, the IBM Tivoli Intelligent Orchestrator product uses the Deployment Engine and workflows to maintain a standard approach to data center operations.

Workflows can be run in several different ways:

- A workflow assigned to an object can be run when the associated logical operation is triggered.
  - This can be triggered by the Resource Manager.
  - Alternatively, SOAP commands can be used to trigger a logical operation.
- Workflows not associated with logical operations can be executed from the GUI or through the SOAP interface.

2.3.4 Workflow transitions and variables

Upcoming sections offer descriptions of the transition types allowed in workflows:

- Logical operations
- Java programs
- Java plug-ins
- Workflows

Archived

Logical operations as transitions

Logical operations provide a layer of abstraction from the physical operations defined by specific device drivers or workflows. Different device drivers can implement device-specific operations for the same set of logical operations. For example, the logical operation *Add IP Address* can be implemented differently on Linux, AIX®, and Solaris™. These logical operations can be used as workflow transitions.

Figure 2-7 shows how logical operations are related to workflow transitions and device models.

Figure 2-7 Workflow component relationships

Creating workflows based on logical operations means greater reuse because logical operations can be used by other workflows.

Java programs as transitions

Both Java programs and Java plug-ins can be wrapped by a workflow. Often, you can define multiple wrapper workflows for a single Java plug-in or Java program,

Archived

each calling a different method and providing only a portion of the functionality available in the Java code.

Where possible, you should use wrapper workflows instead of direct calls to the Java code. This will make your logic more easily understood by anyone working with the workflow.

For example, these six workflows all are used to wrap the build-in Java plug-ins for the TIO/TPM resource manager:

- RM_Add_Server
- RM_Allocate_Ip_Address
- RM_Allocate_Server
- RM_Choose_Server_for_Removal
- RM_Release_Server
- RM_Remove_Server

Java plug-ins as transitions

Java plug-ins provide the interface for interaction with data center devices. Every action that is performed in the data center is implemented by the Deployment Engine using a Java plug-in.

Java plug-ins are implemented by Enterprise Java Beans (EJB), so the versatility of the Deployment Engine is limited to the versatility of the (external) commands available through the EJB implementation.

It is possible, but not recommended, to program new Java plug-ins to enhance functionality. This is usually done to extend the existing functionality or when developing new IBM Tivoli Intelligent Orchestrator workflows and automation packages to a specific purpose. Instead of coding your own Java plug-ins, you should rather provide special functions as Java scripts, which (as of IBM Tivoli Intelligent Orchestrator Version 3) is the preferred method.

Workflows as transitions

Workflows themselves can be run as transitions in a workflow. This enables the development of modular solutions for data center provisioning operations.

Warning: It is possible to create a workflow that references itself indirectly. (Although the GUI does not allow you to drag a workflow into itself, you can drag a workflow into a second workflow and then drag the second workflow into the first to create a recursive loop.) A workflow created this way will fail when it is executed.

Archived

Using variables in transitions

For each of the workflow components and for the workflow itself, a set of variables is used to store input and return values:

Input variables	Parameters or arguments passed to the workflow or transition
Output variables	The returned values of the workflow or transition
Local variables	Variables that are used internally in the workflow or transition

Note: Transitions and workflows can have local variables that are used within the transition but not used for input and output of data.

Summary of workflow component relationships

The relationships between the workflow components are:

- Logical operations are implemented by workflows.
- Workflows are made up of logic and transitions.
- Transitions can be logical operations, Java scripts, Java plug-ins, or workflows.
- Java scripts provide the preferred method of implementing external commands and offering a range of functionality.
- Java plug-ins are Enterprise Java Beans, implementing external commands and offering a range of functionality.

Figure 2-8 on page 39 shows these relationships.

Archived

Figure 2-8 Workflow component relationships summary

2.3.5 Building workflows

Although workflows can be developed using a standard text editors and later imported into the IBM Tivoli Intelligent Orchestrator environment, the preferred tool to use for workflow development is the Automation Package Development Environment (APDE). Alternatively, you can use the TIO/TPM Web based User Interface to perform all of the following operations:

- Creating a workflow
- Cloning a workflow
- Editing a workflow
- Adding a transition to a workflow
- Working with variables
- Tagging a workflow with a logical operation
- Assigning the workflow to an TIO/TPM object
- Inheriting multiple workflows from an automation package

Archived

- Removing a workflow from an TIO/TPM object
- Exporting a workflow to a file
- Importing a workflow from file
- Deleting workflows

Chapter 3, “Architectural design” on page 45 and Chapter 6, “Developing workflows” on page 175 go into detail on how to plan, design, and create workflows for your solution.

2.4 Automation packages basics

IBM Tivoli Intelligent Orchestrator and IBM Tivoli Provisioning Manager is shipped with a number of automation packages. Out of the box, it covers major networking devices, software components, and platforms, such as IBM AIX, various distributions of Linux, Microsoft® Windows, and others that exist in a typical data center. It is fair to assume that future releases of this product will include:

- Automation package enhancements
- New automation packages for additional platforms not currently supported

In general, a user (data center administrative staff responsible for provisioning and deployment of Data Control Model objects) does not have to be concerned with the development of an automation package. However, extending an existing driver to satisfy a specific requirement may be desired or even required. For example, one could create a new automation package that extends the existing standard Cluster Operation Templates automation package that shipped with the product to include workflows and logical operations that apply to a particular environment. In this example, operations like removing a server from a cluster could be modified to include a bare metal installation of a particular operating system required by the processes and policies of the customer.

2.4.1 What is an automation package

An automation package, also referred to as *TC driver*, *device driver, TC Driver package,* or simply *driver*, is a collection (or container) of commands, shell scripts, workflows, logical operations, and Java scripts and plug-ins that applies to the operation of one specific type of software component or a physical device.

These resources are packaged into a single file with the .tcdriver extension. The Microsoft IIS Application driver (iis.tcdriver) or a Cisco CSS 1100 content Switch driver (cisco-css-11000.tcdriver), which are installed during the installation process of the IBM Tivoli Intelligent Orchestrator, are examples of available automation packages.

An automation package is an abstraction layer representing various manufacturers' products and related versions. For example, each automation package may include a make and model number. This abstraction ties the Deployment Engine workflows to a higher-level abstraction for all data center assets. For example, all Cisco CSS model 11050 switches in a data center can perform the same action using the same workflow, which is dependent on the make and model and not on a specific instance.

It is always desirable to assemble the full suite of logical operations supporting the data center devices into an automation package.

When creating a new set of automation packages for managing a resource, you should include all of the high-level operations that apply for that device model. Include only those components in the package that uniquely apply to that resource. There may be other dependencies, but these should be referenced by specifying the dependencies in the tc-driver manifest described later in Chapter 8, "Automation package content and packaging" on page 233.

There may not always be a one-to-one relationship between operations and automation packages. If some operations are shared between automation packages, consider packaging them together. For example, when packaging the drivers for the load balancer BigIP, the versions for both 3.3 and 4.1 were packaged together. Most of the Cisco switching equipment drivers were packaged together.

Logical device

A logical device is an abstraction of common behaviors and interfaces. For example, all switches assign a port to a VLAN no matter the make and model. The tasks performed on a logical device are called logical operations (for example, Switch.Move Port to VLAN).

Logical devices can be an abstraction of a physical device (an inter-networking device, such as a switch, a router, a firewall, or a boot server) or a virtual device (such as a Software Product or a file system).

Archived

Logical operation

Logical operations, also referred to as *DCM interactions*, are EJB methods implemented by workflows to carry out a specific function. These are invoked whenever the corresponding workflows are executed. It is always desirable to create workflows that implement logical operations whenever possible. There are at least two reasons to do so:

- Reuse

  The logical operations can be shared among multiple workflows.

- GUI usage

  The GUI is written in such a manner that the fast function buttons and drag-and-drop functionality make their use easier.

2.4.2 Automation package .tcdriver file

The automation package is a compressed Java jar file that includes various directories (depending on the elements in the package). The directories contain all the elements required to manipulate the specified application or device, such as the manifest XML file describing the different elements, a doc directory, for overall documentation, all the developed workflows, Java plug-ins, scripts, and external commands.

Chapter 8, “Automation package content and packaging” on page 233 provides a process and guidance through all the steps required to create an automation package.

2.4.3 Managing automation packages

The IBM Tivoli Intelligent Orchestrator provides a management interface that interacts directly with the DCM database to dynamically build a visual representation of the managed data center. This interface renders data collected dynamically by the Data Acquisition Engine, the Deployment Engine, the Application Controller, and the Global Resource Manager.

It provides two different methods of access:

- Web-based interface

  This interface provides real-time access to the IBM Tivoli Intelligent Orchestrator environment. It is the primary method of configuring, defining, customizing, and operating the data center.

  Even though it is possible to create automation packages using the GUI, it is *not* the recommended method. Adding an automation package via the GUI does not create the associated .tcdriver file.

- Command-line interface

  The command-line interface is used by the system operators and administrators primarily to access system internal properties and various functions. It is also considered an integration interface to external fault management systems.

  IBM Tivoli Intelligent Orchestrator and IBM Tivoli Provisioning Manager provides the **tc-driver-manager** command for managing automation packages and has the following format:

  ```
  tc-driver-manager <command option> <package name>
  ```

  where:

  <**command option**> One of the options shown in Table 2-1

  <**package name**> The automation package name to be operated on

Table 2-1 tc-driver-manager command line

tc-driver-manager command option	Description
forceInstallDriver	Un-installs and installs the named driver.
getDescription	Retrieves tcdriver's description.
getDocumentation	Retrieves tcdriver's documentation.
getDriverStatus	Shows tcdriver's status.
installDriver	Installs a new driver named on the command line.
installNoItems	Installs the named automation package on the command line without creating workflows described in the Items section of the driver manifest.
listAllStr	Lists all currently installed drivers.
listDeviceModels	Un-installs the named automation package in the command line.
uninstallDriver	Un-installs the named automation package in the command line.

Chapter 3, "Architectural design" on page 45 and Chapter 8, "Automation package content and packaging" on page 233 go into detail on how to plan, design, and create automation packages.

Archived

Archived

3

Architectural design

The workflows built for IBM Tivoli Intelligent Orchestrator provide detailed specification of the activities performed during a provisioning operation. In this chapter, we discuss the steps required for planning and designing workflows and automation packages from an architectural point of view.

This chapter covers the following topics:

- Functional overview of workflows and automation packages
- Designing the automated provisioning solution
- Determine Automation Package functionality
- Verify automation feasibility
- Define high-level operations
- Identify transitions
- Design implementation procedures
- Architecting best practices
- Practical aspects of designing automation packages and workflows
- Authentication and authorization internals
- Naming conventions
- Documentation guidelines

© Copyright IBM Corp. 2004, 2006. All rights reserved.

3.1 Functional overview

This section gives an short overview about the functionality of workflows, workflow components, and automation packages provided by IBM Tivoli Intelligent Orchestrator. It contains information about following objects and how they work:

- Logical operations, Java programs, Java plug-ins, and workflows
- Transitions and variables
- Feasibility and control of workflows
- Automation packages

3.1.1 Workflow overview

As we have seen in Chapter 2, “IBM Tivoli Intelligent Orchestrator concepts” on page 21, in IBM Tivoli Intelligent Orchestrator terms, a workflow is a representation of the set of steps that must be followed in a data center environment in order to carry out a specific provisioning operation. The individual steps that make up a workflow are called *transitions*.

Figure 3-1 shows the *Configuration* pane of IBM Tivoli Intelligent Orchestrator Web UI. The workflows and their transitions discussed below are managed through this item.

Figure 3-1 Configuration pane

There are four different types of transitions:

1. Logical operation

   A logical operation can be defined as a task that is abstracted from its implementation. An example would be a data center task of adding an IP address. It is a logical operation in that it makes no assumptions about the

implementation. For example, adding an IP address to Linux is very different from adding an IP address to Windows.

2. Workflow

   Transitions can also be other nested workflows. In this way, workflows make it possible to create a library of processes that can meet any data center process requirement.

3. Java plug-in

   A Java plug-in is the Java class that contains the interface or protocol code that interacts with the devices in the data center.

4. Java program

   An external Java program that provides specialized functionality. The package (jar) file containing the Java classes that are called has to be present in the class path of the TIO/TPM server machine but contrary to Java plug-ins, no special bindings are required, and the external Java programs does not have to be registered with TIO/TPM.

When a workflow is executed, the transitional steps are carried out one after another in list order until the final step is finished and the workflow has been completed.

Feasibility of workflows

Because workflows are used to carry out provisioning operations in a data center, the transitions enable a wide variety of activities to be performed on systems and network devices. The set of activities for a particular provisioning operation may affect multiple systems and network devices in the data center.

Control of workflows

Workflows are controlled by the Deployment Engine. Through the execution of workflows and their components, operations are carried out in the data center. Essentially, the IBM Tivoli Intelligent ThinkDynamic Orchestrator product uses the Deployment Engine and workflows to maintain a standard approach to data center operations.

Using variables in transitions

For each of the workflow components and for the workflow itself, a set of variables is used to store input and return values:

Input variables Parameters or arguments passed to the workflow or transition

Output variables The returned values of the workflow or transition

Archived

Local variables Variables that are used internally in the workflow or transition

Transitions and workflows can have local variables that are used within the transition but not used for input and output of data.

3.1.2 Overview of automation packages

IBM Tivoli Intelligent Orchestrator is shipped with a number of automation packages. Out of the box, it covers major networking devices, software components, and platforms, such as IBM AIX, various distributions of Linux, Microsoft Windows, and others that exist in a typical data center.

Figure 3-2 shows the most of the groups of device drivers provided by IBM Tivoli Intelligent Orchestrator.

Figure 3-2 Groups of device drivers

An automation package is a collection (or container) of references to Java scripts, shell scripts, workflows, Java plug-ins, and logical operations that applies to the operation of one specific type of software component or a physical device.

An automation package is an abstraction layer representing various manufacturers' products and related versions. For example, each automation package may include a make and model number. This abstraction ties the Deployment Engine workflows to a higher-level abstraction for all data center assets.

When creating a new set of automation packages for managing a resource, you should not implement all of the high-level, logical operations that apply for that device model. Include only those components in the package that uniquely apply to this solution. There may be other dependencies, but these should be referenced by specifying the dependencies in the tc-driver manifest file that is the backbone of the automation package definition. The tc-driver manifest file will be discussed in detail in Chapter 8, "Automation package content and packaging" on page 233.

Logical device

A logical device is an abstraction of common behaviors and interfaces. The tasks performed on a logical device are called logical operations. Logical devices can be an abstraction of a physical device or a virtual device.

Logical operation

Logical operations, also referred to as *DCM interactions*, are EJB methods implemented by workflows to carry out a specific function. These workflows are invoked whenever the corresponding logical operation is executed. It is always desirable to invoke workflows through their associated logical operation whenever possible.

3.2 Designing the automated provisioning solution

When setting out to develop the automated provisioning solution for a specific product, it is more than likely that manual installation procedures that are independent from IBM Tivoli Intelligent Orchestrator have already been developed. The task at hand is to automate these procedures, hopefully without having to modify them.

The main items in the automated provisioning solution are the workflows that perform the various provisioning tasks. These are normally packaged into an Automation Package, which is used by IBM Tivoli Intelligent Orchestrator. However, before designing these tasks, it is necessary to clearly understand the

manual processes and determine what is being delivered to the customer and how.

In the following section, we will take a closer look at the requirements for a set of deliverables that a software vendor could provide to the customer. This set is, in our terms, called the Orchestration Solution Package.

3.2.1 The Orchestration Solution Package

When architecting and developing Automation Packages, you may discover that customers need additional files and executables on top of what is provided in the Automation Package or that some of the resources that are included in the Automation Package may also have to be referenced from outside of the IBM Tivoli Intelligent Orchestrator environment. An example of this would be software images and installation tools, which, during the customized provisioning processes, might be used from the IBM Tivoli Configuration Manager environment through calls from the automated provisioning processes (workflows). The same might true for various installation and customization files, such as model response files and the like.

The point is that in many cases, the Automation Package alone may not provide the best way of distributing the objects needed to support or implement a solution. The Automation Package is specifically designed to provide the best means for packaging and distributing pre-defined workflows, scripts, Java plug-ins, and their related DCM objects. You cannot expect a large, versatile customer environment, in which multiple systems management solutions have been deployed to provide special capabilities to support the customers processes, to only use the function set provided by IBM Tivoli Intelligent Orchestrator (TIO/TPM). Therefore, to be able to support as many customer environments as possible, you should package any software as you normally would, and, in addition, provide Automation Packages and documentation to allow the customer to implement the new solution using the existing processes and tools.

In the following, we will use the term *Orchestration Solution Package* to denote the combined bundle of deliverables (like a CD-ROM) for a solution. The purpose of the Orchestration Solution Package is to provide all the objects (for example binaries, license keys, and documentation) required to successfully deploy a solution in a customer environment, including TIO/TPM definition templates, workflows, and DCM objects, to allow for fully automated installation and customization.

The content of the provisioning solution package is dependent on the solution at hand and the customer environment in which the solution will be used. For example, the content of a Orchestration Solution Package could be:

- Code libraries
- License keys
- Sample response files
- Installation and customization tools
- Automation package(s) for IBM Tivoli Intelligent Orchestrator and IBM Tivoli Provisioning Manager
- Skeleton XML files for defining devices, such as software definitions -packages and -stacks, and new object property definitions needed in the customers Data Center Model
- Other binaries to support additional integration points, such as sample IBM Tivoli Monitoring (ITM) Resource Models, and template IBM Tivoli Enterprise™ Console (TEC) rule files
- Instructions on how to import, customize, and use the Automation Package and related deliverables
- Other relevant product documentation

The content of the Automation Package should be:

- Workflows for integration to relevant tools that might be used by the customer to perform specific tasks, such as software installation or license management. Some customers may prefer to install software through Tivoli Configuration Manager instead of standalone by TIO/TPM. However, stand-alone procedures *must* be provided in order to allow customers that do not use external tools to include your solution into their operating environment.
- Java plug-ins, custom-written Java jar files implemented by the automation package's Java plug-ins.
- Java scripts, custom written Java files used to provide special functions to be referenced from workflows
- Scripts, and other executables used to automate the provisioning processes.
- Automation package documentation.

The recommendations listed above do not apply exclusively to vendors of solutions with software or firmware content. Vendors of strict hardware solutions are also encouraged to provide skeleton DCM definitions to allow the customer to easily discover and define the new devices in the data center model - and it goes

Archived

without saying that current and detailed documentation is a requirement and cannot be neglected.

As demonstrated, the main purpose of the Automation Package is to provide the packaged set of workflows, standard workflows, scripts, executables, and Java code that support the automated provisioning and de-provisioning of a device (or a device type).

To recapitulate the major building blocks of Automation Packages and their purpose, please refer to Figure 3-3.

Figure 3-3 Orchestration Solution Package overview

The purpose of the Automation Package is to provide the means by which provisioning of specific implementations of objects of a predefined type can be manipulated. In layman's terms, this means that you enable users of TIO/TPM to easily automate the install, deploy, customize, operate, and remove procedures for a specific solution within their own environment.

3.2.2 Architecting Orchestration Solution Packages

To architect and design a Orchestration Solution Package for a solution (hardware or software based) that is going to be provisioned by TIO/TPM, the software architect needs to follow a structured approach that will help define the actual scope and implementation details. The logical flow is:

1. Define Orchestration Solution Package content and scope.
2. Determine Automation Package functionality.
3. Verify automation feasibility.
4. Determine device types to be supported.
5. Define high-level operations.
6. Define low-level operations.
7. Break down operations into workflows.
8. Define external prerequisites, dependencies, and interfaces.
9. Define location and naming conventions for external files.
10. Identify transitions.
11. Create specifications for new code to be developed.
12. Develop recovery policies.
13. Design templates for DCM object and related files.
14. Design implementation procedures.
15. Ensure proper documentation.

Before starting out on this path, a minimum background knowledge about IBM Tivoli Intelligent ThinkDynamic Orchestrator, its workflows, and Automation Packages may be required. For more information of these issues, you should consult Chapter 6, “Developing workflows” on page 175 and Chapter 8, “Automation package content and packaging” on page 233 to gain insight in the development and packaging processes.

3.2.3 Define Orchestration Solution Package content and scope

The first step in architecting an Orchestration Solution Package is to decide the content and scope of the package. This may seem to be an obvious decision, but in reality, many third-party solutions rely on already established networking and middleware solutions or components - and a vital decision when architecting and defining the scope of the Orchestration Solution Package is to clearly determine which components and functions are included in the package and which are left out.

Many solutions today rely on the existence and proper configuration of related components. As examples, many application systems rely on database management systems and application run time platforms, such as IBM WebSphere Application Server. These in turn rely on an existing networking environment, which may also be the case for hardware specific solutions, such as switches and firewalls.

Archived

The main decision to be made by the software architect is to clearly define the scope and deliverables of the Orchestration Solution Package.

Software based solutions are often bundled in such a way that by acquiring an application, the underlying middleware software is provided as part of the solution. Given this fact, the software architect has to determine if the Orchestration Solution Package provided with the solution should include Automation Packages for the related middleware products or not.

For example, an ISV bundles together with its software solution a Web application server, a RDBMS, and an LDAP directory running on Windows, AIX, and Linux platforms. Would the ISV want to provide Automation Packages for all four products (its own application plus the three middleware products) on all three platforms? It is not likely. In this case, the Orchestration Solution Package for TIO/TPM may include installation images for the required middleware products, but only one Automation Package for TIO/TPM, assuming that the middleware vendors provide the Automation Packages for their specific components.

This IBM Redbook presents a case study scenario in Chapter 10, “Case study: Trade3 composite application” on page 281, where it is assumed that the deliverables from the Trade3 application provider would be the Trade3 application executables, the response file templates to be used for installation, and an Automation Package specifically made for the Trade 3 application. The Trade3 application provider would not provide Automation Packages for the required middleware applications: IBM WebSphere Application Server and IBM DB2 systems.

Typical deliverables an ISV could include are:

- Hardware
- Software (for the actual solution and, optionally, prerequisite software)
- Templates
  - Response files for installation
  - TIO/TPM Data Center Model definitions
- Automation Package(s), including templates functions for:
  - Clonable workflow templates
  - Workflows for device operation
  - Java plug-ins and Java scripts for device interaction
- Other, non-TIO/TPM related deliverables, for example, software signatures, ITM Resource Models or TEC rules
- Documentation

3.2.4 Determine Automation Package functionality

Once it has been decided which components (software and Automation Packages) to deliver as part of the Orchestration Solution Package, the next step is to determine the functionality of the Automation Packages included in the Orchestration Solution Package.

For a hardware based solution, setup and initial customization may be considered manual functions that will be considered prerequisites to automating the provisioning, but for software based solutions, these functions may be included in the Automation Package.

For both types of solutions, however, these functions may be required to interact with other components in the DCM when implementing, deploying, and operating the solutions. For example, the Trade3 application requires access to a database (which may or may not reside on a system that is part of the same application, or even the same application tier), and creating, priming, and cataloging the database may or may not be included in the Automation Package. A similar example using switches, routers, and firewalls could also have been used. The point is that the software architect has to decide which functions to include in the Automation Package and which to leave out.

To determine which functions to implement in the Automation Package, the installation and customization guides for any solution are valuable sources of inspiration and information. Ideally, these documents provide step-by-step information on all the required activities to set up, deploy, and operate a given solution.

Typical functions that should be considered are:

- Installation
- Customization
- Operation
- Removal
- Cleanup

Prior to provisioning, the solution's various objects, such as File Repositories, Devices, Automation Packages, Software, and so on, have to be defined to the IBM Tivoli Intelligent ThinkDynamic Orchestrator environment so that the solution can be provisioned. You might consider including workflows for performing these setup tasks in the Orchestration Solution Package.

Archived

3.2.5 Verify automation feasibility

To make sure that your solution may be integrated into the IBM Tivoli Intelligent ThinkDynamic Orchestrator and IBM Tivoli Provisioning Manager environments, it is vital that you ensure that all required functions that must be performed against your device(s) can be executed through a command-line interface, and that a minimum of operator responses are required (preferably none).

If parts of the provisioning functions of your device(s) require interaction through a GUI, for example, a Web-based interface, they cannot be implemented through the TIO/TPM functions, and thus, they cannot be integrated into this environment.

If this is the case, consider developing SOAP, SNMP, TELNET, SSH, serial, or plain command line interfaces in addition to the GUI functions to allow for scripted automation. You should remember that IBM Tivoli Intelligent ThinkDynamic Orchestrator supports EXPECT scripts, which allow for programmable operator interaction with interfaces that are designed to use an interactive request-reply type of dialog like, for example, the telnet protocol.

3.2.6 Determine device types to be supported

Once it has been determined which hardware and software components to provide automated provisioning functions for, and which functions to provide, the software architect needs to relate the components provided with the solution to the device-types in the IBM Tivoli Intelligent ThinkDynamic Orchestrator Data Center Model (DCM). This step is critical to determining which devices and logical operations have to be implemented as part of the Orchestration Solution Package.

The number of components provided may vary tremendously from one solution to the other, depending on the complexity of the solution and whether or not installation and configuration of prerequisites are included. If, for example, an Automation Package is being architected for a router solution, both ACL and firewall device(s) may have to be manipulated/supported in addition to the router device, if the router solution is software based, Software products and software stacks may have to be defined along with any related software and hardware devices that have support included as part of the actual automation solution.

3.2.7 Define high-level operations

Once the devices that will be supported by the Orchestration Solution Package have been determined, it should be decided which high-level operations to implement. In the solution specific Automation Package, all high-level operations will be implemented by workflows.

To provide a complete set of automation functions for a solution, we advise that all logical operations for the particular device type be supported. In an actual implementation scenario, chances are that all of the logical operations for a device will be referenced from other high-level logical operations, which are controlled by the implementor. By high-level logical operations, we refer to logical operations that operate on TIO/TPM device types, such as Applications, Tiers, Pools, Servers, and Software Stacks.

Note: The software architect should remember that although it is possible to define new custom logical operations in IBM Tivoli Intelligent Orchestrator V3, we do not recommend doing so because defining custom logical operations involves modifying TIO/TPM's critical Java classes.

For example, the template workflow Cluster.AddServer.Template, which implements the Cluster.AddServer logical operation, invokes the SoftwareStack.Install logical operation as one of the transitions for the stack defined for the tier being operated on. The SoftwareStack.Install logical operation installs a predefined software stack on the server that is in transition to the tier. A customer could use the Cluster.AddServer.Template workflow as a model for implementing its own process for adding a server to an application tier and modify it to execute a workflow provided by an ISV for installing the ISV's software solution on the server. The workflow provided by the ISV will most likely implement the logical operations SoftwareStack.Install, Software.Install, or both, depending on the solution. As a consequence, the Software.Install logical operation should be supported by the ISV Orchestration Solution Package if the solution includes software, instead of defining a completely new logical operation. A similar reasoning could have been made for the Switch.Move Port To VLAN logical operation, if the ISV solution involves hardware.

In addition to the operations mapping the logical operations, other high-level operations may be defined as required. These may be required to support particular device functions or special needs related to installation, customization, or operation.

3.2.8 Define low-level operations

The software architect should now carefully break down the functionality of each high-level operation into functional group related tasks, which are used repeatedly throughout all the high-level operations, in order to define low-level operations implementing these sub-functions.

The definition of low-level operations helps avoid implementation of the same set of tasks in multiple operations, thus ensuring that both development, testing, implementation, and customization is made easier.

As an example, we can look at interacting with a switch. Whenever we need to interact with the switch to perform operations such as moving a port to a VLAN or deactivating a port, it is likely that the same protocol and credentials are used. Therefore, it makes sense to identify one and only one generic, reusable low-level operation that interacts with the switch.

3.2.9 Break down operations into workflows

When it has been decided which high- and low-level operations to provide in the Automation Package, workflows that implement these operations have to be defined. This involves further breaking down of each operation into a series of reusable building-blocks (workflows) that, when executed, perform the desired function. Variables are used to pass information from workflow to workflow.

During this process, the software architect has to remember that input and output parameters for the high-level workflows that implement logical operations have to comply with the specifications of the logical operations they implement. In addition, all high-level workflows should be designed as clonable templates to be customized by the organization implementing the Automation Package, and that they should have a modular structure in order to allow the implementing organization to modify them and model their own processes.

When breaking down the operations, the main considerations should be:

- Reuse
- Be easy to customize
- Openness
- Modularity
- Exit points (to be used in actual implementations)

As an example of this breakdown, let us once again turn to the switch example. The operation for interacting with the switch itself could be implemented as an independent workflow, and by calling this workflow whenever we need to interact with the switch, we have increased both reuseability and ease of customization.

3.2.10 Define external prerequisites, dependencies, and interfaces

Now that the workflows that need to be implemented have been defined, the software architect has to make sure that the prerequisites for the successful execution of each workflow will been established. This involves identifying prerequisites and dependencies for each workflow, and identifying and designing interfaces to components external to the workflows.

Since the Automation Package basically is a collection of little bits and pieces that will work in conjunction with customer specified objects, the final

implementation of many of the prerequisites and dependencies is not the responsibility of the Orchestration Solution Package provider. However, it is the provider's responsibility to document the requirements and dependencies needed to automate the provisioning of a particular solution, and to provide samples and templates to help the implementor in establishing the needed objects. The provider should also consider creating installation verification procedures to ensure that all the prerequisites and dependencies are met.

In this context, the following definitions for the terms are applied:

Prerequisite Existence or proper configuration of an internal or external object that is required for successful execution of a workflow.

Dependency Existence or proper configuration of an internal or external object that is required to successfully operate the provisioned solution.

Interfaces Protocols for ensuring interaction between internal and external objects, such as workflows, solution devices, or external files. In this context, the use of variables to pass information between workflows is considered an interface.

When looking at dependencies, prerequisites, and interfaces, the following should be considered:

- Platform affinity
- External file existence, such as templates, scripts, and interpreters
- Global and local DCM properties
- SAPs (protocols and credentials)
- Software Stacks, Software Products, and Software Product parameters
- Routes, VLANs, ACLs, and other networking definitions
- Proxy servers, such as Blade, Boot, and Terminal Servers
- Solutions external to IBM Tivoli Intelligent ThinkDynamic Orchestrator that support the data center operations, for example, IBM Tivoli Configuration Manager, Citrix and VMWare servers

One example of a prerequisite is the existence of a Software Product in the data center model with a pre-determined variable and parameter configuration. If the workflow that implements the Software.Install logical operation requires a specific parameter to exist, this may be considered a prerequisite for the workflow, since the workflow is likely to fail if the parameter has not been specified or configured correctly.

Regarding dependencies, let us turn to the Trade3 application example. The application uses a database, and even though it may not be considered part of the Software Stack that is installed for the Trade3 tier, the application depends on

a properly configured DB2 client and a catalog entry pointing to the database to be used. If this dependency is not in place, the application will not work correctly, even though it may install successfully.

One obvious example of an interface is the interaction with a switch. In order to successfully send and execute a switch command, the protocol to be used, along with any required credentials needed to authenticate with the switch, has to be clearly identified, and defined, most likely in a SAP.

3.2.11 Define location and naming conventions for external files

In case the automation procedures for a specific solution requires use of external files such as templates, software images, scripts and the like, a standard for the naming and placement of these files must be established.

We advise allowing the implementor of the Automation Package to specify the top-level (parent) path of the location tree in order to allow for local customization. In addition, the solution provider should try to avoid building in firm requirements regarding file locations. For example, a customer may want to place all response files used for application installation on a dedicated server, and perhaps even execute response file generation tasks on this server, in order to off load the TIO/TPM system. The design of the Automation Package and placement of files should allow for this by not requiring specific local paths and files to exist on the TIO/TPM server, and by providing customization to allow for specifying the system on which to store the files.

3.2.12 Identify transitions

Now that the functionality of each workflow, the interaction between workflows, and the use of internal (DCM objects) and external resources have been determined, it is time to design the inner workings of the workflows. In other words, determine which transitions should be executed in which sequence in order to establish the required functionality.

When designing the internals of the workflows, the emphasis should be on reusing existing workflows and transitions. During the workflow design process, we highly advise you to seek inspiration by looking at existing workflows implementing operations similar to the ones you are designing.

Even though the highest possible number of transitions (workflows, Java scripts, and Java plug-ins) are being reused and cloned, it is likely that new transitions will have to be developed in order to interact with new devices, especially when developing workflows for interacting with hardware based solutions, such as switches and routers.

As an example, let us look at the outline of a reusable workflow to execute commands on a switch. The series of transactions could be something like the following:

1. Get information regarding the physical device.
2. Fetch protocol and credential information for the execute operation.
3. Build a command to be executed.
4. Execute the command.
5. Validate result.

The first three steps can most likely be done using existing transitions, perhaps in combination with a script that builds the command to be sent to the switch. Depending on the capabilities of the switch's interface, step 4 may have to be implemented as a Java plug-in that is developed specifically for this device (or group of similar devices from the same vendor). Step 5 may or may not require special processing through a script, depending on the actual circumstances.

3.2.13 Create specifications for new code to be developed

As the example in the previous section shows, specialized code may have to be developed in order to provide the necessary functions to automate the provisioning of a solution. This may be the case for both purely software based and hardware based solutions.

Since the workflow implementation in IBM Tivoli Intelligent Orchestrator V3 does *not* support branching (if.. then.. else), it is likely that special executables have to be developed in order to dynamically set up run time parameters or provide interfaces to external systems.

Based on the functional breakdown of the required automation operations, it is the responsibility of the software architect to create coding specifications for the development of these executables. Whether to use Java plug-ins, shell scripts, or other types of executables is entirely up to the software architect and the current implementation, but to ease troubleshooting and maintenance, we recommend that scripts are used as much as possible, and the use of compiled code or Java plug-ins are limited to a minimum. However, the Java platform provides some unique functions that may make interacting with solutions and devices very easy, in which case they should be utilized.

3.2.14 Develop recovery policies

Depending on the complexity of the automation procedures for a particular solution, and the number of dependencies and prerequisites, you might need to develop recovery workflows that may be invoked in case a transition within a workflow fails.

Naturally, recovery policies should be customizable to allow implementors to apply their own processes, but solution specific recovery actions should be implemented in order to ensure that the devices are not left in non-operational states in case of workflow malfunction.

3.2.15 Design templates for DCM object and related files

Now that the workflows implementing all the automation processes required by the solution have been designed, templates for the required and prerequisite objects in the data center model can be developed. This includes assigning device drivers or logical operation workflows to the devices to ensure that the correct workflows are invoked when a logical operation is performed against the object(s).

3.2.16 Design implementation procedures

When all the code (workflows, Java programs, and Java plug-ins) and sample DCM objects have been created, it must be packed into the automation package file to allow for easy distribution. The automation package file is a compressed JAR file with a .tcdriver extension and is also referred to as a tcdriver file. More details about the packing process are provided in Chapter 8, “Automation package content and packaging” on page 233.

The packaging process is basically designed to handle distribution and installation of the automation functions provided in the Automation Package. It is not designed to include code images and sample DCM files, even though these types of data may be included and referenced from a post-installation workflow. This function may be used to import sample DCM definitions and code images as part of the Automation Package installation in order to allow for easy installation by the implementor.

This post-installation workflow should be used in conjunction with manual procedures instructing the implementor on what to define manually, how to do it, and which parameters and variables to customize. If it is chosen to make use of the post-installation-workflow, this naturally has to be defined, coded, tested, and included in the tcdriver file.

3.2.17 Ensure proper documentation

Whether or not you chose to use the post-installation-workflow or not, instructions on how to define DCM objects to support the new automation functions, perform customization, and use the functions has to be provided. Refer to Chapter 8, “Automation package content and packaging” on page 233 for details on the automation package documentation files.

3.2.18 Architecting best practices

Throughout the architecting process, the software architect has to keep in mind that the Orchestration Solution Package provides bits and pieces that need to be integrated with an existing customer environment. Chances are that the solution provided by the ISV is a part of a bigger whole, and thus the ISV must provide the highest possible level of flexibility to the final implementor (the customer) of the Orchestration Solution Package, in order to allow the customer to use the ISV provided devices and Automation Package(s) to support local processes and procedures.

To allow for this solution, a few simple rules should be followed:

- Ensure that all required functions that must be performed against your device(s) can be executed through a command line interface, and that a minimum of operator responses are required (preferably none).
- Implement all logical operations for the devices your solution supports/uses.
- Reference logical operations where possible.
- Reference and use standard workflows as much as possible.
- Wrap calls to Java code (plug-ins and programs) in workflows.
- Create device specific workflows.
- Allow for customization.
- Use small reusable building blocks.
- Do not make assumptions about customer policies.

Workflow executables

When designing the executables for an Automation Package, you should consider providing a three-tier architecture, as in the following:

- Workflows implementing logical operations
  - Workflows implementing specific device operations
    - Workflows interacting with the device
    - Java plug-ins interacting with a device

This architecture will allow for sufficient modularity to provide easy customization and reuse.

Use scripts as much as possible, and Java plug-ins when it makes good sense.

Archived

3.3 Practical aspects of designing automation packages and workflows

This section discusses some of the practical aspects of designing workflows and automation packages and should give you information and guidelines about the following:

- Introducing the environment
- Building the development environment
- Design considerations for automation packages and workflows
- Reuseability of workflow transitions
- Authentication and authorization internals
- Naming conventions
- Documentation guidelines

3.3.1 Introducing the environment

In this section, we discuss the prerequisites to be implemented for building an architecture for workflows and their components. This gives the basic guidelines for developing the automation packages and their containing components (that is, workflows, logical operations, Java plug-ins, and so on).

The first step in developing workflows and automation packages is to have the target ISV solution implemented. The target solution can be hardware based or software based. Examples are software applications, storage devices, servers, back-end services, network devices and appliances, and so on.

Put into IBM Tivoli Intelligent Orchestrator terms, the installation of the target solution involves the creation and configuration of the following major components, as described in Table 3-1.

Table 3-1 Components of the application environment

Component	Description
Network Management	The Network Management, also referred as a Network Pool, contains all the active components of the network environment, that is, Router, Switch Fabrics, Load balancers, and so on.
Application Pool	This component of the application environment contains the Servers for operating the first tier of the application, for example, an application running on top of a Web application server.

Archived

Component	Description
Resource Pool for Application	This pool is the repository for available servers to be provisioned to the application pool. The goal is to maintain resource utilization and performance of that pool at acceptable service levels.
Back-end Pool	This part of the application environment contains the back-end servers for the application, for example, database servers, host applications, and so on.
Resource Pool for Back-end	This pool is the repository for available servers to be provisioned to the back-end pool. The goal is to maintain resource utilization and performance of that pool at acceptable service levels.

A working IBM Tivoli Intelligent Orchestrator environment is also required. This involves installing and configuring the following components at a high level:

- Shell execution environment (Cygwin on Windows, for example).
- SSH communications between the TIO/TPM servers and the managed environment.
- IBM DB2 UDB.
- IBM Directory Server (LDAP).
- IBM WebSphere Application Server and Client and proper Fix Pack levels.
- IBM Tivoli Intelligent Orchestrator.
- A data center model should be defined with all the objects that will take part of the development effort.

Note: Please remember that the environment of IBM Tivoli Intelligent Orchestrator has to be installed on a minimum of two servers operating in a homogeneous operating environment. For more detailed information, please refer to *Provisioning On Demand: Introducing IBM Tivoli Intelligent OrchestratorProvisioning On Demand Introducing IBM Tivoli Intelligent ThinkDynamic Orchestrator*, SG24-8888.

The environment in which IBM Tivoli Intelligent Orchestrator manages the components described on Table 3-1 on page 64 is referred to in this section as the Management Environment.

Note: In the following figures, we show the management environment by showing IBM Tivoli Intelligent Orchestrator installed on a single box symbolically, in order to reduce the complexity of these pictures.

Figure 3-4 shows the logical view of an application tier managed by IBM Tivoli Intelligent Orchestrator.

Figure 3-4 Logical View of an application tier managed by TIO/TPM

The *Provision* arrows shown in Figure 3-4 illustrate the summary of the steps and required processes for a provisioning operation.

Based on the complete installation of all necessary components of the application and managing environment, the logical data center model should be defined using the Inventory tab (shown in Figure 3-5) of IBM Tivoli Intelligent Orchestrator.

Figure 3-5 Inventory tab

For instructions on how to define the data center model, refer to *Provisioning On Demand: Introducing IBM Tivoli Intelligent OrchestratorProvisioning On Demand Introducing IBM Tivoli Intelligent ThinkDynamic Orchestrator*, SG24-8888.

In Figure 3-6, you will see how IBM Tivoli Intelligent Orchestrator provides an overview of an existing data center as an example of an properly defined data center model.

Figure 3-6 Overview of an existing data center

A major step to establishing the data center model is to define the network environment properly. This definition contains the following points:

- Active components of network, including:
  - Servers including NICs
  - Switch Fabric and Routers
  - Load Balancer
  - Firewall
  - Other network devices
- TCP/IP configuration including:
  - Real IP address (RIP)
  - Virtual IP address (VIP)
  - Subnets
  - Virtual LAN (VLAN) configuration
- Interaction between network components

Archived

Figure 3-7 shows the physical view of an application environment to be managed by IBM Tivoli Intelligent Orchestrator.

Figure 3-7 Physical View of an application tier managed by TIO/TPM

The *Provision* arrows shown in Figure 3-7 illustrate the summary of the steps and required processes for a provisioning operation.

A list of the processes for provisioning an additional server to an application tier could look like the following:

1. Allocate bare metal server.
2. Install and configure the operating system on a server.
3. Install and configure the software package, including the application on server.
4. Add the server to the application tier.

The necessary processes and procedures for manual installation and removal of an application have to be already in place. Also, they should be properly tested and documented for further automation package development.

3.3.2 Building the development environment

In the previous section, we have described the environment that has to be in place for the development of workflows and automation packages.

In order to develop automation packages and workflows, we advise that the ISV build a development environment similar to the production environment of the customer. The minimum components of a development environment for building workflows and automation packages are:

- IBM Tivoli Intelligent Orchestrator environment
- A successfully defined and loaded Data Center Model close to the customer's environment
- Available managed systems
- An additional desktop PC with APDE and a Java SDK installed for developing workflows and packaging automation packages. This environment may also be used to develop external Java programs that may be necessary for a particular solution.

Note: Please note that the Orchestration Solution Package provided by ISVs should take the customer's processes and guidelines into consideration whenever possible.

3.3.3 Design considerations for automation packages and workflows

In this section, we will cover the design considerations needed for automation packages and their workflows and workflow components. This section contains:

- Considerations about existing processes
- Design preparation
- Enhancement of design
- Structure and modularity of the design
- Determine reusable workflows

Considerations about existing processes

Whenever possible, identifying the current processes defined by the customer regarding systems and Security Management is a key area in which the ISV also must translate what the customer currently does in its business operations into a management function within the IBM Tivoli Intelligent Orchestrator application. The ISV has to determine the processes of the customer's business organization and to gather the existing structure in whatever form that may be (ISO9000, ISO9001, IT Infrastructure Library® (ITIL®), or home-grown by the customer) to use as a basis for implementing the workflows in IBM Tivoli Intelligent Orchestrator.

Archived

In summary, the customer's processes and environment could be:

- Customer' guidelines and implemented processes about:
  - Systems Management
  - Security Management
- The application to be managed
- The IT environment to be managed (hardware, operating systems, network protocols, and so on)

As a workflow is simply an automation of an existing manual process, this manual process has to be already in place and properly tested. While this may seem obvious, it underscores the fact that without a manual process, there can be no automated process. And, if the manual process is poorly designed, then the automated process is likely to redo the flaws in the manual process.

In practical terms, when developing a workflow, we are trying to model something that already exists in the real world into the IBM Tivoli Intelligent Orchestrator. The elegance, or lack thereof, of the manual process and the fidelity with which we are able to model it within IBM Tivoli Intelligent Orchestrator will determine our success in implementing orchestration and provisioning.

Design preparation

The major part in doing the design preparation for a workflow is considering the existing processes and how they can be implemented in using workflows. In our example, we use a conceptual workflow for the reconfiguration of an existing process. We assume that the manual processes for these actions already exist.

The conceptional workflow contains the subprocesses listed below:

- Initiate process
- Configuration of process
- Re-initiate process

We used the same approach when developing workflows for our case study scenario chapter. These workflows are presented in Chapter 10, "Case study: Trade3 composite application" on page 281.

Figure 3-8 on page 71 shows the flowchart of this first step of the workflow design.

Figure 3-8 Flowchart for step 1 of designing a workflow

A description about each process of this workflows follows:

Start of workflow This label marks the beginning of this workflow.

Initiate process The process to be configured must be in the initiated state before the configuration work can be changed.

Configuration of process The configuration or reconfiguration of the running process.

Re-initiate process The configured process has to be re-initiated and, as a result of that, in the initiated state before the configuration work can take place.

End of workflow This label marks the end of this workflow.

We will use this simple workflow for showing how to enhance its design in the section below.

Enhancement of design

Normally, before initiation or re-initiation of a process, some values, also known as parameters, attributes, or variables, of the operational environment have to be changed, for example, due to Systems Management or operational issues.

Archived

In this case, we have to distinguish the processes for initiation and re-initiation in a more detailed way. This is shown in Table 3-2.

Table 3-2 Distinguishing the processes

Generic process	Distinguished process
Initiate process	Change value Initiate process
Re-initiate process	Change value Re-initiate process

Figure 3-9 shows the flowchart of the second step of the workflow design.

Figure 3-9 Flowchart for step 2 of designing a workflow

A description of each process or label of this workflow follows:

Start of workflow This label marks the beginning of this workflow.

Value change This process changes the values of the object's parameters, attributes, or variables of the operational environment.

Initiate process The process to be configured must be in the initiated state before the configuration work can be changed.

Archived

Configuration of process
Configuration or reconfiguration of the running process.

Value change This process changes values of an object's parameters, attributes, or variables of the operational environment.

Re-initiate process The configured process has to be re-initiated and, as result of that, in the initiated state before the configuration work can take place.

End of workflow This label marks the end of this workflow.

We will use this enhanced workflow for building a structured and modular design in the section below.

Structure and modularity of the design

The objective here is to develop the workflows shown in Figure 3-9 on page 72 into a more structured and modular workflow.

For example, the workflow step named Re-initiate process can be split into two independent processes. Table 3-3 shows the result.

Table 3-3 Modularized the process of re-initiation

Generic process	Distinguished process
Re-initiate process	Terminate process Initiate process

However, the second process *Initiate process* is the same as the process in the beginning of the workflow.

Archived

Figure 3-10 shows the flowchart of the third step of the workflow design.

Figure 3-10 Flowchart for step 3 of designing a workflow

A description about each process or label of this workflow follows:

Start of workflow This label marks the beginning of this workflow.

Value change This process changes the values of the object's parameters, attributes, or variables in the operational environment.

Initiate process The process initiates the process or engine within this conceptual workflow and changes its state to started.

Configuration of process
Configuration or Reconfiguration of the running process.

Value change This process changes the values of an object's parameters, attributes, or variables in the operational environment.

Terminate process The process terminates the process or engine within this conceptual workflow and changes its state to stopped.

Archived

Initiate process The process initiates the process or engine within this conceptual workflow and changes its state to started.

End of workflow This label marks the end of this workflow.

Determine reusable workflows

The final step of designing the architecture of a workflow as part of an automation package is to ascertain that already existing and clonable workflows are delivering the functionality they were delivering before.

Sources for these kind of reusable workflows could be:

- Shipped with IBM Tivoli Intelligent Orchestrator
- Workflows and workflow components developed by customers and ISVs
- IBM Open Process Automation Library (OPAL)

For more detailed information about the IBM Open Process Automation Library (OPAL) and its functionality, please refer to:

http://www.developer.ibm.com/tivoli/workflow.html

3.3.4 Reuseability of workflow transitions

As described in 3.2.12, “Identify transitions” on page 60, the emphasis should be on reusing existing workflows and transitions as well as developing workflows and transitions that can be reused.

Let us use the TMA Assign Policy Region workflow as an example. The TMA Assign Policy Region workflow is shipped with IBM Tivoli Intelligent Orchestrator.

Example 3-1 shows the workflow TMA Assign Policy Region.

Example 3-1 Workflow TMA Assign Policy Region

```
# ------------------------------------------------------------------
# Licensed Materials - Property of IBM
# 5724-F75
# (C) Copyright IBM Corp. 2003 - 2005
# All Rights Reserved
# US Government Users Restricted Rights -Use, duplication or
# disclosure restricted by GSA ADP Schedule Contract with IBM Corp.
#
# Assign an endpoint to a Policy Region
workflow TMA_Assign_Policy_Region(in DeviceID, in Policy_Region, in
TMA_Label, in WorkingDirectory, in WorkflowExecutionID, in subdir)
LocaleInsensitive
```

```
# 1.0 Create the command to invoke
  var TIO_Server_ID
  var FileName_default = "tmasetpr.sh"
  var CommandString_default = Jython(WorkingDirectory + "/" +
(FileName_default or ""\) + " " + (TMA_Label or ""\) + " " +
(Policy_Region or ""\))
  log debug Jython("CommandString_default = " + CommandString_default)
  var TimeoutInSeconds_default = "300"

# 2.0 Copy the script from the TIO server to the TMR server
  array All_Script_Names = { FileName_default }
  TMA_Copy_Scripts_To_Temp_Dir(All_Script_Names, subdir, DeviceID,
WorkingDirectory, TIO_Server_ID, WorkflowExecutionID)

# 3.0 Invoke the command on the TMR server
  log info Jython("Device.Execute Command")
  Device.ExecuteCommand(DeviceID, CommandString_default,
WorkingDirectory, "default", TimeoutInSeconds_default, "error", <null>,
<null>, <null>)
```

The workflow uses three transitions provided by IBM Tivoli Intelligent Orchestrator to assign an Tivoli endpoint as part of a Tivoli Management Region to an existing Policy Region (see Table 3-4).

Table 3-4 Transitions of Workflow TMA_Assign_Policy_Region

Transition	Description
TMA_Copy_Scripts_To_Temp_Dir	This workflow copies the necessary shell scripts (defined in the array named All_ScriptNames) to the target endpoint.
Device.Execute Command	This logical operation executes the shell script *tmasetpr.sh* on the target endpoint.

IBM Tivoli Intelligent Orchestrator provides workflows templates, logical operations, Java plug-ins, and Java programs that can be reused, and most likely will be, in any workflow development.

3.3.5 Authentication and authorization internals

This section discusses how IBM Tivoli Intelligent Orchestrator interacts with the managed objects in term of authentication and authorization. The following issues will be discussed:

- Service Access Points

- Storing passwords and how to use inside workflows

Identifying the current processes regarding Security Management is a key area in which the ISV also must translate what the customer currently does in its business operations into a management function within the IBM Tivoli Intelligent Orchestrator application. The ISV has to determine the security area of the customer's business organization and to gather the existing structure in whatever form that may be (ISO9000, ISO9001, IT Infrastructure Library (ITIL), or home-grown by the customer) to use as a basis for implementing the security processes in IBM Tivoli Intelligent Orchestrator.

Managing hardware objects

IBM Tivoli Intelligent Orchestrator uses a combination of protocols and credentials called service access points (SAPs). SAPs identify which network protocols you will use when communicating with devices on your network. Implementing SAPs means defining your protocol method and authorization control for each device, server, and tier defined on IBM Tivoli Intelligent Orchestrator.

Archived

For example, Figure 3-11 shows how to add a new Service Access Point to a server.

Figure 3-11 Adding a new Service Access Point to a server

A number of protocols are available from IBM Tivoli Intelligent Orchestrator to be used when accessing services on managed objects, such as servers, tiers, and network devices. These kinds of protocols are known as Fiber-Channel, IPX™, IPv4, IPv6, and so on.

For the IPv4 protocol type, the most common subtypes are:

FTP File Transfer Protocol (GET/PUT)

Telnet Insecure Shell (for example, ksh)

ICMP Internet Control Message Protocol (PING)

SCP Secure Copy (GET/PUT)

SSH Secure Shell

SNMP Simple Network Management Protocol (GET/SET)

Telnet/FTP

If the type of protocols to be used for accessing the service are Telnet or FTP, password credentials for the Service Access Point may be required.

Figure 3-12 shows how to add new password credentials for accessing the Telnet service. This also similar for the FTP service.

Figure 3-12 Password credentials for Telnet and FTP protocols

SSH/SCP

If the type of protocols to be used for accessing the service are SSH or SCP, RSA credentials for the Service Access Point are normally required; however, password credentials can also be used for the SSH and SCP protocols.

Archived

Figure 3-13 shows how to add new RSA credentials for accessing the SSH service. This also similar for the SCP service.

Figure 3-13 RSA credentials for SSH and SCP protocols

SNMP

If the type of protocol to be used for accessing the service is SNMP, SNMP credentials for the Service Access Point are required for snmp-set and snmp-get operations. Password credentials can also be used for the SNMP protocol.

Figure 3-14 on page 81 shows how to add SNMP credentials for accessing the service SNMP.

Archived

Figure 3-14 SNMP credentials

For more detailed information about the use of Service Access points and their credentials, please refer to the *IBM Tivoli Intelligent Orchestrator Operator's Guide*, SC31-1421.

Using SAPs for security purpose

As explained above, IBM Tivoli Intelligent Orchestrator uses protocol and credentials called Service Access Points (SAPs). Service Access Points identify which network protocol will be used when communicating with objects in the managed network environment. Implementing Service Access Points means defining the protocol method and authorization control to be used for each device, server, and tier defined on IBM Tivoli Intelligent Orchestrator.

The ISV has to determine how the developed workflows actions will authenticate to each device defined in the Data Center Model (DCM). At the end, the customer is responsible for the definition and implementation of this communication using his own guidelines and protocols.

Readable passwords, as part of the password credentials for the protocols Telnet/FTP, SSH/SCP, and SNMP, are not acceptable in environments having a proper Security Management. In addition, the customer would not agree to use readable passwords for the operators of IBM Tivoli Intelligent Orchestrator and perhaps for the "man in the middle", especially when using Telnet and FTP.

We recommend implementing the Service Access Points using SSH and SCP with RSA credentials for managing devices, servers, and tiers within the Data Center Model of IBM Tivoli Intelligent Orchestrator.

Storing run time credentials in SAPs

Besides providing a permanent store for credentials used when communicating with devices, SAPs may also be used to store credentials to be used when authenticating with subsystems (for example, database or application servers) during the provisioning.

By providing your own application specific combination of the context, search key, and domain - and specifying a protocol of *unknown* - you can store encrypted user ID and password information that can be retrieved from your workflows and, optionally, can passed on to scriptlets that interact with the subsystems.

The credentials sorted can be of any type (password or RSA) depending on how they are intended to be used in the specific application context.

For example, to configure the WebSphere Application Server on which IBM Tivoli Intelligent Orchestrator is running, you have to authenticate with the WebSphere environment using the default wasadmin/wasadmin user name and password. These credentials can be stored in a SAP defined for the TIOServer system, and used by workflows that need to execute the **`wsadmin`** command. Figure 3-15 shows an example of a SAP definition for storing application specific password credentials for accessing the TIO/TPM WebSphere Application Server using a domain name of Management, a context of TIO, and user ID of wasadmin.

Figure 3-15 SAP for application specific credentials

Managing software objects

When manually installing an application on a server, many of the steps need the authentication information (user ID and the password) of the account that the application uses to run or install. For example, the user ID and password of an

Administrator account are required for implementing a new application to run a service on MS Windows 2000, or the credentials of a database instance owner to create or change database settings.

In Chapter 10, “Case study: Trade3 composite application” on page 281, when installing an application running on top of the IBM WebSphere Application Server, we had to specify the user ID and password of the application’s database instance owner to catalog the database server and create an JDBC™ connection to the database used by the application. This is because the Trade 3 application installation method is interactive and prompts for several parameters, including the credentials of the database instance owner.

This scenario is a typical when the application installation method has not been planned with the “automated provisioning” mind set. Solutions to this problem could be hardcoding the credentials into scripts, or response files, and that, of course, would not be accepted by customers with high-level security standards. With that in mind, the software architect could opt, for example, to do the following:

- Use an application specific SAP definition to store subsystem credentials. Since SAPs cannot be associated with software components, the SAP should be defined on either the application tier or individual server level.

  Note: This is one way to store the information for the user ID and password in an encrypted way.

- Define the user name and password as parameters of the Software Resource Template definition in TIO/TPM.
- Define the user name and password as local variables of the software product.
- Define the user name and password as local variables of the workflow that implements the operation.
- Change the installation method of the application, if possible.

Archived

Credentials defined as parameters in a Software Resource Template

Figure 3-16 shows how to define a user ID and password as a parameter of the software package defined for the application Trade3.

The parameters defined for authentication are:

- DB2Username
- DB2Password

Figure 3-16 User ID and password defined in a Software Resource Template

Credentials defined as software module variables

Figure 3-17 shows the user ID and password defined as variables of the software product definition for the Trade Web Application.

The variables defined in the Deployment Engine scope are:

- DB2Username
- DB2Password

Figure 3-17 User ID and password defined as software product variables

Credentials hardcoded in a workflow

In case the credentials used to access, for example, a database are guaranteed not to change (which is not very likely), they can be hardcoded into a workflow using the syntax shown in Example 3-2 on page 85.

Example 3-2 Hardcoding database access credentials

```
var DB2Username=”tradeUser”
var DB2Password=”smartway”
```

Credentials defined as part of the installation package

An alternate way to store the user ID and password needed for the installation of an application is to code it directly into the installation code or configuration files, if possible. In this case, the user ID and password are not stored in the repository of IBM Tivoli Intelligent Orchestrator.

We will show how this has been accomplished for the Trade3 application, as an example. Without any changes, the manual installation process for Trade3 prompts for the following parameters:

- Type of database
- Path of database driver
- Name of database owner
- Password of database owner

Note: The information required by the installation process is shown in the following examples (Example 3-3 and Example 3-4) using emphasized characters.

Example 3-3 shows portion of the original installation script Trade3.xml without any modifications.

Example 3-3 Installation script Trade3.xml

```
< ... >

<target name="installResources" depends="init" description="Install the
trade3 JDBC and JMS resources via wsadmin">
   <echo> " Installing Trade3 JDBC/JMS resources on WAS Node ${node} "
</echo>
   <input
        message="Please enter the database type [db2 or oracle]:"
          property="dbtype"
   />
   <input
          message="Please enter the database username:"
          property="dbuser"
   />
   <input
          message="Please enter the database password:"
          property="dbpass"
```

```
   />
   <input
         message="Please enter the Oracle database hostname:"
         property="dboraclehost"
         onlyIfDbtypeEquals="oracle"
   />
   <input
         message="Please enter the Oracle database SID:"
         property="dboraclesid"
         onlyIfDbtypeEquals="oracle"
   />
   <input
         message="Please enter the fullpath to the JDBC driver
db2java.zip or classes12.zip (e.g. c:/sqllib/java/db2java.zip)"
         property="basepath"
   />
    <path id="base.path" path="${basepath}"/>
   <pathconvert dirsep="/" property="dbdriverpath" refid="base.path"/>
   <echo> "Database Classpath path converted ${dbdriverpath}" </echo>

   <echo> "Creating TradeDataSource JDBC Datasource"</echo>
   <wsadmin
         script="createTrade3RDBMSResources.jacl">
            <arg value="${dbtype}"/>
            <arg value="${node}"/>
            <arg value="${dbdriverpath}"/>
            <arg value="${dbuser}"/>
            <arg value="${dbpass}"/>
            <arg value="${dboraclehost}"/>
            <arg value="${dboraclesid}"/>
   </wsadmin>

   <echo> "Creating Trade JMS Queues and Topics"</echo>
   <wsadmin
         script="createTrade3JMSResources.jacl">
            <arg value="${node}"/>
   </wsadmin>
  </target>

< ... >
```

Example 3-4 on page 87 shows a portion of the installation script Trade3Provision.xml of the Trade3 application after changes have been made for use with IBM Tivoli Intelligent Orchestrator.

Example 3-4 Installation script Trade3.xml changed for use with TIO/TPM

```
< ... >

<target name="installResources" depends="init" description="Install the
trade3 JDBC and JMS resources via wsadmin">
   <echo> " Installing Trade3 JDBC/JMS resources on WAS Node ${node} "
</echo>


   <echo> "Creating TradeDataSource JDBC Datasource"</echo>
   <wsadmin
         script="createTrade3RDBMSResources.jacl">
            <arg value="db2"/>
            <arg value="${node}"/>
            <arg value="C:/IBM/SQLLIB/java/db2java.zip"/>
            <arg value="tradeUser"/>
            <arg value="smartway"/>
            <arg value="${dboraclehost}"/>
            <arg value="${dboraclesid}"/>
   </wsadmin>

   <echo> "Creating Trade JMS Queues and Topics"</echo>
   <wsadmin
         script="createTrade3JMSResources.jacl">
            <arg value="${node}"/>
   </wsadmin>
  </target>

< ... >
```

Summary

- Storing a user ID and password of an user needed for the provisioning of an application as a parameter within the software package definition means storing these values in a database of the IBM Tivoli Intelligent Orchestrator environment, and that they are readable for every user that authenticates against the TIO/TPM graphical interface.
- Using SAPs to store encrypted credentials provides an object specific (Application or tier) way of providing different credentials to different implementations of the same Software Product.

Archived

- Encrypted coding of a user ID and password as a local variable is only available within workflows. This will store these values encrypted only in the repository of IBM Tivoli Intelligent Orchestrator, but it will also be readable throughout the TIO/TPM graphical interface.
- The software architect and developer should consider changing credential requirements for the solution to be provided and take security measures as appropriate (that is, hardcode users' credentials and encrypted passwords) within the product code whenever possible. This will ensure a smooth transition from the manual implementation process of a solution to an automated process performed by TIO/TPM.

3.3.6 Naming conventions

A major point in designing naming conventions to be used with IBM Tivoli Intelligent Orchestrator is to clearly identify a workflow, or a plug-in and its relationships, to the other components of the provisioning environment. This section covers the naming convention from a structured view to propose a naming convention that will allows a comprehensive affiliation of automation packages, workflows, and workflow components.

> **Note:** The convention presented in this section is not the only way for setting a naming convention. They are meant to be used as a start point for developing new workflows, workflows components, and automation packages. Customers would apply their own naming policies when using your workflows in their TIO/TPM environment.

This section discusses the use of a naming convention for:

- Workflows and workflow components
  - Logical operations
  - Workflows
- Automation packages

Where possible, terminology should be used that is commonly applied in the specific domain, and names should be kept reasonably short. Company naming principles could also be applied, but as with naming conventions in general, the main objective is to implement consistent terminology and syntax so that the functions of the transition are clear to people working with them.

Archived

Naming convention for logical operations

A naming convention for logical operations must follow the following syntax:

```
{DeviceTypeName}.{Operation}
```

Where:

{DeviceTypeName}	Name of the device type
{Operation}	Name of the operation implemented by the logical operation

As discussed earlier in this chapter, logical operations provide a layer of abstraction from the physical operations defined by specific device drivers or workflows.

Some examples are:

- Cluster.AddServer
- Device.Execute Command
- BootServer.Install Image

Naming convention for workflows

IBM Tivoli Intelligent Orchestrator provides a naming field 256 characters long for naming workflows. The only rules that apply to workflow naming are that:

- The workflow name cannot contain blanks.
- The workflow name cannot contain special characters.

In addition to that, there is also a description field that can further describe the workflow.

The following is the naming syntax to be used with workflows:

```
{TargetObject} {WorkflowAction}[ {SubSection}] [{CMSection}]
```

Where:

{TargetObject}	Name of the target object (that is, ISV solution, application name, device name, and so on) to be managed by IBM Tivoli Intelligent Orchestrator using this workflow.
{WorkflowAction}	Action of the workflow. Usually the operation performed by the workflow.
{SubSection}	Optional field. The section that describes the structure of workflows used as transitions.
{CMSection}	Section for purposes of Change Management.

Archived

The {SubSection} above could be defined as:

```
{SubWorkflow}[_{SubSection}]
```

Where:

{SubWorkflow} Name of the transition workflow

{SubSection} Optional field

The {CMSection} above could be defined as:

```
{Vendor} {##.##} | {'Template'}
```

Where:

{Vendor} {##.##} Short name or ID of the ISV that developed the workflows followed by a release number.

or

{'Template'} Ending the workflow's name with 'Template' shows that this workflow is a template to be customized for a customer's use.

All different parts of a workflow name are separated from each other by using an space(" "). Fields in { ...} are mandatory and fields in [...] are optional.

Fictitious examples:

- T3 Install InstallApp ITSO 01.00

  TargetObject = T3 Install

  WorkflowAction = InstallApp

  Vendor and release number = ITSO 01.00
- AIX Software Install Template

  Target Object = AIX

  WorkflowAction = Software Install

  Template workflow

Naming conventions for automation packages

When developing new automation packages, we recommend using the following naming convention:

```
{TargetObject}-{DevDrvGrp}-{CM_section}.tcdriver
```

Archived

Where:

{TargetObject} Name of the target object (that is, ISV solution, application name, device name, and so on) to be managed by IBM Tivoli Intelligent Orchestrator

{DevDrvGrp} Name of the Device Driver Group

{CMSection} Section for purposes of Change Management

The {CMSection} above could be defined as:

```
{Vendor} {##.##} | {'Template'}
```

Where:

{Vendor} {##.##} Short name or ID of the ISV that developed the workflows followed by a release number.

or

{'Template'} Ending the workflow's name with 'Template' shows this workflow is a template to be customized for customer's use.

Usage examples are:

```
T3-Cluster-ITS001.00.tcdriver
```

Note: The name of the automation package file must be exactly the same as the automation package name defined in the automation package definition file (tc-driver.xml). For details, refer to the Chapter 8, "Automation package content and packaging" on page 233.

3.3.7 Documentation guidelines

This section covers the need for a complete and structured documentation of the automation package. It recommends how to build a *Reference Documentation* about the package and contains information about:

- Automation packages documentation
- Workflows, plug-ins, Java programs, and logical operations documentation
- Variables documentation

We also provide an example of documentation used in Chapter 10, "Case study: Trade3 composite application" on page 281.

Archived

For more detailed information about the documentation required when packaging an automation package, file please refer to Chapter 8, “Automation package content and packaging” on page 233.

Note: All the templates for documenting workflows and automation packages presented in this section are available for downloading. Refer to Appendix G, “Additional material” on page 625.

Automation packages

On top of the README file that is packaged with the automation package file, we recommend the following data to be part of the automation package:

Name	Name of the application package following the naming convention defined in 3.3.6, “Naming conventions” on page 88
Release	Release Number or version of the automation package (see 3.3.6, “Naming conventions” on page 88)
Description	Complete and structured description about the functionality of the regarding automation package
TC-Driver.xml	Name of the TC-Driver.xml file

Additional information has to be added for Change Management purpose:

Approved	A check box to show that the automation package has been tested, works as expected, and is ready for production.
Date	Date of the last update of this automation package.
Date of File	Date of the last update of the TC-Driver.xml file.

Using this data and additional information, we built a form for documenting an automation package. This piece of documentation should be on top of the Reference Documentation about an automation package.

Figure 3-18 on page 93 shows a portion of the form for documenting the automation package with the information discussed above.

Archived

Automation Package Documentation			
Name			Approved
Release		Date	
Description			
TC-Driver.xml			
Date of File			

Figure 3-18 Automation package documentation

Workflows, plug-ins, Java programs, and logical operations

Taking a look at the internal documentation of an workflow inside IBM Tivoli Intelligent Orchestrator, we see a great deal of available information:

Figure 3-19 shows an example of the internal documentation of the workflow ITSO WebAppInstall.

Figure 3-19 Internal documentation of a workflow

We used this as the basis for building a documentation form. This form uses the data described below:

Name Name of the application package following the naming convention defined in 3.3.6, “Naming conventions” on page 88.

Archived

Description	Complete and structured description about the functionality of the workflow.
Approved	The check box that is used to show that the workflow is tested, working properly, and ready for production.
Category	Category of this workflow with the value of Not assigned, Reusable, DCM Accessor, or TC Transaction.

Additional information has to be added for Change Management purposes:

Implementation	Name of the logical operation implemented by this workflow.
TC-Driver	Name of TC-Driver/automation package this workflow is part of.
Implementation	Object's type of implementation, with the following check boxes marked as Logical Operation, Workflow, or Java plug-in or -program.
Release	Release number or version of the automation package (see 3.3.6, "Naming conventions" on page 88).
Date	Date of the last update of this workflow.
Export	Name of the export file of this workflow.
Date of Export	Date of the updated export file.

Archived

Figure 3-20 shows the main part of a documentation template that can be used for the following objects: workflows, Java plug-ins, Java programs, and logical operations.

Object Documentation							
Name							Approved
TC-Driver							
Implementation	Log. Operation	Workflow	Java Plug-in	Simple Com.			
Release			Date				
Category	Not assigned	Reusable	DCM Accessor	TC Transact.			
Description							
Export							
Date of Export							

Figure 3-20 Object documentation

The above template can be enhanced to include object specific information as follows:

- Workflows

  Figure 3-21 shows the part of the object documentation that should contain the additional informations, if the object is a workflow.

Workflow	
Logical Operation	

Figure 3-21 Workflow documentation

The object specific documentation of a workflow contains the following data and information:

Logical Operation Logical operation implemented by this workflow

Archived

- Java plug-in

  Figure 3-22 shows the part of the object documentation that should contain the additional informations, if the object is a Java plug-in.

Java Plug-in	
Class Name	

Figure 3-22 Java plug-in documentation

The object specific documentation of a Java plug-in contains the following data and information:

Class Name Name of the Java class used by the Java plug-in

- Java program

  Figure 3-23 shows the part of the object documentation that should contain the additional information, if the object is a Java program.

Simple Command	
Java Plug-in	

Figure 3-23 Java program documentation

The object specific documentation of a Java program contains the following data and information:

Class Name Name of the Java class used in the workflows

Method Name Name of the method called from the workflow

Description Description of the functions provided by the methods, and the arguments passed to and from the method.

In order to enhance the overall documentation of a workflow, Java plug-in, Java program, or logical operation, a table (shown in Figure 3-24 on page 97) could be created to specify the following:

- Logical operations implemented by the automation packages workflows
- Java plug-ins used by the workflows
- Java programs used by the workflows
- Other data center objects associated to this object

Archived

Data Center Objects assigned to this object:	
Type	Name

Figure 3-24 Documentation Data Center Objects assignment

The list contains the information described below:

Type Type of object associated

Name Name of object associated

Variables documentation

IBM Tivoli Intelligent Orchestrator uses different types of variables, including:

- Input variables (global)
- Output variables (global)
- Local variables

They can be encrypted or not.

Archived

Figure 3-25 shows the different types of variables used in workflows.

Figure 3-25 Variables of the workflow ITSO WebAppInstall

We recommend you document the following items:

Name Name of the variable, following the naming convention defined in 3.3.6, “Naming conventions” on page 88

Type The implementation type of the variable, with the check boxes marked as Input, Output, Local, or Encrypted

Default Value Default Value of this Variable, if needed

Description Description of this variable and its use

Figure 3-26 shows how to document the embedded variables of the object. This section should be copied for each of the variables.

Variables				
Name				
Type	☐ Input	☐ Output	☐ Local	☐ Encrypted
Default Value				
Description				

Figure 3-26 Documentation of variables

Documentation example

In this section, we provide an example of documentation following the guidelines for documenting automation packages, workflows, and related objects presented in this chapter. The examples provided here are based on the sample case study scenario developed for the IBM WebSphere Application Trade3 application in Chapter 10, “Case study: Trade3 composite application” on page 281.

Figure 3-27 through Figure 3-33 on page 101 show this documentation example in detail.

IBM	IBM Tivoli Intelligent ThinkDynamic Orchestrator **Automation Package Documentation Form**	

Figure 3-27 Headline of automation package documentation

Automation Package Documentation			
Name	T3-Software Products-ITSO 01.00.tcdriver	x	**Approved**
Release	01.00	**Date**	March 9th, 2004
Description			
Automation package for provisioning and unprovisioning the IBM WebSphere Application Trade3 to an existing application cluster as part of the fictitious customer's operating environment			
TC-Driver.xml	T3-Software Products-ITSO 01.00.tcdriver		
Date of File	March 9th, 2004		

Figure 3-28 Automation package documentation

Archived

Object Documentation							
Name	T3 Install CopyImage ITSO 01.00					x	**Approved**
TC-Driver	T3-Software Products-ITSO 01.00.tcdriver						
Implementation		Log. Operation	X	Workflow		Java Plug-in	Simple Com.
Release	01.00		**Date**	March 9th, 2004			
Category		Not assigned	X	Reusable		DCM Accessor	TC Transact.
Description							
Copy the ‚tar'-file of the IBM WebSphere Application Trade3 to the server to be provisioned.							
Export	T3 Install CopyImage ITSO 01.00 03902004.xml						
Date of Export	March 9th, 2004						

Figure 3-29 Object documentation

Data Center Objects assigned to this object:	
Type	**Name**
Workflow	T3 Install ITSO 01.00

Figure 3-30 Assigned object of data center

Workflow	
Logical Operation	Software.Install

Figure 3-31 Workflow specific documentation

Archived

Variables				
Name	DeviceID			
Type	x Input	Output	Local	Encrypted
Default Value				
Description	ITITO internal ID of device (server) to provisioned			
Name	SoftwareID			
Type	x Input	Output	Local	Encrypted
Default Value				
Description	ITITO internal ID of software package to installed on target server			
Name	ClusterID			
Type	Input	Output	x Local	Encrypted
Default Value				
Description	ITITO internal ID of apllication cluster to which the server should be provisioned			

Figure 3-32 Documentation of the workflow's variables

Name	DestinationPath			
Type	Input	Output	x Local	Encrypted
Default Value				
Description	Workflow internal use of the software package property T3TargetDirectory			
Name	DE Device ID			
Type	Input	Output	x Local	Encrypted
Default Value				
Description	ITITO internal DeviceID of the Deployment Engine			
< ... >				
Name	UserNameDefault			
Type	Input	Output	x Local	x Encrypted
Default Value	Administrator			
Description	Name of the user of an administrative account on the target operating system			

Figure 3-33 Documentation of the workflow's variables

Archived

Archived

4

Workflow development environments

Tivoli Intelligent Orchestrator and Tivoli Provisioning Manager Version 3.1 both provide two different facilities to help you develop workflows:

- The Web based Workflow Composer
- The Automation Package Development Environment

In addition, a plain text editor may be used to create and maintain workflows; however, to test and run these, the flat-file versions of the workflows must be imported into the TIO/TPM environment through one of the facilities provided by TIO/TPM in order to compile and test the workflows.

The Workflow Composer is installed as part of the base system - TIO or TPM - but the APDE needs to be installed selectively on workstations used to develop and maintain workflows and automation packages.

© Copyright IBM Corp. 2004, 2006. All rights reserved.

The Automation Package Development Environment (APDE) is an Eclipse-based plug-in environment that connects to the TIO/TPM server and the data center model to build workflows and automation packages that can be used to manage data centers (see Figure 4-1).

Figure 4-1 APDE architecture

The Eclipse platform is structured as a core runtime engine that includes additional features that are installed as plug-ins. A plug-in contributes additional functionality to the Eclipse environment platform.

Table 4-1 summarizes the functional differences between the different workflow development environments.

Table 4-1 Development environment functional comparison

	task	Text editor	Workflow Composer	APDE
workflows	create		x	x
	export		x	x
	import		x	x
	edit	x[a]	x[b]	x[c]
	compile		x	x
Java code	create	manual		x
	edit	manual		x
	compile	`javac` command		x
	register plug-in			

Archived

	task	Text editor	Workflow Composer	APDE
Automation packages	create tcdriver.xml	manual		automatic
	build	`jar` command		x
	install	`tc-driver-manager` command		x
	uninstall	`tc-driver-manager` command		x

a. Requires that the workflow has been created using APDE or exported from Workflow Composer
b. Through drag-and-drop style programming
c. Context sensitive help available

Archived

4.1 Preparing the TIO/TPM environment

By default, the central TIO/TPM is configured not to enforce access control, which implies that all users that are authorized to execute workflows can manipulate any DCM object in any way they like. This may not necessarily be the optimal way of controlling changes to the environment. To set up an authorization schema that will work for your organization requires a good understanding of the TIO/TPM authorization mechanisms, and thorough planning.

In addition to controlling the access to functions and objects in the data center model, the TIO/TPM administrator should consider if the central database environment supporting TIO/TPM is configured adequately to support the database use of several simultaneous APDE users.

Neither security and access control or database performance are topics that are specifically related to automation package and workflow development; however, the following provides a brief discussion of these topics in the context of workfow and automation package development.

4.1.1 Security and authentication considerations

Prior to letting a number of APDE developers loose in the TIO/TPM environment, the TIO/TPM administrator needs to plan for having these extra users active in the system. The TIO/TPM administrator needs to set up authorizations to adequately protect business assets and resources in the data center model while providing authorized users access to the functions and resources required to perform their job.

To allow the administrator to do this, TIO/TPM provides two related mechanisms to control privileges:

User roles Used to authorize users to perform specific functions or sets of functions required to assume the responsibilities of a particular role, such as Cluster Domain Operator, Workflow Operator, or Security Operator.

Access privileges Controlled through sets of Access Groups and Access Permissions.

Access Groups are used to group several DCM objects, and as placeholders for authenticating users to manipulate the resources in the group. Users are authorized to issue specific logical device operations against the resources in an Access Group by linking one or more access permissions to an access group, and specifically subscribe users one or more access

permissions.

Access Permissions are used to group logical device operations that a user may perform against resources in an Access group.

Note: The enforcement of access privileges require that access control has been enabled for the entire TIO/TPM environment by a user with Admin or Security Operator authority.

The relationships between Users, Security Roles, Access Groups, and Access Permissions are depicted in Figure 4-2.

Figure 4-2 TIO/TPM users, roles, and permissions

To ease the understanding of the TIO/TPM authorization mechanism, the following statements may be helpful:

- Security roles define WHAT the user is allowed to do in terms of actions to be performed.
- Access Groups define subsets of the data center model. These are used to determine WHICH resources a user can access.

- Access permissions define HOW resources in an Access Group can be manipulated (in terms of issuing logical device operations against the resources in the related Access Group) by authorized users.
- By authenticating a user to an Access Group through the linked Access Permission, the TIO/TPM Administrator allows the user to perform the Logical Device Operations in the Access Permission against the resources in the Access Group.

When combined, the users privileges can be restricted in such a way, that users may be limited to only perform a specific set of actions (logical device operations) against a limited, well-defined set of DCM resources, for example, all related to the same customer or application.

Security roles

Each user in the TIO/TPM environment is assigned a specific set of roles that the user is authorized to perform. The minimum roles required to be able to create, compile, and execute workflows, logical device operations, and tcdrivers are:

- Workflow, TCDriver, Device Driver - Edit
- Workflow, TCDriver, Device Driver - View
- Workflow - Execution

All of these roles should be assigned to a single security role in TIO/TPM, for example, named Workflow Developer, and each of the TIO/TPM users that works with workflow and device driver development should have this security role assigned. Figure 4-3 shows the security role properties as displayed in TIO/TPM.

Figure 4-3 APDEDeveloper security role properties

Managing activity authorizations through security roles allows for easy maintenance and the highest degree of transparency for the security administrator.

Note: You should be aware that the Login Role is required to be able to login to and use the TIO/TPM Web UI. The above security roles will allow users to develop and test workflows and device drivers through the APDE interface - and not the Web UI. To allow the users to perform Web UI workflow interactions, you should add the Login Role to the roles of the Workflow Developer security role.

TIO/TPM actually already has a specific security role defined, which could be used. The Workflow Operator role has all the roles necessary assigned, but in addition, this security role includes the Task - Edit, Task - View, and Login roles, which strictly are not necessary for APDE workflow developers. For a complete list of the built-in security roles, please refer to the Tivoli Information Center at `http://publib.boulder.ibm.com/infocenter/tiv3help/index.jsp` and navigate to **Intelligent Orchestrator** → **Reference** → **Predefined user roles**.

To authorize users for a specific security role, modify the user definition, and add the necessary roles, as shown in Figure 4-4.

Figure 4-4 Assigning security roles to users

Now, the users are authorized to perform the activities necessary to develop and test workflows, and at this point, where we assume that the global Access Control is disabled, the users will be able to access and manipulate any object in the data center model. If, however, global Access Control was to be enabled, the users will not be able to access any DCM objects, since they have not been specifically authorized to perform actions against any DCM objects.

To authorize users to manipulate DCM objects while global access control is enabled, we have to assign the users to one or more access permission groups, where each has been assigned to an access group. So let us take a look at access groups and permissions.

Access groups

An access group defines a subset of DCM resources that a user may manipulate. Multiple subsets (access groups) may be nested to allow for a modular approach to access group administration and easy maintenance.

The DCM resources associated with an access group are added statically, on a one-by-one basis. In Version 3.1 of TIO/TPM, there is no way to define generic filters specifying resources to be included in an access group, so resources are added to the access group from the Web UI page of each individual resource. Also, when new resources are added (for example, using a workflow), it is the responsibility of the resource creator to include the resource in the correct access group.

Note: Please remember that access groups and access permission groups are only consulted to restrict user access to DCM resources if the global TIO/TPM configuration setting Access Control is enabled.

Figure 4-5 shows the definitions for an access group that only contains resources related to the QA-Customer

Figure 4-5 Resources in the QA_Customer Access Group

Once the access group has been defined, an set of access permissions may be associated with the access group, and finally users can be authorized to manipulate the access group resources as specified in the access permission set. As seen in Figure 4-5, no access permissions has been assigned to the access group, and hence, neither has any users.

Access permissions

Access permissions specify a set of logical device operations that may be performed against the objects in the access group(s) to which the access permissions are assigned. If, for example, you want to limit a user to manipulate only software and devices related to a particular access group, you would create an access permission group containing only the logical device operations related to the DCM, devices, and software.

Figure 4-6 Access permissions

For a complete list of the built-in access permissions, please refer to the Tivoli Information Center at `http://publib.boulder.ibm.com/infocenter/tiv3help/index.jsp` and navigate to **Intelligent Orchestrator** → **Reference** → **Access Permissions**.

Archived

Finally, to apply the permissions to a particular user, you must assign the access permissions to an access group, and for each user, authorize the user to the access permissions (see Figure 4-7).

Figure 4-7 Access Group with associated access permissions and users

In Figure 4-8 on page 113, we have tried to visualize the relationship between access groups and access permissions, and the resulting authorizations that will be assigned to the users that are associated with access permission.

Archived

Figure 4-8 Resource authorizations using Access Groups and Permissions

4.1.2 TIO/TPM Database performance considerations

The APDE interacts heavily with the central TIO/TPM database, so the more active workflow developers your environment has to support, the greater the risk that you will experience database performance degradation, hurting both the developer's productivity, and the overall performance and stability of the entire TIO/TPM environment.

Database performance tuning is naturally a task that should be performed by the DBA, who should be familiar with the implementation details of your particular environment; however, if you experience long response times from the APDE environment, or the DBA receives warnings regarding insufficient heap size, you should consider increasing application or the application control heap sizes for the TIO/TPM database. By default, the heap size variable, APP_CTL_HEAP_SZ, is set to 128 4 KB pages.

To check the current value of APP_CTL_HEAP_SZ, you can run the following command:

```
db2 get db cfg for <TIO database name>.
```

Follow the steps below to increase the application control heap size available

1. Log on to the DB2 server locally, using a DB2 administrator account:

   ```
   db2 connect to <databasename> user <username> using <password>
   ```

2. Update the database configuration using the following command:

   ```
   db2 update db cfg for <TIO database name> using app_ctl_heap_sz 1024
   ```

3. Stop the database to activate the changes:

   ```
   db2stop force
   ```

4. Start the database:

   ```
   db2start
   ```

By default, the TIO/TPM installer sets the application heap size variable, APPLHEAPSZ, to 3072. If the TIO/TPM database was created manually, consider setting the APPLHEAPSZ variable to 3072. To do this, use the following commands to update your DB2 configuration:

```
db2 connect to <databasename> user <username> using <password>
db2 update db cfg for <TIO database name> using applheapsz 3072
db2stop force
db2start
```

Important: Before setting specific database performance parameters, you should check for further information and recommendations in the release notes of the specific product level (Fix Pack) you have installed.

4.2 The development environments

With TIO/TPM V3, three specific environments are provided:

- The Workflow Composer
- The Automation Package Development Environment
- Standard text editor

In the following, we will take a closer look at the first two.

Archived

4.2.1 The Workflow Composer

The Workflow Composer is a Web-based user interface that has been available since Tivoli Orchestrator and Provisioning Manager Version 2.1.

The Workflow Composer is restricted to workflow development only, but provides a easy-to-use drag-and-drop style workflow development environment that allows the novice user to quickly become productive and gain experience in developing workflows.

To open a specific workflow, you may use the Workflow Composer, open your browser, log in to the Orchestration or provisioning environment, and navigate to **Tasks** → **Configuration**, as shown in Figure 4-9.

Figure 4-9 Navigate to Configuration

Once in the Configuration context, expand the **Workflows** category (by clicking on the expand icon ▸ next to the Workflows label) and open a workflow by double-clicking any of the workflows shown in the list.

Instead of listing all workflows in the navigation pane in order to select the one of your interest, you can double-click directly on the Workflows label (instead of expanding it) and use the search facilities in the All Workflows dialog, as shown in Figure 4-10.

Figure 4-10 Workflow search

From here, you can open a particular workflow by double-clicking the name or by selecting **Properties** ( ) from the object context menu icon ( ) to the right of the workflow.

> **Tip:** Instead of double-clicking the workflow of your interest, you may wish to use your browser's capabilities to open the Workflow Composer - or any dialog for that matter - in another window, allowing you use the TIO/TPM UI while editing your workflow.
>
> To open a new window, right-click the workflow of your interest, and select **Open in New WIndow**.

Finally, you will reach the Workflow Composer, in which you can see the entire workflow, as shown in Figure 4-11.

Figure 4-11 Workflow browsing in Workflow Composer

Now, to provide functionality in the workflow, you select a particular workflow element from the list in the top of the Workflow Composer dialog, and drag-and-drop it onto the designated line in the body of the workflow.

4.2.2 The Automation Package Development Environment

In addition to the Workflow Composer, TIO/TPM V3.1 provides an Eclipse based development environment that can be installed on workstations used by workflow developers.

Besides providing a development environment supported by an intelligent (parsing) editor with build-in context sensitive help, the Automation Package Development Environment (APDE) provides an integrated environment for performing all actions necessary to create, build, and compile an entire automation package.

The main features of APDE are:

- Automation package, device driver, and workflow creation wizard.
- Workflow editor with real-time help on available workflow elements (syntax), available workflows, java plug-ins, and java classes.
- Automatic workflow compilation when workflows are saved.
- Compilation errors/warnings are reported visually.
- Workflow execution with real-time workflow execution results and history.
- Ad hoc DCMQuery execution.
- TCDriver Install/Uninstall - with limitations.

For more details, please refer to the *IBM Tivoli Intelligent Orchestrator Workflow Developer's Guide Version 3.1*, GC32-1652, which is available on the installation CD-ROM labeled APDE (disk 9) or online at the Tivoli Information Center Web site: `http://publib.boulder.ibm.com/infocenter/tiv3help/index.jsp`. From the navigation pane, select **Intelligent Orchestrator** → **Workflow Developer's Guide**.

APDE architecture

As already mentioned, the Automation Package Development Environment is a plug-in to the open-source Eclipse platform. Most of the interaction with the TIO/TPM environment is based on direct database manipulation, performed through the IBM DB2 client, as shown in Figure 4-12 on page 118. For performance reasons, the Java-RDBMS interface is not used.

The Eclipse platform is structured as a core run time engine that includes additional features that are installed as plug-ins. A plug-in contributes additional functionality to the Eclipse environment platform.

Archived

Figure 4-12 APDE architecture

The database interface handles all requests regarding workflow creation, update, and compilation; in addition, it also handles object (device driver, workflow, Java plug-in, and so on) lookups. Furthermore, the workflow execution history presented in the APDE is gathered directly from the TIO/TPM database, while the workflow execution within the TIO/TPM environment is initiated through the SOAP interface.

4.2.3 Installing APDE

The Automation Package Development Environment is based on the following components:

- Eclipse V3.01
- IBM DB2 or Oracle Client
- IBM Java2 Runtime Environment Version 1.4.2

Restriction: It should be noted that APDE has only been tested on Windows 2000 and Windows XP platforms.

The Eclipse Version 3.01 development environment is provided on the APDE installation media, but before starting the installation, you should obtain copies of the database client and the IBM JRE™.

TIO/TPM supports the use of both IBM DB2 and Oracle database environments, and the APDE workstation needs a native client to connect and communicate

with the TIO/TPM database. In the following, we will discuss the setup of a DB2 Client environment to connect to a TIO/TPM database implemented on IBM DB2 Version 8.2.

The Automation Package Development Environment can be installed on a developer workstation either manually - or by using the TIO/TPM platform itself by means of the IBM JRE, Eclipse, and APDE automation packages provided with TIO/TPM V3.1 Fix Pack 2. For more details on installing and deploying APDE using the tcdrivers, please refer to the latest information available at the IBM Open Process Automation Library (OPAL) at:

http://www.ibm.com/software/ondemandcatalog/automation

The following provides instructions on how to install APDE manually.

Manually installing and verifying prerequisites

As discussed, the prerequisites required for installing and configuring the APDE environment are:

1. IBM Java2 Runtime Environment Version 1.4.2
2. Database client
3. Eclipse

IBM Java2 Runtime Environment Version 1.4.2

To check which version of the JRE is the default version in your environment, issue the **`java -version`** command in a command-line window. When inspecting the output, verify that you are indeed using the IBM JRE and that the version is 1.4.2 or higher. The following shows the output from an environment that honors the Java run time requirements imposed by APDE:

```
C:\>java -version
java version "1.4.2"
Java(TM) 2 Runtime Environment, Standard Edition (build 1.4.2)
Classic VM (build 1.4.2, J2RE 1.4.2 IBM Windows 32 build
cn1411-20040301a (JIT enabled: jitc))
```

If your current Java run time environment is not IBM Java2 Version 1.4.2 or later, you should obtain a copy from IBM developerWorks® at http://www-128.ibm.com/developerworks/java/jdk and install it on the APDE workstation.

Database client

To access the TIO/TPM database, a database client must be installed on the APDE workstation. This client must support the JDBC interface as well as the TCP/IP communication protocol. Since the TIO/TPM database may be implemented on both IBM DB2 and Oracle databases, you must install the client

code that supports your particular environment. In our case, the TIO/TPM database is implemented on a IBM DB2 V8.2.1 UDB Database Server, so in the following, our discussion will be limited to the setup and configuration of the DB2 Client V8.2.1 to allow APDE to access the TIO/TPM DB2 database.

The main activities involved are:

- DB2 Client installation
- Cataloging the TIO/TPM Database Server node and database
- Verifying connectivity

Installing the DB2 client is pretty straightforward, and the documentation in the DB2 Information Center at `http://publib.boulder.ibm.com/infocenter/db2help/index.jsp` (navigate to **Installing** → **Database systems** → **DB2 Universal Database™ for Linux, UNIX®, and Windows** → **DB2 clients**) provides all the details necessary to succeed.

Once the DB2 client is installed you have to let the DB2 CLient know how and where to find the TIO/TPM database. To achieve this, you will need to issue two **`db2 catalog`** commands in the DB2 command-line interface. The following describes the steps in detail:

1. Open the DB2 command-line interface by issuing the **`db2cmd`** command from a command-line window.
2. Catalog the TIO/TPM Database Server system by issuing the following command:

   ```
   db2 catalog tcpip node <node-name> remote <host-name> server <port>
   ```

   Where:

   node-name — A logical name assigned to the server. Any name can be used, as long it is not longer than eight characters. A name of TIODBSRV might be a good suggestion.

   host-name — The fully qualified name of the TIO/TPM Database Server. If the name cannot be resolved, use the IP address of the TIO/TPM Database Server. In our case, the TIO/TPM Database Server is located on a system named TIOServer.demo.tivoli.com.

   port — The port number used by the DB2 instance in which the TIO/TPM database is implemented. In our case, the port number is 50100.

Archived

Using the parameters relevant to our environment, the command used to catalog the TIO/TPM Database Server system to the DB2 Client is:

```
db2 catalog tcpip node TIODBSRV remote TIOServer.demo.tivoli.com
server 50100
```

If in doubt on how to catalog the TIO/TPM Database Server to your client, contact your local TIO/TPM administrator or your DBA.

3. Catalog the TIO/TPM database by issuing the following command:

```
db2 catalog database <dbname> as <alias> at node <node-name>
authentication server
```

Where:

dbname The name of the database as it is known at the TIO/TPM Database Server system. If the default values were used during TIO/TPM installation, the database name will be TC. In our environment, the database name is TIODB.

alias The logical name used on the DB2 Client to access the database. Using the same name as is used on the TIO/TPM Database Server system may prevent a lot of confusion, so the default value should be TC. In our environment, we used an alias name of TIODB.

node-name The logical name assigned to the TIO/TPM Database Server system as used in the previous **`catalog tcpip node`** command. We used TIODBSRV.

Using the parameters relevant to our environment, the command used to catalog the TIO/TPM database to the DB2 Client is:

```
db2 catalog database TIODB as TIODB at node TIODBSRV authentication
server
```

If in doubt on how to catalog the TIO/TPM database to your client, contact your local TIO/TPM administrator or your DBA.

4. To verify the connectivity between your DB2 Client and the TIO/TPM database on the TIO/TPM Database Server system, use the following command to connect to the database:

```
db2 connect to <alias> user <userid> using <password>
```

Where:

alias The logical name of the database as it is known to the DB2 Client. You specified this name in the previous **`catalog database`** command. In our environment, the database alias is TIODB.

Archived

user The user ID assigned to you by the TIO administrator or the DBA.

password The password associated with the user that you supply.

Using the parameters relevant to our environment, the command used to coconut the TIO/TPM database from the DB2 Client is:

```
db2 connect to TIODB user tiodb using <password>
```

> **Note:** Verifying the connectivity to the TIO/TPM database is not strictly required. The authentication configuration in APDE allows for reusing the credentials used centrally on the TIO/TPM server, thus removing the need to maintain a large number of access authorities. Please refer to item 6 on page 128.

5. Close the DB2 command-line interface by issuing the **exit** command in the window.

Eclipse

As mentioned, the Eclipse development environment installation image is provided on the APDE installation CD-ROM. The exact location is:

```
<cdrom-drive>\eclipse\eclipse-SDK-3.0.1-win32-lucene.1.4.3.zip
```

As an alternative, you can download the Eclipse SDK from http://www.eclipse.org/downloads/index.php. You will need to download the entire Eclipse SDK, and not just the platform run time.

To install Eclipse on the APDE workstation, simply extract the zipped Eclipse SDK V3 archive to your preferred location (we suggest c:\ibm). While unpacking, you should make sure that the directory names recorded in the zip archive is recreated on the workstation. All the files will be unpacked to the eclipse sub-directory of the target directory of the unpack operation.

If Cygwin is installed on the APDE workstation, the following command will achieve the task:

```
unzip
/cygdrive/<cdrom-drive>/eclipse/eclipse-SDK-3.0.1-win32-lucene.1.4.3.zi
p -d /cygdrive/c/ibm
```

If you are using Cygwin to unpack Eclipse, the execution privileges are not set for any files in the Eclipse directory tree, and hence, you will not be able to start Eclipse successfully. Therefore, you should modify execution permissions for all files in the Eclipse directory tree by issuing the following command:

```
chmod -R +x /cygdrive/c/ibm/eclipse/*
```

Now, you should be able to start the Eclipse environment using the following command:

```
c:\ibm\eclipse\eclipse.exe
```

Next, to make sure that the Eclipse environment is started with the correct version of the Java2 Runtime Environment, and that the JRE executables are found in the path, we suggest that you create a small script called APDE.cmd to set the correct environment and start Eclipse.

Assuming that the IBM JRE V1.4.1 is installed in the c:\ibm\java141 directory, the APDE.cmd file for invoking Eclipse would look like this:

```
@echo off
set JAVA_HOME=C:\IBM\Java141\jre
set PATH=%JAVA_HOME%\bin;%PATH%
start c:\ibm\eclipse\eclipse.exe
```

The natural location for the APDE.cmd file would be the Eclipse directory (c:\ibm\eclipse in our case). You may also want to create a shortcut on the Desktop of the APDE workstation to APDE.cmd script, for easy access to the Automation Package Development Environment.

APDE installation

At last, it is time to install the Automation Package Development Environment on top of the newly installed Eclipse environment. The installation process is fairly simple: unpacking another archive on top of the Eclipse installation directory.

The archive file that contains the APDE plug-in is located in the apde subdirectory on the installation media and is named com.ibm.tivoli.orchestrator.tcdriverdevelopment.zip. To install the plug-in, simply unpack the archive to the same directory you used for unpacking the Eclipse archive. In our case, the directory is c:\ibm, and the command used in the Cygwin environment is:

```
unzip -o
/cygdrive/<cdrom-drive>/apde/com.ibm.tivoli.orchestrator.tcdriverdevelo
pment.zip -d /cygdrive/c/ibm
```

Again, you have to make sure that the files can be executed, by means of the **chmod** command:

```
chmod -R +x /cygdrive/c/ibm/eclipse/*
```

At this point, both Eclipse and the APDE plug-in has been installed, but you have to perform some basic customization before you can create your first workflow.

4.2.4 Configuring the Eclipse environment

Once installed, the Eclipse environment has to be configured in order to control where to store user settings and work files. This involves the following activities:

- Defining an Eclipse workspace
- Configuring the automation package perspective

which will be described in detail in the following.

Defining an Eclipse workspace

When the Eclipse environment is started, the user is prompted to (unless the Eclipse environment has been configured not to) to select which workspace should be loaded. A workspace is basically a directory in which specific configuration information used by the users is stored. By default, Eclipse will assume that the workspace is a subdirectory to the Eclipse installation directory; however, the workspace may be loaded from any location accessible from the workstation.

When opening Eclipse, at least for the very first time, you will be prompted to specify which workspace to load, as shown in Figure 4-13.

Figure 4-13 Select workspace

Configuring the automation package perspective

In the Eclipse environment, a perspective provides a specific set of functionality used to complete a task. The automation package perspective is the workbench that is used to write workflows, and create and build automation packages. This perspective will be loaded as the default perspective after installation of the APDE Eclipse plug-in.

To configure the automation package perspective, you should complete the following steps:

1. Start Eclipse, and select a workspace.
2. From the Eclipse menu, select **Window** → **Preferences**.
3. In the main Workbench panel, de-select the Build automatically option (as shown in Figure 4-14) to avoid background recompilation of all your workflows whenever a specific number of changes (defined through the Build Order dialog) have been applied.

Figure 4-14 APDE Workbench defaults

Archived

4. Expand the **Automation Package** perspective by clicking the + sign to the left of the label. Your dialog should look similar to the one shown in Figure 4-15.

Figure 4-15 Defining the TIO/TPM server to APDE

Define the TIO/TPM Server to the APDE by providing the following parameters:

Server Name The name or IP address of the Tivoli Intelligent Orchestrator server that you will use to run your workflows.

Port The port to use for the Tivoli Intelligent Orchestrator server. The default, the port is 9080.

Userid Supplies a user ID that is allowed to log in to the TIO/TPM environment and authorized to create, compile, and execute workflows. The default TIO/TPM administrator user that we used is tioappadmin.

Password The password for the TIO/TPM user. The default password for the tioappadmin user is tioappadmin.

At this point, you may enable/disable certain options that apply to automation package installation. The values shown in Figure 4-15 provide a good starting point.

5. (optional) Click the **Crypto** sub-category of Automation Package perspective in order to configure password encryption for accessing the TIO/TPM environments. You should see a dialog similar to Figure 4-16.

Figure 4-16 Configuring APDE encryption

APDE allows you to define two encryption keys:

Configuration key This key is optional and is used to encrypt all passwords used for database and TIO/TPM server access.

Database key This key is required and is used to encrypt sensitive information to be stored in the database.

If you wish to enable this feature in the Automation Package perspective, you should import an XML file containing the keys used to access the TIO/TPM environment.

A XML file containing valid keys, named crypt.xml, is available on the TIO/TPM server in the %TIO_HOME%\config directory. This file should be made available to the APDE workstation in order to enable it to be imported in the current Automation Package perspective.

To import the settings, use the **Import** button, specify the directory in which the crypt.xml file resides, and press **OK**. Alternatively, you may enter the database and configuration encryption keys directly in the appropriate fields.

Press **Apply** when finished.

The APDE encryption configuration is stored in a file named crypto.xml in the .metadata\.plugins\com.ibm.tivoli.orchestrator.tcdriverdevelopment\config directory relative to your workspace, and it should look similar to Example 4-1.

Example 4-1 crypto.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!--this file stores the encryption keys -->
<keys-configuration>
  <!-- database-key is a MANDATORY key and used to encrypt sensitiv
      information in the database.
      It is a different key then the configuration one.
   -->
  <database-key>MVgfWF5bELqhx4Wo9HZwYQ3W72fIO7+/</database-key>

  <!-- configuration-key is an OPTIONAL key.
      When configured, that is the key used to encrypt *ALL* passwords
      in configuration files.
      Comment the line out if you choose not to have this encryption
      (ONLY FOR DEVELOPMENT).
   -->

  <!-- below is a sample configuration key -->
  <!--

<configuration-key>OyY7it/9rWIpqx+nosgyenaDVKt2DtCo</configuration-key>
   -->

<configuration-key>31IxUmEE8Yl5dqTxwhVJ2dC5c6KF3C8Z</configuration-key>
</keys-configuration>
```

If you maintain this file manually, you should obtain the database key from the %TIO_HOME%\config\crypt.xml file on the TIO/TPM server.

6. To configure access from the APDE to the TIO/TPM database click on the **Database** sub-category of Automation Package perspective. You will see a dialog similar to the one shown in Figure 4-17 on page 129.

Archived

Figure 4-17 Configuring database access for the APDE

In the Database dialog, you can provide information to instruct the APDE how to interact with the TIO/TPM database. These definitions are closely related to the way the database client is configured (see “Database client” on page 119).

As was the case for encryption settings, you can either import a XML file containing the definitions used on the TIO/TPM server, or you can supply all the values manually.

In you opted to use the password encryption in the previous step, the preferred method is importing an existing set of values, since the password used to access the database will have to be supplied in its encrypted form. If encryption is not used, you may still use the import method, but you may have to re-enter the password in its non-encrypted form.

– Importing an existing XML file

  To import the database settings used on the TIO/TPM server, make the dcm.xml file in the %TIO_HOME%\config directory of the TIO/TPM server available to the APDE client, and specify the location in which it resides in the **Import from** field. Press **Import**, and verify the settings.

– Supplying values manually

If you do not have access to the dcm.cml file, or want to control the settings yourself, specify the following parameters:

DB Type Select DB2 or ORACLE. In our environment we used DB2.

JDBC Driver Select the proper driver for your environment. For DB2, the value should be COM.ibm.db2.jdbc.app.DB2Driver.

DB URL The URL to which the database server responds. The value should be constructed from the following template:

<drivertype>:<dbtype>:<dbname_as_known_to_the _dbserver>.

In our environment, the value is jdbc:db2:TIODB.

DB Name The alias used when cataloging the TIO/TPM database to your database client. The default value is TC, but in our environment, we used TIODB.

Username The name of the user authorized to access the TIO/TPM database.

Password The password of the user authorized to access the TIO/TPM database.

If a database encryption key has been defined, the password should be supplied in it encrypted form. If no database encryption key is provided, the non-encrypted database password should be used.

Press **Apply** when done.

The APDE database configuration is stored in a file named dcm.xml in the .metadata\.plugins\com.ibm.tivoli.orchestrator.tcdriverdevelopment\config directory relative to your workspace, and it should look similar to Example 4-2 on page 131.

Archived

Example 4-2 dcm.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<config>
  <database>
    <type>DB2</type>
    <driver>COM.ibm.db2.jdbc.app.DB2Driver</driver>
    <url>jdbc:db2:TIODB</url>
    <name>TIODB</name>
    <username>tiodb</username>
    <password>HrtGaD6vjfZmj1qNNpzbPg==</password>
  </database>
</config>
```

Instead of using the graphical interface to configure new workspaces or to automatically deploy APDE, the dcm.xml file and the related directory structure may be applied manually.

7. (optional) The final option in the Automation Package perspective configuration dialog, Workflow editor preferences, allows you to customize the look of the APDE editor. To modify the default setup, select the **Workflow Editor Preference** sub-category of the Automation Package Perspective and use the Workflow Editor Preferences dialog shown in Figure 4-18 to select the colors that you want to define for different aspects of your workflow code.

Figure 4-18 Workflow editor preferences

8. Press **OK** to save your APDE configuration and close the Automation Package perspective configuration.

Configuring the Eclipse environment

In addition to the specific configuration of the Automation Package perspective, you may want to modify the following options in the Eclipse environment in order control your APDE environment:

- Workflow file encoding

In addition, specific configuration of the APDE project may apply to your environment. Please refer to “Configuring the APDE project” on page 135 for further details.

Setting a workflow file encoding to ASCII

From the APDE menu:

1. Select **Window** → **Preferences** → **Team** and select **File Content**.
2. Click **Add Name** and type wkf.
3. Ensure that the value of the field in the content column is ASCII. If it is not, click on the text Binary to change it. In the drop-down list that appears, select the value ASCII, as shown in Figure 4-19.

Figure 4-19 Customizing file encryption

4. Click **OK** to save your settings.

4.2.5 Showing APDE views

If you closed a view and you want to re-open it, you can select **Window** → **Reset perspective** to set your perspective to its default state. You must be in the Automation Package perspective to show workflow-related views. You can also click **Window** → **Show View** → **Other** and select **Automation Package** to display other available views. Click the view that you want to work with.

4.2.6 Defining an APDE project

Before starting your development efforts using the APDE Eclipse plug-in, you should define an APDE Project in which all your definitions and workflows are stored.

An APDE project is for most practical purposes synonymous with an Automation Package (also known as a tcdriver). However, there is no requirement to create the Automation Package for a project, so it may also be used as a simple container for managing a workflow source.

To create a new APDE project, open Eclipse, select a workspace, and ensure that your have opened the Automation Package Perspective. Then select **File** → **New** → **Other** or press Alt+Shift+N and wait for the appearance of the Select a Wizard dialog, as shown in Figure 4-20.

Figure 4-20 Create new APDE project

Archived

From the Select a Wizard dialog, expand the **Automation Package** category, select **Automation Package Project**, and press **Next**. You will now be prompted, through the New TCDriver project dialog shown in Figure 4-21, to provide a name (and, optionally, a location) for the new project.

Figure 4-21 Naming a new APDE project

If there are no dependencies to existing tcdrivers, you can press **Finish** to complete the APDE Project creation. If however, you want to specify relationships to other tcdriver files, press **Next** and select the desired dependencies form the TCDriver dependencies dialog (see Figure 4-22 on page 135).

Archived

Figure 4-22 Setting APDE project dependencies

For the APDE_Driver project we are defining here, Figure 4-22 shows that we make this project dependent upon the core tcdriver. This means that the core tcdriver has to be installed (on the TIO/TPM server) prior to installation of the APDE_Driver tcdriver, and the APDE_Driver tcdriver must be uninstalled before the core tcdriver can be uninstalled.

We create this dependency primarily because we anticipate that our tcdriver will use the standard workflows and logical device operations defined in the core tcdriver. We might have specified more dependencies; however, at this point, we do not know exactly which tcdrivers to use.

Pressing **Finish** in the TDCDriver dependencies dialog will eventually create the project and close the New TCDriver Project Wizard.

Configuring the APDE project

Now that the project has been defined, you are ready to start creating and coding workflows and Java plug-ins to implement your functionalities particular to your automation package. However, before you put too much effort into coding, you may want to perform some additional configuration of the APDE project.

Preventing the bin directory from being overwritten

By default, Java projects and automation packages use the bin directory of the APDE project directory structure as the output for Java compilation. The contents of this directory will be deleted each time a compilation has occurred.

To prevent your directory contents from being deleted, you can modify the Java Build Path of your project by right-clicking the **APDE** project and selecting **Properties** → **Java Build Path**. Change the Default output folder to <APDE_project_name>/classes.

Remove errors related to referenced items in the tc-driver.xml file

The APDE checks the existence of all items referenced by your tc-driver.xml file, but there might be missing items, such as JAR files, created from the source code in the project by the build.xml file. Missing items can be resolved by creating a .tcdriverignore file in the project and adding the name of the item to the file. For example, if you want to remove errors related to lib/core.jar in the core project, add lib/core.jar to the .tcdriverignore file.

4.3 The text editor based development environment

The text editor based application package and workflow development environment is basically included in this discussion in order to allow us to discuss how to achieve various functions manually using the command-line environment of the APDE Workstation and the TIO/TPM server and a basic text editor, such as vi or Notepad. To set up an automation package and workflow develop environment based on these tools, there are no special requirements that must be honored.

Archived

5

Workflow development quickstart

Tivoli Intelligent Orchestrator (TIO) and Tivoli Provisioning Manager (TPM), through Automation Packages, automate the manual provisioning and deployment process. They use pre-built workflows to provide control and configuration to datacenter resources, such as applications, devices, and other resources that provide the services to IT. These workflows, coupled with the necessary documentation and resources, provide the Automation Packages that can then be automated and executed in a consistent, repeatable, and error-free manner that ensure availability and resilience of vendor products.

This section is intended to provide a brief, but thorough, overview of the necessary components of a sound Automation Package, as well as practices and samples that will streamline the development process. For purposes of this IBM Redbook, the Tivoli Intelligent Orchestrator (TIO) product will be referenced only, although Automation Packages developed will function in either TIO or TPM.

© Copyright IBM Corp. 2004, 2006. All rights reserved.

5.1 Development Environment

TIO workflows can be developed in several tools: Automation Package Development Environment (APDE), a favorite editor, or the Workflow Composer GUI. The ideal environment is through the APDE, as it not only provides additional support for workflow development, but also provides capabilities to test and build the final Automation Package.

5.1.1 Workflow composer

The workflow composer is a GUI-based tool that is designed to allow you to develop and maintain workflows in TIO/TPM. The workflow composer is accessed through the TIO/TPM GUI console and provides drag and drop component capability for workflow development. Workflows developed in the workflow composer can be exported, allowing a developer to work from a favorite editor. The workflow composer does not provide the capability to automatically generate an automation package (.tcdriver); therefore, this process must be performed manually. The composer is best suited for user modifications based on unique customer environmental requirements.

5.1.2 Editor

Workflows for automation packages can also be written using an editor of the developer's preference. A workflow can be completely written using an editor or an initial workflow can be created in the workflow composer and then exported and modified in an editor. Workflows developed in this manner can also be imported back into the TIO/TPM using the workflow composer. As with the workflow composer, a developer would need to complete the automation package manually.

5.1.3 Automation Package Development Environment (APDE)

The APDE is an Eclipse based plug-in that provides a developer with a complete automation package development environment. From the APDE, a developer can develop/test and refine a set of workflows for an automation package. The APDE also provides the capability to generate the automation package (.tcdriver). This is the recommended environment for automation package development. Some of the features of the APDE are:

- Automation package skeleton environment
- Base TC-INF/tc-driver.xml file automatically generated
- New automation package project creation
- Content assist
- Automation package generation (built tc-driver)

- Configuration with TIO/TPM database
- Color coded editor preferences

The Eclipse development platform itself can be downloaded from http://www.eclipse.org/ and the Eclipse plug-in for TIO/TPM workflow development can be obtained at http://www-18.lotus.com/wps/portal/automation.

5.2 Understanding Orchestrator and Provisioning Automation Library

The Orchestrator and Provisioning Automation Library (OPAL) is available to all TIO/TPM customers as a means to extend the number of resources that can be managed by downloading new and updated Automation packages developed by IBM and other vendors:

http://www-18.lotus.com/wps/portal/automation

Software and hardware vendors interested in placing their Automation Packages in the OPAL catalog can do so through the 'Ready For Tivoli' program. For more information, please refer to the following site:

http://www.developer.ibm.com/tech/validation/middleware.html

5.3 Workflow development

A robust automation package will include a number of integrated workflows representing the tasks required (with sufficient error handling and recovery) to successfully provision and de-provision a solution in a totally automated environment without human intervention.

A workflow is a sequenced set of transitions (commands) that run in a synchronous manner to accomplish the tasks needed to automatically provision systems and solutions. Examples of a workflow are

- Modifications to data center infrastructure (route changes and VLAN assignments)
- Configuration and allocation of servers (software installation and configuration)
- Specific command actions, such as rebooting a server or powering off a device
- Install and un-install of software images

Archived

When developing suitable workflows that meet the datacenter requirements, the following areas should be considered as part of the implementation:

- The Automation Package and its respective workflows should follow a standard naming convention.
- Determine the necessary functions to develop that best suits the provisioning of a software product or device.
- Use of Logical Operations.
- Use of informational headers and comments throughout a workflow to provide a better understanding of the source and function of the workflow.
- Variable naming.
- The Tivoli Intelligent Orchestrator Data Center Model Query capability (DCMQuery) should be used whenever possible to gather and update information from defined DCM objects.
- Developers should take advantage of the internal methods for logging transitional information, as well as handling errors and exceptions. to better manage TIO in production environments.

Throughout the development of the workflows and supporting documents, it is important to maintain this naming convention.

5.4 Automation Package Baseline Workflow Requirements

The following are baseline requirements and recommendations for the type of workflows needed to develop an automation package that can successfully deploy and un-deploy a software solution or alter the characteristics of a device.

5.4.1 Software solutions

TIO/TPM provides the concept of logical operations where the same type of action can be performed for multiple solutions and the underlying workflows determine the specific steps taken to accomplish the task on the target system. Workflows implementing logical operations are listed in the TIO/TPM console under “Logical Devices”. We recommend that the logical operation for an action be used or implemented (such as copying a file, executing a command, or installing software) when developing workflows. In addition, we recommend that the software object and any customer customizable software object parameters or variables required for the successful execution of the workflows be automatically created (through xml in the tc-driver.xml file) when the automation package tcdriver is installed. The tc-driver.xml file needs to include the stanzas

for creating the device model, software category, software product, and any related variables and properties.

For software products, the automation package must contain, at a minimum, a set of workflows for implementing the following logical operations for automatic deploy and un-deployment of the software product:

- Software.Install
- Software.Uninstall
- Software.Start
- Software.Stop

Table 5-1 shows the minimum set of workflows needed for each of the software products.

Table 5-1 Software related logical operations

Logical operation	Software product	Software patch	Software stack	Description
Install	Yes	Yes	Yes	Product install
UnInstall	Yes	Yes	Yes	Product uninstall
Start	Yes			Application or service start
Stop	Yes			Application or service stop
CheckStatus	Optional			Check Application or service status

5.4.2 Hardware products or devices

Device workflows will be highly dependent on the function and interface capabilities exposed by the device. For example, the logical operations needed to integrate with a switch will be different than those needed for a storage system. However, most devices have common concepts that can be related to logical operations, such as starting, stopping, recycling, and managing, the environment of the device. For example, turning the port on or off on a switch or starting and stopping a device.

We recommend (where feasible) that the device object and any customer customizable variables required for the successful execution of workflows be automatically created (through XML in the tcdriver.xml file) when the automation package is installed.

Archived

Table 5-2 shows the recommendations and guidelines for the base line device workflows.

Table 5-2 Hardware related logical operations

Logical operation	Server	Storage system	Switch	Network device	Description
Turn On/Off, Enable/Disable, Boot/Shutdown Start/Stop	Yes	Yes	Yes	Yes	Workflows to query the device and perform the operations to start/stop, boot/shutdown, or initialize/terminate the device.
Reboot or recycle	Yes	Yes	Yes	Yes	Workflows to query the device and perform the operations to reboot, recycle, or reset a device.
Get device status	Yes	Yes	Yes	Yes	Workflow to query the device and performs the operations to determine the status of the device.
Get device attribute	Yes	Yes	Yes	Yes	Workflow to query the device and perform the operation to query an attribute.
Set device attribute	Yes	Yes	Yes	Yes	Workflow to query the device and performs the operations to alter the device.

Archived

5.5 Defining and classifying workflows (logical operations)

Workflows should be developed to implement logical operations, like those described in the Automation Package Baseline Workflows section. Classifications of logical operations facilitate the automated deployment features of TIO/TPM, as it deploys servers and software products. Logical operations define an action that should be implemented by an orchestration automation package or a particular workflow. For example, Software.Install is a logical operation, abstraction, that a workflow implements to install a software product. The Software.Install logical operation makes no assumptions of the operating environment but performs an action that is constant across all operating systems. Workflows implementing this logical operation will perform the required operations for each operating system.

The following are examples of classifying the workflows for the software solution logical operations:

- Install a software solution.

  ```
  workflow MyCompany_Install_Product(in SoftwareID, in DeviceID)
  implements Software.Install
  ```

- Uninstall a software solution.

  ```
  workflow MyCompany_Uninstall_Product(in SoftwareID, in DeviceID)
  implements Software.Uninstall
  ```

- Start a software solution.

  ```
  workflow MyCompany_Start_Product(in SoftwareID, in DeviceID)
  implements Software.Start
  ```

- Stop a software solution.

  ```
  workflow MyCompany_Stop_Product(in SoftwareID, in DeviceID)
  implements Software.Stop
  ```

The following are examples of setting device based workflows to logical operations:

- Initialize a device.

  ```
  workflow MyDevice_Initialize(in DeviceID) implements
  Device.Initialize
  ```

- Turn on a switch port.

  ```
  workflow My_Switch_Turn_Port_ON(in SwitchID, in PortModule, in
  PortNumber) implements Switch.TurnPortON
  ```

Archived

- Turn off a switch port.

  ```
  workflow My_Switch_Turn_Port_ON(in SwitchID, in PortModule, in
  PortNumber)implementsSwitch.TurnPortOFF
  ```

5.5.1 Use logical device operations where possible

When performing an action, we recommend that defined logical operations be included in the workflow instead of using Java plug-ins or creating additional workflows. Defined logical operations can be found under the Logical Devices area on the Tivoli Intelligent Orchestrator Configuration and Workflow console tab. For example, to execute a command on a target system, one should use the logical device operation of:

```
Device.ExecuteCommand(DeviceID, Jython(command_path + "/" +
command_name + " start"), "/", "default", "60", <null>, <null>, <null>,
<null>)
```

By using the logical operation, it allows the system to determine the specific actions needed to perform the task on the different target systems. Be careful to not use the underlying defaults for a logical operation, as this represents the core implementation for the logical operation. So use Device.ExecuteCommand instead of Default_Device_Execute_Command:

```
Default_Device_Execute_Command(DeviceID, Jython(command_path + "/" +
command_name + " start"), "/", "default", "60", <null>, <null>, <null>,
<null>)
```

5.6 Workflow headers and copyright

All workflows should have an introductory comment section that identifies the company and solution. The workflow header should clarify the assumptions and requirements that need to be met within the object definition (software product, software patch, server, switch, and so on) in order for the workflow to automate successfully. The header should clarify the relationship the workflow has to other workflows in the automation package. The header should also include the company copyright statements. Copyright statements should be included in all automation package components, including workflows, documentation, external scripts, and Java programs.

Example 5-1 on page 145 is an example of a workflow header for a workflow that is part of an automation package representing a software solution. Any additional relevant information can also be appended to the header.

Example 5-1 Sample workflow header

```
#-----------------------------------------------------------------
#Licensed Materials - Property of MyCompany
#
#(C) Copyright MyCompany 2004#All Rights Reserved
#
# Workflow: MyCompany_Solution_Function
# Automation Package: MyCompany_Solution
# Description: This workflow installs the core services for the
MyCompany
# solution.
#
# Input Variables:
# SoftwareID – DCM object ID for the Software Product to be installed
on
# the target server or device.# DeviceID - DCM object ID of the server
endpoint.
#
# Output Variables:
# ReturnCode - Return Code of the MyScript.sh script
#
# DCM Object Properties:
#
# The following parameters and variables must be configured properly on
# the Software Product defined in the Data Center Model in order for
the
# functions of this workflow to succeed.
# DCM Software Product Variables:
#  Solution Install Key: MySolutionKey
#  Status: NotInstalled
#
#  DCM Software Product Parameters:
#  Silent Install Parameter Option: Option1
#  Silent Install Parameter Option2:Option2

#Description: Workflow uses the parameters defined with the software
#  product definition in the Data Center to build a silent install
option
#  file.
#  image files are pulled from the repository location defined in
#  Data Center Software Product Object definition.
#-----------------------------------------------------------------
```

Archived

5.7 Workflow comments

Comments should be used throughout the workflow to document the function of the workflow and the expected results. Comments can describe individual statements such as a DCM Query command and can also describe blocks of code, such as an if/else section. Comments can be combined with logging to make the workflow and the workflow execution log more readable and understandable for trouble-shooting activities.

Comments in a workflow are noted by the “#” sign.

The following is an example of workflow comments:

```
var MSG1 = "Execution successful"
var MSG2 = "Execution failed"
var MSG3 = "Execute the installation script"
# Execute the installation script using the files copied to the target
system
log info MSG3
Device.ExecuteCommand(DeviceID, commandString, "/tmp", "default",
"1800", "error", ReturnCode, ErrorMessage, <null>)
# Test for success / failure of the installation script.
if Jython(ReturnCode == "0") then
   log info Jython("MyScript.sh: " + (MSG1 or ""\))
else
log info Jython("MyScript.sh: " + (MSG2 or ""\))
endif
```

5.8 Variable naming conventions and handling

Customizable values required within workflows that the customer must modify (user tailored values) in order for the successful execution of the workflow should be defined as part of the DCM definition for the device or software product. This way, the customer only has to customize the properties for the DCM object without modifying the variable within the workflow source. DCMQueries or specific workflows (GetSoftwareParameter) should be used to retrieve the variable values for use in the workflow.

5.8.1 Variable encryption

Passwords and other sensitive data should never be stored in an unencrypted format, variables used to store sensitive data should be declared as encrypted. Workflows that receive or return sensitive data should declare the parameters as encrypted, along with command strings that contain passwords. Where possible,

“Service Access Points” should be used to store telnet passwords instead of DCM object parameters or variables.

The following is an example of a workflow with encrypted inputs:

```
workflow MyWorflow_Action(in IPAddress, in encrypted Password, in
encrypted Username)
```

The following is an example of an encrypted variable:

```
var encrypted telnetPwd
```

5.8.2 Variable naming conventions

A good practice is to define variables at the beginning of the workflows to facilitate ease of maintenance. The following represents the recommended best practices for naming variables and parameters:

Passed variables

Capitalize the first letter for external or passed variables. Variable names cannot match Jython reserved words (timeout, error).

```
workflow MyCompany_Jumpstart_Get_Client_Properties(in ServerID, out
ClientHostName, out ClientInterfaceName) LocaleInsensitive
```

Internal variables

Start variables with lower case for internal variables. Do not use spaces in variable names. Variable names cannot match Jython reserved words (timeout, error).

```
var softwareName
var serverName
var sendStatus
```

Constants

Constants should be in all uppercase. Consider using variables with a descriptive name as a constant for inputs that are potentially cryptic.

```
ERRORMSG1 = "This is an error"
var ENTIRESYSTEM = “0”
var DEPLOYMENTENGINE = “5”

java:com.thinkdynamics.kanaha.de.javaplugin.datacentermodel.GetProperty
(ENTIRESYSTEM, cardinalNodeID, "hacmp.publicSSHKey", publicSSHKey)
```

Archived

5.8.3 Variable handling

Workflow inputs should be validated as early as possible within a workflow. Ideally, this should occur before performing any actions that would need to be undone in the event of a failure. This includes input variables passed to the workflow and variables obtained via Get DCM Property transitions. At a minimum, variables should be tested to ensure they are not null. Additional tests for numeric, alpha, and alpha-numeric can also be performed. Variables obtained by user input to a workflow should have extensive validation. If possible, the input should be examined to ensure that it is the correct object type.

Examples:

```
var testMe
var result
var NUMERIC = "^[0-9]+$"
var ALPHA = "^[a-zA-Z]+$"
var ALPHANUMERIC = "^[a-zA-Z0-9]+$"
if Jython(testMe==None) then
   throw NULL_VARIABLE "testMe is Null"
endif
String_Replace_All(testMe, NUMERIC, "true", result)
if Jython(result != "true") then
   throw NONENUMERIC_VARIABLE "testMe is not numeric"
endif
String_Replace_All(testMe, ALPHA, "true", result)
if Jython(result != "true") then
   throw NONALPHA_VARIABLE "testMe is not alphabetic"
endif
String_Replace_All(testMe, ALPHANUMERIC, "true", result)
if Jython(result != "true") then
   throw NONALPHANUMERIC_VARIABLE "testMe is not alpha-numeric"
endif
```

5.8.4 Error handling within workflows

In order for a workflow to complete successfully or exit gracefully when errors are encountered, the workflow should provide sufficient error checking. In addition to checking input parameters and variables as described in the previous section, workflows should take advantage of the throwing and catching support provided in Tivoli Intelligent Orchestrator. These techniques provide support to allow the workflow to recover on its own or shift the error to other less obtrusive areas of the workflow. The following error handling techniques are supported in the workflow language.

Archived

Internal notification of workflow failures

When a workflow fails, if the DCM object was being acted upon is in an undetermined state, then the DCM object should be put into maintenance mode within the DCM. For example, if a Server is being rebuilt by a Sparepool.InitializeServer workflow and there is a failure, then the server should be put in maintenance mode. An error message should also be written to the TIO workflow log:

```
try
 . . . Workflow statements
catchall
   log info Jython(RECOVERY_MESSAGE)
   Set_Server_In_Maintenance("true", ERROR_MESSAGE, ServerID)
   rethrow
endtry
```

Workflow logging

When examining the workflow execution log, it is often difficult to determine what actions are occurring without having to refer back to the workflow source. In situations like this, it is beneficial if the workflow writes informational messages to the workflow log

For example, when a workflow performs a Device.ExecuteCommand, the actual command that is being executed is not logged. Some workflows perform multiple device execute commands in a row, which makes interpreting the workflow log very difficult:

```
var commandString
var MSG1 = "Command: "
...
commandString = Jython("/tmp/ifcfg.ksh " + (NetworkInterfaceName or
""\) + " " + (IPAddress or ""\) + " " + (netMask or ""\))
log info Jython(MSG1 + (commandString or ""\))
Device.ExecuteCommand(DeviceID, commandString, workingDirectory,
"default", "120","error", <null>, <null>, <null>)
```

For example, DCMQuery commands are used in variable assignments and are not captured in the workflow log. It is only when the variable is used in a later transition that it is logged. A good practice is to log the results of DCM queries:

```
# Get serviceSubnetID by using serviceSubnetName serviceSubnetID =
DCMQuery(/subnetwork[@name=$serviceSubnetName\])
log info Jython("DCMQuery serviceSubnetID = " + (serviceSubnetID or
""\))
```

In some cases, a workflow transition is expected to fail, and the failure should be ignored in order for the workflow to continue on. This action is implemented using a try/catch all statement. Only the single statement to be ignored should be in the try statement. Note that in the example below, even the logging of the message is outside the try loop. If it was inside, and a logging error occurred, this would also get ignored:

```
var MSG1 = "Execute Linux Restart Networking Command"
. . .
# When the network is successfully restarted, connectivity to the linux
# endpointis lost and this transition will fail on a timeout.
# Therefore this error needs tobe ignored.
log info MSG1
try
   Device.ExecuteCommand(DeviceID, COMMANDSTRING, <null>,
CREDENTIALSKEY,
   "60", "error", <null>, <null>, <null>)
catchall
   noop
endtry
```

In some cases, when a workflow fails, a recovery action should be performed. This action is implemented via a try/catch statement that encompasses the entire workflow. In most cases, the recover action only documents that an error has occurred and does not actually perform a complete recovery. In this situation, the **rethrow** command must be used to cause the workflow to fail and the original error message to be presented in the workflow log. If **rethrow** is not used, the workflow will be logged as ending successfully:

```
try
 . . . Workflow statements
catchall
   log info Jython(RECOVERY_MESSAGE)
   Set_Server_In_Maintenance("true", ERROR_MESSAGE, ServerID)
   rethrow
endtry
```

5.9 DCMQuery

The DCMQuery is the preferred means for obtaining and updating information in the TIO/TPM DCM. Java plug-ins included with the product are release dependant and the trend is to do more through DCMQueries rather than relying on embedded Java plug-ins. Some of the Java plug-ins in the current release have already been deprecated. The online help within Tivoli Intelligent Orchestrator provides information and examples on DCMQueries. Understand

that DCMQuery uses an XPATH like query to the database underlying the DCM to retrieve values.

Using the following DCMQuery, here is a brief explanation of the constructs that comprise the statement:

```
DCMQuery(/softwareproduct[@id=$SoftwareID]/@version)
Query the database (DCMQuery).
```

- Start at entire database (/). The slash directs the query to search from the root of the DCM.
- Search for a software product.
- Limit the search to rows that have a unique identifier of $SoftwareID ([@id = $SoftwareID]). The brackets are used to limit a query. You may see examples with more than one bracketed phrase in the expression. This is a making a further limit on the query.
- Retrieve the value of the version column (/@version).

Notes concerning DCMQuery commands:

- A dollar sign ($) in a DCM query expression followed by letters or digits is used to represent a variable in a DCM query.
- Brackets ([]) in a DCM query expression are used to group conditional and order by expressions.
- A back slash (/) in a DCM query expression represents an absolute path to the required element.
- Attributes are specified by a @ prefix and are separated from values by an equal (=) sign in a DCM query expression.
- Values are enclosed in double quotes (" ") in a DCM query expression.
- To use a value that contains a double quote, you can define a variable that has that value and use the variable instead in the query.
- To define a null value, use "<null>". For example, @name="<null>".

The following are some samples taken from the TIO/TPM online help information and other sources.

5.9.1 Using DCMQuery to retrieving DCM information

Retrieve all NIC IDs that have a server name of "*Nile*" and that are not managed:

```
DCMQuery(/server[@name="Nile"]/nic[@managed="N"])
```

Retrieve all the server names, ordered alphabetically by name, in a cluster called “RiversEdge”:

```
DCMQuery(/cluster[@name="RiversEdge Web"]/server/@name[orderBy@name])
```

Retrieve the server name from a cluster named “RiversEdge” using a variable ($serverName) that had been previously defined in a workflow:

```
DCMQuery(/cluster[@name="RiversEdge Web"]/server[@name=$serverName])
```

Retrieve information where some objects have two fields that link to another object (For example, accessRule has sourceSubnetwork and destinationSubnetwork attributes).

To navigate from accessRule to a subnetwork, use `accessRule/sourceSubnetwork` or `accessRule/destinationSubnetwork` instead of `accessRule/subnetwork`, as shown below:

```
DCMQuery(/acl[@name="acl-1
New1"]/accessRule/sourceSubnetwork/@ipaddress)
```

A relationship between two DCM objects can be navigated using a DCMQuery in two directions. For example, `/server[@name=\"Server1\"]/bladeadminserver` or `bladeadminserver [@name=\"Blade1\"]/server`.

Aliases are used to differentiate between relationships. For example, in the query below, accessRule has two fields connecting to a subnetwork `sourceSubnetwork` and `destinationSubnetwork`. Two relationships were generated for this scenario, one for each relation field:

```
/acl[@name="acl-1 New1"]/accessRule/sourceSubnetwork/@ipaddress
/acl[@name="acl-1 New1"]/accessRule/destinationSubnetwork/@ipaddress
```

In order to navigate in the opposite direction, you must use an alias object to differentiate between the relationships. For example, `/subnetwork[@ipaddress="10.1.8.0"]/AclRuleSource/acl/@name` will link the subnetwork to the sourceSubnetwork field from the accessRule, and `subnetwork[@ipaddress="10.1.8.0"]/AclRuleDestination/acl/@name` will link the subnetwork to the destinationSubnetwork field from the accessRule.

5.9.2 Using DCMInsert to insert data into the DCM

When you insert data into the DCM, it is assumed that the object does not already exist and you are creating it for the first time. Your XML data consists of top-level and embedded elements and, consequently, there are two methods for inserting data into the DCM. To insert data into the DCM, use the DCMInsert command, as shown in the examples below.

Inserting a top-level object: If a top-level object is being added to the DCM, the only requirement is the DCMInsert command and the XML data. For example, to insert a new blade server chassis that is not managed, has a network interface name of Management network, and an IP address of 10.1.8.77:

```
DCMInsert <<EOF
<blade-admin-server name="blade-chassis-admin">
<network-interface name="Management network"ipaddress="10.1.8.77"
managed="false"/>
</blade-admin-server>
EOF
```

Note: When you insert a top-level object (parent) into the DCM using the Web-based user interface, you do not have to specify a parent element in the Parent field.

Inserting an embedded element: If a child element is being added to the DCM, you must first retrieve the parent element's ID to determine the location in the DCM where the new data should be added. In the following example, the parent element is Customer, and the

```
DCMInsert parent = DCMQuery(/customer[@name="Rivers Edge Retail"])
```

expression will retrieve the Customer ID, and using this ID, can determine the precise location in the XML that will contain the new data:

```
DCMInsert parent = DCMQuery(/customer[@name="Rivers Edge Retail"])
<<EOF
<application name="Online Store New" priority="5">
<cluster name="RiversEdge Web" virtualipaddress="10.1.1.132"
virtual-tcp-port="80" min-servers="2" max-servers="10"
pool="Apache-Redhat" vlan="510" tier ="1" fabric="Default Fabric"/>
</application>
EOF
```

5.9.3 Using DCMUpdate to update data in the DCM

To update data in the DCM, use the DCMUpdate command, as shown in the example below:

```
DCMUpdate (/cluster[@name=\"RiversEdge Web1\"]) <<EOF
   <cluster name="RiversEdge Web2" is-device-model="Simulator"
   min-servers="2" max-servers="10" pool="IIS-Win2K New1" vlan="0001"
   tier ="1" fabric="Default Fabric New1"/>
EOF
```

Note: You can only update one object at a time, so embedded objects cannot be updated at the same time you update a top-level object.

5.9.4 Using DCMDelete to delete data from the DCM

To delete data from the DCM, use the DCMDelete command. To delete a customer application with a name of Online Store:

```
DCMDelete (/application[@name="Online Store"])
```

To delete a customer application with an ID of 123:

```
DCMDelete (/application[@id="123"])
```

Note: You cannot delete an object that cannot be identified through a single ID (the primary key is composed from several fields). You can delete all properties for a specific object, but you cannot delete a specific property for an object. Cascading delete is not supported in the DCM query language. If a DCM object that you want to delete has a relationship to other DCM objects, those objects must be deleted first. For example, to delete a router switch, you have to delete the router first and then the corresponding switch.

5.10 Use Jython for string concatenation and evaluation

Jython can now be used to easily concatenate strings. The variables should be populated at the time the assignment is made. Therefore, the assignment should occur just before the string is to be used. The format “(Action or ""\)” should be used to prevent a cryptic error message that occurs if the value of the variable is null.

Example:

```
var commandString ... commandString = Jython((JSHomeDir or ""\) +
"/JumpStart.ManageResource ACTION=" +(Action or ""\)
```

5.11 Using scriptlets in workflows

Workflow scriptlets should be used in place of external script files. Scriptlets can represent Bash, Perl, Expect, or Visual Basic® scripts and the source is embedded within the workflow. Scriptlets are more transparent when viewing the workflow, as all processing is visible, rather than hidden in a separate script. Scriptlets allow for passing of variables and provides three constructs for passing information back to the Tivoli Intelligent Orchestrator deployment engine.

TIOsetVar Allows one to update a variable in the workflow that is used outside of the scriptlet.

TIOthrow Provides support for throwing an error from within the scriptlet.

TIOlog Provides for logging transition information in the deployment engine logs from within the scriptlet.

Example:

```
scriptlet(osLevel) language=bash
target=DCMQuery(/server[@id=$DeviceID]) credentialskey="default" <<EOF
RC="$?"
if [ "$RC" -ne 0 ] ; then
   TIOlog "File not found set return code = $RC"
   TIOsetVar rc "$RC"
else
# The file is present. Check it for the right level
   TIOlog "File found: return code = $RC"
   RC="$?"
fi
return 0
EOF
```

External scripts that are developed and invoked by a workflow need to return meaningful information to the workflow to aid in problem determination.

Scripts that end successfully must exit with a return code of zero. They can optionally pass back a message in stdout indicating that the execution was successful. Scripts that fail must pass back a non-zero return code. This return code should have a unique value for each type of failure. The script should also return back a reason for the failure via a message in stderr. It is not acceptable to only return a error code, and to have the error messages documented in some external document. If possible, the error message should also suggest how the problem could be resolved.

Archived

5.12 Using Java in workflows

If Java needs to be used, then use Java assignment statement rather than coding a Java plug-in. The Java variable can refer to any Java class in the classpath. The following example retrieves the home directory from TIO/TPM using Java:

```
var HomeDirJava = Java[java.lang.System#getProperty("tio.home")]
```

5.13 Supporting documentation for automation packages

Here we discuss the supporting documentation for automation packages.

5.14 Automation package documentation (/doc)

The automation package must have a user's guide or readme file covering the content of the automation package, including steps necessary for the customer to install the driver and customize the solution. The user's guide or readme file needs to be in HTML UTF-8 format so that it can be displayed on the Device Driver "Documentation" tab within the TIO/TPM Administrative Console. The overall purpose of the user's guide is to provide information on the content of the automation package, planning information needed to properly handle the solution covered by the automation package, instructions on how to install the driver, and Data Center configuration information needed in order for the workflows to operate successfully. The following is a recommended outline and suggested topics for the user's guide.

5.15 License (/license)

Each company providing an automation package for IBM Tivoli Intelligent Orchestrator or IBM Tivoli Provisioning Manager needs to clarify how customers can receive the driver and what support they will be granted when encountering problems. The license agreement should contain the "License Use" information and "Terms and Conditions" the company grants to customers. The file or files included in the automation package should be in a readable format: flat file, HTML, or PDF. Multiple files may be included for the different translated languages where support is provided.

Archived

5.16 Repository (/repository)

The repository is for the purpose of storing files and executables that will be needed by the workflows for installing and managing solutions on target systems. By creating a directory within the repository using the company name (also used on the automation package), the files in the directory will not conflict with files from other automation packages. Where possible, the file names should reflect the same naming connotation as the automation package, including the company name, but with scripts and installable files, the files will represent those needed by the solution being installed or managed.

For automation packages where the customer needs to modify silent install or configuration files, those can be described in the configuration section of the readme.html file. Customer modified files not in the automation package will need to be copied to the /repository/MyCompany directory. Once put in the directory, the workflows programmed to reference the /repository/MyCompany directory will be able to pick up the files shipped with the automation package and those files modified by the customer.

5.17 Script and binary files run on the Orchestrator Server (/bin)

Automation packages may have files that will need to be executed on the TIO server. These script and Java files are defined in the tc-driver.xml file and are copied to the $TIO_HOME/bin when the driver is installed. These files need to follow a naming convention to prevent files from being overwritten when subsequent automation packages are installed. The file name should include the company name or the unique prefix used by the files in the company's solution.

5.18 Java plug-ins (/java-plugin)

Embedded workflows, logical operations, and, if needed, Java class calls should be used prior to including a Java plug-in in a workflow. The trend is to do more through logical operations and eventually grandfather the Java plug-in supported in Tivoli Intelligent Orchestrator or Tivoli Provisioning Manager.

Archived

When a Java plug-in is needed, the XML definition for the plug-in is included in the /java-plugin directory. The XML describing the Java program should have a <name> tag that contains the company or solution prefix. The plug-in name should not contain any spaces. Underscores are used between words in a plug-in name. The start of each word should be capitalized. For example:

```
<?xml version="1.0" encoding="UTF-8" ?>
<com.thinkdynamics.kanaha.de.command.DriverType>
<driverTypeDeId>
 com.MyCompany.javaplugin.datacentermodel.GetServerNICIDByMAC
</driverTypeDeId>
<description>Get Server NICID by MAC Address</description>
...
<name>MyCompany_Get_Server_NIC_ID_By_MAC</name>
```

In addition, Java plug-ins should be stored in the companies name space. For example,

```
<?xml version="1.0" encoding="UTF-8" ?>
<com.thinkdynamics.kanaha.de.command.DriverType>
<driverTypeDeId>
   com.MyCompany.javaplugin.GetServerNICIDByMAC
</driverTypeDeId>
<description>Get Server NICID by MAC Address</description>

...
```

5.19 The lib directory (/lib)

The /lib directory in the automation package is used for any Java JAR files shipped with the automation package. This includes when Java plug-ins or classes are included and needed for the successful operation of the workflows in the automation package. When the automation package is loaded into TIO/TPM, the contained JAR files will copied to the TIO/TPM servers $TIO_HOME/drivers/lib directory and made available to the deployment engine class path. The names of the files should contain the company's name or solution prefix.

Archived

5.20 Putting it all together: creating the automation package

The content of the automation package can be created with any preferred editor or development tool. Developing automation packages using APDE will allow for the generation or the automation package from within the environment.

5.21 Naming the automation package

All automation package (.tcdriver) files get copied to the TIO/TPM $TIO_HOME/drivers directory. The $TIO_HOME/tools/tc-driver-manager utility is used to install/uninstall the (.tcdriver) file. Automation package names need to be unique to prevent one package from overlaying another package. The same holds for workflow names and other content within the automation package. Following a standard naming convention will also assist in identifying and differentiating the workflows and device driver files for TIO/TPM.

Use the following convention for Automation package naming:

- The name should contain no spaces.
- Use hyphens (-) to separate words to identify company, product, and resource.
- First letter of each word should be capitalized.
- The automation package name will be used in the tcdriver.xml file between the <driver-name> and </driver-name> tags.

We recommend that the name of the company or unique identifying abbreviation prefix be used at the front of the automation package name. The following are examples based on the above recommendations:

- MyCompany-MyProduct.tcdriver
- MyCompany-MyProduct-Resource.tcdriver

Archived

5.22 Components of the automation package

An automation package, also referred to as a TC-Driver, is an archive file. This archive file contains the collection of components and logical operations that manage the operations of a software entity or physical device. The TC-Driver also includes the components that provide for the configuration of the Data Center Model (DCM) representation of the product within TIO/TPM. Included in the archive file (.tcdriver) are the following directories containing files for supporting the managed software entity or physical device:

- **/TC-INF** Contains the XML manifest file (tc-driver.xml) that defines the contents of the automation package and is used to define the software object or physical device, along with associated workflows, within TIO/TPM DCM.
- **/workflow** Contains the set of workflows needed to deploy, un-deploy, or alter the attributes of a software component or physical device. Workflows may define a logical operation that can be performed on the software object or physical device. These files have a .wkf extension.
- **/doc** Contains documentation for the automation package; like the automation package user's guide (readme.html), but in html UTF-8 format. When the automation package is loaded into TIO/TPM, this document is viewable from the device driver "Documentation" page in the console.
- **/license** Contains a company's "License Agreement". This file provides the "Terms and Conditions" and warranties governing the download and use of the automation package within a customer's environment. The agreement should be in .pdf, .txt, or .html format for easy viewing. There may be multiple files representing different languages and file formats.
- **/repository** This directory contains files that are copied to the target system. These could include scripts, silent install option files, or configuration files that are used within the workflows to perform a specific action. These files should be copied into a company's repository using the tcdriver.xml directives.
- **/bin** This directory contains any script, or executable file that needs to execute on the TIO/TPM server.
- **/java-plugin** This directory contains the XML definitions for Java plug-ins when needed to. Note: The trend is to move away from using Java plug-ins in workflows and rely on DCM interaction and workflows.

Archived

- **/lib** This directory contains any Java JAR files shipped with the automation package. This includes Java plug-ins or classes included for the operation of a workflow. These files will copied to the TIO/TPM /drivers/lib directory and made available to the deployment engine class path.

5.23 Manually creating the automation package

The following example provides a method for performing the automation package generation manually.

The correctly named files must then be placed in the correct directory structure. Using a JAR command, the files in each directory will be included in the archive file representing the automation package.

Create the following directory structure from a given root directory and then copy the appropriate files for each directory into the build directory:

```
root root/TC-INF root/doc root/license root/workflow root/repository
root/bin root/java-plugin root/lib
```

From the root directory, issue the following command to build the MyCompany.tcdriver:

```
jar –cMf  MyCompany.tcdriver  TC-INF  doc  license  workflow
repository  bin  java-plugin  lib.
```

5.24 A sample README file

Recommended outline:

- Introduction
- Components
- Requirements
- Logical Operations
- Planning
- Configuration
- Installation
- Troubleshooting
- Uninstalling the driver
- Limitations
- Trademarks

Archived

Introduction

An overview describing the product or the solution that the automation package implements, including information on the features provided in the automation package. For example, there is a description of the software product or device, along with information describing how the content of the automation package provisions or manages the software product or device.

Components

This section should list the files included in the automation package, along with an explanation of what the files handle. When there are a lot of workflow files, it is helpful to have a description to assist the customer in understanding the relationship between the workflows and the solution being deployed. Table 5-3 shows an example.

Table 5-3 Automation package components

File	Description
bin/myCompany.cmd	Description of executable to run on the TIO server.
repository/MyCompany/filename	Description of files used by the workflow for distributing the workflows. Files will be stored in the %TIO_HOME%\repository\MyCompany directory.
doc/MyCompany_Solution.html	Description of the documentation files for this automation package.
TC-INF/tc-driver.xml	XML manifest file used to load the driver into Tivoli Intelligent Orchestrator.
workflow/myCompany_Solution_Install.wkf	Workflow that will read the solution variables and parameters and then install the solution using the repository files.
workflow/ myCompany_Solution_UnInstall.wkf	Workflow that will read the solution variables and parameters and then uninstall the solution using the repository files.
workflow/ myCompany_Solution_Stop.wkf	Workflow file used to stop the solution.
workflow/ myCompany_Solution_Start.wkf	Workflow file used to start the solution.
workflow/ myCompany_Solution_CheckStatus.wkf	Workflow file used to check the running status of the workflow.

Archived

Requirements

The requirements section defines system requirements, such as resources or dependencies, that must be in place to install a package and work with the product.

The following is an example of this:

- Software
  - Windows 2000/SP4
  - TIO/TPM V2.1/FP1
- Hardware
  - IBM System p™ Server
  - 8 GB of memory

Logical operations

Defines the logical operations that the automation package workflows implement.

For example:

```
Software.Install
Software.Uninstall
Software.Start
Software.Stop
```

Planning

Provides information for planning the deployment of the software product or device into TIO/TPM. The Web site link and support links may also go here.

Configuration

This section defines the DCM objects that will be created during the installation and all parameters associated to the automation package that will be defined.

Here is an example of this section for a software product:

During the installation of this driver, the following Data Center Model objects and their associated variables and parameters as well as the workflows that will be installed in the Device Driver will be created.

Archived

Software product information

Software Product Category MyCompanyName
Software Product MyProductName
Software Product Properties

Name	ProductName
Category	MyCompany
Device Driver	Version: X.XX
Description	Product description
Software Type	
Package Path	Path to the repository for example $TIO_HOME/repository/MyCompany

Software product parameter and values

For example:

installDirectory	C:\Program Files\MyCompany\MyProduct
installExecPath	C:\Cygwin\tmp
licenseFile	MyLicense.lic
installTmpDirectory	/tmp
installResponseFile	MyInstall.iss
uninstallResponseFile	MyUninstall.iss

Device driver information

Device Driver MyCompany-MyProduct

Workflow/Logical operation information

Logical Operation	Workflow Name
Software.CheckStatus	MyCompany_CheckStatus_MyProduct
Software.Install	MyCompany_Install_MyProduct
Software.Uninstall	MyCompany_Uninstall_MyProduct

Information is provided here to describe the parameters that can be modified to meet the demands of each unique installation environment. Files not included in the automation package that may need to be copied to the repository should be defined here. Any other information regarding accepted modifications or creation of files that are not part of the automation package should be defined here.

Installation

Perform the following actions to install the automation package:

1. Copy the <automation package name>.tcdriver to the $TIO_HOME/drivers directory.
2. Change the directory to $TIO_HOME/tools.

Archived

3. Run the following command from the $TIO_HOME/tools directory

   On Windows:

   ```
   tc-driver-manager.cmd installDriver <automation package name>
   ```

   On UNIX:

   ```
   tc-driver-manager.sh installDriver <automation package name>
   ```

Once the automation package has been installed, and the device definition and DCM have been created, variables and parameters can be customized for the environment.

Troubleshooting and problem determination

This section should contain information pertinent to trouble-shooting problems with the particular automation package. This can include information on where to look for errors within the Tivoli Intelligent Orchestrator logs.

For example:

- For all errors encountered during the installation of the drivername tcdriver, refer to the following log file:

  ```
  $TIO_LOGS/tdcrivermanager.log
  ```

- For all errors associated with the execution of the workflows, refer to the following log files:

  ```
  $TIO_LOGS/deploymentengine and $TIO_LOGS/j2ee console.log
  ```

By default, debug messages are not written to the log files. To increase the log level, change the settings in $TIO_HOME/config/log4j.prop:

```
log4j.appender.errorfile.threshold=debug
log4j.appender.consolefile.threshold=debug
```

In addition to the logs, workflows in the automation package should be written to handle error checking and log information during execution. This section should include assistance and debugging information for the error conditions and codes being logged within workflows contained in the automation package.

Archived

Uninstalling the automation package

This section should contain information on the steps needed to uninstall the automation package, for example, if there are actions needed to remove the solution from a server or software stack prior to the automation package being uninstalled. The following steps will uninstall the automation package from a TIO server:

Change the directory to $TIO_HOME/tools:

- On Windows:

```
tc-driver-manager.cmd uninstallDriver <automation package name>
```

- On UNIX:

```
tc-driver-manager.sh uninstallDriver <automation package name>
```

Limitations

Provide license and expressed warranty and liability clauses here.

Trademarks and service marks

Corporate trademark and copyright clauses here.

5.25 The XML Manifest file (\TC-INF directory)

The tc-driver.xml file located in the TC-INF directory is an XML representation of the automation package content, the relationship of the content, and the relationship of this automation package to other packages or device drivers loaded into TIO/TPM. The tc-driver.xml file is used to configure/deconfigure the driver when it is installed/uninstalled in TIO/TPM.

> **Note:** If using the Automation Package Development Environment, the XML Manifest file is automatically generated. This section outlines the sections of this document for reference, or if a developer chooses to develop the Automation Package in a different development tool.

This XML file needs to have XML stanzas to represent the device model, DCM Object, logical operations, workflows, documentation, repository files, external scripts, and any other file included in the automation package. XML statements are also needed to identify the relationship between the workflows and the DCM objects. Portions of this file are automatically generated when using the APDE to develop an automation package.

Archived

Automation package solution name

The manifest file starts with the version of the template and the format of the tc-driver, followed by the automation package name and version. Note that the character set encoding needs to be set to UTF-8:

```
<?xml version="1.0" encoding="UTF-8"?>
<tc-driver>
 <tc-driver-format>2.0</tc-driver-format>
 <driver-name>MyCompany-Solution</driver-name>
 <version>1.2.1</version>
<description>My Product Description</description>
```

Dependencies

Dependencies on other device drivers or automation packages should be referenced in XML without including the content of those packages. For example, the following stanza declares that this automation package has a dependency on the Core device drivers within Tivoli Intelligent Orchestration and on the Image-Software-Stack:

```
<dependencies>  <dependency name="Core"/>  </dependencies>
```

Depending on the software product or physical device represented by the automation package, the manifest tc-driver.xml files needs to have the following sections.

Automation package documentation

This is the file located in the /doc directory of the automation package and is installed when the TC-Driver is installed in TIO/TPM. This document is viewable from the Device Driver documentation tab in the TIO/TPM console. Also, on import, the file name is treated as case sensitive and the XML stanza needs to match the exact file name for the file in the /doc directory. Otherwise, an error will be encountered.

Use the following convention for documentation naming and creation:

- Match the name of the automation package.
- The document is in HTML format with UTF-8 encoding.

The following example is based on the above convention:

```
<documentation location="doc/MYCompany_Solution.html" />
```

Archived

Automation package class files

The actions section lists all of the separate classes that are necessary to install the components of the automation package:

```
<property name="tc.pkg"
location="com.thinkdynamics.kanaha.tcdrivermanager.action"
/>

 <actions>
<action name="command" class="${tc.pkg}.SimpleCommandActions" />
<action name="copy-file" class="${tc.pkg}.CopyFileActions" />
<action name="java-plugin" class="${tc.pkg}.JavaPluginActions" />
<action name="workflow" class="${tc.pkg}.TxtWorkflowAction" />
 </actions>
```

Automation package items

This section lists all the items to be installed. The order that the <items> are listed in your XML file is important and needs to follow a specific pattern. This pattern provides the order that each item will be installed and uninstalled. Flat files installed first, followed by Java plug-ins, and workflows. Uninstallation of the automation package follows a reverse order: workflows are uninstalled first, followed by Java plug-ins, and all flat files.

Items section for creating repository files

This section is needed when there are files to be included in the /repository directory for the driver. During installation the MyCompany_Solution directory in $TIO_HOME/repository/ will be created and files will be copied to that location. The <items> tag can contain multiple <item> tags.

The following example copies the SampleFile.sh from the automation package to the $TIO_HOME/repository/MyCompany_Solution directory and changes the mode of the file to allow execution:

```
<item name="repository/SampleFile.sh" action="copy-file"> <param
name="dest.path"
value="${tc.home}/repository/MyCompany_Solution/SampleFile.sh"/> <param
name="chmod" value="755"/>  </item>
```

Items section for listing workflow files

The following item declarations identify the workflows included in the package that are in the /workflow directory. A stanza is needed for each workflow. You should list the workflows first that are referenced by the other workflows included in the automation package:

```
<item name="workflow/MyCompany_Solution_CheckStatus.wkf"
action="workflow"><param name="editable" value="false" /> </item>
<item name="workflow/MyCompany_Solution_Install.wkf" action="workflow">
<param name="editable" value="false" />  </item>
<item name="workflow/MyCompany_Solution_Start.wkf" action="workflow">
<param name="editable" value="false" />  </item>
<item name="workflow/MyCompany_Solution_Stop.wkf" action="workflow">
<param name="editable" value="false" />  </item>
<item name="workflow/MyCompany_Solution_Uninstall.wkf"
action="workflow">  <param name="editable" value="false" />  </item>
```

Automation package device model

This section lists all the items to be installed with the automation package. The items in this stanza will be listed in the TIO/TPM console under the System Configuration Device Drivers tab.

> **Note:** It is not necessary to include all the workflow files that may have been included in the <items> section. In the items section, all the files in the automation package need to be declared. Under device-models, only the direct logical operations or main workflows should be listed.

This example creates the MyCompany_Solution device model under the Device Drivers Software Products category.

```
<device-models>
   <device-model name="MyCompany_Solution" category="Software
Products">
      <workflow name="MyCompany_Solution_CheckStatus" />
      <workflow name="MyCompany_Solution_Install" />
      <workflow name="MyCompany_Solution_Start" />
      <workflow name="MyCompany_Solution_Stop" />
      <workflow name="MyCompany_Solution_Uninstall" />
   </device-model>
</device-models>
```

Archived

Automation Package DCM Software Object Definition

This section provides the XML line items for creating a DCM software object as well as any parameters that can be accessed by the workflow. We recommend that all software products be created under a software category that represents the solutions from the company or the type of solution. The software category name cannot be the same as the name used for the “device-model name” above. If they are the same name, workflow executions may encounter an unrelated Software Object/Device Object error on execution. The software product stanza needs to include the software category, solution name, directory (package_path) for the source files, and the parameters and properties that define the software product:

```
<software-category name="MyCompany" />
<software>
   name=" MyCompany " description=" MyProduct descriptoin"
service="false"
   type="SOFTWARE" version="n.n" category=" MyCompany "
   package_path="/TIO_HOME/repository/MyCompany_Solution"


<use-workflow workflow-id ="MyCompany_Solution_CheckStatus" />
<use-workflow workflow-id ="MyCompany_Solution_Install" />
<use-workflow workflow-id ="MyCompany_Solution_Start" />
<use-workflow workflow-id ="MyCompany_Solution_Stop" />
<use-workflow workflow-id ="MyCompany_Solution_Uninstall" />
<parameter name="InstallDirectory" value="my install directory " />
<parameter name="CompanyName" value="My Company Name" />
<property component="DEPLOYMENT_ENGINE" name="Installed Server Status"
value="uninstalled" />
<property component="DEPLOYMENT_ENGINE" name="Uninstall Options File"
value="\MyCompany\_uninstall" />
</software>
```

On the software product definition, the following items are needed to set up the software product DCM Object definition correctly:

<use-workflow workflow-id>

Required for each workflow that implements a logical operation and would be called directly as part of the automatic provisioning of the software product. This XML line will cause the workflow to be listed on the workflows tab of the DCM Software Product definition. Only the workflows for logical operations like install, uninstall, start, stop, and check status should be included

<parameter name>

Provided for each customer customizable parameter for the software product. This will create the entries with default values on the “General” tab of the DCM object software product definition. The users guide provided with the product should clearly specify how these parameters need to be customized in order for the solution to be automatically deployed using the workflows in the automation package.

<property component>

Used to declare the default variables for the software product where the values need to be called and possibly updated between execution of workflows. DCMQuery calls would be used within the workflows to retrieve and update these variables. The user’s guide for the automation package should clearly identify how the values for these variables need to be modified for the successful execution of the workflows in the customer’s environment.

Automation package post install workflow

The post install workflow section of the tcdriver.xml file names a workflow along with its parameters that will be executed after all the items defined in the tc-driver.xml file are installed and defined in the data center model. The named workflow could be one that was installed by the current automation package, or one that had been previously installed. Use the post-install workflow to complete updating the DCM with specific system variables:

```
<post-install-workflow name="MyCompany_Install_Finalization ">
   <param name="command variable name" value="value>
</post-install-workflow>
```

Sample automation package tc-driver.xml file

The following is an example of a tc-driver.xml file that covers all of the previously defined XML stanzas:

```
<?xml version="1.0" encoding="UTF-8"?>

<tc-driver>
   <tc-driver-format>2.0</tc-driver-format>
   <driver-name>MyCompany_Solution</driver-name>
   <version>2.1.0</version>
   <description>Sample MyCompany tcdriver file.</description>
<dependencies>
   <dependency name="Core"/>
</dependencies>
<documentation location="doc/MYCompany_Solution.html" />
```

```
<property name="tc.pkg"
location="com.thinkdynamics.kanaha.tcdrivermanager.action" />
<actions>
   <action name="command" class="${tc.pkg}.SimpleCommandActions" />
   <action name="copy-file" class="${tc.pkg}.CopyFileActions" />
   <action name="java-plugin" class="${tc.pkg}.JavaPluginActions" />
   <action name="workflow" class="${tc.pkg}.TxtWorkflowAction" />
 </actions>
<items>
   <item name="repository/SampleFile.sh" action="copy-file">
      <param
name="dest.path"value="${tc.home}/repository/MyCompany_Solution/SampleF
ile.sh"/><param name="chmod" value="666"/>
 </item>
   <item name="workflow/MyCompany_Solution_CheckStatus.wkf"
      action="workflow"><param name="editable" value="false" />
   </item>
   <item name="workflow/MyCompany_Solution_Install.wkf"
      action="workflow"> <param name="editable" value="false" />
   </item>
   <item name="workflow/MyCompany_Solution_Start.wkf"
      action="workflow"> <param name="editable" value="false" />
   </item>
<item name="workflow/MyCompany_Solution_Stop.wkf" action="workflow">
<param name="editable" value="false" /> </item><item
name="workflow/MyCompany_Solution_Uninstall.wkf" action="workflow">
<param name="editable" value="false" /> </item>
<device-models>
   <device-model name="MyCompany_Solution" category="Software
Products">
      <workflow name="MyCompany_Solution_CheckStatus" />
      <workflow name="MyCompany_Solution_Install" />
      <workflow name="MyCompany_Solution_Start" />
      <workflow name="MyCompany_Solution_Stop" />
      <workflow name="MyCompany_Solution_Uninstall" />
   </device-model>
</device-models>

<software-category name="MyCompany" />

<software name="MyCompany"
   description="MyCompany Network Configuration" service="false"
   type="SOFTWARE" version="3.0"
   package_path="/TIO_HOME/repository/MyCompany_Solution"
   category="MyCompany">
```

```
   <use-workflow workflow-id ="MyCompany_Solution_CheckStatus" />
   <use-workflow workflow-id ="MyCompany_Solution_Install" />
   <use-workflow workflow-id ="MyCompany_Solution_Start" />
   <use-workflow workflow-id ="MyCompany_Solution_Stop" />
   <use-workflow workflow-id ="MyCompany_Solution_Uninstall" />
   <parameter name="InstallDirectory" value="MyCompany" />
   <parameter name="CompanyName" value="My Company" />
   <property component="DEPLOYMENT_ENGINE"
      name="Installed Server Status" value="uninstalled" />
   <property component="DEPLOYMENT_ENGINE"
      name="Uninstall Options File" value="\MyCompany\_uninstall" />
</software>

<post-install-workflow name="MyCompany_Install_Finalization ">
   <param name="command variable name" value="value>
</post-install-workflow>
</tc-driver>
```

Archived

Archived

6

Developing workflows

In this chapter, we show some of the techniques for developing workflows. We approach this task with the perspective of someone who needs to automate the installation of a software product, but the techniques should be easily applied by someone who needs to provision hardware or for a product that has both software and hardware components.

We begin with a discussion of workflows in the context of IBM Tivoli Intelligent Orchestrator (what are workflows and why do we need them?). We then describe the process of how to create a workflow, beginning with the establishment of scope and strategy, and continue on to the practical details of workflow coding.

Finally, we give some suggestions on how to improve your workflows to make them robust and extensible in design.

© Copyright IBM Corp. 2004, 2006. All rights reserved.

6.1 Workflows overview from a developer's perspective

A workflow is simply an automation of an existing manual process or a step of a process. While this may seem obvious, it underscores the fact that without a manual process, there can be no automated process. And, if the manual process is poorly designed, then the automated process is likely to exaggerate the flaws in the manual process.

As an analogy to software development, a workflow could be thought of as a *class* which models a real-world manual provisioning process. The class is *instantiated* as an object and its methods get invoked each time a logical operation has a need for the execution of the manual process in the Data Center.

In practical terms, when developing a workflow, we are trying to model something that exists in the real world in the TIO/TPM environment. The elegance of our manual process and the fidelity with which we are able to model it within TIO/TPM will determine our success in implementing orchestration and provisioning.

6.1.1 Workflows in the context of TIO/TPM

Workflows interact with a very large, all encompassing framework in IBM Tivoli Intelligent Orchestrator, known as the Data Center Model (DCM).

TIO/TPM objects

In TIO/TPM, workflows interact with a framework of objects that includes, among other things, owners and users of applications (called *customers*), applications, clusters, servers, server pools, software objects, and software stacks, just to mention a few.

These objects are hierarchical in nature. At the top are *customers*, which is a collective group of those using an application. An *application* is an enterprise tool, such as ERP, SCM, Human Resources, or e-business, which is deployed on a cluster. A *cluster* is a collection of one or more identical servers hosting the application components. *Servers* may also be included in a *pool*, which is a collection of available servers waiting to be provisioned into a cluster. Servers in a cluster have identical software stacks installed. A *software stack* is a group of software objects that are deployed on a server.

A *software object* is an TIO/TPM representation of a deployable piece of software, such as the Linux operating system, IBM WebSphere Application Server, or a solution provided by an Independent Software Vendor (ISV).

Logical operations

Similar to objects in object oriented software development, TIO/TPM objects have both “fields” and “methods”. We will discuss fields later when we discuss attributes, properties, and variables. Methods for an TIO/TPM object are known as *logical operations*.

An example is a commonly used logical operation for a cluster object called cluster.AddServer. This logical operation is triggered by the provisioning process when performance levels for the cluster have fallen below the acceptable level established by the TIO/TPM policies. What does cluster.AddServer do? Obviously, it adds a server to a cluster. But what kind of server should be provisioned and where should it be acquired?

To answer this, you must understand that logical operations are similar to abstract methods in Java, which means that the logical operation must be implemented by some coding that is appropriate for the instantiated object. In our example, the class of object is a cluster and perhaps the instantiated object is named “SUSE Linux Human Resource Servers”. When an object, such as a cluster, is set up, the developer must implement each of the necessary logical operations. He or she does this by assigning a workflow to a logical operation for that object. It is a workflow that implements a logical operation.

Table 6-1 shows these relationships in a more structured way.

Table 6-1 DCM objects and Java code analogies

Java	TIO/TPM	Example
Class	Device model	Software
Method	Logical operation	Software.Install
Object	Device	SUSE Linux HRServer
Method implementation	Workflow	SUSE.v2.InstallOnIntel

Workflows

While a workflow can exist independently of an TIO/TPM object, when it is attached to an object, it is done so as an implementation of a logical operation for a particular object.

Continuing our example, as seen in Figure 6-1, when the cluster.AddServer logical operation of our cluster of SUSE Linux servers has been triggered, a custom workflow is run, which has been tailored for SUSE Linux servers running the appropriate software stack to service the particular application. This workflow will eventually find the assigned server for the operation in the resource pool. This operation is performed by the RM Allocate Server Java plug-in.

Figure 6-1 Add_Server workflow example

Before the workflow may add the server to our cluster, the workflow must determine which software stack is needed for the cluster. As seen in Figure 6-1, this is handled by the transition FindDeviceSoftwareStack Java plug-in. The workflow then proceeds by managing the allocation of a new IP address to the server, before it finally initiates another logical operation for the software stack called SoftwareStack.Install.

The SoftwareStack.Install logical operation is also an “abstract” method (the EJB Proxy Class Name is com.thinkdynamics.ejb.dcm.interaction.SoftwareStackComponentProxy and the EJB Proxy Class Method is install), which means that it must be implemented by another workflow that has been developed for installing the particular software stack on the particular instantiated server.

Attributes and variables for TIO/TPM objects

We said objects in TIO/TPM have “fields”, known as attributes and variables. When setting up a new object, such as a software object, you must define certain attributes, such as the object name. Different types of objects have different attributes. You may add your own variables and provide them with values. These variables then become a placeholder for values to be referenced by workflows.

As an example, if you were setting up a software object to represent IBM DB2 Universal Database Server, you might set up a variable called “TCP_PORT” and provide it a value of 50000. A workflow that installs the DB2 server will then know which TCP/IP port to use for configuring the server. Other useful properties in this case might be an “InstallDir” variable with a value of c:/ibm/sqllib and a “binDir” variable with a value of c:/ibm/sqllib/bin.

As of TIO/TPM Version 3.1, a new way of managing variables used specifically for software installation and operation has been introduced. Software Resource Templates (SRTs) are basically a set of custom-defined key-value pairs that may be recognized by workflow implementing software related LDOs, such as SoftwareInstallable.Install. For variables that related specifically to software installation and deployment, we recommend that you use SRTs rather than variables. The variables should be used to reflect TIO implementation specifics, such as directory paths, names, and other control information used to control the behavior of the DCM objects rather than software installation.

Workflows and the Data Center Model

To summarize, implementing IBM Tivoli Intelligent Orchestrator requires the following steps:

1. The creation of objects in the Data Center Model that represent the real Data Center.
2. Defining attributes, properties, and variables of the DCM objects that describe the object characteristics and providing them with appropriate values.
3. Developing workflows that mimic manual processes in the data center, such as provisioning a server.

Archived

For Independent Software Vendors (ISVs), whose solutions are software based, the focus should be on creating a software object that contains

1. Attributes, properties and variables that represents its state when implemented
2. Workflows that represent how it is processed in a typical data center.

Workflows might include implementations of the software related logical operations, such as Software.Install, Software.Uninstall, Software.Start. Software.Stop, SoftwareStack.Install, and so on.

6.1.2 Transitions

Each workflow is made up by control logic and one or more transitions, which is the only type of object in the TIO/TPM environment that actually performs actions. Which actions to perform is determined by the standard task object referenced by a transition. The only objects that may be referenced are:

Java programs Java plug-ins

Compiled Java code that performs generic, or very specific, functions, such as passing parameters on to the shell interpreter of the operating platform, issue low-level operating system commands, facilitate communication, manipulation of the TIO/TPM repository, and so on.

Whenever a Java plug-in is instantiated by being referenced from a workflow, a simple command is created behind the scenes. This simple command is deleted automatically when the referencing workflow is deleted.

Workflows

A transition in one workflow may reference another workflow. This allows the developer to define “sets” of transitions that perform a predefined series of tasks that can, and should, be reused by referencing the same workflow from other transitions.

Logical operations

Logical operations may be regarded as “super-sets” of tasks in the sense that, when used, TIO/TPM consults its database to identify the specific workflow that implements the particular logical operation for the device that is the target of the operation.

It is like telling your children that you will buy pizza for dinner. The logical operation BuyPizza does not specify if they will get cheese or pepperoni, but the selection of the

first pizza parlor you meet will determine this. If you (or your children) have a particular pizza preference, you would implement the Logical Operation: BuyPizza through a customized workflow named BuyCheesePizza, thus making sure that you get the topping you anticipate.

Since both logical operations and workflows may be considered as containers, or macro commands, implementing a specific series of transitions, the fact of the matter is that the only type of object that does any practical work is the Java Programs and plug-ins, and since these are invoked from within the WebSphere environment that hosts the IBM Tivoli Intelligent ThinkDynamic Orchestrator, they are all implemented as Java plug-ins. Having said this, it should be noted that a huge set of Java plug-ins are provided by TIO/TPM which will allow for, for example:

- Communication to external systems and devices
- Local and remote execution of shell commands, scripts, and executables
- Retrieval and storage of DCM related data
- A multitude of generic device and implementation specific functions

When developing workflows, we highly recommend using the provided existing logical operations and the provided workflows and Java plug-ins as much possible.

6.1.3 Workflow development challenges

Developing a workflow is a somewhat awkward task compared to straight-forward application development. The nature of the workflow is such that the developer needs to provide various bits and pieces that fits into the big picture in various different places. By allowing these to be customized to fit into a customer environment, and combining them with standard components provided by Tivoli and other vendors, a complete set of automation functions can be compiled, providing all the automated capabilities to provision (deploy, install, customize, and operate) and de-provision (remove) any third-party solution in the IBM Tivoli Intelligent ThinkDynamic Orchestrator environment.

The functions that the developer needs to provide through the workflow is best identified by referencing the Installation, Customization, and Operations Guides of any solution. These documents will hopefully provide descriptions of a set of structured tasks needed to perform the various high-level functions of the provisioning. The main goal of the developer is to provide skeleton, and operational, bits of code that may be customized for a given customer environment in order to automate all the provisioning and de-provisioning tasks.

Since the developer cannot make any presumptions, apart from the fact that IBM Intelligent ThinkDynamic Orchestrator is being used to facilitate the provisioning processes, the main challenges are:

- To keep an open mind during the design and development process
- Reuse as many standard components as possible
- Avoid hardcoding names, values, or processes that may not be generic in nature

all because the bits and pieces have to be embedded into a bigger picture.

Figure 6-2 illustrates the fact that an automation task for any device or software product is compiled from bits and pieces (workflows, simple commands, and logical operation implementations) from various sources and that they ultimately have to support the customer's process and not have one dictated or imposed on them.

Figure 6-2 Orchestration Package development scope

Archived

6.1.4 Workflow prerequisites

IBM Tivoli Intelligent Orchestrator is only one part of an organization's data center operations. As such, certain conditions are essential for its successful implementation.

A well-functioning existing process must be in place

The On Demand operating environment has three aspects: integration, virtualization, and automation. IBM Tivoli Intelligent Orchestrator focuses on virtualization and automation, and it assumes that the process of integration has been addressed.

Workflows have the ultimate purpose of provisioning resources. What IBM Tivoli Intelligent Orchestrator offers is a mechanism to *virtualize* and *automate* the provisioning process, not to *integrate* the process.

This may seem obvious, but it is often assumed that a product as powerful as TIO/TPM can do all things. The case may be that an existing provisioning process is inadequate because it is poorly implemented and not integrated. If this is the case, then automating the existing process without changes using TIO/TPM is likely to only compound the inadequacies.

Install/Uninstall process must be more robust

A manual provisioning process is labor intensive. For example, the manual setup and installation of a new server in a cluster may take one or two days time. Because of this, the manual process may be run infrequently. But IBM Tivoli Intelligent Orchestrator automates the process, removing much or all of the labor resources involved in provisioning. Because of this, a manual process that is run one or two times a year may be run on even a daily basis.

This has ramifications for the robustness of the provisioning process. Implementers of TIO/TPM workflows must address issues such as preparation and cleanup, authorization, and logging of the process, and any side effects generated by the provisioning process. How will a weekly or daily provisioning process affect other processes in the data center and the larger IT organization?

To summarize, workflow developers must assess the adequacy and robustness of existing provisioning processes before designing TIO/TPM workflows.

Recovery actions must be defined

Workflows must address what will happen if the process fails in some way. In previous versions of TIO/TPM, this was known as *recovery actions*. from version 2 and forward, recovery actions have been replaced by the try-catch-endtry constructs combined with the ability to throw exceptions from a workflow. This

enables you to identify problems, and develop code fragments that will be invoked specifically to handle specific error situations.

An example of throw try-catch-endtry workflow logic is shown below in Example 6-1. In this example, the recovery action is limited to setting the “failed” attribute of the server to `true` and to place the server in a maintenance state.

Example 6-1 Example of recovery actions implemented by try-catch-endtry

```
## this workflow illustrates how to use throw try-catch-endtry to
implement
## recovery actions

workflow LAB9  LocaleInsensitive
var wrkflow="LAB9"
var msg=Jython["Workflow " + wrkflow + " failed: "]

## set up a loop to try three times
var retries="3"
var attempt="1"
var retry="true"

while Jython[retry == "true"] do

  ##  try a command, and catch exceptions
  try
    #  do something meaningfull - if it fails, throw an exception
    noop
    throw FAILED "Houston, we have a problem!"
  catch FAILED exceptionMsg
    ##  perform cleanup actions to prepare for
    if Jython[int(attempt) <= int(retries)] then
      log warning Jython["attempt: " + attempt + "  " + msg +
exceptionMsg]
    else
      var newMsg=Jython["operation failed after " + retries +
"attempts"]
      throw QUIT newMsg
      retry="false"
    endif
  catchall
    ## for all other exceptions - send them back to the caller
    rethrow
  endtry

  attempt=Jython[int(attempt) + 1]
```

```
done
```

6.2 Preparing to write a workflow

This section describe some of the preparation tasks that need to take place when developing workflows. Additional information on this matter can be found in the *Workflows Developer's Guide* manual. The IBM Orchestration and Provisioning Automation Library is available at the following Web site:

http://www.developer.ibm.com/tivoli/workflow.html

6.2.1 Establish workflow scope and review workflow objects

The process begins by establishing what the workflow is intended to accomplish and continues with a study of the context within the workflow will be run.

Workflow scope

As previously stated, workflows may be implementations of logical operations. A logical operation is generic by nature. For example, Cluster.AddServer does not define which cluster or type of cluster for which it is run. Those questions are all left up to the implementing workflow.

A workflow should be much more specific, which means that many more questions are answered when the workflow is developed and less questions are determined when the workflow is executed. Any questions that are left unanswered until execution should be defined in variables and well documented.

Review workflow objects

The workflow developer must then identify and analyze all artifacts that will be used in the workflow. For example, if the workflow were intended to install software on a server, the following might be important artifacts:

- Target server
- Installation software
- Pool from which target server is provisioned
- Cluster to which target server is deployed
- VLANs involved
- Network interface card of target server
- Repository server for install software (if any)
- Model response files for automated software installation

Archived

After identifying these items, the workflow developer must then:

1. Figure out how they are defined within TIO/TPM
2. Identify the variables each object has that are relevant to the workflow
3. Understand how to access the variables

Defining the scope and identifying objects and variables provides an analysis that is essential before any meaningful design can occur.

6.2.2 Detail the workflow operations and their order

As stated previously, developing a good workflow requires having a good manual process defined. Detailing the operations is the process of listing the steps in the manual process, with possible modifications, keeping in mind that those steps will be executed by TIO/TPM.

Workflow operations example: RPM Install

Table 6-2 shows operations that would install a software product onto a Linux server using the Red Hat Package Manager (RPM). The steps are taken from the RPM Install workflow, which is an implementation of the Software.Install logical operation.

Table 6-2 RPM software install workflow operations

Operations	Description
Get Software Product Attributes	Which Software Product is to be installed?
Get Server IP Address	What is the IP address of the server where the software is to be installed?
Get Repository ServerID	On what server is our install file located?
Get Package Name	What is the RPM package name?
Copy Package	Copy the RPM file from the install server.
Execute RPM Install	Run the RPM install command.

Workflow operations example: remove server from cluster

Table 6-3 on page 187 shows operations that would remove a server from a cluster. The steps are taken from the Cluster.RemoveServer.Template workflow, which is an implementation example for the Cluster.RemoveServer logical operation.

Archived

Table 6-3 Cluster.RemoveServer.Template workflow operations

Operations	Description
Get Current Request ID	Data gathering phase.
Get Cluster Attributes	Data gathering phase.
Get Pool Attributes	Data gathering phase.
Get VLAN Attributes	Data gathering phase.
Prerequisite checking	Using information from the data gathering phase, validate that the key components in the workflow are available for use. In this workflow, an example would be the software server responsible for un-installing the Software Products defined in the stack, and the existence of the ACL IDs to enable on the target firewall when the server is moved back to the management VLAN.
RM Choose Server for Removal	Resource Manager to select a random allocated server from the application cluster (dedicated servers will not be affected).
Get Server Attributes	Get the server attributes, such as NIC ID, to use in subsequent transitions.
Get NIC Attributes	Get the properties of the NIC.
Get Real IP by Network Interface	Get the IP address of the NIC.
Get Software Stack for Device	Determine the software stack associated with the server.
LoadBalancer.Rmv Real IP from Virtual IP	Remove the server's real IP addresses from the VIP.
SoftwareStack.Uninstall	Remove the Software Products associated with the stack.
RM Allocate IP address	Get the next available IP address for the subnet of the management VLAN.
IP System.Add IP Address	Add IP to DCM.
IPSystem. Apply Routing Table	Update the routing table.
Switch.Move Port to VLAN	Move the server from a production VLAN to a management VLAN.
SNMP Ping	Verify that un-managed interface can still be reached.

Archived

Operations	Description
Update SAPs for Group of Servers	Update the service access points for the server.

6.2.3 Create a new workflow or create one from a template

When using the APDE environment provided by IBM Tivoli Intelligent Orchestrator, you can create a new workflow by clicking **Add Workflow** on the Workflows page, or you can create a new workflow by copying/cloning an existing one. Workflows can also be developed using XML and then imported into IBM Tivoli Intelligent Orchestrator.

Adding a workflow manually

To manually create a workflow:

1. Open the Configuration tab in the navigation pane, and open the **Workflows** tab.
2. Click **Edit** → **Add Workflow** in the upper right. A form will open for you (Figure 6-3).

Figure 6-3 Add workflow

3. Open the **Edit** icon on the workflow line, select **Properties**, and change the Name and Locale as necessary.
4. In the Implements section, you may assign choose a logical device operation for the new workflow. You should select a logical operation that implements a similar underlying function as the new workflow.
5. Click **Save**.

6. From the menu bar at the top of the WebUI, select **Compile** → **Compile** to save the new workflow.

Cloning an existing workflow

To create a new workflow using a copy of an existing workflow:

1. Open the Configuration tab in the navigation pane, and open the **Workflows** tab.
2. Define the search criteria to easily find the workflow you want to clone and press **Find** or press **List All**.
3. From the resulting list of workflows, open the one you want to copy.
4. Click the **Edit** icon to the far right on the workflow line, select **Properties**, and provide a new name for the workflow. Press **Save**.
5. From the menu bar at the top if the WebUI, select **Compile** → **Compile** to save the new workflow.

Either way, you are left with a new workflow that is ready for the customization to your workflow requirements.

6.2.4 Add new transitions to the workflow

Transitions are steps in a workflow that are needed to run in sequential order to complete the workflow. For a given workflow, you may need to modify, add, or remove transition steps as required, or create a new transition for your workflow using the same steps.

To add a transition to a workflow:

1. In the navigation tree, identify the workflow, command, Java plug-in, or logical operation you want to add as a new transition to your workflow, and then drag it onto your workflows editing area. Your cursor will change its shape to include a “+” character when the transition area is ready to accept the item (see Figure 6-4 on page 190). Release the mouse button to add the item as a new transition. You can also copy transitions from already existing workflows by dragging them either from the navigation tree or from another browser window onto your workflows transition area. In this case, a copy of the selected transition is created in your workflow.
2. If necessary, use the same drag-and-drop facility to rearrange the transitions within your workflow.

Archived

Figure 6-4 Dragging a simple command onto the Transition Area

6.2.5 Map the variables

The next step is to determine the variables that will be passed from one transition to another within the workflow. Transitions have three types of variables: *input*, *output*, and *local*. The purpose of this step is designing the workflows so that an output variable of one transition is used to provide the input value of a subsequent transition. The goal is to ensure that all input variables of all transitions in the workflow have been given proper values, either as a constant (default) value or as the output value of a previous transition. Local variables have a scope limited to the transition itself, so they are not considered in this step.

The invocation method (this)

When creating variables during a transition mapping, IBM Tivoli Intelligent Orchestrator will create a variable in the local variables of the **`this`** (invocation) method. The name will default to the name of the variable that is being mapped. Often, this name is not meaningful in the context in which it is being used. As a standard, this name should be changed to reflect how the variable will be used. You can change the name of the generated variable by accessing the local variables under the **`this`** method in the GUI workflow builder.

Process to map variables

1. Identify a transition that has input variables that requires mapping. An input variable will be passed into the workflow when it is called.
2. Provide a value for the variable in one of two ways:
   a. Provide a constant (default) value for the variable by clicking the **ABC** icon and entering a default value.

b. Drag an appropriate output variable from a previous transition onto the input variable that requires mapping. You can drag both existing variables and unmapped variables displayed as question marks. If you drag an unmapped variable, a new variable will be created automatically. You can also use local variables to map values between transitions.

3. Repeat steps 1 and 2 for every transition that requires variable mapping.
4. Check the **Compile after save** check box at the bottom of the transaction list.
5. Click **Save Changes**.

> **Important:** For a workflow to run successfully, all input variables must be mapped correctly.

6.2.6 Validate and update the recovery actions

If necessary, create an appropriate recovery script for the workflow. A recovery script does not fix the problem that has occurred in the workflow, but is considered an appropriate action to take while the reason for the failure can be determined. For example, if you are running a workflow that adds a server to a cluster, then an appropriate recovery action is to set the status of the server to `Failed` and put the server in maintenance. When the server is in maintenance, any future deployment requests will not be able to access the resource.

6.2.7 Validate and test the new workflow

Run the workflow in a development environment to ensure that the end result is what was intended. The test case should include intended function and unintended function (error case).

6.3 Workflow elements

A TIO/TPM workflow may contain the elements shown in Figure 6-5:

Figure 6-5 Workflow elements

Variable declaration

In workflows, all variables have to be declared prior to use. Variables can represent numeric values, characters, character strings, or memory addresses. All variables are treated as strings.

> **Note:** Workflow parameters (input or output) are will be implicitly declared as variables when the workflow is instantiated at runtime

Perform the following actions to declare a new variable to your workflow:

Workflow composer:

From the Workflow Element list at the top of the Workflow Composer WebUI, drag the variable workflow element icon var to the appropriate line in your workflow, and the following code sample is inserted into your workflow:

Assign attributes (name and encryption) to the variable by double-clicking of the variable name, and fill in the variable attributes dialog:

APDE:

Place the cursor on the line in your workflow where you want the conditional processing to take place, and type in the line:

```
var [encrypted] <variable_name>
```

Optionally, you can assign a value to the variable at declaration time by adding an assignment immediately after the variable declaration, as described in “Assigning values” on page 192.

Assigning values

In workflows, variables may be assigned values by means of an assignment. An assignment represents an expression used to assign a value to a variable. Variable values are always treated as text. Please note that workflow parameters (input or output) are will be implicitly declared as variables when the workflow is instantiated at run time.

Archived

Perform the following actions to declare assign values to variables in your workflow:

Workflow composer:

From the Workflow Element list at the top of the Workflow Composer WebUI, drag the assignment workflow element icon to the appropriate line in your workflow, and the following code sample is inserted into your workflow:

```
var var1
```

Assign attributes (name and encryption) to the variable by double-clicking of the variable name, and fill in the variable attributes dialog:

APDE:

Place the cursor on the line in your workflow where you want the conditional processing to take place, and type in the line:

```
var [encrypted] <variable_name>
```

Optionally, you can assign a value to the variable at declaration time by adding an assignment immediately after the variable declaration, as described in “Assigning values” on page 192.

Variable definition and manipulation

Parameter A parameter defines the characteristics of an element in your program. Parameters, which are synonymous with arguments, are used to customize a program and pass a value to a routine.

Variable A variable is a symbol or name that stands for a value. Variables can represent numeric values, characters, character strings, or memory addresses.

Assignment An assignment represents an expression used to assign a variable. This variable is text if it is a string variable or a value if it is an integer variable.

Expression An expression represents a combination of at least one operand and one or more operators that represent a value.

Archived

Array Arrays hold simple ordered collections of scalar variables. Tivoli Provisioning Manager workflow development includes array support as a:

- Variable as a workflow input and output parameter
- Variable in foreach or while loop

Looping and iteration

Workflows may use the iterator and loop constructs to control repeated processing of parts of the code.

Loop A loop represents a question in a program. If that answer requires an action, the action is performed and the original question is asked again until the action is no longer valid.

Iterator An iterator represents a pass through a programming loop (a group of instructions).

Break Used to terminate a loop prior to completion as determined by the conditions set up in a while of foreach statement.

Conditional processing

Within workflows, you may use the if-then-else elements to perform conditional processing.

To include a conditional test in your workflow, do the following:

Workflow composer:

From the Workflow Element list at the top of the Workflow Composer WebUI, drag the conditional workflow element icon to the appropriate line in your workflow, and the following code sample is inserted into your workflow:

```
if "" then
    <no operation>
else
    <no operation>
endif
```

APDE:

Place the cursor on the line in your workflow where you want the conditional processing to take place, and press Ctrl-Space to get the context-sensitive

pop-up. Select If-then or if-then-else to get the proper code samples inserted into your workflow.

In both cases, you will have to provide the condition to be tested yourself, by replacing the word “condition” inside the parentheses shown.

Conditional A conditional statement is an action that occurs only if a specific condition is met. Programming languages use the word if for conditional expressions.

Comment A comment represents a statement in a program that provides information about the program, or an element of the program. It does not affect the program or the way it runs.

Exception handling

try A try represents a statement where an error can occur.

throw A throw represents a statement that is passed to another area of the workflow.

catch A catch represents a statement to handle errors that occur in a try statement.

catchall A catch all represents a statement to catch exceptions of any type that occur in a try statement.

finally A finally represents a statement that runs after all error processing has completed.

Imbedded custom code

scriptlet A scriptlet represents a list of commands that can be run without user interaction. The workflow composer supports scripts written in Bash, Perl, and Expect.

Datacenter Model manipulation

DCM Insert A DCM insert statement represents a call to the data center model to insert new information into the database.

DCM Update A DCM update statement represents a call to the data center model to update new information into the database.

DCM Delete A DCM delete statement represents a call to the data center model to delete information from the database.

Archived

Miscellaneous

Check Device Locale Check the language (locale) for the device. If a locale has been specified, a workflow will fail if the target device for the workflow does not match the locale.

Comment A comment represents a statement in a program that provides information about the program, or an element of the program. It does not affect the program or the way it runs.

Log A log element logs details associated with the workflow when it is run. You can choose to log messages of type: Info, Warning, or Error.

@depreciated Used to mark workflows as backlevel. If a workflow references other workflows that has been tagged with the @depreciated tag, a warning will be raised when compiled.

@requirepermission This tag is used to enforce that the user executing the workflow has certain rights. This is only enforced if the global setting Access Control is set to On.

6.3.1 Variables

All variables and arrays used in workflows have to be declared before they can be used. The only exceptions from this rule are input and output variables that are implicitly declared, and item-variables used in foreach loops.

The var or array statements are used to declare variables and arrays respectively, and these may optionally be used to assign values too.

Here are a few examples:

```
var ServerID
var i = “0”
var serverName = DCMQuery(/server[@id=$ServerID]/@name)
var serverLabel = Jython[ ServerName + “(“ + serverID +
```

Remember, that variable and array names are case sensitive, and that all variables are treated as strings.

Variable naming conventions

A good practice is to define variables at the beginning of the workflows to facilitate ease of maintenance. The following represents the recommended best practices for naming variables and parameters.

Passed variables

Capitalize the first letter for external or passed variables Variable names cannot match Jython reserved words (timeout, error):

```
workflow MyCompany_Jumpstart_Get_Client_Properties(in ServerID, out
ClientHostName, out ClientInterfaceName) LocaleInsensitive
```

Internal variables

Start variables with lower case for internal variables. Do not use spaces in variable names. Variable names cannot match Jython reserved words (timeout, error):

```
var softwareName
var serverName
var sendStatus
```

Constants

Constants should be in all uppercase. Consider using variables with a descriptive name as a constant for inputs that are potentially cryptic:

```
ERRORMSG1 = "This is an error"
var ENTIRESYSTEM = “0”
var DEPLOYMENTENGINE = “5”

java:com.thinkdynamics.kanaha.de.javaplugin.datacentermodel.GetProperty
(ENTIRESYSTEM, cardinalNodeID, "hacmp.publicSSHKey", publicSSHKey)
```

Variable handling

Workflow inputs should be validated as early as possible within a workflow. Ideally, this should occur before performing any actions that would need to be undone in the event of a failure. This includes input variables passed to the workflow and variables obtained via Get DCM Property transitions. At a minimum, variables should be tested to ensure they are not null. Additional tests for numeric, alpha, and alpha-numeric can also be performed. Variables obtained by user input to a workflow should have extensive validation. If possible, the input should be examined to ensure that it is the correct object type.

Examples:

```
var testMe
var result
var NUMERIC = "^[0-9]+$"
var ALPHA = "^[a-zA-Z]+$"
var ALPHANUMERIC = "^[a-zA-Z0-9]+$"
if Jython(testMe==None) then
   throw NULL_VARIABLE "testMe is Null"
```

```
endif
String_Replace_All(testMe, NUMERIC, "true", result)
if Jython(result != "true") then
   throw NONENUMERIC_VARIABLE "testMe is not numeric"
endif
String_Replace_All(testMe, ALPHA, "true", result)
if Jython(result != "true") then
   throw NONALPHA_VARIABLE "testMe is not alphabetic"
endif
String_Replace_All(testMe, ALPHANUMERIC, "true", result)
if Jython(result != "true") then
   throw NONALPHANUMERIC_VARIABLE "testMe is not alpha-numeric"
endif
```

6.3.2 Comments

Comments should be used throughout the workflow to document the function of the workflow and the expected results. Comments can describe individual statements, such as a DCM Query command, and can also describe blocks of code, such as an if/else section. Comments can be combined with logging to make the workflow and the workflow execution log more readable and understandable for trouble-shooting activities.

Comments in a workflow are noted by the “#” sign.

The following is an example of workflow comments:

```
var MSG1 = "Execution successful"
var MSG2 = "Execution failed"
var MSG3 = "Execute the installation script"
# Execute the installation script using the files copied to the target
system
log info MSG3
Device.ExecuteCommand(DeviceID, commandString, "/tmp", "default",
"1800", "error", ReturnCode, ErrorMessage, <null>)
# Test for success / failure of the installation script.
if Jython(ReturnCode == "0") then
   log info Jython("MyScript.sh: " + (MSG1 or ""\))
else
log info Jython("MyScript.sh: " + (MSG2 or ""\))
endif
```

6.3.3 String manipulation

When developing workflows, you will most probably incur a situation in which you need to manipulate string variables.

All string manipulation within the workflow engine is handled by Jython (a Java implementation of the popular Python programming language) which is imbedded in TIO/TPM V3.

The string methods available for you to manipulate strings are listed in “Jython string methods” on page 584. Among the most popular ones are find, replace, split, strip, upper, lower, and lshift. In addition, the Jython language provides facilities to extract substrings (slice).

Examples on how to manipulate strings are provided in the following section.

Basic string operations

All workflow variables are considered strings. When manipulated by Jython, they can be treated either as a value (in the case of concatenation or comparison) or as an object, that is, manipulated through invoking a method.

Concatenation and default values

The plus (+) operator is used to concatenate two (or more) strings. For example, the following workflow code:

```
var firstName=”Morten”
var lastName=”Moeller”
var name = Jython[“my name is ” + firstname + " “ + lastname]
```

will assign the value of “my name is Morten Moeller” to the variable named name.

To assign default values to be used in case the variables have no values, you may use the *or* operator. The following lines:

```
var firstName
var lastName
var name = Jython[“my name is ”+(firstname or “John”)+" “+(lastname or
“Doe”]
```

will assign the value of “my name is John Doe” to the variable named name, since neither firstName nor lastName has a value.

Indexing, substringing, or slicing

To select a character or series of characters (a substring) from a string, use indexing.

For example:

```
var t = Jython("abcdef"[2])
```

will assign the value of *c* to the variable t, and

```
var t = Jython("abcdef"[2:5])
```

will assign a value of *cde* to the t variable.

Many of the string functions test conditions, so they are often used in conjunction with the if and while statements. For details, please see “Conditional processing” on page 194. In addition to testing conditions, strings also support methods to test the nature of the string. These are islower, isupper, isalnum, isnum, isalpha, isspace, and istitle. These methods test to see if all the characters in the strings meet these conditions.

Splitting and joining

Jython also provides methods for splitting and joining strings. Bear in mind that these functions apply only to single string variables and not to workflow variables of the type array.

Here is an example in which we convert a US date format (mm/dd/yy) to European format (dd/mm/yy):

```
# var us_date = “mm/dd/yy”
# split the date into three words using the / as delimiter
array words=Jython[us_date.split(“/“)]
# eu_date= Jython( join(words[1],”/”,words[0],”/”,words[2]\) )
```

6.3.4 Number manipulation

When numbers are used in workflows, they are all represented as strings. Therefore, in order to perform numerical operations on these variables, you have to typecast them into the proper number format within the Jython statement.

The following shows a couple of examples of typecasting string variables to different number formats:

```
# typecast the string “4.67” to a floating point
log info Jython[float(“4.67”)]

# represent the value of the variable i as an integer
var i = “9”
log info Jython[ int(i) ]
```

Table 6-4 shows the operations supported for all numeric types (except complex). The operations are sorted by ascending priority. All numeric operations have a higher priority than comparison operations.

Table 6-4 Jython numeric operations

Operation	Result
x + y	Sum of x and y.
x - y	Difference of x and y.
x * y	Product of x and y.
x / y	Quotient of x and y.
x // y	(floored) Quotient of x and y.
x % y	Remainder of x / y.
-x	X negated.
+x	X unchanged.
abs(x)	Absolute value or magnitude of x.
int(x)	X converted to integer.
long(x)	X converted to long integer.
float(x)	X converted to floating point.
hex(x)	X converted to hexadecimal.
complex(re,im)	A complex number with real part re, imaginary part im. im defaults to zero.
c.conjugate()	Conjugate of the complex number c.
divmod(x, y)	The pair (x // y, x % y).
pow(x, y)	X to the power y.
x ** y	X to the power y.

Archived

And here are a few examples of performing numeric operations:

```
var i = "444.67"
var j = "9"

# divide i with j
log info Jython[float(i) / float(j)]

# find the remainder of the division (always an integer)
log info Jython[int(float(i) % float(j))]

i = 1
```

6.3.5 Manipulating DCM objects

When developing workflows, it is more than likely that you need to manipulate properties of the DCM objects (Servers, SoftwareProducts, and so on) that the workflow deals with. To facilitate this, the following four workflow functions are provided:

- DCMQuery
- DCMInsert
- DCMUpdate
- DCMDelete

The DCMQuery

The DCMQuery is the preferred means for obtaining information in the TIO/TPM DCM. Java plug-ins included with the product are release dependant and the trend is to do more through DCMQueries rather than relying on embedded Java plug-ins. Some of the Java plug-ins in the current release have already been deprecated. The online help within IBM Tivoli Intelligent Orchestrator V3 provides information and examples on DCMQueries. Understand that DCMQuery uses an XPATH like query to the database underlying the DCM to retrieve values.

Using the following DCMQuery, here is a brief explanation of the constructs that comprise the statement:

```
DCMQuery(/softwareproduct[@id=$SoftwareID]/@version)
```

- Query the database (DCMQuery).

  Start at entire database (/). The slash directs the query to search the entire document. In our case, it is a database and it means to search all tables.

- Search SOFTWARE_PRODUCT table (softwareproduct[...]).

  Limit the search to rows that have a unique identifier of $SoftwareID ([@id = $SoftwareID]). The brackets are used to limit a query. You may see examples

with more than one bracketed phrase in the expression. This is a making a further limit on the query.

- Return the value of the version column (/@version).

> **Note:** DCMQuery syntax tips
>
> - A dollar sign ($) in a DCM query expression followed by letters or digits is used to represent a variable in a DCM query.
> - Brackets ([]) in a DCM query expression are used to group conditional and order by expressions.
> - A back slash (/) in a DCM query expression represents an absolute path to the required element.
> - Attributes are specified by a @ prefix and are separated from values by an equal (=) sign in a DCM query expression.
> - Values are enclosed in double quotes (“ “) in a DCM query expression.
> - To use a value that contains a double quote, you can define a variable that has that value and use the variable instead in the query.
> - To define a null value, use “<null>”. For example, @name=”<null>”.
> - Ordering the return values of a DcmQuery that returns multiple values may be achieved by adding [orderBy@attribute].
> - For example:
>
> ```
> /server[@id=$Server]/nic[@managed="Y" and
> @failed!="<null>"]/networkinterface/@ipaddress[orderBy@managed]
> ```
>
> The above example lists the IP addresses (sorted by the managed attribute) for all network interfaces hosted by any managed NIC on a server with an ID equal to the value of the variable named $ServerID.

The following are some samples taken from the TIO/TPM online help information and other sources:

- Retrieve all NIC IDs that have a server name of “*Nile*” and that are not managed:

  ```
  DCMQuery (/server[@name="Nile"]/nic[@managed="N"])
  ```

- Retrieve all the server names, ordered alphabetically by name, in a cluster called “RiversEdge”:

  ```
  DCMQuery (/cluster[@name="RiversEdge
  Web"]/server/@name[orderBy@name])
  ```

- Retrieve the server name from a cluster named “RiversEdge” using a variable ($serverName) that had been previously defined in a workflow:

  ```
  DCMQuery (/cluster[@name="RiversEdge
  Web"]/server[@name=$serverName])
  ```

- Retrieve information where some objects have two fields that link to another object (for example, accessRule has sourceSubnetwork and destinationSubnetwork attributes).
- To navigate from accessRule to a subnetwork, use either **accessRule/sourceSubnetwork** or **accessRule/destinationSubnetwork**:

  ```
  DCMQuery(/acl[@name="acl-1
  New1"]/accessRule/sourceSubnetwork/@ipaddress)
  ```

- A relationship between two DCM objects can be navigated using a DCM query in two directions. For example:

  ```
  /server[@name=\"Server1\"]/bladeadminserver
  ```

  or

  ```
  /bladeadminserver [@name=\"Blade1\"]/server
  ```

- Aliases are used to differentiate between relationships. For example, in the query below, accessRule has two fields connecting to a subnetwork: sourceSubnetwork and destinationSubnetwork. Two relationships were generated for this scenario, one for each relation field:

  ```
  /acl[@name="acl-1 New1"]/accessRule/sourceSubnetwork/@ipaddress
  /acl[@name="acl-1 New1"]/accessRule/destinationSubnetwork/@ipaddress
  ```

  In order to navigate in the opposite direction, you must use an alias object to differentiate between the relationships.

  For example:

  ```
  /subnetwork[@ipaddress="10.1.8.0"]/AclRuleSource/acl/@name:
  ```

  will link the subnetwork to the sourceSubnetwork field from the accessRule, and:

  ```
  /subnetwork[@ipaddress="10.1.8.0"]/AclRuleDestination/acl/@name
  ```

  will link the subnetwork to the destinationSubnetwork field from the accessRule.

Inserting new objects into the DCM

When you insert data into the DCM, it is assumed that the object does not already exist and you are creating it for the first time. Your XML data consists of top-level and embedded elements and, consequently, there are two methods for inserting data into the DCM. To insert data into the DCM, use the DCMInsert command, as shown in the examples below.

Inserting a top-level object

If a top-level object is being added to the DCM, the only requirement is the DCMInsert command and the XML data. For example, to insert a new blade server chassis that is not managed, has a network interface name of Management network, and an IP address of 10.1.8.77:

```
DCMInsert <<EOF
<blade-admin-server name="blade-chassis-admin">
   <network-interface
    name="Management
    network"ipaddress="10.1.8.77"
    managed="false"/>
</blade-admin-server>
EOF
```

Note: When you insert a top-level object (parent) into the DCM using the Web-based user interface, you do not have to specify a parent element in the Parent field.

Inserting an embedded element

If a child element is being added to the DCM, you must first retrieve the parent element's ID to determine the location in the DCM where the new data should be added. In the following example, the parent element is Customer, and the `DCMInsert parent = DCMQuery(/customer[@name="Rivers Edge Retail"])` expression will retrieve the Customer ID, and using this ID, can determine the precise location in the XML that will contain the new data.

```
DCMInsert parent = DCMQuery(/customer[@name="Rivers Edge Retail"])
<<EOF
  <application name="Online Store New" priority="5">
    <cluster name="RiversEdge Web"
      virtualipaddress="10.1.1.132"
      virtual-tcp-port="80"
      min-servers="2"
      max-servers="10"
      pool="Apache-Redhat"
      vlan="510"
      tier ="1"
      fabric="Default Fabric"/>
  </application>
EOF
```

Archived

Updating DCM objects

To update data in the DCM, use the DCMUpdate command, as shown in the example below:

```
DCMUpdate (/cluster[@name=\"RiversEdge Web1\"]) <<EOF
  <cluster
    name="RiversEdge Web2"
    is-device-model="Simulator"
    min-servers="2"
    max-servers="10"
    pool="IIS-Win2K New1"
    vlan="0001"
    tier ="1"
    fabric="Default Fabric New1"/>
EOF
```

Note: You can only update one object at a time, so embedded objects cannot be updated at the same time you update a top-level object.

Deleting objects from the DCM

To delete data from the DCM, use the DCMDelete command. To delete a customer application with a name of Online Store:

```
DCMDelete (/application[@name="Online Store"])
```

To delete a customer application with an ID of 123:

```
DCMDelete (/application[@id="123"])
```

Note: You cannot delete an object that cannot be identified through a single ID (the primary key is composed from several fields). You can delete all properties for a specific object, but you cannot delete a specific property for an object. Cascading delete is not supported in the DCM query language. If a DCM object that you want to delete has a relationship to other DCM objects, those objects must be deleted first. For example, to delete a router switch, you have to delete the router first and then the corresponding switch.

Archived

6.3.6 Testing, conditional processing, and looping

The following shows a few examples of how you can use existence and containment testing to see if a string has a value or a certain pattern of characters is contained in a string:

Testing for any value:

```
# has the variable been declared
if Jython[lastName]

# has a value been assigned
if Jython[lastName != None]

# has a null-value been assigned
if Jython[lastName.isspace()]
```

Testing for a specific value:

```
# has the variable a specific value
if Jython[lastName == “Moeller”]

# do the two variables have different values
if Jython[lastName != firstName]
# is the value of the variable uppercase only
if Jython[lastName.isupper()]
```

String containment testing:

```
# does the value of the string contain a blank
if Jython[“ “ in “Morten Moeller”]...

# does the lower value of the variable not contain the string “ll”
if Jython[“ll” not in lastName.lower()]

# does the value of the variable start with “M”
if Jython[lastName.startswith(“M”)]

# is the lowercase translation of last 3 characters of the stripped
value
# of the varaiable equal to “ler”
if Jython(lastName.strip(\).lower(\).[-3:] == “ler”)
```

Archived

Numeric testing:

```
# is the integer value of the variable i greated than 1
if Jython[int(i) > 1]

# is the long inte
```

6.3.7 Using Java in workflows

If Java needs to be used, then use a Java assignment statement rather than coding a Java plug-in. The Java variable can refer to any Java class in the classpath. The following example retrieves the home directory from TIO/TPM using Java:

```
var HomeDirJava = Java[java.lang.System#getProperty("tio.home")]
```

6.3.8 Logging

The workflow language allows you to write messages to the workflow execution log at run time. This function is facilitated by the *log* statement.

Using the log statement, messages are tagged with a severity level, which may be one of info, warning, error, or debug. The information to be written to the execution log may be a string, a variable, or a string composed by a Jython statement.

The following shows a few examples of the use of the log statement:

```
log debug “Workflow startedx..”
var MSG1 = "Execution successful"
var MSG2 = "Execution failed"
var MSG3 = "Execute the installation script"
# Execute the installation script
log info MSG3
Device.ExecuteCommand(DeviceID, commandString, "/tmp", "default",
"1800", "error", ReturnCode, ErrorMessage, <null>)
# Test for success / failure of the installation script.
if Jython(ReturnCode == "0") then
  log error Jython["MyScript.sh: " + (MSG1 or "")]
else
  log info Jython["MyScript.sh: " + (MSG2 or "")]
endif
```

6.3.9 Handling workflow exceptions

When automating any process, you almost always need capabilities to raise exceptions to signal error conditions or deviations from normal processing. In the TIO/TPM workflow language, you use the **`throw`** command to raise an exception, and a try-catch-endtry sequence to test for conditions.

A try-catch-endtry sequence may be constructed from the following statements:

try Sets up the exception handler.

.... Code to try, which may raise exceptions (in workflows, use the throw statement, in scriptlets, use the TIOthrow function).

catch <exception> <messageVariable>
Specific exception to test for.

... Code to handle a specific exception.
You may have multiple catch sections within the same try-endtry sequence in order to handle different exceptions.

catchall <messageVariable>

... Code to handle otherwise unhandled exceptions.

finally

... Code to cleanup. This is always executed no matter the status of the exceptions.

endtry Stops the exception handler.

If your **`throw`** command is embedded in a try-endtry sequence, the exception may be handled within your workflow; if not, the execution of the current workflow terminates, and the exception is sent to the calling workflow (if any).

The following demonstrates throwing exceptions and immediately terminating a workflow:

```
# some execution that fails
var error = “true”
if Jython[error == “true”] then
  throw errorException “an error occured”
endif
```

Archived

To handle the exception locally (within the workflow), you can embed it in a try-catch-endtry sequence:

```
try
  # some execution that fails
  var error = “true”
  if Jython[error == “true”] then
    throw errorException “an error occured”
  endif
catch errorException exceptionMessage
  log warning Jython[“caught errorException: “ + exceptionMessage]
  # code to recover the situation
  error = “false”
endtry
```

Now, we may want to add a test for any (unspecified) exceptions. The catchall statement is used to catch any exception, not previously tested for, using the catch statement. In the following example, we have added a catchall statement to send any exceptions besides the errorException back to the caller.

```
try
  # some execution that fails
  var error = “true”
  if Jython[error == “true”] then
    throw errorException “an error occured”
  endif
catch errorException exceptionMessage
  log warning Jython[“caught errorException: “ + exceptionMessage]
  # code to recover the situation
  error = “false”
catchall exceptionMessage
  log error Jython[“caught UNKNOWN exception: “ + exceptionMessage]
  rethrow
endtry
```

In the example above, the recovery is only tried once. If you want to be able to retry, for example, three times, you can add a while loop, and use the **rethrow** function to send the error information back to the caller:

```
var attempts = 1
while Jython[not error or error==None or error==”true” or
error.isspace()] do
  number_of_retries = 3
  try
    # some execution that fails
    var error = “true”
    if Jython[error == “true”] then
```

```
      throw errorException “an error occured”
    endif
  catch errorException exceptionMessage
    log warning Jython[“attempt “ + attempts + “ caught errorException:
“ + exceptionMessage]
    if Jython[int(attempts) >= 3] then
      rethrow
    else
      # code to recover the situation
      error = “false”
    endif
  catchall exceptionMessage
    log warning Jython[“caught UUNKNOWN exception: “ +
exceptionMessage]
    rethrow
  finally
    attempts = Jython[int(attempts) + 1]
  endtry
done
```

Notice that besides sending the exception back to the caller using **rethrow**, we use the finally statement (which is always executed, no matter the status of exceptions) to increment the attempts counter.

6.3.10 Embedding scripts

With the TIO/TPM V3 the workflow engine allows you to embed scripts into your workflows. These embedded scripts are known as *scriptlets*, and should be used in place of external script files.

Scriptlets can represent Bash, Perl, Expect, or Visual Basic scripts and the source is embedded within the workflow. Scriptlets are more transparent when viewing the workflow as all processing is visible, rather than hidden in a separate script. Scriptlets allow for passing of variables and provides three constructs for passing information back to the Tivoli Intelligent Orchestrator deployment engine:

TIOsetVar Allows one to update a variable in the workflow that is used outside of the scriptlet.

TIOthrow Provides support for throwing an error from within the scriptlet.

TIOlog Provides for logging transition information in the deployment engine logs from within the scriptlet.

Archived

Example:

```
scriptlet(osLevel) language=bash
target=DCMQuery(/server[@id=$DeviceID]) credentialskey="default" <<EOF
RC="$?"
if [ "$RC" -ne 0 ] ; then
   TIOlog “info” "File not found set return code = $RC"
   TIOsetVar “rc” "$RC"
else
# The file is present. Check it for the right level
   msg="File found: return code = $RC"
   TIOlog “error” “msg”
   TIOthrow “errorException” “msg”
   RC="$?"
fi
return 0
EOF
```

External scripts that are developed and invoked by a workflow need to return meaningful information to the workflow to aid in problem determination.

Scripts that end successfully must exit with a return code of zero. They can optionally pass back a message in stdout indicating that the execution was successful. Scripts that fail must pass back a non-zero return code. This return code should have a unique value for each type of failure. The script should also return back a reason for the failure via a message in stderr. It is not acceptable to only return a error code, and to have the error messages documented in some external document. If possible, the error message should also suggest how the problem could be resolved.

6.4 Creating your first simple workflows

To get started with understanding the basics of workflow development, we suggest that you refer to Chapter 9, “Case study: basic workflows” on page 259 and go through the case study to develop a few basic workflows.

Archived

7

Extending your workflows with Java

The following chapter provides basic guidance on how to develop Java programs to extend the capabilities of your workflows.

Archived

© Copyright IBM Corp. 2004, 2006. All rights reserved.

7.1 Workflows and Java programs

TIO/TPM V3 allows you to extend the functionality of your workflows by calling either Java programs or Java plug-ins directly from a workflow.

The main difference between the two is that a Java plug-in for IBM Tivoli Intelligent Orchestrator is a Java class file that runs in the context of TIO/TPM. In other words, TIO/TPM provides a container in which the plug-in executes. This is similar to a Java servlet, which runs inside a servlet container, or an applet, which runs inside a Web browser.

Because the plug-in runs inside the TIO/TPM container, the developer must remember two things:

- TIO/TPM as a container provides services to the plug-in, such as providing input values and reading output values.
- TIO/TPM constrains the plug-in, requiring the plug-in to implement certain functions and restricting how it operates.

With TIO/TPM v3, the support for creating custom Java plug-ins has been replaced by the capability of calling external Java programs directly. This provides greater flexibility and simplifies the development of specialized code, however, at a cost. Since the Java programs do not have to be registered with TIO/TPM, there is no obvious way of identifying which external (to the TIO/TPM environment) classes are being used, and management of these external modules can be a nightmare, if not managed properly.

If you decide to develop and use custom Java programs, we recommend that you wrap each package by a workflow, just like most Tivoli provided Java plug-ins are wrapped by a corresponding workflow. This will help maintain your environment. Java packages are typically named according to the Internet domain naming standard (for example, com.ibm.thinkcontrol.PathHelper) and the suggested naming standard for the wrapper workflows would be to reuse the package name, replacing the dots with underscores, so the corresponding workflow name would be com_ibm_thinkcontrol_PathHelper.

In the workflows that are delivered with TIO/TPM V3, you will find a lot of calls to external Java programs. Look for lines in the workflows similar to either one of the following:

```
var y=Java[<class>#<method>(<args>)]
java:<class>#<method>(<args>)
```

7.2 Developing Java programs

To develop your own Java programs that are needed to accomplish specific tasks for a workflow, the tool of choice is the Eclipse platform. This is most likely already available on your workstation since it is used as the basic platform for workflow development.

However, in some cases, you may need to develop or maintain your Java source from a plain-text editor environment. This is also possible, but requires that IBM Java SDK V1.4.2 has been installed on your workstation, and that any external classes that are referenced from the Java program are available to the system on which the code is compiled and packaged.

In the following, we will demonstrate how to develop and package a simple Java program that transforms a path from an UNIX/Cygwin format to a pure Windows format. The inspiration for this program has been found in the convertToCygwinPath method of the com.thinkdynamics.ak\kanaha.util.PathHelper class.

For the high-level design of our Java program, that we will need to reference methods from the following class:

com.thinkdynamics.kanaha.util.PathHelper

and the program needs to both receive and return a variable of type string.

7.2.1 Install the Java Software Development Kit

The Java Software Development Kit provides libraries and programs for compiling Java source files (with a .java extension) into Java class files (with a .class extension) and then compacting them into a compressed file format called a jar file (with a .jar extension).

The Software Development Kit is referred to in shorthand as the *SDK*, or sometimes mistakenly as *JDK*™ (for Java Development Kit). There are three separate SDKs: J2SE™, J2EE™, and J2ME™, which are used for developing standard Java components, enterprise Java components, and micro-device Java components. Be sure to download the J2SE SDK. At the time of the writing of this IBM Redbook, the current version for the J2SE SDK is 1.4.2_04.

The download page will offer both the SDK and the *Java Runtime Environment* (JRE), which provides files needed to run Java applications. Be sure to select the J2SE SDK, not the JRE.

You may see a reference to the Java 2 SDK, SE V1.4.2_04; the term Java 2 is often confusing to people. Java 2 is a marketing term used by Sun™. The

important factor is SE, which is a shorthand for J2SE and v1.4.2_04, which means Java 1.4.2 with a patch level of 4.

The download is available at `http://java.sun.com`. Assuming you wish to download the J2SE SDK for Windows, you will be prompted to save the file j2sdk-1_4_2_04-windows-i586-p.exe. Save it to your temp directory and then launch the download.

When launched, the wizard will prompt you for an installation directory (see Figure 7-1). We chose c:\java\j2sdk1.4.2_04. We also selected all options.

Figure 7-1 Java SDK Installation Wizard

After installation is finished, you will find a new directory created in c:\java\sdk1.4.2_04.

7.2.2 Develop a Java program using Eclipse

Even though this is not intended to be a full-blown description on how to use Eclipse, the following will demonstrate the basic steps of creating an eclipse Java project, define your package, classes, and methods, and compile and package your package.

The required steps are:

1. Create and customize a Java project.

2. Define package and class(es).
3. Create the source code.
4. Compile and build the package.
5. Activating the package.
6. Testing the Java code and creating the wrapper workflow.

Create and customize a Java project

Follow this procedure to create a Java project in your Eclipse workspace:

1. Open Eclipse, select the desired workspace, and select **File** → **New** → **Project**, and select **Java Project** from the New Project dialog shown in Figure 7-2.

Figure 7-2 Create a new Java Project

Click **Next**.

Archived

2. To provide a name and basic properties for the project, supply a name (we used ITSO PathHelper Project) and set the properties shown in Figure 7-3.

Figure 7-3 Naming and customization

In order to avoid overwriting your source, you may want to select the **Create separate source and output folders** option from the Create Java Project dialog.

Press **Next**.

3. To specify the location of your source and output files, modify the information in the **Source** tab of the Java Settings dialog (Figure 7-4).

Figure 7-4 Specifying source and build locations

Notice that we specified the lib subdirectory of the project as the output folder. This ensures that the source is not overwritten.

Press **Next**.

Archived

4. The final step is to specify the build path. From the Libraries tab on the Java Settings dialog, use the Add External JARs to specify the path to the TIO/TPM class library (in %TIO_HOME%\lib) on the TIOServer (Figure 7-5).

Figure 7-5 Setting build path for you Java project

As shown in Figure 7-5, we only need the datacenter.jar for our project; however, you should import all the package archives containing all the external classes you need for your particular program

5. You have now created the Java project. To start using it, press **Finish**.

> **Note:** You may receive a warning message from Eclipse indicating that you should shift to the Java perspective. You can accept this with no problems.

Expanding the project in the eclipse Project Explorer window will reveal a structure similar to Figure 7-6 on page 221.

Figure 7-6 Initial Java project content

Define package and class(es)

Now the time has come to define the package and class required for our Java program.

1. To create the package, right-click the newly created project (ITSO PathHelper Project, in our case) and select **New** → **Package**.

   Provide a name for the package in the Java Package dialog (Figure 7-7).

Figure 7-7 Creating Java package com.itso.kanaha.util

Archived

2. Create the package (Figure 7-8).

Figure 7-8 Creating java class ItsoPathHelper

Create the source code

Now that the project has been defined with packages and classes, you can create the source code for your methods.

The source code to convert a path name from Cygwin (UNIX) to Windows notation is shown in Example 7-1.

Example 7-1 convertToWindowsPath Java source code

```
package com.itso.kanaha.util;

import com.thinkdynamics.kanaha.util.PathHelper;

public class ItsoPathHelper {


   public static String convertToWindowsPath(String cygwinPath) {
```

```
   cygwinPath = PathHelper.onlyBackwardSlash(cygwinPath);
   String cygdrive = "cygdrive";
   int len = cygdrive.length();
   int at = cygwinPath.indexOf(cygdrive);
   if(at > 0)
   {
      char driveLetter = cygwinPath.charAt(at + len + 1);
      cygwinPath = driveLetter + ":" + cygwinPath.substring(at + len +
2);
   } else
   {
      cygwinPath = "%CYG_ROOT%" + cygwinPath ;
   }
   return cygwinPath;
}
```

When using the Eclipse platform, your Java code is checked for correct syntax as you go along, which is extremely helpful to catch missing references and typos.

Compile and build the package

When you are convinced that the Java code provides the expected functionality, you are ready to build the package. There is no need to compile the code, since this is performed behind the covers by Eclipse every time you save your source.

Archived

To build the package jar file, which eventually will have to be copied to the classpath of the TIO/TPM server, follow these steps:

1. Right-click the package and click **Export**. Then, from the Export dialog (Figure 7-9), select the JAR file and click **Next**.

Figure 7-9 Exporting the package jar file

Archived

2. Then, you need to specify what to include in the JAR file that will be created. Make sure that you check the **Export generated class files and resources** and **Compress the contents of the** JAR file check boxes (Figure 7-10).

Figure 7-10 Exporting the package jar file

When specifying the destination of the JAR file, you may use any location. As was the case for setting up the build path, you can specify the location of the %TIO_HOME%\lib directory of the TIO/TPM server to store your class file directly in its final destination. Alternatively, you can specify any local file, probably one that will be included later in the build process of a tcdriver, to ensure that the class file is distributed along with workflows and other files that are specific to a particular driver.

In our situation, we simply specified the root directory of the project, and named the JAR file ItsoPathHelper.jar.

Click **Next**.

3. Before the JAR file is created, you have to specify whether or not you will accept class files that does not compile successfully in the resulting JAR file,

and, optionally, a location for the XML file describing the JAR file (Figure 7-11).

Figure 7-11 Specifying Java package output file

In our situation, we did not want to include class files with compilation problems, and the location of the JAR description is the root of the project.

Click **Next**.

Archived

4. The final step before the JAR file is created is to specify a location for the manifest file, which describes the package. In our situation, we simply store the manifest file in the root directory of the project, as shown in Figure 7-12.

Figure 7-12 Specifying Java project manifest file

When done, click **Finish**, and after a short while in which the package is built and exported, the JAR file will be available in the destination specified. In our situation, the ItsoPathHelper.jar is located in the project root-directory as shown in Figure 7-13.

Figure 7-13 Eclipse Java project content after successful export

Activating the package

After the JAR file has been generated, it has to be copied to the %TIO_HOME%\lib directory on the TIO/TPM server. Naturally, if you specified this location when building the JAR file, this step is not necessary.

In order for the TIO/TPM WebSphere server to recognize the new package, it has to be restarted. Use the tiostop/tiostart scripts to restart the TIO/TPM server processes. When the TIO/TPM server has been restarted, calls to the classes in the newly created package will be possible.

Testing the Java code and creating the wrapper workflow

To test the new Java programs, you can call the methods you created from your workflows by adding statements similar to the following:

```
var x=Java[<class>#<method>(<args>]
```

In our example, in which we only have a single method in the Java class, we use the following:

```
wPath=Java[com.itso.kanaha.util.ItsoPathHelper#cnvertToWindowsPath(<var
>)]
```

As previously mentioned, you should consider creating a wrapper workflow to ease maintenance of your environment. Unlike Java plug-ins, Java programs are not registered in the TIO/TPM environment, which requires quite a bit of detective work to figure out which classes are called from which workflows, and which package files provide these classes. By creating a workflow that wraps the direct

calls to the methods of the class, you can use the standard TIO/TPM reporting functions to create where-used and used-by reports.

A wrapper workflow for the com.itso.kanaha.util.ItsoPathHelper class is shown in Example 7-2. Notice that even though the class only has one method, and that this only uses one input argument and produces one output argument, the passing of arguments is handled by arrays to allow for the requirements of methods that may be added at a later point, and to demonstrate how we would handle wrapping of classes with multiple methods. This approach has been chosen to minimize maintenance, since the alternative would have been to create individual wrapper workflows for each method.

Example 7-2 Wrapper workflow for the com.itso.kanaha.util.ItsoPathHelper class

```
workflow com_itso_kanaha_util_ItsoPathHelper(in method, in array argIn, out array
argOut)  LocaleInsensitive

var wrkflow="com_itso_kanaha_util_ItsoPathHelper"
var msg=Jython["Workflow " + wrkflow + " failed: "]
array supported_methods={"convertToWindowsPath"}
var isMethodKnown

# validate input
foreach m in supported_methods do
  if Jython[method == m] then
    isMethodKnown="true"
    break
  endif
done

# exit if method not found
if Jython[ not isMethodKnown or isMethodKnown == None ] then
  log error Jython[msg + "method '" + method + "' is unknown"]
  throw methodNotFound method
endif

# call Java program
if Jython(method=="convertToWindowsPath") then
  # map input array to arguments for the java call
  var inpath=argIn[0]
  # call the method
  argOut[0] = Java[com.itso.kanaha.util.ItsoPathHelper#convertToWindowsPath(inpath)]
endif
```

Archived

To test the Java program, you can now call the wrapper workflow from any other workflow. The simple workflow shown on Example 7-3 demonstrates how to perform native calls to standard Java classes (java.lang.System) to get the path of the directory referenced by %TIO_HOME%, the Tivoli provided com.thinkdynamics.kanaha.util.PathHelper class to convert a Windows path name to UNIX (Cygwin) syntax, and our wrapper workflow to convert a Unix path name to Windows format.

Example 7-3 Sample workflow calling Java

```
## this workflow illustrates how to call Java programs from a workflow

workflow LAB10  LocaleInsensitive

var tio_home

# get the unix version of tio_home
tio_home=Java[java.lang.System#getProperty("tio.home.unix")]
log info Jython["<b>unix_path:".ljust(20) + tio_home + "</b>"]

# get the windows version of tio_home
tio_home=Java[java.lang.System#getProperty("tio.home")]
log info Jython["<b>home_path:".ljust(20) + tio_home + "</b>"]

# convert the windows version of tio_home to cygwin format
tio_home =
Java[com.thinkdynamics.kanaha.util.PathHelper#convertToCygwinPath(tio_h
ome)]
log info Jython["<b>cygwin_path:".ljust(20) + tio_home + "</b>"]

# convert cygwin version of tio_home to windows
#    native call to Java method
#tio_home =
Java[com.itso.kanaha.util.ItsoPathHelper#convertToWindowsPath(tio_home)
]
#    using wrapper workflow
array args
args[0]=tio_home
array result
com_itso_kanaha_util_ItsoPathHelper("convertToWindowsPath", args,
result)
tio_home=result[0]
log info Jython["<b>windows_path:".ljust(20) + tio_home + "</b>"]
```

On execution, the workflow above will produce output similar to Figure 7-14.

Text	Execution Time	
Start workflow 'LAB10' (id=3216)	2006-04-05 03:44:56.953	
unix_path: /cygdrive/c/ibm/tivoli/THINKC~1	2006-04-05 03:44:57.184	
home_path: C:\ibm\tivoli\thinkcontrol	2006-04-05 03:44:57.284	
cygwin_path: /cygdrive/C/ibm/tivoli/thinkcontrol	2006-04-05 03:44:57.384	
Start workflow 'com_itso_kanaha_util_ItsoPathHelper'	2006-04-05 03:44:57.644	
End workflow 'com_itso_kanaha_util_ItsoPathHelper' (id=3218)	2006-04-05 03:44:58.736	
windows_path: C:\ibm\tivoli\thinkcontrol	2006-04-05 03:44:58.896	
End workflow 'LAB10' (id=3216)	2006-04-05 03:44:58.926	
	Total: 8, displayed: 8.	

Figure 7-14 Converting pathnames using Java programs

7.2.3 Manually creating the package file

In case you do not want to use the Eclipse environment to help ease the development and packaging of your Java programs, you can perform the steps described in the previous section manually. The following outlines the steps necessary to succeed:

1. Create a directory structure for your source code.

   To successfully compile your code, the source code has to reside in a directory structure that reflects the package naming. For our package, which is named com.itso.kanaha.util.ItsoPathHelper, we need the following structure:

   ```
   <basedir>\com\itso\kanaha\util\ItsoPathHelper
   ```

   To reuse the source code that was created using Eclipse, the <basedir> section of the directory structure should (in our case) be replaced with the root directory of the Eclipse project. In our case, the Eclipse workspace is located in c:\ibm\eclipse\workspace, so the <basedir> part would translate to:

   ```
   "c:\ibm\eclipse\workspace\ITSO PathHelper Project"
   ```

2. Create the source code.

   Use your favorite editor to create the Java program source code. In our example, we reuse the file from the Eclipse project, which was called ItsoPathHelper.java.

3. Create a compile, package, and copy script.

   Now, since you probably will have to compile and package your code multiple times before you get it right, we recommend that you create a script for compiling and packaging the Java source. This script should include steps for compiling your source code, packaging the JAR file, and (optionally) copying the JAR file to the TIO/TPM server system.

Please refer to Example 7-4 for an example of how these steps may be implemented.

Example 7-4 Command to manually compile and build the class file

```
set basedir=”c:\ibm\eclipse\workspace\ITSO PathHelper Project”
set java_home=c:\ibm\java142
set TIOServer=TIOServer
set SereverRootShare=C$
set tio_home=ibm\tivoli\thinkcontrol

echo Compiling the java program
pushd %basedir%\com\itso\kanaha\util
%java_home%\bin\javac -g -verbose -classpath
%tio_home%\lib\datacentermodel.jar -sourcepath
%base_dir%\com\itso\kanaha\util -d %basedir% ItsoPathHelper.java
popd
pause

echo packaging the class file
pushd %basedir%
%java_home%\bin\jar -cf %basedir%\com\itso\kanaha\util\ItsoUtil.jar
com\itso\kanaha\util\ItsoPathHelper.class
popd
pause

echo copying the package file to the orchestrator lib directory
copy %basedir%\com\itso\kanaha\util\ItsoUtil.jar
\\%TIOServer%\%SereverRootShare%\%tio_home%\ibm\tivoli\thinkcontrol\
lib
```

Archived

8

Automation package content and packaging

This chapter provides guidelines and the required steps for installing, deploying, and managing IBM Tivoli Intelligent Orchestrator automation packages.

We also provide a simple guide describing the required steps in the development and deployment of a sample driver for and Apache Web server.

This chapter contains the following topics:

- 8.1, “Introduction” on page 234
- 8.2, “Automation package anatomy” on page 234
- 8.3, “The manifest file” on page 237
- 8.4, “Packaging and deploying an automation package” on page 243

© Copyright IBM Corp. 2004, 2006. All rights reserved.

8.1 Introduction

As described in Chapter 2, “IBM Tivoli Intelligent Orchestrator concepts” on page 21, IBM Tivoli Intelligent Orchestrator and IBM Tivoli Provisioning Manager provide a number of automation packages covering major networking devices and software components that are used in a data center. Most of these automation packages can be obtained from the IBM On Demand Automation Catalog. Business Partners are encouraged to promote and make their automation packages available for Clients and other Business Partners via the IBM On Demand Automation Catalog, which is located at the following Web site:

http://www.ibm.com/software/ondemandcatalog/automation

In this section, we describe the major elements of a typical automation package and the process for creating an automation package.

8.2 Automation package anatomy

The current implementation of an automation package contains the binaries and metadata information that is related to specific objects in the Data Center Model. For example, an automation package could contain all the elements (workflows, Java plug-ins, scripts, and so on) required to manipulate a network device in the data center, such as a Cisco 3548 Switch. Another example would be an automation package that permits IBM Tivoli Intelligent Orchestrator to install and configure the Apache Web server.

All the elements of an automation package are compressed into a single Java jar file with the .tcdriver extension. Refer to 3.3.6, “Naming conventions” on page 88 for naming conventions. This section uses the Apache Web server automation package’s elements to describe the anatomy of the automation package file.

8.2.1 Directories

The automation package is a compressed Java jar file that includes various directories (depending on the elements in the package). For example, the Apache Web server automation package file is named apache.tcdriver and contains six folders: TC-INF, workflow, bin, command, repository, and doc.

Archived

For automation packages that include additional items, such as Java plug-ins and additional jar files, appropriate directories must be created to house these items. A list of all directories is as follows:

- command
- java-plugin
- TC-INF
- workflow
- lib
- doc
- bin
- repository

The command directory

This directory will contain all the definitions for the simple commands to be used in this automation package. A simple command performs an action on the data center, such as installing an application on a server, saving the configuration of a switch, and so on. Simple commands are defined in IBM Tivoli Intelligent Orchestrator via specialized XML files. Once defined, these XML files must be placed in the commands directory.

For example, in the case of the Apache automation package, the following simple commands are used:

- Apache Red Hat Config Parameters.xml
- Apache Start.xml
- Apache Stop.xml
- Expect Install of Apache.xml

Once the automation package is installed, these simple commands will be available for reuse by any workflow that might need to perform these operations.

The java-plugin directory

This directory holds all the additional Java plug-ins that are installed and used on a specific device or software related to the automation package. A Java plug-in enables the IBM Tivoli Intelligent Orchestrator to communicate with devices defined in the data center.

Java plug-ins are invoked exclusively by IBM Tivoli Intelligent Orchestrator workflows and are the basic elements from which workflows are built. Java plug-ins are defined in IBM Tivoli Intelligent Orchestrator via specialized XML files. Once defined, these XML files must be placed in the java-plugin directory.

For example, there are Java plug-ins for executing remote commands via SSH, to get and set attributes of a server, or get information about a Software Product.

Archived

The TC-INF directory

This directory contains the automation package's manifest file. The manifest is an XML file describing different elements of an automation package. This includes information, such as the name of the driver, its version number, and other associated information, such as the driver's template and its dependencies.

The workflow directory

This directory contains the automation package's related workflows developed to manipulate objects in the data center. Once workflows are developed using the IBM Tivoli Intelligent Orchestrator workflows development interface, they must be exported to XML-formatted files and placed in the workflow directory.

For example, the Apache automation package implements two workflows:

- Apache Install - Debian.xml
- Apache Install - Red Hat.xml

The lib directory

This directory contains the custom-written Java jar files implemented by the automation package's Java plug-ins. These files will be copied to the %cygwin%/home/thinkcontrol/drivers/lib directory when the automation package is installed.

The doc directory

This directory contains the automation package's overall documentation file that is referenced inside the manifest file via an XML documentation tag. The automation package design and deployment can be documented as follows:

- The overall solution implementation document is included in the automation package (referenced by a documentation tag) and is placed in the doc directory.
- The name, description, and version number of the driver referenced in the automation package manifest.
- Association to a new or existing category described in the automation package manifest.
- Name and description of the workflows, documented in the associated workflow's XML file.
- Name and description of each step (transition).
- Name and description of each variable.

Archived

The bin directory

Any additional script files that are required by the automation package must be placed in this directory. These scripts are run on the IBM Tivoli Intelligent Orchestrator server and are not replicated on the target systems. For example, the Apache automation package makes use of an Expect script named ApacheConfig.exp.

The repository directory

The repository directory holds all the scripts used by the automation package that are meant to be run in the target system. These scripts are first copied to the target system by IBM Tivoli Intelligent Orchestrator and then executed from there.

> **Note:** The location of the repository is controlled by the global TIO/TPM variables called PackageRepositoryHost and PackageRepositoryDir.

8.3 The manifest file

This section describes the automation package's manifest file using the manifest file created for the Apache automation package as an example. As described earlier, the manifest file should be located in the TC-INF directory.

The name of the manifest file must be tc-driver.xml.

The following lists the main XML tags (sections) that make up the manifest file:

- <driver-name>
- <version>
- <dependencies>
- <action>
- <property>
- <items>
- <device-models>
- <post-install-workflow>
- <software-products>

There are additional tags, such as <description> and <documentation>, that describe the driver further.

Archived

The following examples illustrate the manifest file used in the Apache automation package manifest's main sections.

8.3.1 The <driver-name> and <version> tags

These main sections specify the driver's name and its associated version. Example 8-1 displays the main section of the Apache automation package. It also shows the <description> and <documentation> tags, where:

<driver-name> Specifies the automation package name: apache. This name must exactly match with the file name of the automation package: apache.tcdriver.

<version> The version number (1.0) associated with the automation package. It shows up when the **`tc-driver-manager command`** is issued.

<description> Should be a short description of the automation package's functionality.

<documentation> Provides the file name of the text file used to describe the contents of the automation package.

Example 8-1 Automation package name and version

```
<?xml version="1.0" encoding="UTF-8"?>
<tc-driver>
   <tc-driver-format>1.0</tc-driver-format>
   <driver-name>apache</driver-name>
   <version>1.0</version>
   <description>
      The Apache automation package permits Intelligent Orchestrator to
install and configure the Apache web server server on a Linux Red Hat server
   </description>
   <documentation location="doc/readme.txt"/>
```

8.3.2 The <dependencies> tag

This tag is mandatory, and it defines and lists any other drivers that this automation package depends on. For example, Example 8-2 on page 239 illustrates that the Apache automation package requires core.tcdriver, rpm.tcdriver, and debian-operating-system to be included and referenced.

Example 8-2 Dependencies tag

```
<dependencies>
   <dependency name="core"/>
   <dependency name="rpm"/>
   <dependency name="debian-operating-system"/>
</dependencies>
```

For example, the core.tcdriver includes system-wide workflows, binaries, Java plug-ins, and commands that other drivers can reuse. Example 8-3 shows a portion of the core.tcdriver jar file content.

Example 8-3 core.tcdriver: jar output

```
created: TC-INF/
  created: bin/
  created: command/
  created: doc/
  created: java-plugin/
  created: workflow/
extracted: TC-INF/tc-driver.xml
extracted: bin/ssh_ping.sh
extracted: command/Create Network Interface.xml
extracted: command/Delete Network Interface.xml
extracted: command/Execute Local Command.xml
extracted: command/Get DCM Property.xml
extracted: command/Execute Command Through Terminal Server.xml
extracted: command/Execute Expect Command With Credentials.xml
extracted: command/Execute SSH Ping.xml
extracted: command/Get Cluster Attributes.xml
extracted: command/Get Current Deployment Request Id.xml
extracted: command/Get Default SAP For Operation.xml
extracted: command/RM Allocate Ip Address.xml
extracted: command/Lock DCM Object.xml
extracted: command/Get Device Service Access Point.xml
extracted: command/Get Load Balancer Attributes.xml
extracted: command/Get NIC Attributes.xml
extracted: command/Get Network Interface Attributes.xml
extracted: command/Get Network Interface By IP Address.xml
extracted: command/Get Pool Attributes.xml
extracted: command/Get Route Attributes.xml
extracted: command/Get Router Attributes.xml
extracted: command/Get SAP Attribute.xml
extracted: command/Get SAP Connection Points.xml
extracted: command/Get SAP Credentials Information.xml
extracted: command/Get SNMP Read Community for DCM Object.xml
extracted: command/SNMP Ping.xml
extracted: command/Get SNMP Write Community for DCM Object.xml
```

Archived

8.3.3 The <property> tag

The <property> tag is optional. It is used to define a macro substitution that can be utilitized for any future quoted string references in the manifest file.

For example, the property named tc.pkg in Example 8-4 is used to assign "`com.thinkdynamics.kanaha.tcdrivermanager.action`" to the tc.pkg string. Any occurance of ${tc.pkg} in an attribute string inside the manifest file, such as the Actions attribute in Example 8-4, is substituted with:

```
com.thinkdynamics.kanaha.tcdrivermanager.action
```

Example 8-4 Property tag

```
<property name="tc.pkg"
location="com.thinkdynamics.kanaha.tcdrivermanager.action"/>
```

8.3.4 The <actions> tag

The <actions> tag (Example 8-5) is used to list all required classes (the ones required, for example, by the workflows that are specified in the Items section) that are necessary to install items such as Java drivers, commands, and others, such as binary programs and the operating system's shell scripts.

Example 8-5 Actions tag

```
<actions>
   <action name="command" class="${tc.pkg}.SimpleCommandActions"/>
   <action name="copy-file" class="${tc.pkg}.CopyFileActions"/>
   <action name="java-plugin" class="${tc.pkg}.JavaPluginActions"/>
   <action name="workflow" class="${tc.pkg}.WorkflowActions"/>
</actions>
```

8.3.5 The <items> tag

The <items> tag is used to list items to be installed with this automation package. Each item identifies a certain operation that will be performed with this automation package. Example 8-6 on page 241 lists logical operations that are implemented by the Apache automation package.

Archived

The order that the items are listed in the <items> tag needs to follow the order that the items need to be installed in the IBM Tivoli Intelligent Orchestrator database during the automation package install. Flat files are installed first, followed by Java plug-ins, simple commands, and workflows. The reverse order will be respected during the uninstall of the automation package: workflows are un-installed first, followed by simple commands, Java plug-ins, and all flat files.

Example 8-6 Items tag

```
<items>
      <item name="repository/apache_config.sh" action="copy-file">
         <param name="dest.path"
value="${tc.home}/repository/Apache/apache_config.sh"/>
         <param name="chmod" value="755"/>
      </item>
      <item name="bin/ApacheConfig.exp" action="copy-file">
         <param name="dest.path" value="${tc.home}/bin/ApacheConfig.exp"/>
         <param name="chmod" value="755"/>
      </item>
      <item name="command/Apache Red Hat Config Parameters.xml"
action="command"/>
      <item name="command/Apache Start.xml" action="command"/>
      <item name="command/Apache Stop.xml" action="command"/>
      <item name="command/Expect Install of Apache.xml" action="command"/>
      <item name="workflow/Apache Install - Red Hat.xml" action="workflow"/>
      <item name="workflow/Apache Install - Debian.xml" action="workflow"/>
   </items>
```

8.3.6 The <device-model> tag

The <device-model> tag is used to define a new device model to be installed in the DCM database. It names the workflows to be assigned with each device model.

This tag can be omitted if the automation package does not need to define a device model.

Archived

Example 8-7 illustrates the device models used in the Apache automation package. In this example, the device model named Apache has its category set to Software Products and its associated workflows will to be added to the DCM database by the automation package installation process.

Example 8-7 Device-model tag

```
<device-models>
   <device-model name="Apache" category="Software Products">
      <workflow name="Apache Install - Debian"/>
      <workflow name="Apache Install - Red Hat"/>
   </device-model>
</device-models>
```

Note: Please remember that the linkage between workflows and logical operations for a specific device model is defined in the workflow xml file.

8.3.7 The <post-install-workflow> tag

The <post-install-workflow> is an optional tag that names a workflow and its associated parameters that are to be executed after all items defined in the <items> tag of the manifest are installed. This workflow may be a new one that will be installed by this automation package or a pre-existing workflow in the data center database.

Example 8-8 show a sample of a <post-install-workflow> definition.

Example 8-8 Post-installation-workflow tag

```
<post-install-workflow name ="myWorkflow">
   <param name="myParam" value="myValue" />
</post-install-workflow>
```

Unfortunately, there is no similar capabilities available to reverse the process. In other words, there is no pre-removal workflow option that can be defined to be executed prior to automation package removal/uninstallation.

8.3.8 The <software-products> tag

The <software-products> tag is used to define all the software package definitions that are required by the automation package components. The objects defined by the <software-products> tag will be defined as software packages in the Data Center Model (DCM). Such objects are defined following the same syntax as for the <software> element of the DCM XML definition file. It also supports the use of properties substitutions defined by the <properties> tag (see

8.3.3, “The <property> tag” on page 240) within the software package attribute values.

Example 8-9 shows a sample of a <software-products> tag definition.

Example 8-9 Software-products tag

```
<software-products>
   <software-product name="Trade3" description="Trade3 Websphere Application"
version="1.0" package_path="/home/thinkcontrol/images/t3install"
category="Trade3 Appl">
      <parameter name="DB2BinDir" value="C:/IBM/SQLLIB/bin"/>
      <parameter name="DB2HomeDir" value="C:/IBM/SQLLIB"/>
      <parameter name="DB2TCPPort" value=" 50000"/>
      <parameter name="Trade3DBName" value="trade3db "/>
      <parameter name="WASBinDir" value="C:/IBM/WebSphere/AppServer/bin"/>
      <parameter name="WASAppServerName" value="server1"/>
      <parameter name="WASHomeDir" value="C:/IBM/WebSphere/AppServer"/>
   </software-product>
</software-products>
```

8.4 Packaging and deploying an automation package

This section describes the process for creating an automation package. As an illustration, we list the steps needed to develop and deploy the Apache automation package. The process is generic and can be applied to various automation packages in development and deployment scenarios.

Attention: The names used here to describe items such as workflows and device model are applicable *only* to the Apache automation package example presented in this section.

In order to package and deploy an automation package, perform these tasks:

1. Set up the packaging environment.
2. Create and export all the workflows, simple commands, and logical operations that are required by the automation package.
3. Create all the Java plug-ins associated to the automation package.
4. Create all the supporting script files.
5. Create the automation package readme file.
6. Create the package’s tc-driver.xml manifest file.
7. Perform pre-packaging verification tasks.

8. Create the automation package .tcdriver jar file.
9. Install the .tcdriver file on the IBM Tivoli Intelligent Orchestrator server platform using the **tc-driver-manager command**.
10. Verify the installation.

8.4.1 Setting up the environment for packaging

The automation package creation process can be run on the IBM Tivoli Intelligent Orchestrator server or on a separate machine. When packaging the Apache automation package, we used a separate workstation and then transferred the packaged file onto the IBM Tivoli Intelligent Orchestrator server for deployment.

We performed the following tasks to set up our workstation for the packaging process:

- Install the Java 2 Standard Edition (J2SE) Development Kit. At the time of the writing of this IBM Redbook, the current version is SDK1.4.2_04. The download is available at http://java.sun.com. The SDK is required because creation of an automation package involves creating a compressed Java archives (jar) file using the Java **jar.exe** command included in the JDK.

  By default, the jar utility resides in the %JAVA_HOME%/jdk-1.4.2_04/bin folder. We modified our Windows system PATH (the host OS where Cygwin is installed) to include this path.

> **Note:** The SDK version and level used in this section have no dependencies on the JDK level required during the installation of the IBM Tivoli Intelligent Orchestrator. However, if you are going to use the server platform to build your driver package, you are required to satisfy the version and level of JDK required by the installation process. Refer to *IBM Tivoli Intelligent Orchestrator and Tivoli Provisioning Manager Installation Guide*, SC32-1420 for details on JDK version and level requirements.

- Install Cygwin on the workstation used to perform the packaging.

  This was required because we used a script (buildpackage.sh, shown in Example 8-12 on page 251) to create the .tcdriver file package. We followed the Cygwin installation steps described in the *Provisioning On Demand Introducing IBM Tivoli Intelligent ThinkDynamic Orchestrator*, SG24-8888.

- Create the required directory structure.

  On our packaging workstation machine, we created two temporary directories (folders), called drivers and packages, in our %home% directory.

Within the packages directory, we created an additional directory with a name identical to our automation package name (apache). Therefore, the automation package will be called apache.tcdriver.

Within the apache directory, we created the directory structure of the automation package, as discussed in 8.2, “Automation package anatomy” on page 234.

The following shows the created directory structure:

- %home%/drivers
- %home%/packages
- %home%/packages/apache
- %home%/packages/apache/doc
- %home%/packages/apache/TC-INF
- %home%/packages/apache/workflow
- %home%/packages/apache/repository
- %home%/packages/apache/command
- %home%/packages/apache/bin

where %home% is the Cygwin home directory.

We now continue with the next step, which is exporting the Apache workflows and copying them into the workflow directory created in the steps above.

8.4.2 Exporting the workflows and simple commands

Workflows and simple commands can be exported from the IBM Tivoli Intelligent Orchestrator database to a specified file. We use this facility to export the Apache workflows and simple commands to XML-formatted files to be included in the automation package creation process. Refer to *Provisioning On Demand Introducing IBM Tivoli Intelligent ThinkDynamic Orchestrator*, SG24-8888 for instructions on how to export workflows and simple commands.

Note: When exporting workflows, you can specify whether the current command data or Java plug-in should be included. Normally, these entities should be treated as separate export files.

An instance in which a complete nested export (one that includes command data and Java plug-ins information) might be of interest would be when the content of the workflow is being exported from one system to another and a tc-driver structure is not used.

In our example, these options were left unchecked.

Archived

After all workflows for the automation package have been developed, they must be exported and copied to the workflows directory that will be packaged into the .tcdriver file. For the Apache automation package, the content of the %home%/packages/apache/workflow directory is as follows:

- %home%/packages/apache/workflow/Apache Install - Debian.xml
- %home%/packages/apache/workflow/Apache Install - Red Hat.xml

After all simple commands for the automation package have been developed, they must be exported and copied to the command directory that will be packaged into the .tcdriver file. For the Apache automation package, the content of the %home%/packages/apache/command directory is as follows:

- %home%/packages/apache/command/Apache Red Hat Config Parameters.xml
- %home%/packages/apache/command/Apache Start.xml
- %home%/packages/apache/command/Apache Stop.xml
- %home%/packages/apache/command/Expect Install of Apache.xml

8.4.3 Creating the Java plug-ins XML and JAR files

As described in “Create and load the Java plug-in XML file” on page 607, Java plug-ins are defined to IBM Tivoli Intelligent Orchestrator using XML and Java JAR files. All of the Java plug-ins XML definition files that will be packaged into the .tcdriver file must be copied to the java-plugins directory. The Java JAR files must be copied to the lib directory.

Following our example, the Apache automation package does not implement Java plug-ins. In this case, there is no need to create the java-plugin and lib directories.

8.4.4 Creating all the supporting scripts

After all supporting scripts (Perl scripts, expect scripts, shells, bat files, and so on) are developed, they should be placed into the appropriate directories for packaging.

As described in 8.2, “Automation package anatomy” on page 234, all scripts files that are run on the IBM Tivoli Intelligent Orchestrator server must be copied to the bin directory that will be packaged into the .tcdriver file.

All the script files that are copied to the target systems and executed from there must be copied to the repository directory that will be packaged into the .tcdriver file.

Archived

For the Apache automation package, the content of the %home%/packages/apache/bin directory is as follows:

%home%/packages/apache/bin/ApacheConfig.exp

For the Apache automation package, the content of the %home%/packages/apache/repository directory is as follows:

%home%/packages/apache/bin/apache_config.sh

8.4.5 Creating the automation package documentation: readme file

Automation packages provide a facility to include online documentation. 3.3.7, “Documentation guidelines” on page 91 provides a great deal of information on how to document the entire Orchestration Solution Package. In addition to the documentation described in that section, it is good practice to create a readme file including, for example:

- Basic information about the automation package, such as functionality, version, support contact, and date of creation.
- A listing of the automation package’s directory structure and its contents.
- Issues, constraints, dependencies, known problems, and workarounds.
- Association to a new or existing device driver category.
- What workflows, simple commands, and Java plug-ins are provided.
- Name and description of each transition.

This documentation has to be included in a text formatted file and placed in the doc directory of the automation package directory structure. We recommend the following naming convention for the documentation file:

<automation_package_name>_README.txt

Once the package is installed into the IBM Tivoli Intelligent Orchestrator environment, the readme file can be seen in the IBM Tivoli Intelligent Orchestrator Web interface by first selecting the automation package from the appropriate device driver category used by the automation package, then selecting the Documentation tab of the automation package.

8.4.6 Creating the automation package manifest file

Our next task in creating the Apache automation package is to create the package’s manifest file.

The manifest file is an XML file named tc-driver.xml. It is stored in the %home%/packages/apache/TC-INF/ directory. It contains the name and the version of the Apache automation package, and it describes its dependencies on other automation packages, such as the core.tcdriver. It also contains the dependencies, actions, items, property, and device-models XML tags shown in Example 8-10. For more information, see 8.2, “Automation package anatomy” on page 234.

Example 8-10 Manifest file example

```
<?xml version="1.0" encoding="UTF-8"?>
<tc-driver>
   <tc-driver-format>1.0</tc-driver-format>
   <driver-name>apache</driver-name>
   <version>1.0</version>
   <description>
      The Apache automation package permits Intelligent Orchestrator to
install and configure the Apache web server server on a Linux Red Hat server
   </description>
   <documentation location="doc/readme.txt"/>
   <dependencies>
      <dependency name="core"/>
      <dependency name="rpm"/>
      <dependency name="debian-operating-system"/>
   </dependencies>
   <property name="tc.pkg"
location="com.thinkdynamics.kanaha.tcdrivermanager.action"/>
   <actions>
      <action name="command" class="${tc.pkg}.SimpleCommandActions"/>
      <action name="copy-file" class="${tc.pkg}.CopyFileActions"/>
      <action name="java-plugin" class="${tc.pkg}.JavaPluginActions"/>
      <action name="workflow" class="${tc.pkg}.WorkflowActions"/>
   </actions>
   <items>
      <item name="repository/apache_config.sh" action="copy-file">
         <param name="dest.path"
value="${tc.home}/repository/Apache/apache_config.sh"/>
         <param name="chmod" value="755"/>
      </item>
      <item name="bin/ApacheConfig.exp" action="copy-file">
         <param name="dest.path" value="${tc.home}/bin/ApacheConfig.exp"/>
         <param name="chmod" value="755"/>
      </item>
      <item name="command/Apache Red Hat Config Parameters.xml"
action="command"/>
      <item name="command/Apache Start.xml" action="command"/>
      <item name="command/Apache Stop.xml" action="command"/>
      <item name="command/Expect Install of Apache.xml" action="command"/>
      <item name="workflow/Apache Install - Red Hat.xml" action="workflow"/>
```

```
      <item name="workflow/Apache Install - Debian.xml" action="workflow"/>
   </items>
   <device-models>
      <device-model name="Apache" category="Software Products">
         <workflow name="Apache Install - Debian"/>
         <workflow name="Apache Install - Red Hat"/>
      </device-model>
   </device-models>
</tc-driver>
```

8.4.7 Pre-package verification task

Now that the manifest file for the Apache automation package is created and ready to be packaged, we need to confirm that all of the files are created correctly and are in the right directories. Example 8-11 illustrates the Apache automation package directory structure and its associated workflows, documentation file, and the manifest file to be included in the automation package file.

Example 8-11 Directory layout structure

```
Directory of C:\cygwin\home\packages\apache

03/04/2004  06:02p      <DIR>          .
03/04/2004  06:02p      <DIR>          ..
03/04/2004  06:02p      <DIR>          bin
03/04/2004  06:02p      <DIR>          command
03/04/2004  06:02p      <DIR>          doc
03/04/2004  06:02p      <DIR>          repository
03/04/2004  06:02p      <DIR>          TC-INF
03/04/2004  06:02p      <DIR>          workflow
               0 File(s)              0 bytes

 Directory of C:\cygwin\home\packages\apache\bin

03/04/2004  06:02p      <DIR>          .
03/04/2004  06:02p      <DIR>          ..
04/30/2003  04:55p               3,046 ApacheConfig.exp
               1 File(s)          3,046 bytes

 Directory of C:\cygwin\home\packages\apache\command

03/04/2004  06:02p      <DIR>          .
03/04/2004  06:02p      <DIR>          ..
09/09/2003  02:53p               8,876 Apache Red Hat Config Parameters.xml
04/30/2003  04:55p               7,189 Apache Start.xml
04/30/2003  04:55p               6,562 Apache Stop.xml
04/30/2003  04:55p               8,175 Expect Install of Apache.xml
```

```
               4 File(s)         30,802 bytes

 Directory of C:\cygwin\home\packages\apache\doc

03/04/2004  06:02p      <DIR>          .
03/04/2004  06:02p      <DIR>          ..
09/09/2003  02:54p               4,615 readme.txt
               1 File(s)          4,615 bytes

 Directory of C:\cygwin\home\packages\apache\repository

03/04/2004  06:02p      <DIR>          .
03/04/2004  06:02p      <DIR>          ..
09/09/2003  02:54p              10,675 apache_config.sh
               1 File(s)         10,675 bytes

 Directory of C:\cygwin\home\packages\apache\TC-INF

03/04/2004  06:02p      <DIR>          .
03/04/2004  06:02p      <DIR>          ..
09/10/2003  06:02p               1,843 tc-driver.xml
               1 File(s)          1,843 bytes

 Directory of C:\cygwin\home\packages\apache\workflow

03/04/2004  06:02p      <DIR>          .
03/04/2004  06:02p      <DIR>          ..
04/30/2003  04:56p              27,119 Apache Install - Debian.xml
09/09/2003  02:56p             267,126 Apache Install - Red Hat.xml
               2 File(s)        294,245 bytes
```

8.4.8 Packaging the automation package into a .tcdriver file

The next step in the process is to package the Apache automation package and its associated files into a Java archive (jar) formatted file.

The packaging is done by using the **`jar -cvf`** command to build the .tcdriver file, including all of the files in the automation package directory structure listed in the previous section. We have used the shell script called buildpackage.sh, listed in Example 8-12 on page 251, that executes this task.

Example 8-12 buildpackage.sh script

```
#!/bin/bash
#==============================================================================
# This is bash shell script to create a jar file with a (.tcdriver)extension.
# It executes inside a Cygwin terminal shell.
#
# It is provided “AS IS” to assists in packaging an automation package.
#
# It jars all subdirectories in the ~/packages directory, then placing them
# in the drivers directory.
# It must be executed within the ~/packages directory.
#==============================================================================
#
list=`ls -1`
pwd=`pwd`

#=========================================================
#
# Test for available directories; if successful,
# jar the contents into a single file with its name
# identical to the directory name and .tcdriver as its
# extension.
# For example; apache.tcdriver
#
#=========================================================
#
for d in $list
do
   test -d ${d} && {
      cd ${d}
      echo "jar -cvf ../../drivers/${d}.tcdriver *"
      jar -cvf ../../drivers/${d}.tcdriver *
      cd ${pwd}
   }
done
```

Perform the following tasks to create the package:

1. Open a Cygwin command window.
2. Change the directory to the $TC_HOME/tools directory.
3. Create or cop and paste the UNIX shell script into a file, give it a name, and save it in your current directory. (For example, we move buildpackage.sh into $TC_HOME/tools.)
4. Move to the package directory.

Archived

5. Execute the buildpackage.sh script (or run the **jar -cvf** command manually).

Upon successful execution, we have a new file with the .tcdriver extension in the drivers directory, $TC_HOME/drivers (named apache.tcdriver in this case). This concludes the packaging process.

Java's **jar** command options can be used to view or extract content of the automation package file (illustrated in Example 8-13):

```
jar -tvf filename.tcdriver - to view the driver's table of content
jar -xvf filename.tcdriver - to extract driver's content
```

Example 8-13 Content of the itso-cluster.tcdriver file

```
C:\>jar -tvf C:\cygwin\home\drivers\apache.tcdriver
     0 Fri Nov 28 06:15:52 CST 2003 TC-INF/
     0 Fri Nov 28 06:15:52 CST 2003 bin/
     0 Fri Nov 28 06:15:52 CST 2003 command/
     0 Fri Nov 28 06:15:52 CST 2003 doc/
     0 Fri Nov 28 06:15:52 CST 2003 repository/
     0 Fri Nov 28 06:15:52 CST 2003 workflow/
  1843 Wed Sep 10 18:02:18 CDT 2003 TC-INF/tc-driver.xml
  3046 Wed Apr 30 16:55:38 CDT 2003 bin/ApacheConfig.exp
  8876 Tue Sep 09 14:53:50 CDT 2003 command/Apache Red Hat Config
Parameters.xml
  7189 Wed Apr 30 16:55:46 CDT 2003 command/Apache Start.xml
  6562 Wed Apr 30 16:55:46 CDT 2003 command/Apache Stop.xml
  8175 Wed Apr 30 16:55:46 CDT 2003 command/Expect Install of Apache.xml
  4615 Tue Sep 09 14:54:24 CDT 2003 doc/readme.txt
 10675 Tue Sep 09 14:54:50 CDT 2003 repository/apache_config.sh
 27119 Wed Apr 30 16:56:00 CDT 2003 workflow/Apache Install - Debian.xml
267126 Tue Sep 09 14:56:30 CDT 2003 workflow/Apache Install - Red Hat.xml
```

8.4.9 Deploying the automation package

In order to deploy the newly created automation package, we must install the driver into the IBM Tivoli Intelligent Orchestrator server using the **tc-driver-manager command**. In our Apache example, we performed these tasks:

- Because we created the automation package file on a machine other than the IBM Tivoli Intelligent Orchestrator server, we have to transfer (copy) the apache.tcdriver file into the drivers' directory (%TC_HOME%/thinkcontrol/drivers) on the IBM Tivoli Intelligent Orchestrator server platform.
- On the IBM Tivoli Intelligent Orchestrator server, execute the **tc-driver-manager** command. For details on how to use the

tc-driver-manager command, refer to *Provisioning On Demand Introducing IBM Tivoli Intelligent ThinkDynamic Orchestrator*, SG24-8888.

If the command is successful, a message indicating a successful completion of the automation package installation is shown, as seen in Example 8-14.

> **Note:** Remember to *not* include the .tcdriver extension in the command line. The **tc-driver-manager** command expects the *name* of the automation package, not its file name.

Example 8-14 .tc-driver-manager command: successful driver installation

```
tioadmin@tiosvr ~/drivers
$ ../tools/tc-driver-manager.cmd installDriver apache
ObjectView Persistency Layer version 6.4.38 looking for initialization file:
/C:/cygwin/home/thinkcontrol/config/ObjectV
iew.properties
COM.ibm.db2.jdbc.app.DB2Driver version: 8.1 loaded.
XML alternative persistency storage is set to file:
C:\cygwin\home\thinkcontrol\logs\tcdrivermanager\objectDump.xml
ObjectView Persistency Layer version 6.4.38 finished initialization

Installation successful. (Driver name:apache)
```

8.4.10 Verifying deployment

There are two ways to verify whether the automation package installation process is successful:

1. Using the command line interface:

   Issue the **tc-driver-manager getDriverStatus** command. For example, to check the status of the Apache automation package, we issued the following command:

   ```
   tc-driver-manager.cmd getDriverStatus apache
   ```

Archived

This command creates an output indicating the installation status of the named driver (for example, apache), as shown in Example 8-15.

Example 8-15 tc-driver-manager command: getDriverStatus

```
tioadmin@tiosvr ~/drivers
$ ../tools/tc-driver-manager.cmd getDriverStatus apache
ObjectView Persistency Layer version 6.4.38 looking for initialization file:
/C:/cygwin/home/thinkcontrol/config/ObjectV
iew.properties
COM.ibm.db2.jdbc.app.DB2Driver version: 8.1 loaded.
XML alternative persistency storage is set to file:
C:\cygwin\home\thinkcontrol\logs\tcdrivermanager\objectDump.xml
ObjectView Persistency Layer version 6.4.38 finished initialization
installed
```

2. Using the Web Management Interface:
   - Click the **System configuration and workflow management** tab.
   - Click **Device Drivers**.
   - Click **Software Products**.

   If the installation is successful, the Apache Driver shows up under the Software Products tab, as shown in Figure 8-1.

Figure 8-1 Apache Driver view

Note: After an automation package has been deployed, subsequent modifications to its assigned workflows via the Web-based interface will outdate the workflows in the automation package .tcdriver file.

Therefore, after a new or modified workflow has been tested and is verified to deliver the intended function, it should be assigned and packaged within the appropriate automation package. The automation package manifest file should be updated to reflect the change concerning the new workflow, the associated command, scripts, and plug-ins should be registered with the manifest, and the automation package file should be updated and repackaged accordingly. This keeps the DCM object and the automation package in sync.

Error handling

The **`tc-driver-manager`** command logs errors occurring during the installation of a single driver or the bulk installation of all available drivers during, for example, the DCM reinit process. The directory used is:

%TC_HOME%/thinkcontrol/logs/tcdrivermanager

Archived

8.5 Generating the automation package using APDE

Using APDE to generate the automation package is far easier than the manual process described in the previous section.

To create an automation package for a specific project, expand the project and right-click the build.xml file, and select **Run** → **2 Ant Build...**, as shown in Figure 8-2.

Figure 8-2 Launch the automation package builder

Once the Package Builder is launched, you will be presented with a dialog similar to the one shown in Figure 8-3, in which you can select build options.

Figure 8-3 Select build options

Make sure that the *package* target (available in the Targets tab) is selected, and press **Run**. The build process is started, and in the **Console** pane of the APDE console, you will see the results. Figure 8-4 on page 258 shows a successful build of our TioWS-Labs package. Notice that the automation package file (TioWS-Labs.tcdriver) shows up in the Package Explorer pane.

Figure 8-4 Build results

When selecting the newly created tcdriver file, you will notice that two buttons in the toolbar becomes active. These will allow you to install and uninstall the selected automation package in the TIO/TPM server to which the APDE environment is connected. This is a easy alternative to transferring the tcdriver file to the TPM serve itself, and running the **`tc-driver-manager`** command manually.

Archived

9

Case study: basic workflows

In the current chapter, we will guide the reader through the development of a few basic workflows demonstrating the use of various common workflow development features and techniques.

The following basic workflows are covered:

1. Get_DeploymentEngine_DeviceID
2. LAB1 - Execute_Local_Command
3. LAB2 - Getting information from the DCM using DCMQuery
4. Jython string manipulation
5. Implementing Logical Device Operations
6. Using DCMInsert
7. Working with SAPs
8. Scriptlets and exceptions

© Copyright IBM Corp. 2004, 2006. All rights reserved.

9.1 Get_DeploymentEngine_DeviceID

This workflow demonstrates the lookup of the ID of the TIO Server (Deployment Engine) itself using a Java plug-In.

9.1.1 Objective

Write a workflow to obtain and return the DeviceID of the TIO server from a Java plug-In.

9.1.2 Development steps

Perform the following steps to complete the development of this workflow:

1. Using the TIO Web UI, add a new workflow named Get_DeploymentEngine_DeviceID and make it locale insensitive.
2. Add an output parameter 'DeviceID' to the workflow.
3. Execute the Java lug-In 'Get Deployment Engine Device ID'.
4. Associate the 'DeviceID' variable to the 'DE Device ID' output parameter of the Java plug-in.
5. Compile and execute the workflow.
6. Check the status of the workflow execution.
7. Check the result in the workflow execution history.

Figure 9-1 on page 261 shows the Get_DeploymentEngine_DeviceID workflow.

Archived

Figure 9-1 Get_DeploymentEngine_DeviceID workflow

Example 9-1 shows the completed workflow.

Example 9-1 Get_DeploymentEngone_DeviceID workflow listing

```
workflow Get_DeploymentEngine_DeviceID(out DeviceID) LocaleInsensitive

java:com.thinkdynamics.kanaha.de.javaplugin.datacentermodel.FindDEDevic
eId(DeviceID)
```

9.2 LAB1 - Execute_Local_Command

This workflow demonstrates how to execute a command on the TIO Server itself.

9.2.1 Objectives

A Write a workflow to obtain the IP configuration of your TIO server.

B Allow the workflow to execute any command taken as an input parameter.

9.2.2 Development steps (method A)

Complete the following steps to obtain the IP configuration of your TIO Server:

1. Using the TIO Web UI, add a new workflow - named LAB1 - and make it locale insensitive.
2. Add a variable to the workflow and name it ‘command’.
3. Associate the variable to an expression element.
4. Edit the expression element and give it the explicit value of ‘ipconfig’.

Archived

5. Add the ‘Execute_Local_Command’ workflow.
6. Map the command variable to the input parameter ‘CommandString’.
7. Edit the ‘StartDirectory’ input parameter and give it the explicit value of ‘/’.
8. Compile and execute the workflow.
9. Check the status of the workflow execution.
10. Check the result in the workflow execution history.

Figure 9-2 shows the Execute_Local_IPconfig workflow.

Figure 9-2 Execute_Local_IPconfig workflow

Example 9-2 shows the listing of the completed workflow.

Example 9-2 Execute_Local_IPconfig workflow (step A) listing

```
# Steps A
workflow LAB1 LocaleInsensitive
var command
command = "ipconfig"
Execute_Local_Command(command,<null>,<null>,<null>,"/",<null>,<null>,<null>, <null>)
```

9.2.3 Development steps (method B)

Complete the following steps to allow the workflow to accept input parameters:

1. Delete the expression comment and the definition of the variable 'command'.
2. Add an input parameter to the workflow named 'command'.
3. Add an output parameter to the workflow named 'result'.
4. Assign the ReturnResult parameter returned by ExecuteLocalCommand to the variable 'result'.

Archived

5. Repeat steps 8-10 from 9.2.2, “Development steps (method A)” on page 261, providing a value of 'ipconfig' for the input parameter 'command'.

Figure 9-3 shows the Execute_Local_IPconfig workflow with parameter input.

Figure 9-3 Execute_Local_IPconfig workflow with parameter input

Example 9-3 shows the listing of the completed workflow.

Example 9-3 Execute_Local_IPconfig workflow with parameter input (step b)

```
# Steps B
workflow LAB1(in command, out result)  LocaleInsensitive
Execute_Local_Command(command,<null>,<null>,result,"/",<null>,<null>,<null>,<null>)
```

9.3 LAB2 - Getting information from the DCM using DCMQuery

The following case study demonstrates how to use DCMQuery.

9.3.1 Objective

Write a workflow to provide the host name of your ITIO server to other workflows. Use a Jython string function to strip leading and trailing spaces.

A Call another workflow to obtain the results.
B Use DCMQuery to gather attributes from the Data Center Model.

9.3.2 Development steps (method A)

This method can be used to get the local host name of any system.

1. Using the TIO Web UI, add a new workflow - named LAB2 - and make it locale insensitive.

2. Define an output parameter named 'tio_hostname'.
3. Invoke the LAB1 workflow parsing the string "hostname" and receiving the result in the 'tio_hostname' variable.
4. Use the Jython(<variable>.strip) function to strip leading and trailing spaces.
5. Compile and execute the workflow.
6. Check the status of the workflow execution.
7. Check the result in the workflow execution history.

Figure 9-4 shows a DCMQuery workflow.

Figure 9-4 Workflow with DCMQuery

Example 9-4 lists the DCMQuery workflow.

Example 9-4 DCMQuery workflow listing

```
workflow LAB2(out tio_hostname) LocaleInsensitive
# method A
LAB1("hostname",tio_hostname)
#strip leading and trailing whitespace
tio_hostname = Jython[tio_hostname.strip()]
```

9.3.3 Development Steps (method B): Designed specifically for the TIO Server

This method can be used to get the local host name of any system:

1. Using the TIO Web UI, add a new workflow named LAB2 and make it locale insensitive.
2. Define an output parameter named 'tio_hostname'.
3. Define a variable named DeviceID.

4. Invoke the Get_DeploymentEngine_DeviceID workflow to retrieve the DeviceID of the TIO Server.
5. Obtain the host name of the TIO Server using the DeviceID as key in a DCMQuery call.
6. Assign the host name to the 'tio_hostname' variable.
7. Compile and execute the workflow.
8. Check the status of the workflow execution.
9. Check the result in the workflow execution history.

Figure 9-5 shows the DCMQuery workflow for TIO Server.

Figure 9-5 DCMQuery workflow for TIO Server

Example 9-5 lists the DCMQuery for TIO Server workflow.

Example 9-5 DCMQuery Workflow for TIO Server

```
workflow LAB2(out tio_hostname) LocaleInsensitive
# method B
var DeviceID
Get_DeploymentEngine_DeviceID(DeviceID)
tio_hostname = DCMQuery(/Server[@id=$DeviceID]/@name)
```

Archived

9.4 Jython string manipulation

The following demonstrates some common Jython string manipulation techniques that may be used in your workflows.

9.4.1 Objective

Write a workflow using **`net send`** to display a message on the TIO Server.

The workflow must be able to

- Receive a text message to be displayed on the TIO Server.
- Receive a command as input, execute it, and send the output to the TIO Server.

9.4.2 Development steps

Complete the following steps to create the Jython string manipulation sample workflow:

1. Using the ITIO Web interface, add a new workflow named LAB3 and make it locale insensitive.
2. Add a input parameters 'Info' and 'InfoType'.
3. Add an output parameter named 'ResultValue'.
4. Add the 'command', 'hostname', and 'result' variables.
5. Use an If-then construct to determine if the 'InfoType' received indicates the parsing of a command.
6. If using a command, execute using the LAB1 workflow, and assign the output from LAB1 to the 'result' variable; otherwise, assign the value of the 'Info' parameter to the 'result' variable.
7. Obtain the host name of the TIO Server using workflow LAB2 and assign the output value to the host name variable.
8. Use Jython string concatenation to bind the command **`net send <hostname> <result>`**.
9. Use the LAB1 workflow to execute the **`net send`** command.
10. Compile and execute the workflow.
11. Check the status of the workflow execution.
12. Check the result in the workflow execution history.

Figure 9-6 on page 267 shows the Jython string manipulation workflow.

Archived

Figure 9-6 Jython string manipulation workflow

Example 9-6 lists the completed workflow.

Example 9-6 Jython string manipulation workflow listing

```
workflow LAB3(in Info, in Infotype, out ReturnValue)  LocaleInsensitive
var command
var result
if Jython[ Infotype == "command"] then
  LAB1(Info, result)
else
  result = Info
endif
# get hostname
command = "hostname"
LAB2(command)
# build and execute command
command = Jython( "net send " + hostname + " ' " + result + " ' ")
LAB1(command, ReturnValue)
```

9.5 Implementing Logical Device Operations

The following demonstrates how to implement Logical Device Operations in order to assign your own special processing to a predefined operation for a specific device type.

9.5.1 Objective

Create a workflow that implements the Cluster.AddServer logical operation and displays a message, including server and cluster IDs and names, on the TIO Server when a new server is added to a cluster.

Optional: Log the message in the TIO execution log.

9.6 Development steps

Complete the following steps to develop your LDO workflow:

1. Using the ITIO Web interface, add a new workflow named LAB4, make it locale insensitive, and let it implement the Cluster.AddServer. LogicalOperationNote: Parameters 'ClusterID' and 'ServerID' are added automatically.
2. Add the 'ServerName', 'ClusterName', 'message', and 'result' variables.
3. Optional: Set defaults for 'ClusterID' and 'ServerID' parameters (to ease testing).
4. Use DCMQuery to retrieve the ServerN and Cluster names into 'ServerName' and 'ClusterName' variables.
5. Build a message using Jython concatenation.
6. Log a informational message in the TIO log.
7. Send the message to the TIO Server using workflow LAB3 (Ignore the result from the LAB3 workflow invocation).

9.6.1 Test

1. Select the application cluster currently configured for simulation and add your new workflow.
2. Manually add and remove a server from the cluster and verify that you receive the appropriate messages.

Figure 9-7 on page 269 shows the workflow implementing a LDO.

Archived

Tip: Ensure that the Messenger service is started on all systems.

Figure 9-7 Workflow implementing a Logical Device Operation

Example 9-7 lists the completed workflow.

Example 9-7 Workflow implementing an LDO listing

```
workflow LAB4(in ClusterID,inout ServerID) implements Cluster.AddServer
LocaleInsensitive
var ClusterName
var ServerName
var message
var result
var ClusterId = Jython(ClusterID or "1299")
var ServerId = Jython(ServerID or "1489")
ClusterName = DCMQuery(/Cluster[@id=$ClusterId]/@name)
ServerName = DCMQuery(/Server[@id=$ServerId]/@name)
message = Jython("Adding server '''" + ServerName + "''' (" + ServerId + "\) to cluster
'''" + ClusterName + "''' (" + ClusterId + "\) " )
log info Jython(message)
LAB3(message, "message", result)
```

9.7 Using DCMInsert

The following section demonstrates how to use DCMInsert from a workflow.

9.7.1 Objective

Write a workflow to create a new SAP (ipv4/SSH, host with authentication) to be used as the default SAP for file transfer and command execution at a server. Add both RSA (default) and password credentials.

9.7.2 Development steps

Complete the following steps to create a workflow that uses DCMInsert:

1. Using the ITIO Web interface, add a new workflow named LAB5 and make it locale insensitive.
2. Define an input parameter named 'ServerName'.
3. Create a suitable DCMInsert statement to create a new SAP with RSA credentials for a server. Look in an exported version of your DCM for the xml-syntax.
4. Compile and execute the workflow.
5. Check the status of the workflow execution.
6. Check the result in the workflow execution history.

Challenge

Delete any existing SAPs for the same protocol/domain/context/role combination to allow for automatic re-creation. Remember to remove credentials before deleting the SAP. Consider using the APDE environment to get help for creating the DCM commands.

Figure 9-8 on page 271 shows the DCMInsert workflow.

Archived

Figure 9-8 DCMInsert workflow

Example 9-8 lists the completed DCMInsert workflow.

Example 9-8 DCMInsert workflow listing

```
workflow LAB5(in ServerName) LocaleInsensitive
var ServerID = DCMQuery(/server[@name=$ServerName]/@id)
var SapID
var name
var appprotocol
var appprotocolTypeID
var protocol
var protocolTypeID
var port
var host
var newSAPProtocol = "SSH"
var newSAPPort = "22"
var newSAPHost = "Y"
var CreateNewSAP = "1"
##  list existing SAPs and delete the one to be (re)created
##  identified in the variables newSAPxxx
foreach SapID in DCMQuery(/server[@id=$ServerID]/sap/@id) do
  name = DCMQuery(/sap[@id=$SapID]/@name)
  protocolTypeID = DCMQuery(/sap[@id=$SapID]/@protocoltypeid)
```

```
  protocol = DCMQuery(/protocoltype[@id=$protocolTypeID]/@name)
  appprotocolTypeID = DCMQuery(/sap[@id=$SapID]/@appprotocolid)
  appprotocol =
DCMQuery(/applicationprotocol[@id=$appprotocolTypeID]/@name)
  port = DCMQuery(/sap[@id=$SapID]/@port)
  host = DCMQuery(/sap[@id=$SapID]/@host)
  log info Jython(SapID + " is called:" + name + " Protocol: " +
protocol + "/"        + appprotocol + " port: " + port + " host:" +
host)

  if Jython[ (appprotocol == newSAPProtocol ) and (port == newSAPPort)
and  host == newSAPHost ] then
    ##  we got a hit, now delete the credentials
    var CredID
    var CredName
    var CredProto
    var CredType
    var CredTypeID
    foreach CredID in DCMQuery(/sap[@id=$SapID]/credentials/@id) do
      CredName  = DCMQuery(/credentials[@id=$CredID]/@searchkey)
      CredProto =
DCMQuery(/credentials[@id=$CredID]/@protocolendpointid)
      CredTypeID  =
DCMQuery(/credentials[@id=$CredID]/@credentialstypeid)
      CredType  = DCMQuery(/credentialstype[@id=$CredTypeID]/@name)
      log info Jython("     id:" + CredID + " type:" + CredType + "
search:"+            CredName + " Proto:" + CredProto)
      if Jython(CredType == "RSA") then
        DCMDelete(/rsacredentials[@id=$CredID])
      endif
      if Jython(CredType == "SNMP") then
        DCMDelete(/snmpcredentials[@id=$CredID])
      endif
      if Jython(CredType == "password") then
        DCMDelete(/passwordcredentials[@id=$CredID])
      endif
    done

    ##  and finally we can delete the SAP itself
    DCMDelete(/sap[@id=$SapID])
  endif
done

## ready to create the new SAP
if Jython[ int(CreateNewSAP) == 1 ] then
```

```
  ## The CreateNewSAP variable is only used to control execution during
test
  DCMInsert parent = DCMQuery(/server[@id=$ServerID]) <<EOF
    <sap name="ssh-host" port="$newSAPPort"
app-protocol="$newSAPProtocol"
         is-device-model="SSH Service Access Point" host="true"
         auth-compulsory="true" locale="en_US">
    <default-sap operation-type="execute-command"/>
    <default-sap operation-type="file-transfer"/>
    <credentials search-key="default" is-default="true">
       <rsa-credentials username="Administrator"/>
    </credentials>
    <credentials search-key="passwd" is-default="false">
       <password-credentials username="Administrator"
password="smartway"/>
    </credentials>
    </sap>
  EOF
endif
```

9.8 Working with SAPs

The following section demonstrates how to create Service Access Points from a workflow.

9.8.1 Objective

Write a workflow to copy and execute a script on a remote server.

9.8.2 Development steps

1. Create a script $TIO_HOME/repository/test.sh that appends the current timestamp to the file /tmp/tst.log.

```
#!/bin/sh
echo `date `  : $0 executed  >> /tmp/test.log
exit $?
```

2. Using the ITIO Web interface, add a new workflow and make it locale insensitive.
3. Define input parameter DestinationID.

Archived

4. Use Get_DeploymentEngine_DeviceID to find the objectID of the TIO Server.
5. Use Java function call 'Java[java.lang.System#getProperty("tio.home.unix")]' to get $TIO_HOME.
6. Make sure SSH and SCP HOST SAP for the remote system has been defined.
7. Make sure SSH and SCP CLIENT SAP for the ITIO Server has been defined.
8. Use the Default_Device_Copy_File workflow to copy the script to the remote server.
9. Use the Default_Device_Execute_Command workflow to execute the script on the remote server.
10. Compile and execute the workflow.
11. Check the status of the workflow execution.
12. Check the result in the workflow execution history.

Figure 9-9 shows the creation of an SAP workflow.

Figure 9-9 Create SAP workflow

Example 9-9 lists the completed workflow.

Example 9-9 SAP workflow listing

```
workflow LAB6(in DestinationID) LocaleInsensitive
  var SourceID
  Get_DeploymentEngine_DeviceID(SourceID)
  var TioHome = Java[java.lang.System#getProperty("tio.home.unix")]
  var srcpath = Jython(TioHome + "/repository")
  var srcfile = "test.sh"
  var tgtpath = "/tmp"
  var tgtfile = srcfile
  var srcname = DCMQuery(/Server[@id=$SourceID]/@name)
  var tgtname = DCMQuery(/Server[@id=$DestinationID]/@name)
  var ReturnCode
  var ReturnResult
  var ReturnErrorString

log info Jython("About to copy: " + srcfile + " from " + srcpath + " at
" + srcname +" to " + tgtfile + " in " + tgtpath + " at " + tgtname)
  Device.CopyFile(SourceID, srcpath, srcfile, DestinationID, tgtpath,
tgtfile, <null>, <null>, "120")

var tgtcommand = Jython(tgtpath + "/" + tgtfile)
tgtpath = "/"
log info Jython("About to execute: " + tgtcommand + " from " + tgtpath
+ " on " + tgtname)
Device.ExecuteCommand(DestinationID,  tgtcommand, tgtpath, <null>,
"120", <null>, ReturnCode, ReturnErrorString, ReturnResult)
```

9.9 Scriptlets and exceptions

The following demonstrates how to invoke a simple scriptlet from a workflow.

9.9.1 Objective

Write a workflow that uses a scriptlet to send a message containing a directory listing of a remote system on the TIO Server.

Archived

9.9.2 Development steps

1. Using the ITIO Web interface, add a new workflow and make it locale insensitive.
2. Define an input parameter named ServerName.
3. Get the host name of the TIO Server using Get_DeploymentEngine_DeviceID and a DCMQuery.
4. Get the ObjectID of the target system using a DCMQuery.
5. Define a scriptlet and parse the host name of the TIO Server to the scriptlet.
6. Define the logic of the servlet (you may add logging from the scriptlet using TIOlog). Remember to return a termination code (return $?) to avoid the scriptlet from hanging.
7. Compile and execute the workflow.
8. Check the status of the workflow execution.
9. Check the result in the workflow execution history.

9.9.3 Challenges

1. Modify a variable from the scriptlet to send data back to the workflow.
2. Generate the data for the message in a DOS command shell.
3. Verify connectivity prior to running the scriptlet, and abort if target system does not respond.

Figure 9-10 on page 277 shows part of the initialization scriptlet in a workflow.

Archived

```
workflow LAB7( ServerName )
    var TIODeviceID
    var string = "unknown"
    Get_DeploymentEngine_DeviceID( out TIODeviceID )
    var TIOHostName = DCMQuery(/Server[@id=$TIODeviceID]/@name)
    var DeviceID = DCMQuery(/Server[@name=$ServerName])
    # ######## ------- challenge 3 --v
    var ipAddress
    var pingTimeout = "60"
    var pingTime
    try
    # ######### ------- challenge 3 --^
    log info Jython("Scriptlet is about to start at: " + ServerName + ". Messages are sent to: " + TIOHostName)
      scriptlet ( TIOHostName ) language = bash
    log info Jython("scriptlet executed " + string + "' at " + ServerName + " successfully")
    ## ----- challenge 1 --^
```

Figure 9-10 Workflow with scriptlet - part 1 - initialize

Figure 9-11 shows the scriptlet embedded in the workflow.

```
scriptlet ( TIOHostName ) language = bash
| Parameter | In/Out | Array | Value |
| <param1> | | | TIOHostName |
target: DCMQuery(/Server[@name=$ServerName]) timeout: 120

    cygfile=/tmp/out
    export dosfile=`cygpath -w ${cygfile}`
    export TIOHostName
    ######### -----  challenge 2 --v
      cmd <<EOFDOS
         @echo off
         dir /B /Q /X %TEMP% > %dosfile%
         net send %TIOHostName% "Generated list of %TEMP% in: %dosfile%"
      EOFDOS
      TIOlog info "Return code from filelist generation was: $?"
    ######### -----  challenge 2 --^
    txt=`cat ${cygfile}`
    exe="net send ${TIOHostName} "
    TIOlog info "About to execute ${exe}"
    `${exe} \"${txt}\"`
    ######### -----  challenge 1 --v
      TIOsetVar string "$exe"
      return $?

log info Jython("scriptlet executed " + string + "' at " + ServerName + " successfully")
## ----- challenge 1 --^
```

Figure 9-11 Workflow with scriptlet - part 2 - the scriptlet

Figure 9-12 shows the error handling and cleanup part of the scriptlet.

```
# ######## ------- challenge 3 --v
var ipAddress
var pingTimeout = "60"
var pingTime
try
    Get_Server_Mgmt_IP_Address( DeviceID, out ipAddress )
    Device.Ping( DeviceID, ipAddress, pingTimeout, "default", out pingTime )
catch EmbeddedLDOException
    var msg = Jython("Server " + ServerName + " did not reply from " + ( ipAddress or "unknown" ) + "
    within " + pingTimeout + " seconds")
    log error msg
    throw my_own_SNMP_Ping_Exception msg
catch-all (errorMessage)
    log error Jython("reveived exception: " + errorMessage)
    rethrow
finally
    log info Jython("Server " + ServerName + " responded at " + ( ipAddress or "unknown" ) + " in " +
    ( pingTime or "unknown" ) + " miliseconds")
endtry
# ######## ------- challenge 3 --^
```

Figure 9-12 Workflow with scriptlet - part 3- error handling and cleanup

Example 9-10 on page 279 lists the entire workflow.

Archived

Example 9-10 Workflow with scriptlet listing

```
workflow LAB7 (in ServerName) LocaleInsensitive
var TIODeviceID
var string = "unknown"
Get_DeploymentEngine_DeviceID(TIODeviceID)
var TIOHostName = DCMQuery(/Server[@id=$TIODeviceID]/@name)
var DeviceID = DCMQuery(/Server[@name=$ServerName])
######### ------- challenge 3 --v
var ipAddress
var pingTimeout = "60"
var pingTime
try
    Get_Server_Mgmt_IP_Address(DeviceID,ipAddress)
    Default_Device_Ping(DeviceID,ipAddress, pingTimeout, "default" , pingTime)
  catch EmbeddedLDOException
    var msg = Jython("Server " + ServerName + " did not reply from " + (
              ipAddress or "unknown" \) + " within " + pingTimeout + " seconds")
    log error msg
    throw my_own_SNMP_Ping_Exception msg
  catchall errorMessage
    log error Jython("reveived exception: " + errorMessage)
    rethrow
  finally
    log info Jython("Server " + ServerName + " responded at " + ( ipAddress or
           "unknown" \) + " in " + ( pingTime or "unknown" \) + " miliseconds")
endtry
######### ------- challenge 3 --^
log info Jython("Scriptlet is about to start at: " + ServerName + ".  Messages
         are sent to: " + TIOHostName)
scriptlet (TIOHostName) language=bash target=DCMQuery(/Server[@name=$ServerName])
          timeout = 120 <<EOF
    cygfile=/tmp/out
    export dosfile=`cygpath -w ${cygfile}`
    export TIOHostName
    ######### -----  challenge 2 --v
    cmd <<EOFDOS
        @echo off
        dir /B /Q /X %TEMP% > %dosfile%
        net send %TIOHostName% "Generated list of %TEMP% in: %dosfile%"
    EOFDOS
    TIOlog info "Return code from filelist generation was: $?"
    ######### -----  challenge 2 --^
    txt=`cat ${cygfile}`
    exe="net send ${TIOHostName} "
    TIOlog info "About to execute ${exe}"
    `${exe} \"${txt}\"`
    ######### -----  challenge 1 --v
    TIOsetVar string "$exe"
    return $?
EOF
log info Jython("scriptlet executed '"+string+"' at "+ServerName+" successfully")
######### -----  challenge 1 --^
```

Archived

Archived

10

Case study: Trade3 composite application

Archived

© Copyright IBM Corp. 2004, 2006. All rights reserved.

10.1 Overview of case study

In our case study, we will show detailed examples of best practices around architecting, developing, packaging, and implementing an Automation Package provided for a WebSphere Application Server that uses the WebSphere Edge Server Load Balancer as well.

Our fictitious customer has a request to run the Trade application and to automatically provision additional servers in its datacenter, in which IBM Tivoli Intelligent Orchestrator or Provisioning Manager is already installed. This includes all necessary steps to install the Trade application and the configuration of the supporting load balancer.

In addition, the removal of the Trade application is included.

Figure 10-1 shows an high-level overview of the Trade environment.

Figure 10-1 Trade - standard installation on a single system

10.2 The Trade application

This section describes the main architecture of Trade3 installed on top of an existing IBM WebSphere Application Server on a single system, and also includes the official installations instructions for the Trade3 application.

Archived

10.2.1 Main architecture of the Trade3 application

The IBM WebSphere performance benchmark application Trade3 provides a suite of IBM developed workloads for characterizing the performance of the IBM WebSphere Application Server. The workloads consist of an end-to-end Web application and a full set of primitives. The applications are a collection of Java classes, Java Servlets, Java Server Pages, and Enterprise Java Beans built to open J2EE APIs (see Figure 10-2). Together, these provide versatile and portable test cases designed to measure aspects of scalability and performance.

Figure 10-2 Trade J2EE components - Model-View-Controller architecture

Trade3 is the third release of the IBM WebSphere end-to-end benchmark and performance application, and models an online stock brokerage application. This provides a real world workload driving IBM WebSphere's implementation of J2EE 1.3 and Web Services, including key IBM WebSphere performance components and features (DynaCache, WebSphere Edge Server, and Web Services).

Trade3's design spans J2EE 1.3, including the EJB 2.0 component architecture, Message Driven beans, transactions (1-phase, 2-phase commit), and Web Services (SOAP, WSDL, and UDDI).

For additional informations about the application and to download the code, please refer to:

http://www-306.ibm.com/software/webservers/appserv/benchmark3.html

10.2.2 Trade3 installation instructions

The following provides the official installation instructions for the Trade3 application, as they appear in the ReadMe.html file that is included in the Trade3 installation archive trade3.0_WAS5.0Install.zip, which can be downloaded from http://www-306.ibm.com/software/webservers/appserv/benchmark3.html.

Please note that compared to the original material, the information in the following has been slightly reformatted to fit the current publishing media:

Product WebSphere Application Server Advanced Edition - Single Server
Release Aquila_ASV
Build WebSphere Application Server 5.0

Note: All text in *italic* must be customized to match your system.

Step 1 - Prepare for installation

1. Download and unzip the Trade3 install package.
2. Open a shell/command window to perform the install and change to the t3install directory.
3. Run the WebSphere Application Server **setupCmdLine** script to set up your shell environment:
   - UNIX: **. <WAS_HOME>/bin/setupCmdLine.sh**
   - Win32: **<WAS_HOME>\bin\setupCmdLine.bat**
4. Start the WebSphere V5.0 server
   - UNIX: . **/home/db2inst1/sqllib/db2profile**

     ${WAS_HOME}/bin/startServer server1
   - Win32: **%WAS_HOME%\bin\startServer server1**

Step 2 - DB2

Create the DB2 database for Trade3 and modify DB2 for read stability.

1. If you are running DB2 Version 7, verify that you have run **<*sqllib*>/java12/usejdbc2.[bat | sh]** to configure DB2 to use the JDBC 2.0 driver.
2. Set up a DB2 shell using the following method for UNIX or Win32:
   - UNIX: **su - db2username**
   - Win32:**db2cmd**

3. Execute the following DB2 commands from the Trade3 install directory using the DB2 shell just created:

```
db2 create db trade3db
db2 connect to trade3db
db2 -tvf DB2/Table.ddl
db2 disconnect all
db2set DB2_RR_TO_RS=yes
db2 update db config for trade3db using logfilsiz 1000
db2 update db cfg for trade3db using maxappls 100
db2stop force
db2start
```

4. Execute the following commands to bind DB2 packages for the trade3db.

```
db2 connect to trade3db
cd <sqllib>/bnd
db2 bind @db2cli.lst blocking all grant public
```

Note: Failure to rebind will result in the following run time error:

```
COM.ibm.db2.jdbc.DB2Exception: [IBM][CLI Driver][DB2/NT] SQL0805N
Package "NULLID.SQLLC300" was not found. SQLSTATE=51002
```

Linux Note: Running DB2 on Linux requires the trade3db database to be cataloged to use the TCP/IP communication protocol. Please refer to the WebSphere installation guide for details.

Step 2 - Oracle

As a user with the right Oracle environment variables setup:

1. Set up a database or reuse an existing database.
2. This example assumes a database SID of trade3db.
3. Create the trade user: user name of trade, password of trade.
4. Load the schema for the trade user:

```
sqlplus trade/trade@trade3db @Oracle/Table.ddl
```

Step 3 - Install, configure, and start Trade3

1. Install Trade3 JDBC/JMS resources interactively:
   – UNIX: **$WAS_HOME/bin/ws_ant -f Trade3.xml installResources**
     Win32:**%WAS_HOME%\bin\ws_ant -f Trade3.xml installResources**

Archived

2. Install the Trade3 Application:
   - UNIX:**$WAS_HOME/bin/ws_ant -f Trade3.xml installApp**
     Win32:**%WAS_HOME%\bin\ws_ant -f Trade3.xml installApp**
3. Restart WebSphere to pick up the newly created resources and Trade3 application:
   - UNIX: **$WAS_HOME/bin/stopServer server1 $WAS_HOME/bin/startServer server1**
   - Win32: %**WAS_HOME%\bin\stopServer server1 %WAS_HOME%\bin\startServer server1**

> **Note:** To later uninstall trade3, you can use the command:
>
> `$WAS_HOME/bin/ws_ant -f Trade3.xml uninstall`

Step 4 - Populate the Trade3 database and go trade

1. Direct your browser to `http://localhost:9080/trade`.
2. Click **Configuration** and then click **(Re-)populate Trade database**.

> **Important:** Run the following db2 command to update the DB2 statistics:
>
> ```
> db2 connect to trade3db
> db2 reorgchk update statistics
> ```
>
> This must be done after database population for performance research.

3. Go trade.

10.3 Scoping the job of provisioning Trade

Now, before rushing into development of workflows for the implementation of the Trade application, let us take a minute to properly design the automation package that will be delivered to the organization implementing the application, either your in-house operation organization or an external customer.

To design the automation package, we have to consider the following issues:

- Assumptions
- Prerequisites and limitations
- Functionality
- Content
- Requirements and capabilities
- Prerequisites and their configuration

- Delivery of executables
- Customization
- Transition from test to production

10.3.1 Assumptions

From the installation instructions that comes with the Trade application, we can deduct that the prerequisite infrastructure required consists of the following components:

- An operational database management system (DB2 or Oracle)
- An operational WebSphere Application Server instance named server1

Since most applications are front-ended by a set of HTML pages and scripts, we will add a functioning Web Server (IBM HTTP Server V2), which will be updated during provisioning of the Trade application. These components may reside on any number of systems, and except for the database component, they may even be duplicated on several similar systems to increase availability and reduce bottlenecks.

It is assumed that the customer who will use the Trade Automation Package already have provisioning processes in place to successfully deploy these underlying infrastructure components, and that all of the required networking and storage setup has been established prior to deploying Trade.

Archived

Furthermore, it is assumed that the customer will add the Software Products and or Software Package definitions provided with the Trade Automation Package to their own set of Server Templates and their related Software Stacks, as shown in Figure 10-3.

Figure 10-3 TIO/TPM objects delivered with the Trade Automation Package

10.3.2 Prerequisites and limitations

As described in the Trade Installation instructions, the prerequisites for installing are:

- Application Server (IBM WebSphere Application Server V5.1)
- Database Client (DB2 Client V8.2) if the database server is not installed on the same system as the Application Server.
- Database Server (DB2 Server V8.2)

and for the sake of the example, we will add

- An operational IBM HTTP Server V2 installation

For our purposes, we will pre-install these components, and register the Software Installations with the relevant capabilities in the DCM. Further details are provided in 10.3.5, “Requirements and capabilities” on page 298 and 10.4, “Creating the Test environment” on page 304.

10.3.3 Functionality

From the Trade installation instructions, we also see that the major provisioning tasks required to implement the application are:

- Create and prepare the Trade database.
- Install the Trade application EAR module.
- Stop and start the application.

These steps basically assume that the Trade application is installed on a single system, which hosts both the Application and the Database Server components.

Since we want to support a three-tier configuration, it is necessary for us to add a single infrastructure requirement. In order for the Application Server to access the Database Server, a Database Client must reside on the same system as the Application Server, and, at the database client, we have to catalog the Trade database hosted by the Database Server.

Also, in a production environment you may want to create dedicated server instances for each of the Web, Application, and Database Server components. Since the Trade installation procedures has not been designed to support another WebSphere Application Server server instance than the default server named *server1,* we will restrict our additions of application specific instances to the Web and Database servers. These activities are not included in the standard installation instructions, so we have to add two more provisioning steps.

In addition to all of this, a production-strength Automation Package should include the specific network related provisioning activities, such as updating the load balancer configurations to include the newly provisioned servers, and remove them when the servers are de-commissioned. Typically, using load balancer technology to increase availability and service levels does not apply to databases, since the data used and updated by the online applications needs to be consistent. In high-availability or high-performance configurations, database files may be replicated between servers, but this is not taken into account in this scenario. The bottom line is that for the trade deployment, you should add two additional provisioning tasks to register the newly commissioned Web and Application servers with their respective load balancers; however, in the following, these activities are omitted.

Finally, an important point is that since we have no control over the production environment, the automation package will be restricted to providing ‘integration pieces’ for the actual deployment into a Customer/Application Tier structure. It is the responsibility of the customer to define the application run time structure (Customer, Application Tiers, Server Templates, Load balancers, and so on) and integrate the logical device operations provided by the Trade Automation Package into the Cluster.AddServer and Cluster.RemoveServer LDOs.

Archived

All of this boils down to the fact that in order to create a production-strength Automation Package, the prerequisite components and provisioning activities detailed in Table 10-1 are required.

Table 10-1 Trade provisioning activities

Step	Activity	Target infrastructure component
1.	Create and prepare a database instance.	Database Server
2.	Create and prepare Trade3 database.	Database Instance
3.	Restart the Trade3 database instance.	Database Instance
4.	Catalog the Trade3 database.	Database Client
5.	Create trade resources.	Application Server
6.	Install the trade application.	Application Server
7.	Recycle the Application Server.	Application Server
8.	Register Trade Application with application load balancer.	Application load balancer
9.	Create Apache Service for Trade.	Web Server
10.	Update Web pages to include Trade.	Web Server Instance
11.	Recycle the Trade Apache service to activate the configuration.	Web Server Instance
12.	Register Apache server with front-end load balancer.	Front-end load balancer

In addition to the major activities listed above, our automation package will also provide supporting facilities to operate (stop and start) various components, perform DCM housekeeping, and support the installation.

10.3.4 Content

So from running a few simple workflows to creating a database and installing a WebSphere application, the task at hand has grown into managing the entire application infrastructure and the relationships between the components in an environment similar to the one shown in Figure 10-4 on page 291.

Figure 10-4 Trade infrastructure

Note: In a production environment, you should add activities to register the Application and the Web front end with their respective load balancers (as shown above using dotted lines). For the remainder of this IBM Redbook, we will simplify the scenario, by implementing a communication flow, as shown using the solid lines.

The automation package we are setting out to create is supposed to be installed in a customer environment, in which we expect the customer to be responsible for the deployment of the basic infrastructure components required. In the automation package for Trade, we will provide all the Software Products and related objects required to implement all the major activities identified for the Trade application provisioning, as described Table 10-1 on page 290. These actions relate to the creation, operation, and disposal of specific instances of DCM software resources. The resources are:

Software Installation The anchor point for all other software resources. When the Software Installation is created (from an Software Package definition), actual installation activities may be carried out, if needed.

For the Trade deployment, we will need to be able to create at least three different Software Installations, one for each logical components (database, application, and Web), in order to be able to deploy each component to different physical systems.

Software Instance An instantiation of a Software Installation, for example, a DB2 Instance or a WebSphere Application Server node or server.

For the Trade deployment we will use Software Instance resources to install the Trade application resources and EAR module.

Software Configuration
A specific configuration related to the parenting resource. Software Configurations may, for example, be owned by Software Installations or Software Instances.

In the Trade application package, we will use Software Configurations throughout to provide configuration information to control the creation and configuration of Software Installations and Software Instances.

Application-Data Placeholder to be used for your special purposes.

Foreign-Configuration
A special Software Configuration intended to be used to apply specific configuration information to software resources that are not directly related to the software resource hierarchy of a specific Software Installation.

In the Trade application package, the Foreign-Configuration resource will be used to configure the DB2 Client software for cataloging the trade database.

All software resources managed my TIO/TPM are created from a Software Product, also known as a Software Module. The Software Resource Template(s) related to a Software Product determine which resources to create during installation if the Software Product, and the device drivers associated with each Software Resource Template, controls the behavior of the software resources that are created during the provisioning process. To install a product, a Software Resource Template of type INSTALLATION must exist, and by default, the Software Module.Install logical operation will create a Software Installation object based on this SRT. During the installation process, additional software resources will be created in accordance the Software Resource Templates nested under the INSTALLATION Software Resource Template.

To define exactly what needs to take place during the creation of the Software Installation resource, Software Packages, also known as Software Installable(s), may be associated with the Software Product, and based on the requirements defined for each Software Package, a specific one is picked for the target. The device driver associated with the chosen Software Package determines the

behavior, and thereby the exact actions that will take place during the deployment.

So let us define the software model required to deploy the Trade application in a three-tier environment in Table 10-2 and depicted in Figure 10-5 on page 294.

Table 10-2 The Trade software model resource templates

Software Product	SRT type	Description
Trade DBServer Module	Installation	Anchor object
	Foreign-Configuration	Used to create nested resources on the custom DB2 Server Software Installation object that is hosted on the system on which the Trade DBServer Module Software Product is installed.
	Instance	Used to create the Trade database instance.
	Configuration	Used to create the Trade database.
Trade DBClient Module	Installation	Anchor object.
	Foreign-Configuration	Used to create nested resources on the custom DB2 Client Software Installation object that is hosted on the system on which the Trade DBClient Module Software Product is installed.
	Configuration	Used to catalog the trade database as a configuration related to the DB2 Client Software Installation object hosted on the same system.
Trade Application Module	Installation	Anchor object.
	Foreign-Configuration	Used to create the application instance as a child of the WebSphere Application Server Installation object in order to represent the correct operational relationships in the DCM.
	Instance	Installs the Trade resources and EAR module.
	Configuration	Used to hold parameters for controlling the installation activities.
Trade Web Module	Installation	Anchor object.
	Foreign-Configur ation	Used to create nested resources on the custom IBM HTTP Server Software Installation object that is hosted on the system on which the Trade Web Module Software Product is installed.

Archived

Software Product	SRT type	Description
	Instance	The instantiation of an Apache Server that will support the trade application.
	Configuration	Configuration of the Trade Apache Server.
	Application -Data	Static HTTP pages to front-end the execution of the Trade application.

Figure 10-5 Software components included in the Trade Automation Package

In order to support a full-fledged three-tier configuration, we will have to define Software Products for each of the major Trade components: database, application, and Web. For each of these we need Software Package definitions to link the required files to the Software Products.

For successful installation of a software Package, at lease one Software Resource Template - of type INSTALLATION - must be associated with the Software package. This will in itself not necessarily implement all the required changes to the target platform in order to fully implement the desired

functionality. By nesting one or more Software Resource Templates in the INSTALLATION Software Resource Template, you can control the creation of specific resources related to the installation. For example, to create a DB2 instance on the target system, and the related Instance DCM object, you should create a nested Software Resource Template (of type INSTANCE) as a child of the INSTALLATION Software Resource Template, and associate the appropriate device driver to the INSTANCE Software Resource Template to ensure that your custom workflows for database instance creation are executed.

Software Packages are typically related to one specific operating system platform (through a set of requirements) and to the workflows used to perform the installation through a Device Driver association.

Table 10-3 shows the Trade provisioning activities.

Table 10-3 Trade provisioning activities

Step		Object type	Value
1.	Create and prepare a database instance.		
		Software Definition	Trade DBServer Module
		Software Resource Template	Instance
		Device Driver	Trade_DBServer_Module_Installable_Driver
		Logical Device Operation	SoftwareInstallable.Install
		Workflow	Trade_DBServer_Module_Add_Instance
2.	Create and bind Trade3 database.		
		Software Definition	Trade DBServer Module
		Software Resource Template Type	Configuration
		Device Driver	Trade_DBServer_Module_Installable_Driver
		Logical Device Operation	SoftwareInstallable.Install
		Workflow	Trade_DBServer_Create_Database

Archived

Step		Object type	Value
3.	Recycle the database server to activate the configuration.		
		Software Definition	Trade DBServer Module
		Software Resource Template Type	Instance
		Device Driver	Trade_DBServer_Module_Instance_Driver
		Logical Device Operation	SoftwareInstance.Stop SoftwareInstance.Start
		Workflow	Trade_DBServer_Module_Instance_Stop Trade_DBServer_Module_Instance_Start
4.	Catalog the Trade3 database.		
		Software Definition	Trade DBClient Module
		Software Resource Template Type	Configuration
		Device Driver	Trade_DBClient_Module_Installable_Driver
		Logical Device Operation	SoftwareInstallable.Install
		Workflow	Trade_DBClient_Add_Configuration
5. 6.	Create trade resources and Install Trade application.		
		Software Definition	Trade Application Module
		Software Resource Template Type	Instance
		Device Driver	Trade_Application_Module_Installation_Driver
		Logical Device Operation	SoftwareInstallable.Install
		Workflow	Trade_Application_Add_Instance

Archived

Step		Object type	Value
7.	Recycle the application server to activate the configuration.		
		Software Definition	Trade Application Module
		Software Resource Template Type	Instance
		Device Driver	Trade_Application_Module_Instance_Driver
		Logical Device Operation	SoftwareInstance.Stop SoftwareInstance.Start
		Workflow	Trade_Application_Module_Instance_Stop Trade_Application_Module_Instance_Start
8.	Register Application with Application load balancer (omitted).		
9.	Create Apache Service for Trade.		
		Software Definition	Trade_Web_Module_Installable_Driver
		Software Resource Template Type	Instance
		Device Driver	Trade_Web_Module_Instance_Driver
		Logical Device Operation	SoftwareInstallable.Install
		Workflow	Trade_Web_Module_Installation_Add_Instance
10.	Update Web pages to include Trade.		
		Software Definition	Trade_Web_Module
		Software Resource Template Type	Configuration
		Device Driver	Trade_Application_Module_Installation_Driver
		Logical Device Operation	SoftwareInstallable.Install
		Workflow	Trade_Web_Add_Configuration

Archived

Step		Object type	Value
11.	Recycle the Trade Apache service to activate the configuration.		
		Software Definition	Trade_Web_Module
		Software Resource Template Type	Instance
		Device Driver	Trade_Web_Module_Instance_Driver
		Logical Device Operation	SoftwareInstance.Stop SoftwareInstance.Start
		Workflow	Trade_Web_Module_Instance_Stop Trade_Web_Module_Instance_Start
12.	Register Apache server with front-end load balancer (omitted).		

10.3.5 Requirements and capabilities

To ensure that the prerequisite infrastructure components have been installed on each system that will be hosting any of the Trade Software modules, we will use the Capabilities/Requirements facilities build into the TIO/TPM software model. In addition, the inherent relationships between Trade Software Installations (for example, we need to have the Trade Database Server Module (which creates the Trade database) installed before the database can be cataloged at the system hosting the Trade application modules) need to be taken care of and verified to ensure that all the prerequisite configuration items exist prior to installing any Trade Software Module.

In Figure 10-6 on page 299, the relationships between the various components are depicted, and the capabilities and requirements for each software component are shown in Table 10-4 on page 299.

Archived

Figure 10-6 Trade Software Module and infrastructure relationships

Table 10-4 Trade Software Module capabilities and requirements

Component	Requirements		Capabilities	
	Type	**Value**	**Type**	**Value**
Operating System			OS	os.name=Wndows 2000 Server SP4
DB2 Server	OS	os.name=Wndows 2000 Server SP4	DATABASE	database.family=DB2 database.version=8.2
Trade Database	DATABASE	database.family=DB2 database.version=8.2	APPLICATION	software.title=TRADE software.name=DB
DB2 Client	OS	os.name=Wndows 2000 Server SP4	DATABASE	database.family=DB2Client database.version=8.2
Trade Database catalog entries	DATABASE	database.family=DB2 Client database.version=8.2	APPLICATION	software.title=TRADE software.name=JDBC
	APPLICATION	software.title=TRADE software.name=DB		
WebSphere Application Server	OS	os.name=Wndows 2000 Server SP4	SERVLET_ENGINE	servlet.varsion=2.0
			EBJ_CONTAINER	ejb.version=2.0

Archived

Component	Requirements		Capabilities	
	Type	Value	Type	Value
Trade Application	APPLICATION	software.title=TRADE software.name=JDBC	APPLICATION	software.title=TRADE software.name=EAR
	SERVLET_ENGINE	servlet.varsion=2.0		
	EBJ_CONTAINER	ejb.version=2.0		
IBM HTTP Server	OS	os.name=Wndows 2000 Server SP4	WEBSERVER	jdk.version=2.0
Trade Web Front-end	WEBSERVER	jdk.version=2.0	APPLICATION	software.title=TRADE software.name=WEB
	APPLICATION	software.title=TRADE software.name=EAR		

Important: For successful implementation of the Trade Automation package, the capabilities of the underling middleware software installations must be configured correctly in accordance with Table 10-4 on page 299.

10.3.6 Prerequisites and their configuration

This division between responsibilities leaves us somewhat in the dark when we need to pick up configuration parameters such as installation directories and access credentials for the infrastructure components. For example, in order to create a new database instance in a DB2 Server environment, we must be able to authenticate as the DB2 Administrator, and to install an EAR module in an existing WebSphere Application Server environment, we need to know the installation directory in order to prepare the environment. (Strictly speaking, we should also know the credentials of the WebSphere Application Server Administrator, but since the installation procedures for Trade does not allow us to install in a secure WebSphere environment, we will ignore this fact when designing and developing the Automation Package and related workflows).

When the infrastructure components were installed, these configuration parameters were most likely provided in a Software Resource Template used for the installation; however, we have no way of knowing what the parameter names are. For this reason, we have to provide a way for the customer to provide the parameters required by the workflows we develop. This may be done either by providing instructions for post-installation customization steps that the customer

must complete prior to deploying the Trade application. Instead of asking the customer to define variables for the Trade Software Packages, you might consider creating a workflow that accepts the needed input parameters and performs the necessary customization automatically.

The customization itself may be accomplished using the variable information to look up the actual values, or use a default value if the specified parameter was not found.

Assume that we need to get the installation directory of a DB2 Server installation in order to find the path to the modules to bind. If the Trade DB2 Server Module Installation (provided by us) contains the following variables and the values shown below, we can use this information to identify the Installation SRT for the DB2 Server Installation and pick up the value for the parameter named *install_dir*. If no value is present, we will use the Custom Default value provided by the customer, or the System Default provided by us.

DB2 Server Installation Path SRTType
INSTALLATION

DB2 Server Installation Path parameterName
install_dir

DB2 Server Installation Path Custom Default
c:\ibm\sqllib

DB2 Server Installation Path System Default
c:\Program Files\sqllib

We can use the exact same arguments when discussing the definition of requirements and capabilities of the Software objects in the Trade Automation Package.

10.3.7 Delivery of executables

As described in Appendix A, “Trade3 original installation instructions” on page 523, the executables are delivered from our development organization in a zip archive file. This archive includes the application EAR modules as well as installation and customization scripts for creating the trade database and installation of the Trade application.

To make the application code available to our potential customers, we decide to include it in the Automation Package, primarily for easy installation. However, this comes at a cost, both in lack of license control, less flexibility in customization, and increased size of the Automation Package. If we were creating an Automation Package for a huge application system, several hundreds of MB, this approach may be revised and you should plan for (and document) manual

installation of the installation files and related customization of the Software.Installable/Software.Product objects supporting the installation of your application system.

The approach we have chosen allows us to automatically copy the installation archive to a directory on the TIO/TPM Server, which is known to a File Repository. The only File Repository that we for certain that exists in the customer environment is the LocalFileRepository, which was created during TIO/TPM installation, with a root directory of ${TIO_HOME}/../../SWrepository.

In order to copy the installation archive (trade3Install.zip), we copy this file to the repository directory of the automation Package, and add the following to the tc-driver.xml file in the TC-INF directory:

```
<item name="repository/trade3install.zip" action="copy-file">
   <param name="editable" value="false" /></item>
   <param name="dest.path" value="${repository}/trade3" />
   <param name="chmod" value="755" />
</item>
```

This will copy the trade3install.zip file to the trade3 subdirectory of the %TIO_HOME%\..\..\SWrepository on the TIO/TPM Server system.

In case we had chosen to let the customer be responsible for adding the Trade installation files to his own File Repository, we would have to document the process, including updating the file information in the Software Packages/Software Installables. Since this process may be error-prone, we recommend creating a workflow for customizing the setup, and let the customer provide the required input parameters, such as the name and location of the target File Repository, in order to move the files to a specific location in another File Repository, and update the related objects accordingly.

10.3.8 Customization

In order to allow customers to easily integrate our Trade automation package in their own environment, it is imperative that no parameters are hard coded into the workflows, and that the customer is provided a facility to easily change the default parameters defined during installation.

This is naturally the basic purpose of the Software Configuration Templates, which can be modified according to specific customer needs, but it also applies to a number of other DCM objects created during installation of the automation package, in particular the credentials used to access the database and WebSphere environments (the latter is not the case in our scenario, since the WebSphere Application Server in our test environment has no security enabled), but equally important for the variables assigned to the Software Modules to help

identify the hosting platforms and their configuration parameters (see 10.3.6, “Prerequisites and their configuration” on page 300).

Chances are that the consumers of the Trade automation package also want to be able to decide for themselves in which File Repository to store the installation archives, so somehow they must be given the opportunity (either through a workflow or through installation and customization instructions) on how to move the installation files to another File Repository, and update the Software Packages accordingly.

Most post-installation customization can be automated by creating a workflow that updates the DCM according to customer defined values. We recommend that you create a workflow to control all your pre-defined variables and parameters, and provide default values using the `<post-install-workflow name=”<your workflow>”/>` stanza in the tc-driver manifest file. For the Trade application, we have created the Trade_Customize workflow, which accepts 20+ parameters to control the setup. In this workflow, default values will be created for parameters that are not passed to the workflow.

10.3.9 Transition from test to production

An extension of the customization discussion is the transition to production. Most customers expect a seamless transition from test through staging to production, and here the ability to automatically discover parameters from the current environment and configure the automation package accordingly is imperative.

Since there are may contributors to any production environment, we are in a situation in which we cannot assume that the naming conventions used for our variables confirm to the customer’s environment, especially when we are using system-wide variables as is the case in this scenario for controlling the simulation mode.

For this reason, we strongly advise using either highly qualified names for global variables, as is the case in the implementation demonstrated in the Trade_Get_Simulation workflow, or to use one trade specific variable to point to the global variable. In this way, when moving from test to production, you can change the application variable to accommodate special settings in the new environment.

10.4 Creating the Test environment

In order for us to develop and test the software definitions and workflows needed to deploy the Trade application, we will need a test environment.

In our case, we installed the needed infrastructure components manually, on three individual servers, and updated our Data Center Model to allow us to verify the requirements checking for the installation of the Trade modules.

The following infrastructure components were installed:

- All systems
  - Windows 2000 Server SP4
- DBServer
  - DB2 UDB Enterprise Server V8.2
- WASServer
  - DB2 UDB RunTime Client V8.2
  - IBM WebSphere Application Server V5.1
- WEBServer
  - IBM HTTP Server V2

To register these installations to the DCM, we need the related software objects (Products/Modules, and Packages/Installables) to be defined first using the relevant SoftwareSimulator_X device drivers. In this way, we can either register the Software Installations directly on the three servers, or we can actually install the Software Products using the software installation wizard or the Software Module.Install logical operation.

All the software definitions for the simulated infrastructure components are basically built after the same skeleton: a Software Product with the relevant capabilities, referencing a single Software Package, which requires the Windows 2000 Server SP4 operating system to be installed. We used the Windows 2000 Server SP4 as the base of all the test systems; however, if you are using another Windows flavor, you have to adjust your definitions accordingly, or simply remove the requirement from the Software Packages.

> **Note:** Normally, no device driver association is necessary for Software Product definitions, and this is the case for all of the simulated infrastructure definitions.

In addition, each Software Module includes an Installation Software Resource Template, which, for DB2 CLient and WebSphere Server, includes an Instance template, to reflect the fact that our pre-installed DB2 Client has a default instance named DB2 and the WebSphere Application Server hosts a server named server1. The following shows the details of each of the definitions used to

register the simulated software objects, and the XML files used to create the definitions.

10.4.1 Windows 2000 Server SP4

The Windows 2000 Server SP4 operating system definition is created in the DCM upon installation of the TIO/TPM server. To make sure that it will work in out environment, make sure that the Operating System definition includes the following OS osfamily=Windows 2000 Server SP4 capability, as shown in Figure 10-7.

Figure 10-7 Windows 2000 Server SP4 operating system properties

Also make sure that the device driver associated with the Windows 2000 Server SP4 definition is the one named *Windows Operating System*.

Archived

Note: For the Windows Server® 2000 SP4 operating system definition, we do not associate any installables or images, since it will be discovered automatically by the Common Agent Inventory scanner.

10.4.2 Trade Simulated DB2 Server

The details of the Software Product definition used to register the manually installed DB2 UDB Enterprise Server V8.2 with the DCM are shown in Figure 10-8.

Figure 10-8 Trade Simulated DB2 Server Software Product properties

Archived

The Trade Simulated DB2 Server Software Product has the capability of DATABASE, and for it we define the following details:

database.family	DB2
database.version	8.2
software.title	DB2 UDB
software.name	Enterprise Server
software.version	8.2
software.vendor	IBM

These capabilities will be used as requirements for the Trade DBServer Module to make sure that the server(s) on which the Trade DBServer Module is installed has the required database functionality. In addition, a parameter named install_dir is defined in the Installation Software Resource Template to help the implementation workflows find the DB2 executables.

As Figure 10-8 on page 306 shows, the Trade Simulated DB2 Server Software Module references the Trade Simulated DB2 Server Installable Software Package, and the requirements for this are shown in Figure 10-9.

Figure 10-9 Trade Simulated DB2 Server Installable Software Package

You should notice that the even though the Software Installable does not include any files, and references the SoftwareSimulator_SoftwareInstallable device driver, which means that nothing is actually performed when the Installable is installed, this definition is necessary in order for the Software Installation object to be created on the target system upon installation.

Example 10-1 shows the xml file we used to register the software objects for our DB2 UDB Server (the DCM). The definitions may be manually imported into the DCM using the **xmlimport** command:

```
%TIO_HOME%\tools\xmlimport
file:%TIO_HOME%\xml\Trade_Simulated_DB2_Server.xml
```

For a discussion on how these definitions may be imported as part of the installation of an Automation Package (a tcdriver archive), please refer to 10.4.6, “Building the Trade_Simulated_Infrastructure Automation Package” on page 320.

Example 10-1 Trade_Simulated_DB2_Server.xml

```
<?xml version=”1.0” encoding=”UTF-8”?>
<!DOCTYPE datacenter SYSTEM “file:../xml/xmlimport.dtd”>

<datacenter>
   <software-module name=”Trade Simulated DB2 Server”
                    version=”8.2”
                    vendor=”IBM”
                    title=”Trade DB2 Server 8.2”
                    description=”DB2 Server”
                    is-draft=”false”>
      <software-capability name=”database.family” value=”DB2” />
      <software-capability name=”database.version” value=”8.2” />
      <software-capability name=”software.title” value=”DB2 UDB” />
      <software-capability name=”software.name” value=”Enterprise
Server” />
      <software-capability name=”software.version” value=”8.2” />
      <software-capability name=”software.vendor” value=”IBM” />
      <supported-requirement-type value=”DATABASE” />

    <installable-package name=”Trade Simulated DB2 Server Installable”
                    description=”DB2 Server Installation Package”

is-device-model=”SoftwareSimulator_SoftwareInstallable”
                       locale=”en_US”
                       version=”8.2”
                       priority=”0”
                       file-repository=”LocalFileRepository”
                       status=”tested”>
         <file name=”-1” path=”/” />
         <software-requirement name=”os.family”
                               type=”OS”
                               enforcement=”MANDATORY”
                               hosting=”true”
```

```
                          accept-non-existing="false">
            <software-requirement-value value="Windows 2000 Server SP4"
/>
         </software-requirement>
      </installable-package>

      <software-resource-template name="Simulated DB2 Server
Installation for Trade"
                          software-resource-type="INSTALLATION"

software-resource-device-model="SoftwareSimulator_SoftwareInstallation"
                          multiplicity-type="Unspecified"
                          software-configuration-type="Regular"
                          is-selected="true">
         <template-param name="resource-name"
                     value="Trade DB2 Server Installation"
                     is-changeable="true" />
         <template-param name="install_dir"
                     value="c:/ibm/sqllib"
                     is-changeable="true" />
      </software-resource-template>

   </software-module>

</datacenter>
```

Archived

10.4.3 Trade Simulated DB2 Client

As seen in Figure 10-10, the definition of the software objects for the simulated DB2 Client are almost similar to those of the simulated DB2 Server, the main difference being that an Instance Software Resource Template has been added as a child of the Installation SRT.

Figure 10-10 Trade Simulated DB2 Client Software Product properties

Archived

The only capability we define for the simulated DB2 Client is DATABASE, and the details are:

database.family	DB2Client
database.version	8.2
software.title	DB2 UDB
software.name	Runtime Client
software.version	8.2
software.vendor	IBM

The nested Instance SRT is used to automatically create a Software Instance for the DB2 Client, which is used to host catalog entries such as node and database definitions. The default instance name of DB2 is set up by associating the value DB2 with the parameter named resource-name.

To allow installation, we need a Software Package for the DB2 Client, and (besides the name) this is identical to the one defined for the DB2 Server, as seen in Figure 10-11.

Figure 10-11 Trade Simulated DB2 Client Installable Software Package

Archived

The XML file used to automatically create these objects is shown in Example 10-2.

Example 10-2 Trade_Simulated_DB2_Client.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE datacenter SYSTEM "file:../xml/xmlimport.dtd">

<datacenter>

  <software-module name="Trade Simulated DB2 Client" version="8.2"
vendor="IBM" title="Trade DB2 Server 8.2" description="DB2 Server"
is-draft="false">
    <software-capability name="database.family" value="DB2Client" />
    <software-capability name="database.version" value="8.2" />
   <software-capability name="software.title" value="DB2 UDB" />
   <software-capability name="software.name" value="Runtime Client" />
   <software-capability name="software.version" value="8.2" />
   <software-capability name="software.vendor" value="IBM" />
    <supported-requirement-type value="DATABASE" />
    <installable-package name="Trade Simulated DB2 Client Installable"
                 description="Simulated DB2 Client Installation
Package"

is-device-model="SoftwareSimulator_SoftwareInstallable"
                 locale="en_US" version="8.2"
                 priority="0"
                 file-repository="LocalFileRepository"
                 status="tested">
      <file name="-1" path="/" />
      <software-requirement name="os.family"
                 type="OS"
                 enforcement="MANDATORY"
                 hosting="true"
                 accept-non-existing="false">
            <software-requirement-value value="Windows 2000 Server SP4"
/>
      </software-requirement>
    </installable-package>
    <software-resource-template name="Simulated DB2 Client Installation
for Trade"
                     software-resource-type="INSTALLATION"

software-resource-device-model="SoftwareSimulator_SoftwareInstallation"
                     multiplicity-type="Unspecified"
```

```
                    software-configuration-type="Regular"
                    is-selected="true">
      <software-resource-template name="Trade DB2 Instance"
                          software-resource-type="INSTANCE"

software-resource-device-model="SoftwareSimulator_SoftwareInstance"
                          multiplicity-type="Unspecified"
                          software-configuration-type="Regular"
                          is-selected="true">
               <template-param name="resource-name"
                          value="DB2"
                          is-changeable="true" />
      </software-resource-template>

      <template-param name="resource-name"
                    value="Trade DB2 Client Installation"
                    is-changeable="true" />
      <template-param name="install_dir"
                  value="c:/ibm/sqllib"
                  is-changeable="true" />
   </software-resource-template>

  </software-module>

</datacenter>
```

10.4.4 Trade Simulated WebSphere Application Server

The definitions used for the simulated WebSphere Application Server are almost a repetition of the ones used for the simulated DB2 Client, the main difference being the capabilities, which, for the simulated WebSphere Application Server, are:

- SERVLET_ENGINE
- EJB_CONTAINER:

Figure 10-12 on page 314 shows that the capability details are:

servlet.version	2.0
ejb.version	2.0
software.title	WEBSPHERE
software.name	APSERVER
software.version	5.1
software.vendor	IBM

Archived

and that the name that will be assigned to the instance is server1.

Figure 10-12 Trade Simulated WAS Server Software Product properties

Just as was the case for the two DB2 related definitions, the Software Installable for the simulated WebSphere Application Server uses the simulator, and no files are associated with it.

Figure 10-13 shows the Trade Simulated WebSphere Application Server Installable Software Package

Figure 10-13 Trade Simulated WebSphere Application Server Installable Software Package

To easily define the simulated WebSphere Application Server to our DCM, we prepared the XML file shown in Example 10-3, which may be imported using the **`xmlimport`** command or embedded along with the other XML files in a special automation package for the simulated environment, as demonstrated in 10.4.6, “Building the Trade_Simulated_Infrastructure Automation Package” on page 320.

Example 10-3 Trade_Simulated_WAS_Server.xml

```
<?xml version=”1.0” encoding=”UTF-8”?>
<!DOCTYPE datacenter SYSTEM “file:../xml/xmlimport.dtd”>

<datacenter>

   <software-module name=”Trade Simulated WAS Server” version=”5.1”
                    vendor=”IBM”
                    title=”Trade WAS Server 5.1”
                    description=”WAS Server”
                    is-draft=”false”>
      <software-capability name=”software.title” value=”WEBSPHERE” />
      <software-capability name=”software.name” value=”APPSERVER” />
      <software-capability name=”software.version” value=”5.1” />
      <software-capability name=”software.vendor” value=”IBM” />
      <software-capability name=”servlet.version” value=”2.0” />
      <software-capability name=”ejb.version” value=”2.0” />
      <supported-requirement-type value=”SERVLET_ENGINE” />
      <supported-requirement-type value=”EJB_CONTAINER” />
```

```
      <installable-package name="Trade Simulated WAS Server
Installable"
                  description="WAS Server Installation Package"

is-device-model="SoftwareSimulator_SoftwareInstallable"
                        locale="en_US"
                        version="5.1"
                        priority="0"
                        file-repository="LocalFileRepository"
                        status="tested">
         <file name="-1" path="/" />
         <software-requirement name="os.family"
                           type="OS"
                           enforcement="MANDATORY"
                           hosting="true"
                           accept-non-existing="false">
            <software-requirement-value value="Windows 2000 Server SP4"
/>
         </software-requirement>
      </installable-package>

      <software-resource-template name="Simulated WAS Server
Installation for Trade"
                           software-resource-type="INSTALLATION"

software-resource-device-model="SoftwareSimulator_SoftwareInstallation"
                           multiplicity-type="Unspecified"
                           software-configuration-type="Regular"
                           is-selected="true">
         <software-resource-template name="Simulated WAS Server Server1
Instance"
                              software-resource-type="INSTANCE"

software-resource-device-model="SoftwareSimulator_SoftwareInstance"
                              multiplicity-type="Unspecified"
                              software-configuration-type="Regular"
                              is-selected="true">
            <template-param name="resource-name"
                        value="server1"
                        is-changeable="true" />
         </software-resource-template>

         <template-param name="resource-name"
                     value="Trade WAS Server Installation"
                     is-changeable="true" />
```

```
        <template-param name="install_dir"
                        value="c:/ibm/websphere"
                        is-changeable="true" />
      </software-resource-template>

   </software-module>

</datacenter>
```

10.4.5 Trade Simulated HTTP Server

The final software definition we need to create is the one for the simulated HTTP Server, and it is almost identical to the ones we have already created, especially the one for the DB2 Server.

This time, the capability is WEBSERVER, and the details reflect the attributes of the IBM HTTP Server v2 software product.

Archived

Since we do not care about the default HTTP Server processes created during installation, we only have one Software Resource Template of type Installation, as shown in Figure 10-14.

Figure 10-14 Trade Simulated HTTP Server Software Product properties

The definition of the Software Installable follows the previously discussed guidelines, as seen in Figure 10-15 on page 319.

Archived

Figure 10-15 Trade Simulated HTTP Server Installable Software Package

Again, an XML file (shown in Example 10-4) is used to automatically register the Software Module with the DCM.

Example 10-4 Trade_Simulated_HTTP_Server.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE datacenter SYSTEM "file:../xml/xmlimport.dtd">

<datacenter>

   <software-module name="Trade Simulated HTTP Server"
                    version="2.0"
                    vendor="IBM"
                    title="simulated IBM HTTP Server for Trade"
                    description="Web Server"
                    is-draft="false">
      <software-capability name="jdk.version" value="2.0" />
      <software-capability name="software.title" value="IBM HTTP
Server" />
      <software-capability name="software.vendor" value="IBM" />
      <software-capability name="software.name" value="Apache" />
      <software-capability name="software.version" value="2.0" />
      <supported-requirement-type value="WEBSERVER" />

    <installable-package name="Trade Simulated HTTP Server Installable"
                         description="HTTP Server Installation Package"

is-device-model="SoftwareSimulator_SoftwareInstallable"
                         locale="en_US"
                         version="2.0"
                         priority="0"
```

```
                    file-repository="LocalFileRepository"
                    status="tested">
        <file name="-1" path="/" />
        <software-requirement name="os.family"
                        type="OS"
                        enforcement="MANDATORY"
                        hosting="true"
                        accept-non-existing="false">
           <software-requirement-value value="Windows 2000 Server SP4"
/>
        </software-requirement>
     </installable-package>

     <software-resource-template name="Simulated HTTP Server
Installation for Trade"
                        software-resource-type="INSTALLATION"

software-resource-device-model="SoftwareSimulator_SoftwareInstallation"
                        multiplicity-type="Unspecified"
                        software-configuration-type="Regular"
                        is-selected="true">
     <template-param name="resource-name"
                value="Trade HTTP Server Installation"
                is-changeable="true" />
     <template-param name="install_dir"
                value="c:/ibm/ihs20"
                is-changeable="true" />
    </software-resource-template>

   </software-module>

</datacenter>
```

10.4.6 Building the Trade_Simulated_Infrastructure Automation Package

To easily distribute the definitions for the simulated infrastructure components used by the Trade application, we decided to build an Automation Package, which, when installed, automates the creation of software definitions.

To build this package, we defined a new project in the APDE workspace, and named it Trade_Simulated_Infrastructure. Then we created a subdirectory in the project named xml. Then all the XML files described in this section were copied

to the xml directory, and we updated the TC-INF/tc-driver.xml file, so it looked like Example 10-5.

Example 10-5 Trade_Simulated_Infrastructure tc-driver.xml file

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE tc-driver PUBLIC "-//IBM//DTD TC Driver 2.0//EN"
"http://www.ibm.com/tivoli/orchestrator/dtd/tcdriver.dtd">
<tc-driver>
   <tc-driver-format>2.0</tc-driver-format>
   <driver-name>Trade_Simulated_Infrastructure</driver-name>
   <version>1.0</version>
   <description />
   <documentation location="doc/readme.html" />
   <dependencies />
   <property name="tc.pkg"
location="com.thinkdynamics.kanaha.tcdrivermanager.action" />
   <actions>
      <action name="copy-file" class="${tc.pkg}.CopyFileActions" />
      <action name="java-plugin" class="${tc.pkg}.JavaPluginActions" />
      <action name="workflow" class="${tc.pkg}.TxtWorkflowAction" />
      <action name="import" class="${tc.pkg}.ImportAction" />
   </actions>

   <items/>

   <device-models />

   <dcm>
      <item name="xml/Trade_Simulated_DB2_Server.xml" action="import"/>
      <item name="xml/Trade_Simulated_WAS_Server.xml" action="import"/>
      <item name="xml/Trade_Simulated_HTTP_Server.xml"
action="import"/>
      <item name="xml/Trade_Simulated_DB2_Client.xml" action="import"/>
   </dcm>

</tc-driver>
```

In the tc-driver.xml file, we instruct the Automation Package to copy the XML files to the %TIO_HOME%\xml directory (specified in the <item> sections all using the copy-file action) and import all the information into the DCM by using the import action in the <dcm> section.

Archived

The result is that upon installation of the automation package (using the **`tc-driver-manager i Trade_Simulated_Infrastructure`** command), we will automatically have all the Software Modules defined, along with their related Software Installables and Software Resource Templates.

10.5 Developing the Trade automation package

In the Trade application deployment, the first step we need to perform is the creation of a database instance to host the database, and create the database itself. To accomplish this, we created a Software Product called *Trade DBServer Module*.

When installed, this module will create a new instance in an existing DB2 Server implementation, create a database, and load the tables and indexes from the Table.ddl file provided in the Trade installation archive.

Naturally, this is not as easy as it seems. Since we are not responsible for the implementation of the underlying DB2 Server platform, we have no previous knowledge about the credentials we need to use to access the target system in order to have the right authorizations to create the db2 instance user, create the instance, and manage the database. For this reason, we have set up a SAP (type unknown) to host these credentials, and it will be the responsibility of the customer deploying the Trade Automation Package to customize these credentials accordingly.

Another issue that we will encounter is to find the installation directory of the underlying DB2 Server in order for us to run the necessary commands. Since we have no knowledge about the implementation, we will have to set up a set of variables to help identify the installation directory, and instruct the customer to maintain these in accordance with the actual environment.

Finally, since most of our development effort will be focused around maintaining the objects and their relationships in the DCM, it will come in handy to have some way of controlling whether or not actual deployment actions, such as creating databases, will take place.

These issues apply equally to all the underlying infrastructure implementations, so before starting the creation of the individual resources for each component, let us establish the basics needed to gain access and identify installation directories.

10.5.1 Creating the trade Service Access Point

The Service Access Point is used to store the credentials to be used to access the various infrastructure components. This SAP will be created during installation of the Trade Automation Package, but since the SAP needs a hosting object (typically a server), the only anchor point we can use is the TIO/TPM Server itself, and at installation time, we do not know either the ID or the name. Therefore, we need a workflow to find the ID of the TIOServer, and insert the SAP, and call this workflow from the automation package installation procedure using the post-install-workflow from the tc-driver.xml of the Trade Automation Package:

```
<post-install-workflow name="Trade_Create_SAP" />
```

Example 10-6 shows the workflow used to find the ID of the Deployment Engine and add a Service Access Point named trade.

Example 10-6 Trade_Create_SAP workflow

```
workflow Trade_Create_SAP  LocaleInsensitive


    var DE_Device_ID
    java:com.thinkdynamics.kanaha.de.javaplugin.datacentermodel.FindDEDeviceId(DE_Device_ID)


    DCMInsert parent=DCMQuery(/server[@id=$DE_Device_ID]/@id) <<EOFXML
    <sap name="trade"
        locale="en_US"
        protocol-type="unknown"
        app-protocol="UNKNOWN"
        domain="trade"
        context="db"
        port="0"
        auth-compulsory="true"
        role="host">
        <credentials search-key="dbadmin"
                     is-default="false">
            <password-credentials username="db2admin"
                                  password="AyEOEIqzjjUrtAqUiax6+A=="
                                  is-encrypted="true" />
        </credentials>

        <credentials search-key="dbinstance"
                     is-default="false">
            <password-credentials username="trade3db"
                                  password="AyEOEIqzjjUrtAqUiax6+A=="
                                  is-encrypted="true" />
        </credentials>

        <credentials search-key="dbuser"
                     is-default="false">
```

```
        <password-credentials username="tradeusr"
                              password="AyEOEIqzjjUrtAqUiax6+A=="
                              is-encrypted="true" />
    </credentials>
</sap>
EOFXML
```

You should pay attention to the fact that we use the protocol type UNKNOWN, because this SAP is only going to be used as a placeholder for storing the credentials in an encrypted form.

In Example 10-6 on page 323, you can also see, that a default password may be provided in either clear text (the last <credentials> section) or encrypted (as in the first <credentials> section). To use the encrypted form, you should create the definitions using the TIO/TPM GUI, use the **dcmeexport** command to export the DCM to a flat file, from which you can cut the definitions.

10.5.2 Finding installation directories

To be able to find the installation directories of the various infrastructure components, which have been pre-installed and for which we have no knowledge regarding the use of Software Resource Template parameter names or the like, we will create a set of variables for each of our related Trade modules. Upon installation, it will be the customer's responsibility to customize these according to the environment policies that applies for a specific installation.

The variables that will be applied to all our Software Package/Software Installable definitions are:

install_dir_parameter_name
: The name of the Software Resource Template parameter for which the value holds the installation directory name.

install_dir_softwareresource_type
: The Software Resource type of the SRT in which to find the parameter referenced by the install_dir_parameter_name variable.

install_dir_custom_default
: A custom default to be used if nothing is specified for install_dir_parameter_name or no value is found.

install_dir_system_default
: A system vide default to be used in no other values are found. Provided by the automation package developer.

Having these variables defined for the Software Installables allows the customer to provide references to parameters located in the Software Resource Templates for the underlying infrastructure component. Use a custom default, or simply rely on the default values provided with the automation package.

In the Software definitions provided with the Trade Automation Package, the following system-wide default values will be provided:

Trade DBServer Module Installable C:/Program Files/SQLLIB
Trade DBClient Module Installable C:/Program Files/SQLLIB
Trade Application Module Installable C:/Program Files/WebSphere/AppServer
Trade Web Module Installable C:/Program Files/IBM/IHS20

Now, we just need a workflow to retrieve the installation directories when we need them during the deployment process. Example 10-7 shows how we chose to implement it.

Example 10-7 Trade_Get_Hosting_SRTParameter_Or_Default workflow

```
workflow Trade_Get_Hosting_SRTParameter_Or_Default (in currentInstallableID, in targetInstallationID, in
parameterName, out parameterValue)  LocaleInsensitive

    var
currentSoftwareModuleID=DCMQuery(/softwareinstallable[@id=$currentInstallableID]/softwareinstallationmecha
nism/@moduleid)
    var currentSoftwareModuleName=DCMQuery(/softwaremodule[@id=$currentSoftwareModuleID]/@name)
#   var currentInstallableID=DCMQuery(/softwareinstallation[@id=$currentInstallationID]/@softwareproductid)
    var currentInstallableName=DCMQuery(/softwareinstallable[@id=$currentInstallableID]/@name)
#   var targetSoftwareInstallationName=DCMQuery(/softwareinstallation[@id=$targetInstallationID]/@name)
    var ServerID=DCMQuery(/softwareinstallation[@id=$targetInstallationID]/@managedsystemid)

    var srtParmName=Jython[parameterName + "_parameter_name"]
    var srtType=Jython[parameterName + "_softwareresource_type"]
    var customDefault=Jython[parameterName + "_custom_default"]
    var systemDefault=Jython[parameterName + "_system_default"]

    var componentName = "DEPLOYMENT_ENGINE"
    var componentID = DCMQuery(/kanahacomponent[@name=$componentName]/@id)

 #   foreach id in DCMQuery(/softwareinstallation[@managedsystemid=$ServerID and
@softwaremoduleid=$SoftwareModuleID]/@softwareproductid) do
 #     var name = DCMQuery(/softwareinstallable[@id=$id]/@name)
 #     log info Jython["Installable ID: " + id + "   " + name]
 #   done
 #   var InstallableID=DCMQuery(/softwareinstallation[@managedsystemid=$ServerID and
@softwaremoduleid=$SoftwareModuleID]/@softwareproductid)
 #   var InstallableName=DCMQuery(/softwareinstallable[@id=$InstallableID]/@name)

    log info Jython["Looking for value for " + componentName + " parameter '" + srtParmName + "' in " +
currentInstallableName + "(" + currentInstallableID + ") for installation of " + currentSoftwareModuleName
+ "(" + currentSoftwareModuleID + ")"]
```

```
    var templateKey=DCMQuery(/softwareinstallable[@id=$currentInstallableID]/property[@key=$srtParmName
and @componentid=$componentID]/@value)
    # log debug Jython[“TemplateKey: “ + templateKey ]

    var templateType
    if Jython[templateKey] then

      templateType=DCMQuery(/softwareinstallable[@id=$currentInstallableID]/property[@key=$srtType and
@componentid=$componentID]/@value)
      if Jython[not templateType] then
        templateType=”INSTALLATION”
      endif
      templateType=Jython[templateType.upper()]
      # log debug Jython[“TemplateType: “ + templateType ]


      # Find the srt to pick the parameter from
      var
installationSRTID=DCMQuery(/softwareinstallation[@id=$targetInstallationID]/softwareresourcetemplate/@id)
      var softwareResourceTypeID=DCMQuery(/softwareresourcetype[@name=$templateType]/@id)

      var srtID
      if Jython[templateType == “INSTALLATION”] then
        srtID=installationSRTID
      else
        foreach srt in DCMQuery(/childsoftwareresourcetemplate[@parenttemplateid=$installationSRTID and
@softwareresourcetype=$softwareResourceTypeID]/@id[orderBy@id]) do
          # pick the first one
          srtID=srt
        done
      endif
      # log info srtID

      # get the parameter value
      parameterValue=DCMQuery(/templateparam[@templateid=$srtID and @name=$templateKey]/@value)

    endif


    # if no value, get the custom default
    if Jython[parameterValue == None or parameterValue == “““ ] then
      parameterValue=DCMQuery(/softwareinstallable[@id=$currentInstallableID]/property[@key=$customDefault
and @componentid=$componentID]/@value)
    endif

    # if no value, get the system default
    if Jython[parameterValue == None or parameterValue == “““ ] then
      parameterValue=DCMQuery(/softwareinstallable[@id=$currentInstallableID]/property[@key=$systemDefault
and @componentid=$componentID]/@value)
    endif


    log info Jython[“value for “ + parameterName + “ is “ + (parameterValue or “UNKNOWN”)]
```

As it should be obvious from the workflow listing in Example 10-7 on page 325, this workflow takes three input parameters:

curentInstallableID The objectID of the Software Installable for which the variables have been defined

tragetInstallationID The objectID of the Software Installation that is the (grand)parent of the Software Resource Templates to be inspected

parameterName The anchor name of the variables defined, in our case, “install_dir”

and the resulting value is returned in the variable named *parameterValue*.

The following shows a sample invocation of the Trade_Get_Hosting_SRTParameter_Or_Default workflow:

```
var curentInstallableID = “1234”
var tragetInstallationID = “5678”
var parameterName = “install_dir”
var parameterValue
Trade_Get_Hosting_SRTParameter_Or_Default (currentInstallableID,
targetInstallationID, parameterName, parameterValue)
```

10.5.3 Controlling simulation

As already mentioned, we would like to be able to turn external execution of remove commands and scriptlets (actions that actually modify our target servers) on and off during the development process. This will greatly speed up development of the parts of the deployment workflows that deal with pure DCM interaction and maintenance.

To facilitate this, each workflow that performs actions will define a variable names simulation, and surround each call to scriptlets, Device.ExecuteCommand, as well as non-Trade controlled LDOs, with the following sample code:

```
var simulation
Trade_Get_Simulation(simulation)

......

if Jython[simulation.lower() == "yes" ] then
   # log the intended call and the parametes used
else
   # perform the action
```

```
endif
```

The way we have decided to control the simulation is to create a system wide DEPLOYMENT_ENGINE variable (defined in the KANAHA context) with the name trade_simulation. The Trade_Get_Simulation workflow has been designed to inspect this variable, and enable simulation if the value is any of 1, yes, true or on, as seen in the workflow listing in Example 10-8.

Example 10-8 Trade_Get_Simulation

```
workflow Trade_Get_Simulation(out simulation)  LocaleInsensitive

    # get the DEPLOYMENT_ENGINE component id
    var componentName = “DEPLOYMENT_ENGINE”
    var componentID = DCMQuery(/kanahacomponent[@name=$componentName]/@id)

    # get the value of the trade_simulation variable of the TPM configuration (KANAHA) object
    var variable_name=”trade_simulation”
    simulation=DCMQuery(/dcmobject[@name=”KANAHA”]/property[@key=$variable_name and
@componentid=$componentID]/@value)

    if Jython[not simulation or simulation == “““] then
      simulation=”no”
    endif

    simulation=Jython[simulation.lower()]
    if Jython[simulation == “1” or simulation == “true” or simulation == “yes” or simulation == “on” ]
then
      simulation = “yes”
    endif
```

Since the workflow returns a valid value even if the system-wide trade_simulation variable has not been defined, there is no need for us to ensure that it will be created during the installation of the Trade Automation Package. Customers may add it manually should they want to simulate the Trade deployment.

10.6 Creating the Trade database

The main object we need to define in order to provide facilities to create the DB2 instance and database needed for the Trade application is a Software Product, which holds pointers to Software Resource Templates used to instantiate the necessary resources in the data center model, and other pointers to Software Packages where the specific installation files are referenced. In addition, requirements may be specified for the Software Product, which will help ensure that the functionality required for successful implementation is available on the target system.

When the Software Module.Install logical device operation is executed, the following will take place:

1. TIO/TPM will validate the requirements for the Software Product, and identify the correct Software Package to use based on the requirements set up for the Software Package. If no Software Package can be identified, the installation will fail.
2. The Software Installation object is created on the target Server (with a status of *pending*), and the Installation Software Resource Template from the Software Product, along with all nested (or child) templates, will be copied (cloned) to the Software Installation Object.
3. The SoftwareInstallable.Install logical operation for the Software Package is invoked. The actual workflow that will be executed is the one located in the device driver associated with the Software Package, which implements the SoftwareInstallable.Install LDO.
4. Once the installation has completed, the Software Resource Template associated with the Software Installation object is inspected, and all child resources are created. For each child SRTs of the type Instance, the Software.Installation.AddInstance LDO is called, and the workflow to be used is again associated with the device driver of the Software Installation. If you require any special processing based on the existence of any other Software Resource Templates (as is the case for the Trade DBServer Module), you are responsible for adding your own logic to the workflows implementing the SoftwareInstallation.AddInstance or SoftwareInstallable.Install LDOs.

To create the database for Trade, we are faced with the problem that, when implementing in a customer environment, we have absolutely no knowledge of how the underlying infrastructure is defined in the Data Center Model.

In the following, we make the assumptions that the capabilities of the target servers are:

OS	os.family=Windows 2000 Server SP4
DATABASE	database.family=DB2 database.version=8.2

and we will set up requirements in the Software Product and Software Package that reflect these assumptions.

Archived

10.6.1 The Trade DBServer Module Software Product

For the creation of the database needed for the Trade application, we define a Software Product named Trade DBServer Module. Through the Software Product definition, we will link to a Software Package, named Trade DBServer Module Installable, which holds information about the installation files, and by including the proper Software Resource Templates in the Software Product, we ensure that the expected hierarchy of DCM objects will be created and named according to our standards.

Figure 10-16 Trade DBServer Module properties

In addition, we will associate special capabilities with the Software Product. These capabilities will be applied to the target server upon installation, and thus can be used to later to ensure that the trade database exists. To ensure that the required DB2 Server functionality is available on the servers where the Trade DBServer Module will be installed, we will also add DATABASE related requirements to the Software Product definition.

Capabilities and requirements

The capabilities we assign to the Trade DBServer Module are:

APPLICATION	software.title	TRADE
APPLICATION	software.name	DB
APPLICATION	software.vendor	IBM
APPLICATION	software.version	3.1

These capabilities will allow us, at a later point, to reference the Trade DBServer Module installation with non-existing requirements for other components of the Trade Automation Package. This way, we do not waste time installing modules that rely on the database if the database has not yet been installed.

Archived

By specifying the following requirements for our Trade DBServer Module Software Product:

DATABASE	database.family	DB
DATABASE	database.version	8.2

it is guaranteed that a database server installation exists on the target system at the time of installation of the Trade DBServer Module installation.

> **Note:** In a real-life implementation, the user customer should be given instructions on how to modify the requirements that are set up by default for the Trade DBServer Module in order to integrate it into a specific environment.

Figure 10-17 shows capabilities and requirements for the Trade DBServer Module Software Product.

Requirements and Capabilities

Requirement Name	Requirement Type	Requirement Values
database.family	DATABASE	DB2
database.version	DATABASE	8.2

Capability Type
APPLICATION

Capability Name	Capability Value
software.title	TRADE
software.version	3.1
software.name	DB
software.vendor	IBM

Figure 10-17 Trade DBServer Module capabilities and requirements

Software Package(s)

Now, to help determine which files to use during the installation, we created a Software Package named Trade DBServer Module Installable.

The only file that is referenced by the Software Package is the Table.ddl, which we will need during the installation in order to create the database tables in the trade3db database. Even though this file can be used as input to the DB2 command line processor on any platform, we added, for the sake of the example, a requirement for the Software Package, specifying the following:

OS	os.family	Windows 2000 Server SP4

Archived

During installation, this requirement has to be honored by the target server in order for the TIO/TPM Server to identify a Software Package to be used for the installation, and thus, the Trade DBServer Module will only install on systems the supports this requirement and, naturally, the requirements of the Trade DBServer Module itself.

> **Note:** In a real-life implementation, the user customer should be provided instructions on how to modify the requirements that is set up by default for the Trade DBServer Module Installable in order to integrate it into a specific environment.

The Table.ddl file referenced by the Trade DBServer Module Installable resides inside the trade3install.zip installation archive. This archive will be copied to the TIO/TPM server during installation of the Trade Automation Package, however, since it does not make sense to copy the whole archive to the system where the Trade DBServer Module is installed, we need a way to extract the Table.ddl file to the file repository during automation package. The obvious way is to use a post-install-workflow, similar to the one we used in 10.5.1, “Creating the trade Service Access Point” on page 323.

Example 10-9 shows the Trade_Extract_TableDDL workflow that will be used to make the Table.ddl file available in the File Repository so it can be distributed through the Trade DBServer Module Installable installation.

Example 10-9 Trade_Extract_TableDDL post-install workflow

```
workflow Trade_Extract_TableDDL  LocaleInsensitive

  #  identify packages to gater information from
  var installableName=”Trade DBServer Module Installable”
  var archiveInstallableName=”Trade Application Module Installable”
  # set the name of the file to extract
  var extractFile=”t3install/DB2/Table.ddl”

  # get the root path of the  FileRepository
  var repositoryRoot=DCMQuery(/softwareinstallable[@name=$installableName]/filerepository/@rootpath)

  # get the name of the archive file
  var archiveName=DCMQuery(/softwareinstallable[@name=$archiveInstallableName]/file/@name)
  var archivePath=DCMQuery(/softwareinstallable[@name=$archiveInstallableName]/file/@path)
  var archiveFile=Jython[repositoryRoot + archivePath + “/” + archiveName]


  scriptlet( archiveFile, archivePath, extractFile, repositoryRoot) language = bash  <<EOF
    # convert filenames to cygwin format
    extractFile=‘cygpath -u  ${extractFile}‘
    archiveFile=‘cygpath -u  ${archiveFile}‘
    archivePath=‘cygpath -u  ${archivePath}‘
    repositoryRoot=‘cygpath -u  ${repositoryRoot}‘
```

```
    # build the unzip ocmmand
    command="unzip -o ${archiveFile} ${extractFile} -d ${repositoryRoot}${archivePath}"

    # exxecute it
    TIOlog "info" "executing : ${command}"
    out='${command} 2>&1'
    TIOlog "info" "$?   ${out}"
EOF
```

In the first part of the workflow, you see how the DCMQuery command is used to gather information from the data center model to get file and path names from the Software Packages and the File Repository in which the files are hosted. The second half of the Trade_Extrace_TableDDL shows a very simple example on how to use the scriptlet to execute a bash script on the TIO/TPM server itself. By omitting the target parameter on the scriptlet call, the embedded script will be executed on the TIO/TPM server instead on a target system. You should also note the way the **cygpath** command is used inside the workflow in order to ensure the use of UNIX naming conventions for all files and paths used to set up the **unzip** command.

Now that we have ensured the existence of the Table.ddl file in the File Repository, we can complete the definition of the Trade DBServer Module Installable, as shown in Figure 10-18.

Figure 10-18 Trade DBServer Module Installable software package properties

Notice that the associated device driver, which has to be provided as a integral part of the Trade Automation Package, and hence is developed by us, is named Trade_DBServer_Module_Installable_Driver.

Archived

Variables

We also define a set of variables for the Software Package, which will be used to help find the installation directory of the underlying DB2 UDB Enterprise Server installation, as described in 10.5.2, “Finding installation directories” on page 324. For the Trade DBServer Module Installable, these variables are created with the standard values shown in Figure 10-19, and it is expected that they are customized during the installation of the Trade Automation Package.

Figure 10-19 Trade DBServer Module Installable software package variables

The Trade_DBServer_Module_Installable_Driver Device Driver

To control the installation of the Trade DBServer Module Installable, we need to assign a device driver that, at a minimum, contains a workflow that implements the SoftwareInstallable.Install logical device operation. In addition, we will develop a workflow implementing the SoftwareInsallable.Distribute LDO, which will be used to transfer the Table.ddl file to the target server prior to the creation of the database.

Example 10-10 shows the XML statements embedded in the tc-driver.xml file for the entire Trade Automation Package, which are used to define the device driver to the DCM. The device driver we assign to the Trade DBServer Module Installable will be named Trade_DBServer_Module_Installable_Driver. Details regarding the workflows embedded in the device driver are provided in “Trade_DBServer_Module_Installable_Install” on page 348 and “Trade_DBServer_Module_Installable_Distribute” on page 352.

Example 10-10 XML to define the Trade_DB2Server_Module_Installable_Driver

```
<device-model name="Trade_DBServer_Module_Installable_Driver"
category="Trade">
  <workflow name="Trade_DBServer_Module_Installable_Distribute" />
  <workflow name="Trade_DBServer_Module_Installable_Install" />
</device-model>
```

Before we look at the details of the workflows, we need to take a look at the Software Resource Templates of the Trade DBServer Module Software Product

to gain knowledge about the customization parameters that will be available to the installation workflows.

Software Resource Templates

When modelling the installation activities and resources in TIO/TPM, it is necessary to understand the default behavior of the various software resource types in the TIO/TPM data center model. To translate this default behavior to the task at hand, it quickly becomes clear that we need to design and develop Software Resource Templates to create the desired objects in the DCM as well as workflows implementing the resource specific logical device operations.

The software models used to create the DCM objects related to the Trade DBServer Module are:

INSTALLATION	Creates the Software Installation object representing the installation on a specific server. This object will be used to host child objects such as Instance and Configuration.
INSTANCE	Creates the Software Instance object as a child of the Software Installation. This object represents the trade3 db2 instance used to host the database required by the Trade application.
CONFIGURATION	Creates a Software Resource object, which we will use to represent the trade3db database

The Software Resource Templates will be created in a hierarchy in order to represent the relationships between them. The INSTALLATION SRT is required to instantiate the Trade DBServer Module installation on the target server, which in turn “owns” the INSTANCE that “owns” the CONFIGURATION.

To control the behavior of each of the objects created by the Software Resource Templates, a device driver, which implements the resource specific LDOs, needs to be assigned to each Software Resource Template. If this step is skipped, no implementations for the resource specific LDOs are available, and no actions will be performed. The necessary device drivers have to provide functionality, through the associated workflows, that is specific to our implementation of the Trade DBServer Module, so naturally, these have to be created as part of and delivered with the Trade Automation Package.

Now, if we use the software resource model described above to create and register the objects related to the DB2 Server in the DCM, we will not achieve our goal. Since the Software Resource Templates are parented by the Trade DBServer Module Software Product, the software resources they create will be children of this Software Product, and not the DB2 UDB Enterprise Server installation, which owns the objects that we will actually be implementing, that is,

a DB2 Instance, a DB2 database, and the tables for Trade. Physically, these objects will manifest themselves as children of the DB2 UDB Enterprise Server installation, so within our DCM, we have to reflect this to ensure that the logical model of the data center (in the DCM) is consistent with the physical implementation.

This implies that we will have to create a Software Resource Template structure, which will allow us to create the INSTANCE object (and everything nested under it) as a child of another Software Installation. Luckily, TIO/TPM supports this through the Software Resource Type named FOREIGN-CONFIGURATION. As for all other Software Resource Templates besides INSTALLATION and INSTANCE, TIO/TPM does not offer any automated processing to create the resources parented by a FOREIGN_CONFIGURATION Software Resource Template; it is the responsibility of the implementation workflow to discover the FOREIGN_CONFIGURATION template, and take action accordingly. Normally, the action that would be taken is a call to the HostingEnvironment.Host logical device operation for the hosting resource (the DB2 UDB Enterprise Server, installation in our case). However, since we have no knowledge about the implementation details of the Software Installation that represents the DB2 UDB Server on the target system, we do not know if the IHostingEnvironment.Host logical operation is supported in the device driver associated with the Software Installation, so we have to provide all of this functionality ourselves.

So, in order to create the DB2 related resources in the data center model as children of the DB2 UDB Enterprise Server Software Installation object, we add a FOREIGN_CONFIGURATION Software Resource Template to our software model, making the Software Resource Template hierarchy of the Trade DBServer Module Software Product look like this:

INSTALLATION
  FOREIGN_CONFIGURATION
    INSTANCE
      CONFIGURATION

As a result, the INSTALLATION template will be used to create a Software Installation object on the target server, and associate this with the Trade DBServer Module Software Product. As children of this Software Installation object, TIO/TPM will clone the reminder of the SRT hierarchy (the FOREIGN_CONFIGURATION template and all its children) to the new Software Installation, and the workflow that implements the SoftwareInstallable.Install LDO is responsible for inspecting the SRT hierarchy and take appropriate action(s).

Device driver associations

To control the behavior of the resources that are created from the Software Resource Templates, we may associate a device driver to each of them. TIO/TPM normally applies special processing to the INSTALLATION and

INSTANCE and therefore you will typically find that only these objects have associated device drivers, and that the workflows implementing the SoftwareInstallable.Install, SoftwareInstallable.Uninstall, SoftwareInstance.AddInstance, and SoftwareInstance.RemoveInstance logical operations are responsible for inspecting the nested object hierarchy and perform the necessary tasks.

For the Trade DBServer Module Software Package software model, we associate the device drivers and related workflows implementing the required logical operations, as shown in Table 10-5.

Table 10-5 Trade DBServer Module related devise drivers and workflows

SRT	Type	Name	Workflow
INSTALLATION	device driver	Trade_DBServer_Module_Intallation_Driver	
	LDOs	SoftwareInstallation .AddInstance	Trade_DBServer_Module_Intalla tion_AddInstance
		SoftwareInstallation .Uninstall	Trade_DBServer_Module_Intalla tion_Uninstall
INSTANCE	device driver	Trade_DBServer_Module_Instance_Driver	
	LDOs	SoftwareInstance .RemoveInstance	Trade_DBServer_Module_Instan ce_RemoveInstance
		SoftwareInstance .Start	Trade_DBServer_Module_Instan ce_Start
		SoftwareInstance .Stop	Trade_DBServer_Module_Instan ce_Stop
		SoftwareInstance .UpdateDCM	Trade_DBServer_Module_Instan ce_UpdateDCM

In addition, these workflows will call additional workflows, which are not associated with any particular device driver< in order to execute specific actions, which are repetitive, or provide a very specific set of functions.

Archived

Software Resource naming

The naming of Software Resources that are created based on the templates can be controlled by creating a template parameter named *resource-name*, and by providing a value that matches your requirements. Figure 10-20 on page 340 shows the final Software Resource Template hierarchy for the Trade DBServer Module, and here we see that the names that will be assigned to the objects created by the templates are:

Installation Trade Database Module Installation
Foreign_Configuration
Trade Database Creation
Instance trade3
Configuration trade3db

Software Resource Template parameters

The last step in the complete the software model for the Trade DBServer Module is to associate parameters with each of the templates. These parameters hold configuration information that will be used during the instantiation of each of the software resources. These parameters will be read by the workflows responsible for creating the physical objects represented by the software resources.

Table 10-6 Trade DBServer Module related SRT parameters

SRT	name	value
INSTALLATION	resource-name	Trade Database Module Installation
	trade_dbadmin_SAP_domain	trade
	trade_dbadmin_SAP_context	db
	trade_dbadmin_SAP_searchkey	dbadmin
FOREIGN_CONFIGURATION	resource-name	Trade Database Creation

Archived

SRT	name	value
INSTANCE	resource-name	trade3
	db2type	ese
	port	50010
	trade_dbinstance_SAP_domain	trade
	trade_dbinstance_SAP_context	db
	trade_dbinstance_SAP_searchkey	dbinstance
	trade_dbinstance_user_group.1	DB2ADMNS
	trade_dbinstance_user_group.2	Administrators
CONFIGURATION	resource-name	trade3db
	trade_dbuser_SAP_domain	trade
	trade_dbuser_SAP_context	db
	trade_dbuser_SAP_searchkey	dbuser

As Table 10-6 on page 338 shows, most of these parameters are related to identifying the credentials necessary to perform specific operations. For example, in order to create the DB2 instance, our command has to run using credentials that have dbadmin authorization, and to operate the instance, we will use credentials pointing to the instance owner. For each of the software resources, we associate parameters providing the necessary values to identify the specific set of credentials that was defined in the SAP, which will be created during installation of the Trade Automation Package (see 10.5.1, “Creating the trade Service Access Point” on page 323). In addition, you see in Table 10-6 on page 338, that for the INSTANCE resource, which in our model maps to a DB2 Instance, we have added parameters such as *port* and *db2type*, which will be used for controlling the creation of the instance.

Archived

Finally, we can take a look at the completed Software Resource Template hierarchy for the Trade DBServer Module, as shown in Figure 10-20.

Configuration Templates

Configuration Templates

Trade DB Module Installation

Software Resource Type: Installation Device Model: Trade_DBServer_Module_Installation_Driver

Parameter	Value
resource-name	Trade Database Module Installation
trade_dbadmin_SAP_domain	trade
trade_dbadmin_SAP_context	db
trade_dbadmin_SAP_searchkey	dbadmin

Trade DBServer Instance and Database Creation

Software Resource Type: Foreign configuration

Parameter	Value
resource-name	Trade Database Creation

Trade DB2 Instance

Software Resource Type: Instance Device Model: Trade_DBServer_Module_Instance_Driver

Parameter	Value
resource-name	trade3
db2type	ese
port	50010
trade_dbinstance_SAP_domain	trade
trade_dbinstance_SAP_context	db
trade_dbinstance_SAP_searchkey	dbinstance
trade_dbinstance_user_group.1	DB2ADMNS
trade_dbinstance_user_group.2	Administrators

Trade Database Configuration

Software Resource Type: Configuration

Parameter	Value
resource-name	trade3db
trade_dbuser_SAP_domain	trade
trade_dbuser_SAP_context	db
trade_dbuser_SAP_searchkey	dbuser
trade_dbuser_user_group.1	DB2USERS
trade_dbuser_user_group.2	Administrators

Figure 10-20 Trade DBServer Module Software Resource Templates

Package overview

Based on the parameters and attributes we have defined for the Trade DBServer Module Software Product, along with the related Software Package and the embedded Software Resource Templates, we have created the software model for:

- Creation of the Trade application database instance named trade3
- Creation of the trade3db database

in a way that will be easily customizable for the potential users of the automation package. Figure 10-21 shows a depiction of the resources involved and their relationships.

Figure 10-21 Trade DBServer Module deployment model

Creating the software objects from the Automation Package

Since we want to create the Trade DBServer Module Software Product as part of the Trade Automation Package installation, we need to provide all the definitions and parameters discussed in the previous section in an XML file, which can be included in the automation package, and automatically imported into the DCM during automation package installation.

The easiest way to create this XML file is to create all the definitions manually (using the TIO/TPM Console) and use the **`dcmexport`** utility to export the contents of the entire DCM. Then you can identify the <software-module > definition named Trade DBServer Module and copy the entire `<software-module..> ... </software-module>` section to a flat file, and save it as part of the APDE project for Trade. We used a specific XML directory within the project to store all the XML files for the Trade Automation Package.

The contents of the Trade_DBServer_Module.xml is shown in Example 10-11.

To automatically import this XML file as part of the automation package installation, you will have to add an item definition for the xml file using the **import** function for the <dcm> section in the tc-driver.xml file for the automation package:

```
<dcm>
   <item name="xml/Trade_DBServer_Module.xml" action="import"/>
</dcm>
```

The xml deck needed to define our Software Product and Software Package, along with the embedded file, variable, Software Resource Template, and device driver definitions, are shown in Example 10-11.

Example 10-11 Trade_DBServer_Module.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE datacenter SYSTEM "file:../xml/xmlimport.dtd">

<datacenter>
  <software-module name="Trade DBServer Module"
                   version="3.1"
                   vendor="IBM" title="Trade DB2 Instance, Database and Tables"
                   description="Trade database"
                   is-draft="false">
          <software-capability name="software.title" value="TRADE" />
          <software-capability name="software.version" value="3.1" />
          <software-capability name="software.name" value="DB" />
          <software-capability name="software.vendor" value="IBM" />
          <supported-requirement-type value="APPLICATION" />
          <software-requirement name="database.family"
                                type="DATABASE"
                                enforcement="MANDATORY"
                                hosting="true"
                                accept-non-existing="false">
                    <software-requirement-value value="DB2" />
          </software-requirement>
          <software-requirement name="database.version"
                                type="DATABASE"
                                enforcement="MANDATORY"
                                hosting="true"
                                accept-non-existing="false">
                    <software-requirement-value value="8.2" />
          </software-requirement>

          <installable-package name="Trade DBServer Module Installable"
                               is-device-model="Trade_DBServer_Module_Installable_Driver"
                               locale="en_US"
                               version="3.1"
                               priority="0"
                               file-repository="LocalFileRepository"
                               status="tested">
                   <file name="Table.ddl" path="/trade3/t3install/DB2" />
```

```
                <property component="DEPLOYMENT_ENGINE"
                          name="install_dir_softwareresource_type"
                          value="" />
                <property component="DEPLOYMENT_ENGINE"
                          name="install_dir_parameter_name"
                          value="" />
                <property component="DEPLOYMENT_ENGINE"
                          name="install_dir_custom_default"
                          value="c:/ibm/sqllib" />
                <property component="DEPLOYMENT_ENGINE"
                          name="install_dir_system_default"
                          value="c:\Program Files\sqllib" />

            <software-requirement name="os.family"
                                  type="OS"
                                  enforcement="MANDATORY"
                                  hosting="true"
                                  accept-non-existing="false">
                          <software-requirement-value value="Windows 2000 Server SP4" />
                </software-requirement>
        </installable-package>

        <software-resource-template name="Trade DB Module Installation"
                          software-resource-type="INSTALLATION"
                          software-resource-device-model="Trade_DBServer_Module_Installation_Driver"
                          multiplicity-type="Unspecified"
                          software-configuration-type="Regular"
                          is-selected="true">

           <software-resource-template name="Trade DBServer Instance and Database Creation"
                                  software-resource-type="FOREIGN-CONFIGURATION"
                                  multiplicity-type="Unspecified"
                                  software-configuration-type="Regular"
                                  is-selected="true">

            <software-resource-template name="Trade DB2 Instance"
                                     software-resource-type="INSTANCE"

software-resource-device-model="Trade_DBServer_Module_Instance_Driver"
                                     multiplicity-type="Unspecified"
                                     software-configuration-type="Regular"
                                     is-selected="true">
                <software-resource-template name="Trade Database Configuration"
                                        software-resource-type="CONFIGURATION"
                                        multiplicity-type="Unspecified"
                                        software-configuration-type="Regular"
                                        is-selected="true">
                          <template-param name="resource-name"
                                        value="trade3db"
                                        is-changeable="true" />
                          <template-param name="trade_dbuser_SAP_domain"
                                        value="trade"
                                        is-changeable="true" />
                          <template-param name="trade_dbuser_SAP_context"
```

Archived

```
                                value="db"
                                is-changeable="true" />
                    <template-param name="trade_dbuser_SAP_searchkey"
                                value="dbuser"
                                is-changeable="true" />
                    <template-param name="trade_dbuser_user_group.1"
                                value="DB2USERS"
                                is-changeable="true" />
                    <template-param name="trade_dbuser_user_group.2"
                                value="Administrators"
                                is-changeable="true" />
            </software-resource-template>
            <template-param name="resource-name"
                        value="trade3"
                        is-changeable="true" />
            <template-param name="db2type"
                        value="ese"
                        is-changeable="true" />
            <template-param name="port"
                        value="50010"
                        is-changeable="true" />
            <template-param name="trade_dbinstance_SAP_domain"
                        value="trade"
                        is-changeable="true" />
            <template-param name="trade_dbinstance_SAP_context"
                        value="db"
                        is-changeable="true" />
            <template-param name="trade_dbinstance_SAP_searchkey"
                        value="dbinstance"
                        is-changeable="true" />
            <template-param name="trade_dbinstance_user_group.1"
                        value="DB2ADMNS"
                        is-changeable="true" />
            <template-param name="trade_dbinstance_user_group.2"
                        value="Administrators"
                        is-changeable="true" />
        </software-resource-template>
        <template-param name="resource-name"
                    value="Trade Database Creation"
                    is-changeable="true" />
    </software-resource-template>
    <template-param name="resource-name"
                value="Trade Database Module Installation"
                is-changeable="true" />
    <template-param name="trade_dbadmin_SAP_domain"
                value="trade"
                is-changeable="true" />
    <template-param name="trade_dbadmin_SAP_context"
                value="db"
                is-changeable="true" />
    <template-param name="trade_dbadmin_SAP_searchkey"
                value="dbadmin"
                is-changeable="true" />
</software-resource-template>
```

```
    </software-module>

</datacenter>
```

10.6.2 The Trade_DBServer_Module_Installable_Driver

As mentioned a couple of times already, TIO/TPM will, when the Software Module.Install logical operation is called for a specific Software Product, automatically determine which Software Package to use, and start the installation process by calling the SoftwareInstallable.Install LDO for that Software Package.

The standard logical device operations associated with the Software Installable are SoftwareInstallable.Install and SoftwareInstallable.Distribute. In addition to workflows to implement these two logical device operations, we will need additional workflows to support the installation, especially since we cannot assume that the HostingEnvironment.Host has been implemented for the underlying DB2 UDB Enterprise Server implementation, and we have no control over the functionality of the workflow associated with the SoftwareInstallation.AddInstance for the DB2 Server installation.

For these reasons, all the installation activities, including creations of the DB2 instance, have to be handled from the workflow implementing SoftwareInstallable.Install for the Trade DBServer Module Installable software package.

So, we decided to break up the various tasks into the workflows shown in Figure 10-22 on page 346. We already have a toolbox of ITSO supporting workflows that implement specific functions that will help us perform certain tasks. So, as part of the development of the Trade_DBServer_Module_Installable_Driver, we will have to create one workflow implementing a logical device operation, the SoftwareInstallable.Installa LDO, and three helper workflows that will be used to create resources for the foreign DB2 Server installation.

Figure 10-22 The Trade_DBServer_Module_Installable_Driver

Basically, all of the four main workflows used for installation could have been combined into a single workflow; however, the chosen division allows for easy unit testing as the development progresses, as well as easy maintenance of workflows.

The functionality for each of the workflows are:

- Trade_DBServer_Module_Installable_Install
  Main workflow to initiate the installation process through implementation of the SoftwareInstallable.Install logical device operation.
- Trade_DBServer_Add_Instance
  Creation of the trade3 DB2 Instance on the target server.
- Trade_DBServer_Create_Database
  Creation of the trade3db DB2 database on the target server.

The helper workflow:

- Trade_Get_Simulation
  Retrieves the state of the simulation setting, and sets a variable accordingly.
- Trade_Get_Hosting_SRTParameter_Or_Default
  Retrieves configuration parameters from the prerequisite hosting middleware installation.

Archived

The role of the supporting workflows are:

- ITSO_Find_Supporting_Installation_for_Module_Reqirements
  Finds the supporting Software Installation based on requirements.
- ITSO_Synchronize_Credentials
  Synchronizes credentials between a target system and the TIO/TPM Server hosting the Deployment Engine.
- ITSO_HostingEnvironment_Host
  Creation of the new Software Instance object in the foreign DB2 Server Installation.
- ITSO_Create_User
  Creates a new user and home directories.

Figure 10-23 shows the properties of the Trade_DBServer_Module_Installable_Driver.

Figure 10-23 Trade_DBServer_Module_Installable_Driver Workflows

To define the Trade_DBServer_Module_Installable_Driver to the DCM during installation of the automation package, the statements in Example 10-12 must be present in the automation package manifest file - tc-driver.inf.

Example 10-12 xml definitions for the Trade_DBServer_Module_Installable_Driver

```
<device-model name="Trade_DBServer_Module_Installable_Driver"
category="Trade">
      <workflow name="Trade_DBServer_Module_Installable_Distribute" />
      <workflow name="Trade_DBServer_Module_Installable_Install" />
</device-model>
```

And naturally, all the referenced workflows have to have been defined. This is normally taken care of automatically by the APDE platform.

The following section describes the workflows used in the Trade_DBServer_Module_Installable_Driver in greater detail.

Archived

Trade_DBServer_Module_Installable_Install

The workflow implementing the SoftwareInstallable.Install logical device operation is the anchor of the installation. The one used for the Trade DBServer Module Installable is show in Example 10-13 on page 349. The following list describes the high-level functionality of the workflow:

1. Besides the pretty straight-forward input verification, and gathering of base data, we start the workflow by calling the Trade_Get_Simulation workflow (described in 10.5.3, “Controlling simulation” on page 327).
2. Next we need to find the Software Installation that honors the DATABASE requirements in order to find the location where the db2 executables can be found. This is achieved by a ‘utility’ workflow, developed by us, named ITSO_Find_Supporting_Installation_for_Module_Reqirements. Please refer to 10.10.1, “ITSO_Find_Supporting_Installation_for_Module_Requirements” on page 494 for more details.
3. We also need to find the installation directory for the DB2 Server Installation by passing the supportingSoftwareInstallationID to the Trade_Get_Hosting_SRTParameter_Or_Default workflow described in 10.5.2, “Finding installation directories” on page 324.
4. Our next step is to make sure that the credentials needed to execute scriptlets (file-transfer and execute-command) are in place between the TIO/TPM Server system (hosting the Deployment Engine) and the target system. Because we need to access the target system using credentials that allows us to operate DB2, we will need to use a special searchkey when invoking these scriptlets, and the credentials supporting this searchkey must be in place.

   The credentials to be synchronized are based on the definitions in the trade SAP that was created during installation of the Automation Package, and modified by the customer to reflect the policies in a particular environment. The ITSO_Synchronize_Credentials workflow handles this synchronization for us, and more details are provided in 10.10.2, “ITSO_Synchronize_Credentials” on page 497.
5. Before continuing, we have to commit the creation of the Software Installation. This is normally handled by the SoftwareInstallable.Install LDO, however, since we will be branching off the normal installation path, and we will need the committed Software Installation in the following, we have to commit ourselves. To commit the Software Installation, we update the pending flag (setting the value to “*N*”) by calling the java:com.ibm.tivoli.orchestrator.datacentermodel.helper.SoftwareHelper#updateResourceFlag Java plug-in.
6. Now it is time to create the software resources parented by the FOREIGN_CONFIGURATION template. This is achieved by our own

ITSO_HostingEnvironment.Host LDO, which was developed using inspiration from the standard implementation found in the ITSO_HostingEnvironment_Host workflow. You can find the details of inner workings of the ITSO_HostingEnvironment_Host workflow in 10.10.4, “ITSO_HostingEnvironment_Host” on page 500.

7. Depending on wether or not any exceptions were thrown from the ITSO_HostingEnvironment_Host workflow, we are not ready to terminate the workflow. If any exceptions were received (indicating problems), we have to set the status of the Software Installation back into pending state (pending=”*Y*”), before the exception is transferred back to the SoftwareInstallable.Install LDO; otherwise, we simply terminate.

Example 10-13 Trade_DBServer_Module_Installable_Install

```
workflow Trade_DBServer_Module_Installable_Install(in SoftwareID, in DeviceID, in
SoftwareResourceTemplateID) implements SoftwareInstallable.Install LocaleInsensitive

#####################################################
## 1:  initialize and validate input
#####################################################

var simulation
Trade_Get_Simulation(simulation)

var workflowName=”Trade_DBServer_Module_Installable_Install”
var message = Jython[“<b>” + workflowName + “: </b>”]
var msg

var SoftwareInstallableID=SoftwareID
var ServerID=DeviceID
var ServerName=DCMQuery(/server[@id=$ServerID]/@name)
var DeploymentRequestID
java:com.thinkdynamics.kanaha.de.javaplugin.GetCurrentDeploymentRequestId(DeploymentRequestID)
var SoftwareInstallationID

try
  ####################################################################
  ## 2: Find the supporting installation
  ####################################################################
  # get the ID of the SoftwareModule
  var
SoftwareModuleID=DCMQuery(/softwareinstallable[@id=$SoftwareInstallableID]/softwareinstallationmechanism/@
moduleid)

  var requirementType=”DATABASE”
  var supportingSoftwareInstallationID
  ITSO_Find_Supporting_Installation_for_Module_Reqirements(requirementType, SoftwareInstallableID,
ServerID, supportingSoftwareInstallationID)

  ####################################################################
  ## 3: Find install_dir of the DB2 Server
```

```
  ##################################################################
  var install_dir_parmname="install_dir"
  var install_dir

  Trade_Get_Hosting_SRTParameter_Or_Default(SoftwareInstallableID, supportingSoftwareInstallationID,
install_dir_parmname, install_dir)
  ## fail if no values for install_dir was found
  if Jython[not install_dir] then
    msg= Jython[message + " Cannot find a value for template parameter " + install_dir_parmname + "in SRT
" + DBServerInstallationTemplateID]
    log error msg
    throw missingParameterException msg
  endif

  ##################################################################
  ## 4: Get the dbadmin searchkey
  ##################################################################
  var trade_dbadmin_sap_searchkey_parmname = "trade_dbadmin_SAP_searchkey"
  var trade_dbadmin_sap_searchkey=DCMQuery(/templateparam[@templateid=$SoftwareResourceTemplateID and
@name=$trade_dbadmin_sap_searchkey_parmname]/@value)

  ## fail if no values for trade_dbadmin_SAP_searchkey was found
  if Jython[not trade_dbadmin_sap_searchkey] then
    message = Jython[message + " Cannot find a value for template parameter " +
trade_dbadmin_sap_searchkey_parmname + "in SRT " + DBServerInstallationTemplateID]
    log error msg
    throw missingParameterException msg
  endif


  ##################################################################
  ## 5: Set credentials for server and TIOServer
  ##################################################################
  var dbadmin_sap_applicationprotocoltype=DCMQuery(/applicationprotocol[@name="UNKNOWN"]/@id)
  var dbadmin_sap_protocoltype=DCMQuery(/protocoltype[@name="unknown"]/@id)
  var
dbadmin_sap_domain=DCMQuery(/softwareresourcetemplate[@id=$SoftwareResourceTemplateID]/templateparam[@name
="trade_dbadmin_SAP_domain" and @templateid=$SoftwareResourceTemplateID]/@value)
  var
dbadmin_sap_context=DCMQuery(/softwareresourcetemplate[@id=$SoftwareResourceTemplateID]/templateparam[@nam
e="trade_dbadmin_SAP_context" and @templateid=$SoftwareResourceTemplateID]/@value)
  var
dbadmin_sap_searchkey=DCMQuery(/softwareresourcetemplate[@id=$SoftwareResourceTemplateID]/templateparam[@n
ame="trade_dbadmin_SAP_searchkey" and @templateid=$SoftwareResourceTemplateID]/@value)
  var dbadmin_sap_id=DCMQuery(/sap[@domain=$dbadmin_sap_domain and @context=$dbadmin_sap_context and
@protocoltypeid=$dbadmin_sap_protocoltype and @appprotocolid=$dbadmin_sap_applicationprotocoltype]/@id)
  var dbadmin_credentials_id
  java:com.thinkdynamics.kanaha.de.javaplugin.sap.GetSapCredentials(dbadmin_credentials_id,
dbadmin_sap_searchkey, dbadmin_sap_id)

  var DE_Device_ID
  java:com.thinkdynamics.kanaha.de.javaplugin.datacentermodel.FindDEDeviceId(DE_Device_ID)
  ITSO_Synchronize_Credentials(dbadmin_credentials_id, DeviceID,DE_Device_ID)
```

```
  ##########################################################
  ## 6: Commit the creation of this SoftwareInstallation
  ##########################################################
  SoftwareInstallationID=DCMQuery(/softwareinstallation[@managedsystemid=$ServerID and
@softwaremoduleid=$SoftwareModuleID and @pending="Y"]/@id)

java:com.ibm.tivoli.orchestrator.datacentermodel.helper.SoftwareHelper#updateResourceFlag(SoftwareInstalla
tionID, "false")

  ######################################################################
  ## 7: Create the Foreign Instance object
  ######################################################################
  #log debug Jython["new installation id is: " + SoftwareInstallationID]
  var installationResourceType=DCMQuery(/softwareresourcetype[@name="INSTALLATION"]/@id)
  var instanceResourceType=DCMQuery(/softwareresourcetype[@name="INSTANCE"]/@id)
  var foreignResourceType=DCMQuery(/softwareresourcetype[@name="FOREIGN-CONFIGURATION"]/@id)
  var
installationSRT=DCMQuery(/softwareinstallation[@id=$SoftwareInstallationID]/softwareresourcetemplate[@soft
wareresourcetype=$installationResourceType]/@id)
  var foreignSRTID=DCMQuery(/childsoftwareresourcetemplate[@parenttemplateid=$installationSRT]/@id)
  array objects
  ITSO_HostingEnvironment_Host(supportingSoftwareInstallationID, SoftwareModuleID, foreignSRTID, objects)

 ######################################################################
 ## 8: Install the Trade Application
 ######################################################################
  foreach objectID in objects do
    var targetResourceID=DCMQuery(/softwareresource[@id=$objectID]/@parentresourceid)
    var srtID=DCMQuery(/softwareresource[@id=$objectID]/@resourcetemplateid)
   log info Jython[".....About to call Trade_DBServer_Add_Instance("+targetResourceID+", "+srtID+")"]
    Trade_DBServer_Add_Instance(targetResourceID, srtID)
  done

catchall exception
  ######################################################################
  ## 9: Error handling
  ######################################################################
  msg = Jython[message + exception]
  log error msg
  # delete foreign resources
  foreach objectID in objects do
    log debug Jython["about to delete " + objectID]
    var objID=DCMQuery(/softwareresource[@id=$objectID]/@id)
    if Jython[ objID != None] then
      DCMDelete(/softwareresource[@id=$objectID]/@id)
    endif
  done
  # Set the status pending if creation of freign objects failed

java:com.ibm.tivoli.orchestrator.datacentermodel.helper.SoftwareHelper#updateResourceFlag(SoftwareInstalla
tionID, "true")
  rethrow
endtry
```

Trade_DBServer_Module_Installable_Distribute

The Trade_DBServer_Module_Installable_Distribute workflow implements the logical device operation SoftwareInstallable.Distribute, which is intended to be used to distribute the files associated with a Software Package to the target system.

The following implementation is very generic; in fact, only the destination directory is specific to the Trade application.

The following highlights the main functions of the Trade_DBServer_Module_Installable_Distribute workflow:

1. Initialize and gather data.
2. Build the name of the temporary directory on the target system using the DeploymentID as identifier.
3. Make sure that the destination path has been created by executing a **mkdir** command using the Device.ExecuteCommand logical device operation.
4. Transfer the file from the File Repository to the target system.

Example 10-14 shows how we coded this workflow.

Example 10-14 Trade_DBServer_Module_Installable_Distribute

```
workflow Trade_DBServer_Module_Installable_Distribute(in SoftwareID, in DeviceID) implements
SoftwareInstallable.Distribute LocaleInsensitive

########################################
## 1: initialize and gather data
########################################
var simulation
Trade_Get_Simulation(simulation)

var workflowName="Trade_DB2erver_Module_Installable_Distribute"
var message = Jython["<b> " + workflowName + "</b>: "]
var msg

try
  ##############################################################
  ## 2: fetch control inforamtion
  ##############################################################
  var SoftwareInstallableID=SoftwareID
  var ServerID = DeviceID
  var FileRepositoryID = DCMQuery(/softwareinstallable[@id=$SoftwareInstallableID]/filerepository/@id)
  var SourcePath = DCMQuery(/softwareinstallable[@id=$SoftwareInstallableID]/file/@path)
  var SourceFileName = DCMQuery(/softwareinstallable[@id=$SoftwareInstallableID]/file/@name)

  ##########################################################################################
  ## 3: build the name of the temporary directory on the target system using the DeploymentID
  ##########################################################################################
  var DeploymentRequestID
```

```
  java:com.thinkdynamics.kanaha.de.javaplugin.GetCurrentDeploymentRequestId(DeploymentRequestID)
  var DestinationPath = Jython("/tmp/" + DeploymentRequestID + "/db2")

  ##########################################################
  ##  4: make sure that the destination path has been created
  ##########################################################
  var cmd=Jython["mkdir -p " + DestinationPath]

  if Jython[simulation == "yes" ] then
    log debug Jython["<b>simulation:</b>  executing: " +  cmd]
    log debug Jython["<b>simulation:</b>  Calling FileRepository.GetFile(" + FileRepositoryID + ", " +
SourcePath + ", " + SourceFileName + ", " + ServerID + ", " + DestinationPath + ", " + SourceFileName + ",
<null>)"]
  else
    ##############################################
    ##  5: transfer the file from the FileRepository
    ##############################################
    Device.ExecuteCommand(ServerID, cmd, "/", <null>, <null>, <null>, <null>, <null>, <null>)
    # send the needed file to the target system
    log info Jython["<b>...transfering file: " + SourceFileName + " to " + DestinationPath + "</b>"]
    FileRepository.GetFile(FileRepositoryID, SourcePath, SourceFileName, ServerID, DestinationPath,
SourceFileName, <null>)
  endif
catchall exception
  ##############################################
  ##  6: handle errors
  ##############################################
  msg = Jython[message + " received: " + exception]
  log error message
  rethrow
endtry
```

10.6.3 The Trade_DBServer_Module_Installation_Driver

The Trade_DBServer_Module_Installation_Driver is used to provide Trade specific implementations of the following logical device operations:

- SoftwareInstallation.AddInstance
- SoftwareInstallation.Uninstall

Archived

However, since no INSTANCE Software Resource Templates are created as direct children of the INSTALLATION SRT, the SoftwareInstallation.AddInstance will fail, and thus, there is no need for us to provide an implementation for this LDO. The content of the Trade_DBServer_Module_Installation_Driver is shown in Figure 10-24.

Figure 10-24 Trade_DBServer_Module_Installation_Driver Overview

By assigning the Trade_DBServer_Module_Installation_Driver to the INSTALLATION Software Resource Template in the Trade DBServer Module, we make sure that the specific functionality built into the Trade_DBServer_Module_Installation_Uninstall workflow will be used whenever the Software Product is uninstalled.

Figure 10-25 shows the driver when it has been defined to the DCM.

Figure 10-25 Trade_DBServer_Module_Installation_Driver Workflows

To define the Trade_DBServer_Module_Installation_Driver to the DCM during installation of the automation package, the statements shown in Example 10-15 must be present in the automation package manifest file - tc-driver.inf.

Example 10-15 XML definition of Trade_DBServer_Module_Installation_Driver

```
<device-model name="Trade_DBServer_Module_Installation_Driver"
category="Trade">
      <workflow name="Trade_DBServer_Module_Installation_Uninstall" />
</device-model>
```

And naturally, all the referenced workflows have to have been defined. This is normally taken care of automatically by the APDE platform.

The following section describes the workflows used in the Trade_DBServer_Module_Installation_Driver in greater detail.

Trade_DBServer_Module_Installation_Uninstall

The workflow used to remove an installation of the Trade DBServer Module is named Trade_DBServer_Module_Installation_Uninstall. During removal of the installation, we need to make sure that all the instances that were created based on the FOREIGN-CONFIGURATION Software Resource Template are removed as well. This is achieved by calling the SoftwareInstance.RemoveInstance LDO for each one of them. This LDO is implemented by the workflow assigned in the device driver for the Software Instance. In our case, it will be the one described in “Trade_DBServer_Module_Instance_RemoveInstance” on page 365.

The Trade_DBServer_Module_Installation_Uninstall workflow is listed in Example 10-16 and the main outline is:

1. Initialize to set common variables for simulation and error reporting.
2. Build a list of Software Instance objects that were derived from the FOREIGN-CONFIGURATION template of the Trade DBServer Module Installation.
3. For each Software Instance object, call SoftwareInstance.RemoveInstance to remove it.

Example 10-16 Trade_DBServer_Module_Installation_Uninstall

```
workflow Trade_DBServer_Module_Installation_UnInstall(in SoftwareInstallationID) implements
SoftwareInstallation.Uninstall LocaleInsensitive

#######################################################
## 1: initialize and input validation
#######################################################
var simulation
Trade_Get_Simulation(simulation)

var workflowName=”Trade_DBServer_Module_Installation_UnInstall”
var message=Jython[“<b>” + workflowName + “: </b>”]
var msg

try
  #######################################################
  ## 2: get basic information
  #######################################################
  var ServerID=DCMQuery(/softwareinstallation[@id=$SoftwareInstallationID]/@managedsystemid)
  var SoftwareModuleID=DCMQuery(/softwareinstallation[@id=$SoftwareInstallationID]/@softwaremoduleid)
  var installationResourceType=DCMQuery(/softwareresourcetype[@name=”INSTALLATION”]/@id)
  var instanceResourceType=DCMQuery(/softwareresourcetype[@name=”INSTANCE”]/@id)
```

```
  var foreignResourceType=DCMQuery(/softwareresourcetype[@name="FOREIGN-CONFIGURATION"]/@id)
  var
installationSRT=DCMQuery(/softwareinstallation[@id=$SoftwareInstallationID]/softwareresourcetemplate[@soft
wareresourcetype=$installationResourceType]/@id)
  var foreignSRT=DCMQuery(/childsoftwareresourcetemplate[@parenttemplateid=$installationSRT]/@id)
  var foreignParentResourceID=DCMQuery(/softwareresourcetemplate[@id=$foreignSRT]/softwareresource/@id)

  ##################################################################################
  ## 3: remove all instances derived  from the Instance template in foreign-configuration
  ##################################################################################
  foreach instanceID in  DCMQuery(/softwareresourcetemplate[@softwaremoduleid=$SoftwareModuleID and
@softwareresourcetype=$instanceResourceType ]/softwareresource[@managedsystemid=$ServerID and
@parentresourceid != $foreignParentResourceID]/@id) do
    log info Jython["Removing related instance: " + instanceID]
    SoftwareInstance.RemoveInstance(instanceID)
  done

catchall exception
  ######################################################################
  ## 4: handle errors
  ######################################################################
  msg = Jython[message + exception]
  log error msg
  rethrow
endtry
```

10.6.4 The Trade_DBServer_Module_Instance_Driver

The Trade_DBServer_Module_Instance_Driver device driver is associated with the instances created during the installation process through the definitions in the INSTANCE Software Resource Template.

The purpose of this device driver is to provide implementations for the logical device operations that apply to instances. These are:

SoftwareInstance.RemoveInstance
Uninstalls an instance.

SoftwareInstance.Start
Starts an instance, in our case, a DB2 Instance.

SoftwareInstance.Stop
Stops an instance.

SoftwareInstance.UpdateDCM
Used to update the information in the DCM regarding the instance. In our implementation, we will only update the operational state of the instance.

Figure 10-26 on page 357 depicts the content of the device driver and the helper and supporting workflows required to provide the desired functionality.

Figure 10-26 Trade_DBServer_Module_Instance_Driver Overview

Besides the obvious, self explanatory workflows for each logical device operation, we use two support workflows to remove the instance user and clean up credential definitions, as part of the RemoveInstance LDO. When defined to the DCM, the device driver properties should look similar to the ones shown in Figure 10-27.

Figure 10-27 Trade_DBServer_Module_Instance_Driver Workflows

Archived

To define the Trade_DBServer_Module_Instance_Driver to the DCM during installation of the automation package, the statements shown in Example 10-17 must be present in the automation package manifest file - tc-driver.inf.

Example 10-17 Defining Trade_DBServer_Module_Instance_Driver through XML

```
<device-model name="Trade_DBServer_Module_Instance_Driver"
category="Trade">
      <workflow name="Trade_DBServer_Module_Instance_RemoveInstance" />
      <workflow name="Trade_DBServer_Module_Instance_Start" />
      <workflow name="Trade_DBServer_Module_Instance_Stop" />
      <workflow name="Trade_DBServer_Module_Instance_UpdateDCM" />
</device-model>
```

And naturally, all the referenced workflows have to have been defined. This is normally taken care of automatically by the APDE platform.

The following describes the workflows used in the Trade_DBServer_Module_Instance_Driver in greater detail.

Trade_DBServer_Module_Instance_Start

This workflow implements the SoftwareInstance.Start LDO, and as the name implies, it is used to start the DB2 Instances created for Trade.

The logic of this workflow is pretty straight-forward:

1. Initialize to control simulation and set up base variables.
2. Gather basic data regarding the instance, the related Software Resource Template, parameters, and installation directory.
3. Run a scriptlet to set up the db2 environment and issue the **db2start** command.
4. Update the operational status of the instance in the DCM.

The workflow listing is shown in Example 10-18.

Example 10-18 Trade_DBServer_Module_Instance_Start

```
workflow Trade_DBServer_Module_Instance_Start(in SoftwareInstanceID) implements SoftwareInstance.Start
LocaleInsensitive

##############################################################
##  1:  Initialize
##############################################################
var simulation
Trade_Get_Simulation(simulation)

var workflowName=”Trade_DBServer_Module_Instance_Start”
```

```
var message = Jython[“<b>” + workflowName + “: </b>”]
var msg

try

  ##############################################################
  ##  2:  get base data
  ##############################################################
  var ServerID=DCMQuery(/softwareinstance[@id=$SoftwareInstanceID]/@managedsystemid)

  ## find the SRT id
  var
SoftwareResourceTemplateID=DCMQuery(/softwareinstance[@id=$SoftwareInstanceID]/softwareresourcetemplate/@i
d)

  ## Get parameters
  var db_instance_name=DCMQuery(/templateparam[@name=”resource-name” and
@templateid=$SoftwareResourceTemplateID]/@value)
  var SoftwareInstanceName=db_instance_name
  db_instance_name=Jython[db_instance_name.upper()]

  ##  Get install_dir
  var install_dir_parmName=”install_dir”
  var install_dir=DCMQuery(/templateparam[@templateid=$SoftwareResourceTemplateID and
@name=$install_dir_parmName]/@value)
  if Jython[not install_dir] then
    msg = Jython[message + “ could not find value for install_dir”]
    throw TradeParameterException msg
  endif

  ##############################################################
  ##  3:  run the scriptlet to start the instance
  ##############################################################
  if Jython[simulation.lower() == “yes”] then
    log info Jython[“<b>simulation:</b>...executing scriptlet to start the database”]
  else
    scriptlet ( install_dir, db_instance_name) language=bash target=DCMQuery(/server[@id=$ServerID]/@id)
<<EOFDB2
      [ -z $OSTYPE ] && {
        OSTYPE=‘uname -s‘
      }
      case $OSTYPE in
        cygwin*|CYGWIN*)
          DB2CommandLocation=‘cygpath -u “${install_dir}/bin”‘
        ;;
        AIX*|HP*|hpux*|Linux*|Solaris*|linux*)
          # todo - test this on a real system
          DB2CommandLocation=${install_dir}/bin
        ;;
        *)
          TIOthrow ScriptExitException “${OSTYPE} - Unsupported Operating System”
        ;;
      esac
```

```
      function db2_doit
      {
        command=${DB2CommandLocation}/$1
        TIOlog “info” “About to execute ‘${command}’”
        msg=‘${command}‘
        rc=$?
        TIOlog “info” “(${rc}) ${msg}”
        if [ $((rc)) -gt $((accept_return_code)) ]; then
          TIOthrow “db2CreateInstanceException” “(${rc}) ${command}: ${msg}”
        fi
        return ${rc}
      }

      accept_return_code=0
      inst=‘db2ilist | grep -i ${db_instance_name}‘
      if [ “${inst}” == “${db_instance_name}” ]; then
        export DB2INSTANCE=${db_instance_name}
        accept_return_code=1
        db2_doit “db2start”
      else
        message=”db2 instance ${db_instance_name} does not exist”
        TIOlog “error” ${message}
        TIOthrow “db2InstanceOperateException” “${message}”
      fi
      return 0
    EOFDB2
  endif
catchall exception
  ##############################################################
  ##  4:  handle errors
  ##############################################################
  msg=Jython[message + exception]
  log error msg
  rethrow
endtry

##############################################################
##  4:  Update the DCM Status
##############################################################
# Get DCM status
var status = DCMQuery(/softwareresource[@id=$SoftwareInstanceID]/softwarestate/@name)
var realstatus=”Running”
if Jython[status != realstatus] then

java:com.ibm.tivoli.orchestrator.datacentermodel.helper.SoftwareHelper#updateResourceStatus(SoftwareInstan
ceID, realstatus)
endif
```

Archived

Trade_DBServer_Module_Instance_Stop

This workflow is almost identical to the one described in "Trade_DBServer_Module_Instance_Start" on page 358. The only difference is that the **db2stop** command is issued, in order to stop the instance.

Example 10-19 lists the workflow.

Example 10-19 Trade_DBServer_Module_Instance_Stop

```
workflow Trade_DBServer_Module_Instance_Stop(in SoftwareInstanceID) implements SoftwareInstance.Stop
LocaleInsensitive

##############################################################
##  1:  Initialize
##############################################################
var simulation
Trade_Get_Simulation(simulation)

var workflowName="Trade_DBServer_Module_Instance_Stop"
var message = Jython["<b>" + workflowName + ": </b>"]
var msg

try

  ##############################################################
  ##  2:  get base data
  ##############################################################
  var ServerID=DCMQuery(/softwareinstance[@id=$SoftwareInstanceID]/@managedsystemid)

  ## find the SRT id
  var
SoftwareResourceTemplateID=DCMQuery(/softwareinstance[@id=$SoftwareInstanceID]/softwareresourcetemplate/@i
d)

  ## Get parameters
  var db_instance_name=DCMQuery(/templateparam[@name="resource-name" and
@templateid=$SoftwareResourceTemplateID]/@value)
  db_instance_name=Jython[db_instance_name.upper()]

  ##  Get install_dir
  var install_dir_parmName="install_dir"
  var install_dir=DCMQuery(/templateparam[@templateid=$SoftwareResourceTemplateID and
@name=$install_dir_parmName]/@value)
  if Jython[not install_dir] then
    msg = Jython[message + " could not find value for install_dir"]
    throw TradeParameterException msg
  endif


  ##############################################################
  ##  3:  execute scriptlet to stop the database instance
  ##############################################################
  if Jython[simulation.lower() == "yes"] then
```

```
    log info Jython["<b>simulation:</b>...executing scriptlet to stop the database"]
  else
    scriptlet ( install_dir, db_instance_name) language=bash target=DCMQuery(/server[@id=$ServerID]/@id)
<<EOFDB2
      [ -z $OSTYPE ] && {
        OSTYPE=`uname -s`
      }
      case $OSTYPE in
        cygwin*|CYGWIN*)
          DB2CommandLocation=`cygpath -u "${install_dir}/bin"`
        ;;
        AIX*|HP*|hpux*|Linux*|Solaris*|linux*)
          # todo - test this on a real system
          DB2CommandLocation=${install_dir}/bin
        ;;
        *)
          TIOthrow ScriptExitException "${OSTYPE} - Unsupported Operating System"
        ;;
      esac

      function db2_doit
      {
        command=${DB2CommandLocation}/$1
        TIOlog "info" "About to execute '${command}'"
        msg=`${command}`
        rc=$?
        TIOlog "info" "(${rc}) ${msg}"
        if [ $((rc)) -gt $((accept_return_code)) ]; then
          TIOthrow "db2CreateInstanceException" "(${rc}) ${command}: ${msg}"
        fi
        return ${rc}
      }

      # stop existing instance if it exists
      accept_return_code=0
      inst=`db2ilist | grep -i ${db_instance_name}`
      if [ "${inst}" == "${db_instance_name}" ]; then
        export DB2INSTANCE=${db_instance_name}
        accept_return_code=1
        db2_doit "db2stop force"
      else
        message="db2 instance ${db_instance_name} does not exist"
        TIOlog "error" ${message}
        TIOthrow "db2InstanceOperateException" "${message}"
      fi
      return 0
    EOFDB2
  endif

catchall exception
  ################################################################
  ##  4:  Error handling
  ################################################################
  msg=Jython[message +  exception]
```

```
  log error msg
  rethrow
endtry

##############################################################
##  5:  Update the DCM Status
##############################################################
# Get DCM status
var status = DCMQuery(/softwareresource[@id=$SoftwareInstanceID]/softwarestate/@name)
var realstatus="Not-Running"
if Jython[status != realstatus] then

java:com.ibm.tivoli.orchestrator.datacentermodel.helper.SoftwareHelper#updateResourceStatus(SoftwareInstan
ceID, realstatus)
endif
```

Trade_DBServer_Module_Instance_UpdateDCM

The SoftwareInstance.UpdateDCM LDO is intended to be used to update the data center model with run time information regarding the instance. In the workflow listed in Example 10-20, demonstrates how to query the operational status of the trade3 DB2 instance and report it back the DCM.

The main functionality is:

1. Initialize simulation and error messages.
2. Get basic information about related objects, credentials, location of executables, and so on.
3. Execute a scriptlet that returns the operational state of the instance in a variable accessible from the workflow using the TIOsetVar function.
4. Update the status information in the DCM.

Example 10-20 Trade_DBServer_Module_Instance_UpdateDCM

```
workflow Trade_DBServer_Module_Resource_UpdateDCM(in SoftwareResourceID) implements
SoftwareResource.UpdateDCM LocaleInsensitive

##############################################################
##  1:  initialize
##############################################################
var simulation
Trade_Get_Simulation(simulation)

var workflowName="Trade_DBServer_Module_Resource_UpdateDCM"
var message = Jython["<b>" + workflowName + ": </b>"]
var msg

var realstatus="UNKNOWN"

try
```

```
  ##############################################################
  ##  2:  get base data
  ##############################################################
  var ServerID=DCMQuery(/softwareresource[@id=$SoftwareResourceID]/@managedsystemid)

  # find the SRT id
  var
SoftwareResourceTemplateID=DCMQuery(/softwareinstance[@id=$SoftwareResourceID]/softwareresourcetemplate/@i
d)

  # Get parameters
  var db_instance_name=DCMQuery(/templateparam[@name="resource-name" and
@templateid=$SoftwareResourceTemplateID]/@value)
  var Windows_ServiceName=Jython["DB2 - " + db_instance_name.upper()]
  Windows_ServiceName=Jython[db_instance_name.upper()]

  ####################################################################
  ##  3:  execute scriptlet to query the status of the database instance
  ####################################################################
  if Jython[simulation.lower() == "yes"] then
    log info Jython["<b>simulation:</b>...executing scriptlet to query the status of the database"]
  else
    scriptlet (Windows_ServiceName) language=bash  target=DCMQuery(/server[@id=$ServerID]/@id) timeout=60
<<EOFNET
      [ -z $OSTYPE ] && {
        OSTYPE=`uname -s`
      }
      case $OSTYPE in
        cygwin*|CYGWIN*)
        ;;
       AIX*|HP*|hpux*|Linux*|Solaris*|linux*)
          TIOthrow ScriptExitException "${OSTYPE} - Only Windows/Cygwin is supported"
       ;;
       *)
          TIOthrow ScriptExitException "${OSTYPE} - Unsupported Operating System"
       ;;
      esac

      command="cmd /c net start | grep -c '${Windows_ServiceName}'"

      TIOlog "info" "About to execute: ${command}  "
      msg=`${command}`
      rc=$?
      TIOlog "info" "(${rc}) '${msg}' "
      if [ $((msg)) -gt 0 ]; then
        status="Running"
      else
        status="Not-Running"
      fi

      TIOsetVar "realstatus" ${status}

    EOFNET
  endif
```

```
catchall exception
  ######################################################################
  ##  4: handle errors
  ######################################################################
  msg=Jython[mesage + exception]
  log error msg
  rethrow
endtry

#################################################################
##  4:  update the DCM status
#################################################################
# Get DCM status
var status = DCMQuery(/softwareresource[@id=$SoftwareResourceID]/softwarestate/@name)
log debug Jython[“known status: “ + status + “        real status: “ + realstatus]

if Jython[status != realstatus] then

java:com.ibm.tivoli.orchestrator.datacentermodel.helper.SoftwareHelper#updateResourceStatus(SoftwareResour
ceID, realstatus)
endif
```

Trade_DBServer_Module_Instance_RemoveInstance

The Trade_DBServer_Module_Instance_RemoveInstance workflow is used to remove the DB2 instances created during the installation of the Trade DBServer Module. During execution, this workflow calls the Trade_DBServer_Delete_Database workflow to make sure that the databases are removed before the instance is deleted.

The main functionality of the Trade_DBServer_Module_Instance_RemoveInstance workflow is:

1. Initialize to set up error handling and simulation.
2. Retrieval of basic information to identify resources related to the SoftwareInstance, credentials, and installation directory information to locate the DB2 executables.
3. For each database hosted by the instance, call the Trade_DBServer_Delete_Database workflow to drop it. The workflow for database deletion is described in more detail in “Trade_DBServer_Delete_Database” on page 384.
4. Once all databases are dropped, remove the instance using a scriptlet.
5. Now, to clean up, we delete the instance user (by calling ITSO_Destroy_User, which is described in 10.10.7, “ITSO_Destroy_User” on page 508).
6. In addition to removing the user, we also have to make sure that the cygwin home-directory of the user will be removed as well.

Archived

7. Finally, we have to remove the credentials that were set up for the instance user during installation. The ITSO_Delete_Credentials workflow is called to take care of this task, please refer to 10.10.5, “ITSO_Delete_Credentials” on page 504 for more information regarding the deletion of credentials.

Example 10-21 shows the entire contents of the Trade_DBServer_Module_Instance_RemoveInstance workflow.

Example 10-21 Trade_DBServer_Module_Instance_RemoveInstance

```
workflow Trade_DBServer_Module_Instance_RemoveInstance(in SoftwareInstanceID) implements
SoftwareInstance.RemoveInstance LocaleInsensitive

##############################################################
##  1:  initialize
##############################################################

var simulation
Trade_Get_Simulation(simulation)

var workflowName=”Trade_DBServer_Module_Instance_RemoveInstance”
var message = Jython[“<b>” + workflowName + “: </b>”]
var msg

try
  ##############################################################
  ##  2:  get base data
  ##############################################################
  ## find the SRT id
  var
SoftwareResourceTemplateID=DCMQuery(/softwareinstance[@id=$SoftwareInstanceID]/softwareresourcetemplate/@i
d)
  var ServerID=DCMQuery(/softwareinstance[@id=$SoftwareInstanceID]/@managedsystemid)

  # get the configuration template ID
  var
configurationSRT=DCMQuery(/childsoftwareresourcetemplate[@parenttemplateid=$SoftwareResourceTemplateID]/@i
d)

  ## get dbuser parameters
  var dbuser_sap_searchkey=DCMQuery(/templateparam[@name=”trade_dbuser_SAP_searchkey” and
@templateid=$configurationSRT]/@value)
  var dbuser_sap_id
  var dbuser_credentials_id
  var dbuser_userid
  var dbuser_password
  java:com.thinkdynamics.kanaha.de.javaplugin.datacentermodel.FindDefaultSAP(ServerID, “execute-command”,
dbuser_sap_id)
  java:com.thinkdynamics.kanaha.de.javaplugin.sap.GetSapCredentials(dbuser_credentials_id,
dbuser_sap_searchkey, dbuser_sap_id)
  java:com.thinkdynamics.kanaha.de.javaplugin.sap.FindCredentialsPassword(dbuser_credentials_id, <null>,
<null>, dbuser_password, dbuser_userid)
```

```
  # get database instance user parameters
  var db_instance_name=DCMQuery(/templateparam[@name="resource-name" and
@templateid=$SoftwareResourceTemplateID]/@value)
  db_instance_name=Jython[db_instance_name.upper()]

  # get the instance user credentials
  var dbinstance_sap_searchkey=DCMQuery(/templateparam[@name="trade_dbinstance_SAP_searchkey" and
@templateid=$SoftwareResourceTemplateID]/@value)
  var dbinstance_sap_id
  var dbinstance_credentials_id
  var dbinstance_userid
  var dbinstance_password
  java:com.thinkdynamics.kanaha.de.javaplugin.datacentermodel.FindDefaultSAP(ServerID, "execute-command",
dbinstance_sap_id)
  java:com.thinkdynamics.kanaha.de.javaplugin.sap.GetSapCredentials(dbinstance_credentials_id,
dbinstance_sap_searchkey, dbinstance_sap_id)
  java:com.thinkdynamics.kanaha.de.javaplugin.sap.FindCredentialsPassword(dbinstance_credentials_id,
<null>, <null>, dbinstance_password, dbinstance_userid)

  # get the dbadmin searchkey and userid
  var parentTemplateID
  parentTemplateID=DCMQuery(/softwareresourcetemplate[@id=$SoftwareResourceTemplateID]/@parenttemplateid)
  if Jython[parentTemplateID == None] then
    # get the id of the SRT from which this one was cloned
    var
sourceTemplateID=DCMQuery(/softwareresourcetemplate[@id=$SoftwareResourceTemplateID]/@sourcetemplateid)
    # find the parent two levels up  INSTANCE -> FOREIGN-CONFIGURATION -> INSTALLATION

parentTemplateID=DCMQuery(/softwareresourcetemplate[@id=$sourceTemplateID]/parentsoftwareresourcetemplate/
@parenttemplateid)
  endif
  log info Jython["ParentTemplateID: " + parentTemplateID]

  var installationSRT=parentTemplateID
  var dbadmin_sap_searchkey=DCMQuery(/templateparam[@name="trade_dbadmin_SAP_searchkey" and
@templateid=$installationSRT]/@value)
  var dbadmin_sap_id
  var dbadmin_credentials_id
  var dbadmin_userid
  var dbadmin_password
  java:com.thinkdynamics.kanaha.de.javaplugin.datacentermodel.FindDefaultSAP(ServerID, "execute-command",
dbadmin_sap_id)
  java:com.thinkdynamics.kanaha.de.javaplugin.sap.GetSapCredentials(dbadmin_credentials_id,
dbadmin_sap_searchkey, dbadmin_sap_id)
  java:com.thinkdynamics.kanaha.de.javaplugin.sap.FindCredentialsPassword(dbadmin_credentials_id, <null>,
<null>, dbadmin_password, dbadmin_userid)

  #  get install_dir
  var install_dir_parmName="install_dir"
  var instanceTemplateID=DCMQuery(/softwareinstance[@id=$SoftwareInstanceID]/@resourcetemplateid)
  var install_dir=DCMQuery(/templateparam[@templateid=$instanceTemplateID and
@name=$install_dir_parmName]/@value)
  if Jython[not install_dir] then
    msg = Jython[message + " could not find value for install_dir"]
```

```
    throw TradeParameterException msg
  endif

  ##############################################################
  ##  3:  drop the databases
  ##############################################################
  try
    var ConfigurationTypeID=DCMQuery(/dcmobjecttype[@name="SOFTWARE_CONFIGURATION"]/@id)
    log debug Jython["Dropping databases from " + SoftwareInstanceID]
    foreach db in DCMQuery(/softwareresource[@objecttypeid=$ConfigurationTypeID and
@parentresourceid=$SoftwareInstanceID]/@id) do
      var dbname=DCMQuery(/softwareresource[@id=$db]/@name)
      log info Jython["... Dropping database : " + dbname]
      Trade_DBServer_Delete_Database(db)
    done
  catchall exception
    # ignore errors
    noop
  endtry

  ##############################################################
  ##  5:  remove the database user userID
  ##############################################################
  if Jython[dbuser_userid != dbadmin_userid and dbuser_userid.lower() != "db2admin"] then
    try
      if Jython[simulation.lower() == "yes"] then
        log info Jython["<b>simulation:</b> ...calling ITSO_Destroy_User(" + ServerID + ", " +
dbuser_userid + ")"]
      else
        ITSO_Destroy_User(ServerID, dbuser_userid)
      endif
    catchall
      #ignore errors
      noop
    endtry
  endif

  ##############################################################
  ##  6:  remove the database instance
  ##############################################################
  if Jython[simulation.lower() == "yes"] then
    log info Jython["<b>simulation:</b> ...calling scriptlet to drop the instance "]
    log info Jython["     install_dir: " + install_dir]
    log info Jython["     db_instance_name: " + db_instance_name]
  else
    scriptlet ( install_dir, db_instance_name) language=bash target=DCMQuery(/server[@id=$ServerID]/@id)
credentialskey=dbadmin_sap_searchkey <<EOFDB2
      [ -z $OSTYPE ] && {
         OSTYPE=`uname -s`
      }
      case $OSTYPE in
        cygwin*|CYGWIN*)
           DB2CommandLocation=`cygpath -u "${install_dir}/bin"`
        ;;
```

```
      AIX*|HP*|hpux*|Linux*|Solaris*|linux*)
          # todo - test this on a real system
          DB2CommandLocation=${install_dir}/bin
      ;;
      *)
          TIOthrow ScriptExitException "${OSTYPE} - Unsupported Operating System"
      ;;
    esac

    function db2_doit
    {
      command=${DB2CommandLocation}/$1
      TIOlog "info" "About to execute '${command}'"
      msg='${command}'
      rc=$?
      TIOlog "info" "(${rc}) ${msg}"
      if [ $((rc)) -gt $((accept_return_code)) ]; then
        TIOthrow "db2CreateInstanceException" "(${rc}) ${command}: ${msg}"
      fi
      return ${rc}
    }

    # delete existing user if it exists
    accept_return_code=0
    inst=`db2ilist | grep -i ${db_instance_name}`
    if [ ${inst} == ${db_instance_name} ]; then
      export DB2INSTANCE=${db_instance_name}
      accept_return_code=1
      db2_doit "db2stop force"
      accept_return_code=4
      db2_doit "db2idrop ${db_instance_name}"
    else
      TIOlog "warning" "db2 instance ${db_instance_name} does not exist"
    fi

    return 0
  EOFDB2
endif

##############################################################
##  7:  remove the instance user
##############################################################
if Jython[dbinstance_userid != dbadmin_userid and dbinstance_userid.lower() != "db2admin"] then
  try
    if Jython[simulation.lower() == "yes"] then
      log info Jython["<b>simulation:</b> ...calling ITSO_Destroy_User(" + ServerID + ", " +
dbinstance_userid + ")"]
    else
      ITSO_Destroy_User(ServerID, dbinstance_userid)
    endif
  catchall
    #ignore errors
    noop
  endtry
```

```
  endif

  ##############################################################
  ##  8:  remove credentials from the server
  ##############################################################
  ITSO_Delete_Credentials(ServerID, dbinstance_sap_searchkey)
catchall exception
  ##############################################################
  ##  9:  error handling
  ##############################################################
  msg=Jython[message + exception]
  log error msg
  rethrow
endtry
```

10.6.5 Trade DBServer Module Helper workflows

Besides the four workflows implementing logical device operations for the Trade DBServer Module related device drivers, we need a couple of helper workflows to facilitate the creation of the DB2 instance, as well as the creation and removal of the database for the Trade application:

- Trade_DBServer_Add_Instance
- Trade_DBServer_Create_Database
- Trade_DBServer_Delete_Database

These are all described in the following sections.

Trade_DBServer_Add_Instance

At last we have reached the point where we actually start working with the DB2 Server installation and create instances and databases. As the name indicates, the Trade_DBServer_Add_Instance workflow creates a new DB2 Instance, and then calls Trade_DBServer_Create_Database for creation of the database to be used for hosting data for the Trade Application.

The main functionality of the Trade_DBServer_Add_Instance workflow is:

1. As usual, we start by initializing the workflow, validate input, and gather basic information required by the workflow. In this case, the input validation is omitted.
2. Then we get the parameters for configuring the DB2 Instance from the INSTANCE Software Resource Template.
3. To find the DB2 executables on the target server, we also need to find the location of the installation directory using the Trade_Get_Hosting_SRTParameter_Or_Default workflow. When the data is

retrieved, we store the value in the Software Resource Template for the new Instance parented by the DB2 Server Installation.

4. Now it is time to find the credentials to be used for the DB2 instance user. The keys to look for in the trade SAP are found in the Instance Software Resource Template, and the user ID and password are read from the SAP using Java plug-ins.
5. Before we execute the scriptlet to create the instance, we have to ensure that compatible credentials (using the protocol and searchkeys as keys) exist between the target system and the TIO/TPM Server. For scriptlet execution, we need to support both the file-transfer and execute-command.
6. Before creating the DB2 Instance, we also have to ensure that the instance user has been created and has a valid home directory defined in the cygwin environment. This task is taken care of by the ITSO_Create_User workflow, which is described in detail in 10.10.6, “ITSO_Create_User” on page 505.
7. Finally, we can use a scriptlet to create the instance using the dbadmin credentials. The scriptlet itself is a standard bash script that accepts parameters from the workflow.
8. When the instance has been created, we can call the Trade_DBServer_Create_Database workflow to create the trade3db database and the table schema needed by the Trade application. The database creation workflow is described in “Trade_DBServer_Create_Database” on page 377.
9. The last step is to start the instance to ensure that it is operational. This is handled through the SofteareInstance.Start logical device operation, for the instance. By associating the Trade_DBServer_Module_Instance_Driver to the instance (referenced in our Software Resource Templates for the Trade DBServer Module Software Product), we ensure that our workflow Trade_DBServer_Module_Instance_Start is the one that implements the SoftwareInstance.Start LDO. Please refer to “Trade_DBServer_Module_Instance_Start” on page 358.

Archived

Example 10-22 shows the full listing of the Trade_DBServer_Add_Instance workflow.

Example 10-22 Trade_DBServer_Add_Instance

```
workflow Trade_DBServer_Add_Instance(in SoftwareInstallationID, in SoftwareResourceTemplateID)
LocaleInsensitive

#####################################
##  1:  initialize and validate imput
#####################################
var simulation
Trade_Get_Simulation(simulation)

var workflowName="Trade_DBServer_Add_Instance"
var message=Jython["<b>" + workflowName + ": </b>"]
var msg

var ServerID=DCMQuery(/softwareinstallation[@id=$SoftwareInstallationID]/@managedsystemid)
var
SoftwareModuleID=DCMQuery(/softwareresourcetemplate[@id=$SoftwareResourceTemplateID]/@softwaremoduleid)
var InstanceID=DCMQuery(/softwareresourcetemplate[@id=$SoftwareResourceTemplateID]/softwareresource/@id)

#######################################################################
##  2: Get the parameters from the INSTANCE software resource template
#######################################################################
try
  var
db2_type=DCMQuery(/softwareresourcetemplate[@id=$SoftwareResourceTemplateID]/templateparam[@name="db2type"
and @templateid=$SoftwareResourceTemplateID]/@value)
  var
db_instance_name=DCMQuery(/softwareresourcetemplate[@id=$SoftwareResourceTemplateID]/templateparam[@name="
resource-name" and @templateid=$SoftwareResourceTemplateID]/@value)
  db_instance_name=Jython[db_instance_name.upper()]
  var
db_instance_port=DCMQuery(/softwareresourcetemplate[@id=$SoftwareResourceTemplateID]/templateparam[@name="
port" and @templateid=$SoftwareResourceTemplateID]/@value)

  ##########################################################################
  ##  3: Get and save the installation directory from the hosting Installation
  ##########################################################################
  var install_dir_parmName="install_dir"
  var install_dir

  var sourceModuleID
  foreach id in
DCMQuery(/softwareinstance[@parentresourceid=$SoftwareInstallationID]/softwareresourcetemplate/sourcesoftw
areresourcetemplate/softwaremodule/@id) do
    # get the first one
    sourceModuleID=id
    break
  done
  var sourceInstallationID
```

```
  foreach id in DCMQuery(/softwareinstallation[@managedsystemid=$ServerID and
@softwaremoduleid=$sourceModuleID]/@id) do
    # get the first one
    sourceInstallationID=id
    break
  done

  var sourceInstallationSRT=DCMQuery(/softwareinstallation[@id=$sourceInstallationID]/@resourcetemplateid)
  var sourceInstallableID=DCMQuery(/softwareinstallation[@id=$sourceInstallationID]/@softwareproductid)
  Trade_Get_Hosting_SRTParameter_Or_Default(sourceInstallableID, SoftwareInstallationID,
install_dir_parmName, install_dir)
  if Jython[not install_dir] then
    msg = Jython[message + " could not find value for install_dir"]
    throw TradeParameterException msg
  endif

  ##  Insert the value for installation directory in the SRT for reuse
  DCMInsert parent=DCMQuery(/softwareresourcetemplate[@id=$SoftwareResourceTemplateID]/@id) <<EOFXML
     <template-param name="install_dir" value="$install_dir" is-changeable="true" />
  EOFXML

  ######################################################################
  ##  4: Get the crdentials for the instance user
  ######################################################################
  var dbinstance_sap_applicationprotocoltype=DCMQuery(/applicationprotocol[@name="UNKNOWN"]/@id)
  var dbinstance_sap_protocoltype=DCMQuery(/protocoltype[@name="unknown"]/@id)
  var dbinstance_sap_domain=DCMQuery(/templateparam[@name="trade_dbinstance_SAP_domain" and
@templateid=$SoftwareResourceTemplateID]/@value)
  var dbinstance_sap_context=DCMQuery(/templateparam[@name="trade_dbinstance_SAP_context" and
@templateid=$SoftwareResourceTemplateID]/@value)
  var dbinstance_sap_searchkey=DCMQuery(/templateparam[@name="trade_dbinstance_SAP_searchkey" and
@templateid=$SoftwareResourceTemplateID]/@value)
  var dbinstance_sap_id=DCMQuery(/sap[@domain=$dbinstance_sap_domain and @context=$dbinstance_sap_context
and @protocoltypeid=$dbinstance_sap_protocoltype and
@appprotocolid=$dbinstance_sap_applicationprotocoltype]/@id)

  if Jython[not dbinstance_sap_id or dbinstance_sap_id=""] then
    msg = Jython[message + "Could not find a valid SAP for:  protocol:" +
dbinstance_sap_applicationprotocoltype + "/" + dbinstance_sap_protocoltype + "  domain:" +
dbinstance_sap_domain + " context:" + dbinstance_sap_context]
    log error msg
    throw SAPexception msg
  else
    log info Jython["<b>... finding credentials for searchkey: " + dbinstance_sap_searchkey + " in SAP: "
+ dbinstance_sap_id + "</b>"]
  endif

  var dbinstance_credentials_id

  try
    java:com.thinkdynamics.kanaha.de.javaplugin.sap.GetSapCredentials(dbinstance_credentials_id,
dbinstance_sap_searchkey, dbinstance_sap_id)
    log info Jython[dbinstance_credentials_id or "UNKNOWN"]
  catchall
```

```
    msg = Jython[message + " could not find the credential for searchkey " + dbinstance_sap_searchkey + "
in SAP " + dbinstance_sap_id]
    rethrow
  endtry

  var dbinstance_userid
  var dbinstance_password
  java:com.thinkdynamics.kanaha.de.javaplugin.sap.FindCredentialsPassword(dbinstance_credentials_id,
<null>, <null>, dbinstance_password, dbinstance_userid)

  ##########################################################################################
  ##  5: Ensure that the compatible SAPs have been defined for the instance user searchkey
  ##########################################################################################
  var DE_Device_ID
  java:com.thinkdynamics.kanaha.de.javaplugin.datacentermodel.FindDEDeviceId(DE_Device_ID)
  ITSO_Synchronize_Credentials (dbinstance_credentials_id, ServerID, DE_Device_ID)

  ##################################################################
  ## 6: Ensure that the instanceuser has been created
  ##################################################################
  log info Jython["<b> Creating user " + dbinstance_userid + "</b>"]
  array groups
  var i=arraysize(groups)
  while Jython[ int(i) >= 0 ] do
    var j = arraysize(groups)
    i=Jython[int(j) + 1]
    var groupParamName = Jython["trade_dbinstance_user_group." + i]
    var groupName=DCMQuery(/templateparam[@name=$groupParamName and
@templateid=$SoftwareResourceTemplateID]/@value)
    if Jython[groupName] then
      groups[j] = groupName
    else
      break
    endif
  done

  try
    if Jython[simulation.lower() == "yes"] then
      log info Jython["<b>simulation:</b> About to call : ITSO_Create_User(" + ServerID + ", " +
dbinstance_userid + ", " + dbinstance_password + ", " + groups + ")"]
    else
      ITSO_Create_User(ServerID, dbinstance_userid, dbinstance_password, groups)
    endif
  catchall exception
    log warning Jython["<b>......got exception " + exception + "</b>"]
  endtry

  ##################################################
  ## 7: Create instance as dbadmin
  ##################################################
  var
dbadmin_sap_searchkey=DCMQuery(/softwareresourcetemplate[@id=$sourceInstallationSRT]/templateparam[@name="
trade_dbadmin_SAP_searchkey"]/@value)
```

```
  log info Jython[“<b>Creating Instance “ + db_instance_name + “ on port “ + db_instance_port + “ using
search-key:” + dbadmin_sap_searchkey + “</b>”]

  if Jython[simulation.lower() == “yes”] then
    log info Jython[“<b>simulation:</b> ... calling scriptlet to create the instance”]
    log info Jython[“       install_dir: “ + install_dir]
    log info Jython[“       db_instance_name: “ + db_instance_name]
    log info Jython[“       db_instance_port: “ + db_instance_port]
    log info Jython[“       db2_type: “ + db2_type]
    log info Jython[“       dbinstance_userid: “ + dbinstance_userid]
    log info Jython[“       dbinstance_password: “ + dbinstance_password]
  else
    scriptlet ( install_dir, db_instance_name, db_instance_port, db2_type, dbinstance_userid,
dbinstance_password)  language=bash target=DCMQuery(/server[@id=$ServerID]/@id)
credentialskey=dbadmin_sap_searchkey <<EOFDB2
      [ -z $OSTYPE ] && {
        OSTYPE=‘uname -s‘
      }
      case $OSTYPE in
        cygwin*|CYGWIN*)
          DB2CommandLocation=‘cygpath -u “${install_dir}/bin”‘
        ;;
        AIX*|HP*|hpux*|Linux*|Solaris*|linux*)
          # todo - test this on a real system
          DB2CommandLocation=${install_dir}/bin
        ;;
        *)
          TIOthrow ScriptExitException “${OSTYPE} - Unsupported Operating System”
        ;;
      esac

      function db2_doit
      {
        command=${DB2CommandLocation}/$1
        TIOlog “info” “About to execute ‘${command}’”
        msg=‘${command}‘
        rc=$?
        TIOlog “info” “(${rc}) ${msg}”
        if [ $((rc)) -gt $((accept_return_code)) ]; then
          TIOthrow “db2CreateInstanceException” “(${rc}) ${command}: ${msg}”
        fi
        return ${rc}
      }

      export DB2INSTANCE=${db_instance_name}

      # delete existing instance if it exists
      accept_return_code=0
      inst=‘db2ilist | grep -i ${db_instance_name}‘
      if [ “${inst}” == “${db_instance_name}” ]; then
        echo “db2instance is: ${DB2INSTANCE}”
        accept_return_code=1
        db2_doit “db2stop force”
        ret=$?; if [ $ret -ne 0 ]; then TIOlog “warning” “db2stop force returned $ret”; return 0; fi
```

Archived

```
        accept_return_code=0
        db2_doit “db2idrop ${db_instance_name}”
        ret=$?; if [ $ret -ne 0 ]; then TIOlog “warning” “db2idrop returned $ret”; return 0; fi
      fi

      # create the new instance
      db2_doit “db2icrt -s ${db2_type} -u ${dbinstance_userid},${dbinstance_password} -r
${db_instance_port},${db_instance_port} ${db_instance_name}”
      ret=$?; if [ $ret -ne 0 ]; then TIOlog “warning” “db2icrt returned $ret”; return 0; fi
      db2_doit “db2set DB2COMM=TCPIP -i ${db_instance_name}”
      ret=$?; if [ $ret -ne 0 ]; then TIOlog “warning” “db2set DB2COMM=TCPIP returned $ret”; return 0; fi
      db2_doit “db2cmd -i -w -c db2 update database manager configuration using SVCENAME
${db_instance_port}”
      ret=$?; if [ $ret -ne 0 ]; then TIOlog “warning” “db2 update database manager configuration returned
$ret”; return 0; fi
      export DB2INSTANCE=${db_instance_name}
      db2_doit “db2start”
      ret=$?; if [ $ret -ne 0 ]; then TIOlog “warning” “db2 start returned $ret”; return 0; fi

      TIOlog “info” “Instance creation completed sucessfully $ret”
      return 0

    EOFDB2
  endif
catchall exception
  ######################################################
  ## 8: Error handling
  ######################################################
  log warning Jython[“<b>......IGNORING exception ‘” + exception + “‘</b>”]
  # errors are may occur because of illegal escape charcters in DB2 output
  # ignore
  #rethrow
endtry

#####################################################
## 9: Create the database and tables
#####################################################
log info Jython[“<b>Creating database tables</b>”]
try
  log info Jython[“About to call Trade_DBServer_Create_Database(“ + InstanceID+ “)”]
  Trade_DBServer_Create_Database(InstanceID)
catchall exception
  log error Jython[“<b>......got exception ‘” + exception + “‘</b>”]
  rethrow
endtry

#####################################################
## 10: Start the instance
#####################################################
log info Jython[“<b>Starting the database instance</b>”]
try
  if Jython[InstanceID] then
      SoftwareInstance.Start(InstanceID)
  endif
```

```
catchall exception
  log error Jython[“<b>......got exception “ + exception + “</b>”]
  rethrow
endtry
```

Trade_DBServer_Create_Database

The workflow used to create the DB2 database for the Trade application is very similar the one used to create the instance, the main differences being:

- The functionality of the scriptlet that executes the db2 commands
- The fact that we have to use the dbinstance credentials to access the target system,
- Prior to execution, we need to copy the Table.ddl file from the File Repository to the target system

The main steps in this workflow are:

1. Initialize and gather basic data for processing and error handling.
2. Copy the Table.ddl file from the file repository to the target system using the SoftwareInstallable.Distribute logical device operation.
3. Get the database creation parameters from the CONFIGURATION Software Resource Template.
4. Get the credentials for the instance user in order to execute the scriptlet as a user with the required authorizations.
5. Execute the database creation scriptlet.
6. Remove the Table.ddl file from the target system.

In Example 10-23, you will find a complete listing of the Trade_DBServer_Create_Database workflow.

Example 10-23 Trade_DBServer_Create_Database

```
workflow Trade_DBServer_Create_Database(SoftwareInstanceID)  LocaleInsensitive
#######################################################
## 1: initialize and gather data
#######################################################
var simulation
Trade_Get_Simulation(simulation)

var workflowName=”Trade_DB2erver_Create_Database”
var message = Jython[“<b> “ + workflowName + “</b>: “]
var msg

var DeploymentRequestID
java:com.thinkdynamics.kanaha.de.javaplugin.GetCurrentDeploymentRequestId(DeploymentRequestID)
var ServerID=DCMQuery(/softwareinstance[@id=$SoftwareInstanceID]/@managedsystemid)
```

Archived

```
try

  ######################################################
  ## 2: get information for control and parameter setup
  ######################################################
  var instanceTemplateID=DCMQuery(/softwareinstance[@id=$SoftwareInstanceID]/@resourcetemplateid)
  var sourceModuleID
  foreach id in
DCMQuery(/softwareresourcetemplate[@id=$instanceTemplateID]/sourcesoftwareresourcetemplate/softwaremodule/
@id) do
    # get the first one
    sourceModuleID=id
    break
  done
  if Jython[sourceModuleID == None] then
    msg = Jython[message + " could not find the source Software Module for instance " +
SoftwareInstanceID]
    throw parameterException msg
  endif

  var sourceInstallableID
  foreach id in DCMQuery(/softwareinstallation[@managedsystemid=$ServerID and
@softwaremoduleid=$sourceModuleID]/softwareinstallable/@id) do
    # get the first one
    sourceInstallableID=id
    break
  done
  if Jython[sourceInstallableID == None] then
    msg = Jython[message + " could not find the source Software Installable for module " + sourceModuleID
+ " on " + ServerID]
    throw parameterException msg
  endif

  ##  Get install_dir
  var install_dir_parmName="install_dir"
  var install_dir=DCMQuery(/templateparam[@templateid=$instanceTemplateID and
@name=$install_dir_parmName]/@value)
  if Jython[not install_dir] then
    msg = Jython[message + " could not find value for install_dir"]
    throw TradeParameterException msg
  endif

  var DestinationPath = Jython("/tmp/" + DeploymentRequestID + "/db2")
  var SourceFileName = DCMQuery(/softwareinstallable[@id=$sourceInstallableID]/file/@name)

  ######################################################
  ## 3: get the file containing the ddl
  ######################################################
   if Jython[simulation == "yes" ] then
     log info Jython["<b>simulation:</b> about to call SoftwareInstallable.Distribute(" +
sourceInstallableID + ", " + ServerID + ")"]
   else
     SoftwareInstallable.Distribute(sourceInstallableID, ServerID)
```

```
   endif

  ######################################################
  ## 4: get the parameters to create the database
  ######################################################
  # find the parameters for creation of the tables in the CONFIGURATION SRT
  #  first, find the ID of the CONFIGURATION softwareResourceTemplate hoding the parameters
  var
configurationSRTID=DCMQuery(/childsoftwareresourcetemplate[@parenttemplateid=$instanceTemplateID]/@id)

  #  then get the database name
  var database_name= DCMQuery(/templateparam[@templateid=$configurationSRTID and
@name="resource-name"]/@value)
  database_name=Jython[database_name.upper()]

  # now, get the name of the instance from the INSTANCE SRT, and the dbadm credentials search-key
  var db_instance_name=DCMQuery(/templateparam[@name="resource-name" and
@templateid=$instanceTemplateID]/@value)
  db_instance_name=Jython[db_instance_name.upper()]
  var db_instance_port=DCMQuery(/templateparam[@name="port" and @templateid=$instanceTemplateID]/@value)
  var dbinstance_sap_searchkey=DCMQuery(/templateparam[@name="trade_dbinstance_SAP_searchkey" and
@templateid=$instanceTemplateID]/@value)

  ######################################################
  ## 5: get the credentials
  ######################################################
  var dbuser_sap_applicationprotocoltype=DCMQuery(/applicationprotocol[@name="UNKNOWN"]/@id)
  var dbuser_sap_protocoltype=DCMQuery(/protocoltype[@name="unknown"]/@id)
  var dbuser_sap_domain=DCMQuery(/templateparam[@name="trade_dbuser_SAP_domain" and
@templateid=$configurationSRTID]/@value)
  var dbuser_sap_context=DCMQuery(/templateparam[@name="trade_dbuser_SAP_context" and
@templateid=$configurationSRTID]/@value)
  var dbuser_sap_searchkey=DCMQuery(/templateparam[@name="trade_dbuser_SAP_searchkey" and
@templateid=$configurationSRTID]/@value)
  var dbuser_sap_id=DCMQuery(/sap[@domain=$dbuser_sap_domain and @context=$dbuser_sap_context and
@protocoltypeid=$dbuser_sap_protocoltype and @appprotocolid=$dbuser_sap_applicationprotocoltype]/@id)

  if Jython[not dbuser_sap_id or dbuser_sap_id=""] then
    msg = Jython[message + "Could not find a valid SAP for:  protocol:" +
dbuser_sap_applicationprotocoltype + "/" + dbuser_sap_protocoltype + "  domain:" + dbuser_sap_domain + "
context:" + dbuser_sap_context]
    log error msg
    throw SAPexception msg
  else
    log info Jython["<b>... finding credentials for searchkey: " + dbuser_sap_searchkey + " in SAP: " +
dbuser_sap_id + "</b>"]
  endif

  var dbuser_credentials_id

  try
    java:com.thinkdynamics.kanaha.de.javaplugin.sap.GetSapCredentials(dbuser_credentials_id,
dbuser_sap_searchkey, dbuser_sap_id)
  catchall
```

```
    msg = Jython[message + “ could not find the credential for searchkey “ + dbuser_sap_searchkey + “ in
SAP “ + dbuser_sap_id]
    log error msg
    rethrow
  endtry

  var dbuser_userid
  var dbuser_password
  java:com.thinkdynamics.kanaha.de.javaplugin.sap.FindCredentialsPassword(dbuser_credentials_id, <null>,
<null>, dbuser_password, dbuser_userid)

  ##########################################################################################
  ##  7: Ensure that the compatible SAPs have been defined for the dbuser user searchkey
  ##########################################################################################
  var DE_Device_ID
  java:com.thinkdynamics.kanaha.de.javaplugin.datacentermodel.FindDEDeviceId(DE_Device_ID)

  ITSO_Synchronize_Credentials (dbuser_credentials_id, ServerID, DE_Device_ID)


  ##################################################################
  ## 8: Ensure that the databaseuser has been created
  ##################################################################
  log info Jython[“<b> Creating user “ + dbuser_userid + “</b>”]
  array groups
  var i=arraysize(groups)
  while Jython[ int(i) >= 0 ] do
    var j = arraysize(groups)
    i=Jython[int(j) + 1]
    var groupParamName = Jython[“trade_dbuser_user_group.” + i]
    var groupName=DCMQuery(/templateparam[@name=$groupParamName and
@templateid=$configurationSRTID]/@value)
    if Jython[groupName] then
      groups[j] = groupName
    else
      break
    endif
  done

  try
    if Jython[simulation.lower() == “yes”] then
      log info Jython[“<b>simulation:</b> About to call : ITSO_Create_User(“ + ServerID + “, “ +
dbuser_userid + “, “ + dbuser_password + “, “ + groups + “)”]
    else
      ITSO_Create_User(ServerID, dbuser_userid, dbuser_password, groups)
    endif
  catchall exception
    log warning Jython[“<b>......got exception “ + exception + “</b>”]
  endtry

  ######################################################
  ## 9: create the database
  ######################################################
  log info Jython[“<b>...creating database tables</b>”]
```

```
  if Jython[simulation.lower() == “yes”] then
    log info Jython[“<b>simulation:</b> About to execute database creation scriptlet using
credentials-key: “ +  dbuser_sap_searchkey]
    log info Jython[“ .....install_dir: “ + install_dir]
    log info Jython[“ .....dbuser_userid: “ + dbuser_userid]
    log info Jython[“ .....dbuser_password: “ + dbuser_password]
    log info Jython[“ .....db_instance_name: “ + db_instance_name]
    log info Jython[“ .....database_name: “ + database_name]
    log info Jython[“ .....SourceFileName: “ + SourceFileName]
    log info Jython[“ .....DestinationPath: “ + DestinationPath]
  else
    #  run scriptlet to create the database
    scriptlet(install_dir, dbuser_userid, dbuser_password, db_instance_name, database_name,
SourceFileName, DestinationPath) language = bash  target = DCMQuery(/server[@id=$ServerID]/@id)
credentialskey=dbuser_sap_searchkey <<EOFDB2
      function db2_doit
      {
        command=${DB2CommandLocation}/$1
        TIOlog “info” “About to execute ‘${command}’”
        msg=‘${command}‘
        rc=$?
        msg=$rc
        TIOlog “info” “(${rc}) ${msg}”
        if [ $((rc)) -gt $((accept_return_code)) ]; then
          TIOthrow “db2CreateDatabaseException” “(${rc}) ${command}: ${msg}”
        fi
        return ${rc}
      }

      [ -z $OSTYPE ] && {
         OSTYPE=‘uname -s‘
      }
      case $OSTYPE in
        cygwin*|CYGWIN*)
            DB2CommandLocation=‘cygpath -u “${install_dir}/bin”‘
            DestinationPath=‘cygpath -u “${DestinationPath}”‘
            ;;
        AIX*|HP*|hpux*|Linux*|Solaris*|linux*)
            # todo - test this on a real system
            DB2CommandLocation=${install_dir}/bin
        ;;
        *)
          # TIOthrow ScriptExitException “${OSTYPE} - Unsupported Operating System”
        ;;
      esac

      export DB2INSTANCE=${db_instance_name}
      Temp1FileName=”${DestinationPath}/target1”
      Temp2FileName=”${DestinationPath}/target2”
      TableDDLFileName=”${DestinationPath}/${SourceFileName}”

      # find the instance
      accept_return_code=0
```

```
      # does the instance exist ?
      count=‘db2set -l | grep -i -c ${db_instance_name}‘
      if [ $((count)) -gt 0 ]; then
        TIOlog “debug” “Exporting DB2INSTANCE variable with a value of ${db_instance_name}”
        TIOlog “debug” “Starting database manager for ${DB2INSTANCE}”
        accept_return_code=1
        db2_doit “db2start”
      else
        msg=”Cannot find db2 instance named ${db_instance_name}”
        TIOlog “error” “$msg”
        TIOthrow “db2InstanceIdentificationException” “$msg”
        return 0
      fi

      TIOlog “warning” “db2instance is: $DB2INSTANCE”

      # has the database been created already ?
      inst=‘db2cmd -i -w -c db2 list database directory‘
      count=‘echo $inst | grep -i -c ${database_name}‘
      if [ $((count)) -gt 0 ]; then
        # database exists - drop it
        TIOlog “debug” “Dropping database ${database_name}”
        accept_return_code=0
        db2_doit “db2cmd -i -w -c db2 drop database ${database_name}”
        rc=$?
        if [ $(($rc)) -gt $((accept_return_code)) ]; then return 0; fi
      fi

      #  create the database
      TIOlog “debug” “Creating database ${database_name}”
      accept_return_code=0
      db2_doit “db2cmd -i -w -c db2 create database ${database_name}”
      rc=$?
      if [ $(($rc)) -gt $((accept_return_code)) ]; then return 0; fi

      # create the tables
      filename=${Temp1FileName}
      TIOlog “debug” “Creating tables from ${TableDDLFileName}”
      printf “\n%s\n” “connect to ${database_name} user ${dbuser_userid} using ${dbuser_password};” >
${filename}
      cat ${TableDDLFileName} >> ${filename}
      printf “\n%s\n” “disconnect all;” >> ${filename}
      unix2dos ${filename}
      filename=‘cygpath -m ${filename}‘
      accept_return_code=4
      db2_doit “db2cmd -i -w -c db2 -tvf ${filename} > ${filename}.log”
      rc=$?
      if [ $(($rc)) -gt $((accept_return_code)) ]; then return 0; fi

      # configure the database
      TIOlog “debug” “Configuring database”
      db2_doit “db2set DB2_RR_TO_RS=yes”
      db2_doit “db2cmd -i -w -c db2 update db config for ${database_name} using logfilsiz 1000”
      db2_doit “db2cmd -i -w -c db2 update db config for ${database_name} using maxappls 100”
```

```
      TIOlog “debug” “Stopping database”
      accept_return_code=1
      db2_doit “db2stop force”

      TIOlog “debug” “Starting database”
      accept_return_code=1
      db2_doit “db2start”

      # bind client
      TIOlog “debug” “Binding client access plans”
      accept_return_code=0
      filename=${Temp2FileName}
      bind_dir=‘cygpath -w “${install_dir}\\bnd”‘
      echo $bind_dir
      printf “\n%s\n”  “connect to ${database_name} user ${dbuser_userid} using ${dbuser_password};”
>${filename}
      printf “\n%s\n”  “bind ${bind_dir}\\@db2cli.lst blocking all grant public;” >>${filename}
      unix2dos ${filename}
      filename=‘cygpath -m ${filename}‘
      accept_return_code=0
      db2_doit “db2cmd -i -w -c db2 -tvf ${filename} “
      rc=$?
      if [ $(($rc)) -gt $((accept_return_code)) ]; then return 0; fi
      db2_doit “db2stop force”
      db2_doit “db2start”
      return 0
    EOFDB2
  endif

catchall ScriptException
  ########################################
  ##  10:  handle errors
  ########################################
  msg = Jython[message + “: “ + ScriptException]
  log error msg
  throw dbCreateTableException
finally
  ######################################################
  ## 11: cleanup
  ######################################################
  var dir=Jython[“/tmp/” + DeploymentRequestID]
  var cmd=Jython(“if [ -d “ + dir + “ ]; then rm -r “ + dir + “ ; fi”)
  log info Jython[“<b>...removing temporary files</b>”]
  if Jython[simulation.lower() == “yes”] then
    log info Jython[“<b>simulation:</b> about to execute: “ + cmd]
  else
    Device.ExecuteCommand(ServerID, cmd, “/”, <null>, <null>, <null>, <null>, <null>, <null>)
  endif
endtry
```

Archived

Trade_DBServer_Delete_Database

This workflow is used to drop the DB2 databases created during installation of the Trade DBServer Module.

The Trade_DBServer_Delete_Database workflow is called from the Trade_DBServer_Module_Instance_RemoveInstance workflow, and the main functionality is:

1. Initialize to set up simulation and error messages.
2. Gather and format the information and parameters needed to set up parameters to be parsed to the scriptlet.
3. Execute the scriptlet on the target server to physically drop the database.

Example 10-24 shows the complete listing of the Trade_DBServer_Delete_Database workflow.

Example 10-24 Trade_DBServer_Delete_Database

```
workflow Trade_DBServer_Delete_Database(SoftwareResourceID)  LocaleInsensitive

########################################
##  1:  Initialize
########################################
var simulation
Trade_Get_Simulation(simulation)

var workflowName=”Trade_DBServer_Module_DeleteDatabase”
var message=Jython[“<b>” + workflowName + “</b>”]
var msg

########################################
##  2:  gather base data and parameters
########################################
try
  var install_dir_parmname=”install_dir”
  var DeploymentRequestID
  java:com.thinkdynamics.kanaha.de.javaplugin.GetCurrentDeploymentRequestId(DeploymentRequestID)

  var ServerID=DCMQuery(/softwareresource[@id=$SoftwareResourceID]/@managedsystemid)

  # find the parameters for creation of the tables in the CONFIGURATION SRT
  #  first, find the ID of the softwareResourceTemplate for the current configuration
  var configurationTemplateID=DCMQuery(/softwareresource[@id=$SoftwareResourceID]/@resourcetemplateid)

  #  then get the parameters
  var database_name= DCMQuery(/templateparam[@templateid=$configurationTemplateID and
@name=”resource-name”]/@value)
  database_name=Jython[database_name.upper()]

  # now, get the name of the instance from the INSTANCE SRT, and the dbadm credentials
```

```
  var
instanceTemplateID=DCMQuery(/softwareresource[@id=$SoftwareResourceID]/softwareresourcetemplate/@parenttem
plateid)
  var install_dir=DCMQuery(/templateparam[@templateid=$instanceTemplateID and
@name=$install_dir_parmname]/@value)
  var db_instance_name=DCMQuery(/templateparam[@name="resource-name" and
@templateid=$instanceTemplateID]/@value)
  db_instance_name=Jython[db_instance_name.upper()]
  var db_instance_port=DCMQuery(/templateparam[@name="port" and @templateid=$instanceTemplateID]/@value)

  ## Get dbinstance userid and password parameters
  var dbinstance_sap_searchkey=DCMQuery(/templateparam[@name="trade_dbinstance_SAP_searchkey" and
@templateid=$instanceTemplateID]/@value)
  var dbinstance_sap_id
  java:com.thinkdynamics.kanaha.de.javaplugin.datacentermodel.FindDefaultSAP(ServerID, "execute-command",
dbinstance_sap_id)

  var dbinstance_credentials_id
  java:com.thinkdynamics.kanaha.de.javaplugin.sap.GetSapCredentials(dbinstance_credentials_id,
dbinstance_sap_searchkey, dbinstance_sap_id)

  var dbinstance_userid
  var dbinstance_password
  java:com.thinkdynamics.kanaha.de.javaplugin.sap.FindCredentialsPassword(dbinstance_credentials_id,
<null>, <null>, dbinstance_password, dbinstance_userid)

  ########################################
  ##  3:  drop the database
  ########################################
  if Jython[simulation.lower() == "yes" ] then
    log info Jython["<b>simulation:</b> .... calling scriptlet to drop database"]
    log info Jython["       install_dir: " + install_dir]
    log info Jython["       dbinstance_userid: " + dbinstance_userid]
    log info Jython["       dbinstance_password: " + dbinstance_password]
    log info Jython["       db_instance_name: " + db_instance_name]
    log info Jython["       database_name: " + database_name]
  else
    scriptlet(install_dir, dbinstance_userid, dbinstance_password, db_instance_name, database_name)
language = bash  target = DCMQuery(/server[@id=$ServerID]/@id) <<EOFDB2
      function db2_doit
      {
        command=${DB2CommandLocation}/$1
        msg=`${command}`
        rc=$?
        TIOlog "info" "(${rc}) Executed ${command}"
        if [ $((rc)) -gt $((accept_return_code)) ]; then
          TIOthrow "db2DeleteDatabaseException" "(${rc}) ${command}: ${msg}"
        fi
        return ${rc}
      }

      [ -z $OSTYPE ] && {
         OSTYPE=`uname -s`
      }
```

```
      case $OSTYPE in
       cygwin*|CYGWIN*)
         DB2CommandLocation=‘cygpath -u “${install_dir}/bin”‘
       ;;
       AIX*|HP*|hpux*|Linux*|Solaris*|linux*)
         # todo - test this on a real system
         DB2CommandLocation=${install_dir}/bin
       ;;
       *)
         # TIOthrow ScriptExitException “${OSTYPE} - Unsupported Operating System”
       ;;
      esac

      export DB2INSTANCE=${db_instance_name}

      # find the instance
      accept_return_code=0

      # does the instance exist ?
      count=‘db2set -l | grep -i -c ${db_instance_name}‘
      if [ $((count)) -gt 0 ]; then
        TIOlog “debug” “Stopping database manager for ${DB2INSTANCE}”
        accept_return_code=1
        db2_doit “db2stop force”
        TIOlog “debug” “Starting database manager for ${DB2INSTANCE}”
        accept_return_code=1
        db2_doit “db2start”
      else
        msg=”Cannot find db2 instance named ${db_instance_name}”
        TIOlog “error” “$msg”
        TIOthrow “db2InstanceIdentificationException” “$msg”
        return 0
      fi

      # has the database been created already ?
      inst=‘db2cmd -i -w -c db2 list database directory‘
      count=‘echo $inst | grep -i -c ${database_name}‘
      if [ $((count)) -gt 0 ]; then
        # database exists - drop it
        TIOlog “debug” “Dropping database ${database_name}”
        accept_return_code=0
        db2_doit “db2cmd -i -w -c db2 drop database ${database_name}”
        rc=$?
        if [ $(($rc)) -gt $((accept_return_code)) ]; then return 0; fi
      fi

      return 0

    EOFDB2
  endif

catchall ScriptException
  ########################################
  ##  4:  handle errors
```

```
  ########################################
  msg = Jython[message + ": " + ScriptException]
  log error msg
  throw dbDeleteTableException msg
endtry
```

10.7 Cataloging the Trade database with the DB2 Client

To catalog the db2 instance that is created during the installation of the Trade DBServer Module, you should use a Software Product, which holds pointers to Software Resource Templates used to instantiate the necessary resources in the data center model, and create the necessary db2 catalog entries. In addition, the Software Product will contain a pointer to a Software Package, which is needed to be able to install the Software Product. Furthermore, the requirements must be specified for the Software Product, which will help ensure that the functionality required for successful implementation is available on the target system.

The Software Product will be installed though the Software Module.Install logical device operation. The standard functionally of this is explained in 10.6, "Creating the Trade database" on page 328.

Similar to the issues related to the creation of the database, when creating the catalog entries at the DB2 Client, we are faced with the problem that when implementing in a customer environment, we have absolutely no knowledge of how the underlying infrastructure is defined in the Data Center Model.

In the following, we make the assumptions that the capabilities of the target servers are:

OS	os.family=Windows 2000 Server SP4
DATABASE	database.family=DB2Client database.version=8.2

and we will set up requirements for these capabilities in the Software Product and Software Package to reflect these assumptions.

10.7.1 The Trade DBClient Module Software Product

To catalog the database needed for the Trade application at the Application Server system, we define a Software Product named Trade DBClient Module. Through the Software Product definition, we will link to a Software Package named Trade DBClient Module Installable, and by including the proper Software Resource Templates in the Software Product, we ensure that the expected hierarchy of DCM objects will be created and named according to our standards.

In addition, we will associate special capabilities with the Software Product. These capabilities will be applied to the target server upon installation, and thus can be used to later to ensure that catalog entries for the Trade database exists. To ensure that the required DB2 Client functionality is available on the servers where the Trade DBClient Module will be installed, we will also add DATABASE related requirements to the Software Product definition.

Figure 10-28 shows the Trade DBClient Module properties.

Figure 10-28 Trade DBClient Module properties

Capabilities and requirements

The capabilities we assign to the Trade DBClient Module are:

APPLICATION	software.title	TRADE
APPLICATION	software.name	JDBC
APPLICATION	software.vendor	IBM
APPLICATION	software.version	3.1

These capabilities will allow us, at a later point, to reference the Trade DBClient Module installation as a requirement for other components of the Trade Automation Package. This way, we do not waste time installing modules that rely on the database catalog entries if these have not yet been created.

By specifying the following requirements for our Trade DBClient Module Software Product:

DATABASE	database.family	DB2Client
DATABASE	database.version	8.2
EJB_CONTAINER	ejb.version	2.0

it is guaranteed that the functionality provided by a database client installation, as well as an Application Server implementing an EJB Container, exists on the target system at the time of installation of the Trade DBClient Module installation.

Archived

Note: In a real-life implementation, the user customer should be provided instructions on how to modify the requirements that are set up by default for the Trade DBClient Module in order to integrate it into a specific environment.

Figure 10-29 shows the capabilities and requirements for the Trade DBClient Module Software Product.

Requirements and Capabilities

Requirement Name	Requirement Type	Requirement Values
software.vendor	APPLICATION	IBM
software.version	APPLICATION	3.1
database.family	DATABASE	DB2Client
database.version	DATABASE	8.2
software.name	APPLICATION	DB
ejb.version	EJB_CONTAINER	2.0
software.title	APPLICATION	TRADE

Capability Type
APPLICATION

Capability Name	Capability Value
software.title	TRADE
software.version	3.1
software.name	JDBC
software.vendor	IBM

Figure 10-29 Trade DBClient Module capabilities and requirements

Software Package(s)

To allow the Trade DBClient Module Software Product to be installed, we have to create a Software Package, which determines what needs to be installed. We named it Trade DBClient Module Installable.

For cataloging databases, we do not need to transfer any files to the target system during installation; however, to limit the scope of the installation to systems that have the Windows 2000 Server SP4 operating systems installed, we added a requirement for the Software Package, specifying the following:

OS	os.family	Windows 2000 Server SP4

This will limit the scope of our development effort, since we can be ensured that we are only installing on Windows systems. During installation, this requirement has to be honored by the target server in order for the TIO/TPM Server to identify a Software Package to be used for the installation (and thus, the Trade DBClient

Module will only install on systems the supports this requirement) and naturally the requirements of the Trade DBClient Module itself.

> **Note:** In a real-life implementation, the user should be provided instructions on how to modify the requirements that is set up by default for the Trade DBClient Module Installable in order to integrate it into a specific environment.

The basic attributes of the Trade DBClient Module Installable are shown in Figure 10-30.

Figure 10-30 Trade DBClient Module Installable software package properties

Notice that the associated device driver, which has to be provided as a integral part of the Trade Automation Package, and hence is developed by us, is named Trade_DBClient_Module_Installable_Driver. The workflows associated with this driver are the ones providing the functionality required to catalog the Trade database and the DB2 node hosting it.

Variables

We also define a set of variables for the Software Package, which will be used to help find the installation directory of the underlying DB2 Client installation as described in 10.5.2, “Finding installation directories” on page 324. For the Trade DBClient Module Installable, these variables are created with the standard values shown in Figure 10-31 on page 391, and it is expected that they will be customized for a specific implementation during installation of the Trade Automation Package.

Archived

Figure 10-31 Trade DBClient Module Installable software package variables

The Trade_DBClient_Module_Installable_Driver Device driver

To control the installation of the Trade DBClient Module Installable, we need to assign a device driver that, at a minimum, contains a workflow that implements the SoftwareInstallable.Install logical device operation. Since there are no files needed for the installation, we will not provide support for the SoftwareInstallable.Distribute LDO.

Figure 10-32 Trade_DBClient_Module_Installable_Driver

Example 10-25 shows the XML statements embedded in the tc-driver.xml file for the entire Trade Automation Package, which are used to define the device driver to the DCM. The device driver we assign to the Trade DBClient Module Installable will be named Trade_DBClient_Module_Installable_Driver. Details regarding the workflows imbedded in the device driver are provided in “Trade_DBServer_Module_Installable_Install” on page 348 and “Trade_DBServer_Module_Installable_Distribute” on page 352.

Example 10-25 XML to define the Trade_DB2Server_Module_Installable_Driver

```
<device-model name="Trade_DBClient_Module_Installable_Driver"
category="Trade">
   <workflow name="Trade_DBClient_Module_Installable_Install" />
</device-model>
```

Before we look at the details of the workflows, we need to take a look at the Software Resource Templates of the Trade DBClient Module Software Product in order to gain knowledge about the customization parameters that will be available to the installation workflows.

Software Resource Templates

When modelling the installation activities and resources in TIO/TPM, it is necessary to understand the default behavior of the various software resource types in the TIO/TPM data center model. To translate this default behavior to the task at hand, it quickly becomes clear that we need to design and develop Software Resource Templates to create the desired objects in the DCM as well as workflows implementing the resource specific logical device operations.

The software models used to create the DCM objects related to the Trade DBClient Module are:

INSTALLATION Creates the Software Installation software object representing the installation on a specific server. This object will be used to host child objects, such as Instance and Configuration.

CONFIGURATION Creates a Software Resource object, which we will use to represent the catalog entries of the trade3db database

The Software Resource Templates will be created in a hierarchy in order to represent the relationships between them. The INSTALLATION SRT is required to instantiate the Trade DBClient Module installation on the target server, which in turn “owns” the CONFIGURATION Software Resource Template.

To control the behavior of each of the objects created by the Software Resource Templates, a device driver, which implements the resource specific LDOs, needs to be assigned to each Software Resource Template. If this step is skipped, no implementations for the resource specific LDOs are available, and no actions will be performed. The necessary device drivers have to provide functionality, through the associated workflows, that is specific to our implementation of the Trade DBClient Module, so naturally, these have to be created as part of, and delivered with, the Trade Automation Package.

Now, if we use the software resource model described above to create and register the objects related to the DB2 Client in the DCM, we will not achieve our goal. Since the Software Resource Templates are parented by the Trade DBClient Module Software Product, the software resources they crate will be children of this Software Product, and not the DB2 Client installation that should host the objects that we will actually be implementing, a set of DB2 catalog entries. Physically, these objects will manifest themselves as children of the DB2

Client installation, so within our DCM, we have to reflect this to ensure that the logical model of the data center (in the DCM) is consistent with the physical implementation.

This implies that we will have to create a Software Resource Template structure, which will allow us to create the CONFIGURATION object as a child of another Software Installation; luckily, TIO/TPM supports this through the Software Resource Type named FOREIGN-CONFIGURATION. As for all other Software Resource Templates besides INSTALLATION and INSTANCE, TIO/TPM does not offer any automated processing to create the resources parented by a FOREIGN_CONFIGURATION Software Resource Template; it is the responsibility of the implementation workflow to detect the FOREIGN_CONFIGURATION template, and take action accordingly. Normally, the action that would be taken is a call to the HostingEnvironment.Host logical device operation for the hosting resource (the DB2 Client installation, in this case). However, since we have no knowledge about the implementation details of the Software Installation that represents the DB2 Client on the target system, we do not know if the HostingEnvironment.Host logical operation is supported in the device driver associated with the Software Installation, so we have to provide all of this functionality ourselves.

So, in order to register the DB2 related resources with the data center model as children of the DB2 Client Software Installation object, we add a FOREIGN_CONFIGURATION Software Resource Template to our software model, making the Software Resource Template hierarchy of the Trade DBClient Module Software Product look like this:

INSTALLATION
  FOREIGN_CONFIGURATION
    CONFIGURATION

As a result, the INSTALLATION template will be used to create a Software Installation object on the target server, and associate this with the Trade DBClient Module Software Product. As children of this Software Installation object TIO/TPM will clone the reminder of the SRT hierarchy (the FOREIGN_CONFIGURATION template an all it's children) to the new Software Installation, the workflow that implements the SoftwareInstallable.Install LDO is responsible for inspecting the SRT hierarchy and takes appropriate action(s).

Device driver associations

To control the behavior of the resources that are created from the Software Resource Templates, we may associate a device driver to each of them. TIO/TPM normally applies special processing to the INSTALLATION and INSTANCE and therefore you will typically find that only these objects have associated device drivers, and that the workflows implementing the SoftwareInstallable.Install, SoftwareInstallable.Uninstall,

SoftwareInstance.AddInstance, and SoftwareInstance.RemoveInstance logical operations are responsible for inspecting the nested object hierarchy and perform the necessary tasks.

For the Trade DBClient Module Software Package software model, we associate the device drivers and related workflows implementing the required logical operations, as shown in Table 10-5 on page 337.

Table 10-7 Trade DBClient Module related device drivers and workflows

SRT	Type	Name	Workflow
INSTALLATION	device driver	Trade_DBClient_Module_Intallation_Driver	
	LDOs	SoftwareInstallation .Uninstall	Trade_DBClient_Module_Intallati on_Uninstall

During installation, we will only execute a couple of **`db2 catalog..`** commands, and will not need to be concerned about instances and operational issues. Therefore, we only need to supply the functionality to uncatalog the databases (uninstall). This workflow may call additional workflows, which are not associated with any particular device driver, in order to execute specific actions, which are repetitive, or provide a very specific set of functions.

Software resource naming

The naming of Software Resources that will be created based on the templates can be controlled by creating a template parameter named *resource-name*, and by providing a value that matches your requirements. Figure 10-33 on page 396 shows the final Software Resource Template hierarchy for the Trade DBClient Module, and here we see that the names that will be assigned to the objects created by the templates are:

Installation Trade DBClient Installation
Foreign_Configuration
Trade DBClient Database Cataloging
Configuration Trade DBClient Database Definition

Software Resource Template parameters

The last step in completing the software model for the Trade DBClient Module is to associate parameters with each of the templates. These parameters hold configuration information that will be used during the instantiation of each of the software resources. These parameters will be read by the workflows responsible for creating the physical objects represented by the software resources.

Table 10-8 lists the parameters defined in the Software Resource Templates associated with the Trade DBClient Module.

Table 10-8 Trade DBClient Module related SRT parameters

SRT	Name	Value
INSTALLATION	resource-name	Trade DBClient Database Definition
	trade_dbadmin_SAP_domain	trade
	trade_dbadmin_SAP_context	db
	trade_dbadmin_SAP_searchkey	dbadmin
FOREIGN_ CONFIGURATION	resource-name	Trade DBClient Database Cataloging
	client_instance	DB2
CONFIGURATION	resource-name	Trade DBClient Database Definition
	databaseName	trade3db
	databasAliasName	t3db
	databaseInstanceName	trade3
	databaseNodeName	t3dbnode

As Table 10-8 shows, most of these parameters are related to controlling the database cataloging. For a discussion of the use of the SAP and credential parameters added, please refer “Software Resource Templates” on page 335.

Archived

Finally, we can take a look at the completed Software Resource Template hierarchy for the Trade DBClient Module (Figure 10-33).

Figure 10-33 Trade DBClient Module Software Resource Templates

Package overview

Based on the parameters and attributes we have defined for the Trade DBClient Module Software Product, along with the related Software Package and the embedded Software Resource Templates added, we have created the software model for cataloging an existing instance of the Trade database at a server hosting a DB2 Client installation in a way that will be easily customizable for the potential users of the automation package.

Archived

Figure 10-34 shows a depiction of the resources involved and their relationships.

Figure 10-34 Trade DBClient Module deployment model

Creating the software objects from the automation package

Since we want to create the Trade DBClient Module Software Product as part of the Trade Automation Package installation, we need to provide all the definitions and parameters discussed in the previous section in an XML file, which can be included in the automation package, and automatically imported into the DCM during automation package installation.

The easiest way to create this XML file is to create all the definitions manually (using the TIO/TPM Console) and use the **`dcmexport`** utility to export the contents of the entire DCM. Then you can identify the <software-module > definition named Trade DBClient Module and copy the entire `<software-module..> ... </software-module>` section to a flat file, and save it as part of the APDE project for Trade. We used a specific XML directory within the project to store all the XML files for the Trade Automation Package. The content of the Trade_DBClient_Module.xml is shown in Example 10-26 on page 398.

To automatically import this XML file as part of the automation package installation, you will have to add an item definition for the xml file using the **import** function to the <dcm> section in the tc-driver.xml file for the automation package:

```
<dcm>
   <item name="xml/Trade_DBClient_Module.xml" action="import"/>
</dcm>
```

The XML deck needed to define our Software Product and Software Package, along with the embedded file, variable, Software Resource Template, and device driver definitions, is shown in Example 10-26.

Example 10-26 Trade_DBClient_Module.xml

```
<?xml version=”1.0” encoding=”UTF-8”?>
<!DOCTYPE datacenter SYSTEM “file:../xml/xmlimport.dtd”>

<datacenter>

    <software-module name=”Trade DBClient Module” version=”3.1”
                     vendor=”IBM” title=”Trade DB2 database definitions”
                     description=”Trade database cataloging”
                     is-draft=”false”>

       <software-capability name=”software.title” value=”TRADE” />
       <software-capability name=”software.version” value=”3.1” />
       <software-capability name=”software.name” value=”JDBC” />
       <software-capability name=”software.vendor” value=”IBM” />

     <supported-requirement-type value=”APPLICATION” />
       <software-requirement name=”software.version” type=”APPLICATION”
                             enforcement=”MANDATORY”
                             hosting=”false”
                             accept-non-existing=”true”>
              <software-requirement-value value=”3.1” />
       </software-requirement>
       <software-requirement name=”database.family”
                             type=”DATABASE”
                             enforcement=”MANDATORY”
                             hosting=”true”
                             accept-non-existing=”false”>
              <software-requirement-value value=”DB2Client” />
       </software-requirement>
       <software-requirement name=”database.version”
                             type=”DATABASE”
                             enforcement=”MANDATORY”
                             hosting=”true”
                             accept-non-existing=”false”>
              <software-requirement-value value=”8.2” />
       </software-requirement>
       <software-requirement name=”ejb.version”
                             type=”EJB_CONTAINER”
                             enforcement=”MANDATORY”
```

```
                    hosting="true"
                    accept-non-existing="false">
        <software-requirement-value value="2.0" />
</software-requirement>
<software-requirement name="software.title"
                    type="APPLICATION"
                    enforcement="MANDATORY"
                    hosting="false"
                    accept-non-existing="true">
        <software-requirement-value value="TRADE" />
</software-requirement>
<software-requirement name="software.vendor"
                    type="APPLICATION"
                    enforcement="MANDATORY"
                    hosting="false"
                    accept-non-existing="true">
        <software-requirement-value value="IBM" />
</software-requirement>
<software-requirement name="software.version"
                    type="APPLICATION"
                    enforcement="MANDATORY"
                    hosting="false"
                    accept-non-existing="true">
    <software-requirement-value value="3.1" />
</software-requirement>
<software-requirement name="software.name"
                    type="APPLICATION"
                    enforcement="MANDATORY"
                    hosting="false"
                    accept-non-existing="true">
        <software-requirement-value value="DB" />
</software-requirement>

<installable-package name="Trade DBClient Module Installable"
                    is-device-model="Trade_DBClient_Module_Installable_Driver"
                    locale="en_US"
                    version="3.1"
                    priority="0"
                    file-repository="LocalFileRepository"
                    status="tested">
        <file name="-1" path="/" />
        <property component="DEPLOYMENT_ENGINE"
                  name="install_dir_softwareresource_type"
                  value="" />
        <property component="DEPLOYMENT_ENGINE"
                  name="install_dir_parameter_name"
                  value="" />
        <property component="DEPLOYMENT_ENGINE"
                  name="install_dir_custom_default"
                  value="c:/ibm/sqllib" />
        <property component="DEPLOYMENT_ENGINE"
                  name="install_dir_system_default"
                  value="c:\Program Files\sqllib" />
```

Archived

```
        <software-requirement name="os.family" type="OS"
                              enforcement="MANDATORY"
                              hosting="true"
                              accept-non-existing="false">
                <software-requirement-value value="Windows 2000 Server SP4" />
        </software-requirement>
</installable-package>

<software-resource-template name="Trade DBClient Installation"
                            software-resource-type="INSTALLATION"
                            software-resource-device-model="Trade_DBClient_Module_Installation_Driver"
                            multiplicity-type="Unspecified"
                            software-configuration-type="Regular"
                            is-selected="true">
    <software-resource-template name="Trade DBClient Database Cataloging"
                              software-resource-type="FOREIGN-CONFIGURATION"
                              multiplicity-type="Unspecified"
                              software-configuration-type="Regular"
                              is-selected="true">
        <software-resource-template name="Trade DBClient Database Definition"
                                  software-resource-type="CONFIGURATION"
                                  multiplicity-type="Unspecified"
                                  software-configuration-type="Regular"
                                  is-selected="true">
                <template-param name="resource-name"
                              value="Trade DBClient Database Definition"
                              is-changeable="true" />
                <template-param name="databaseName"
                              value="trade3db"
                              is-changeable="true" />
                <template-param name="databaseAliasName"
                              value="trade3"
                              is-changeable="true" />
                <template-param name="databaseInstanceName"
                              value="trade3"
                              is-changeable="true" />
                <template-param name="databaseNodeName"
                              value="t3dbnode"
                              is-changeable="true" />

        </software-resource-template>
        <template-param name="resource-name"
                        value="Trade DBClient Database Cataloging"
                        is-changeable="true" />
        <template-param name="targetResourceName"
                        value="DB2"
                        is-changeable="true" />
        <template-param name="targetResourceType"
                        value="INSTANCE"
                        is-changeable="true" />
    </software-resource-template>
    <template-param name="resource-name"
                    value="Trade DBClient Installation"
                    is-changeable="true" />
```

```
        </software-resource-template>

    </software-module>

</datacenter>
```

10.7.2 Trade_DBClient_Module_Installable_Driver

The purpose of the device driver that will be associated with the Trade_DBClient_Module_Installable is solely to provide a pointer to the implementation of the SoftwareInstallable.Install logical device operation, which in our case is the Trade_DBClient_Module_Installable_Install workflow, as shown in Figure 10-35.

Figure 10-35 Trade_DBClient_Module_Installable_Driver Overview

The two supporting workflows are used to identify the underlying DB2 Client installation, and to gather configuration data from it, primarily the DB2 installation directory. Figure 10-36 shows the DCM definitions of the driver.

Figure 10-36 Trade_DBClient_Module_Installable_Driver

The XML statements that are added to the tc-driver.xml file of the Trade Automation Package to define the Trade_DBClisnt_Module_Installable_Driver to the DCM during installation of the Automation Package are listed in Example 10-27.

Example 10-27 Trade_DBClient_Module_Installable_Driver xml definitions

```
<device-model name="Trade_DBClient_Module_Installable_Driver"
category="Trade">
    <workflow name="Trade_DBClient_Module_Installable_Install" />
</device-model>
```

In the following section, we provide the details of the workflows that implements logical device operations associated with the Trade_DBClisnt_Module_Installable_Driver.

Trade_DBClient_Module_Installable_Install

As already stated, the purpose of this workflow is to provide specific functionality to install the Trade Module DBClient Installable onto a target system. Since the DB2 Client itself is already installed, and the implementation of this module basically only has to add entries to the DB2 node and database directories of the default instance named DB2, the installation is rather simple, and the main challenges are related to querying the DCM for configuration details of the node and database to be cataloged.

The outline of the Trade_DBClient_Module_Installable_Install workflow is:

1. Initialize and set up variables for error handling and logging.
2. Gather data for control and navigation.
3. Find the supporting installation for the DATABASE requirement in order to locate the DB2 Client installation executables. The utility workflows ITSO_Find_Supporting_Installation_for_Module_Reqirements is used (see 10.10.1, “ITSO_Find_Supporting_Installation_for_Module_Requirements” on page 494 for details).
4. Commit the Software Installation resource to allow for manipulation and query in the subsequent workflows.
5. Create the new resources, modelled after the child software resources templates of the FOREIGN-CONFIGURATION templates, parented by the DB2 Client instance named DB2. This is performed by the utility workflow ITSO_HostingEnvironment_Host, which is described in detail in 10.10.4, “ITSO_HostingEnvironment_Host” on page 500.
6. Call the helper workflow Trade_DBClient_Add_Configuration (see “Trade_DBClient_Add_Configuration” on page 408) in order to perform the

cataloging of the Trade database server node and database in the DB2 Client directories.

7. Handle errors by parsing them back to the caller.

The entire listing of the Trade_DBClient_Module_Installable_Install is shown in Example 10-28.

Example 10-28 Trade_DBClient_Module_Installable_Install

```
workflow Trade_DBClient_Module_Installable_Install(in SoftwareID, in DeviceID, in
SoftwareResourceTemplateID) implements SoftwareInstallable.Install LocaleInsensitive

#######################################################################
## 1: Initialize
#######################################################################

var simulation
Trade_Get_Simulation(simulation)

var workflowName="Trade_DBClient_Module_Installable_Install"
var message=Jython["<b>" + workflowName + ":</b>"]
var msg

var SoftwareInstallationID
try
  #######################################################################
  ## 2: Gather data for control and navigation
  #######################################################################
  var SoftwareInstallableID=SoftwareID
  var ServerID=DeviceID
  SoftwareInstallationID=DCMQuery(/softwareinstallation[@managedsystemid=$ServerID and
@softwareproductid=$SoftwareInstallableID and @pending="Y"]/@id)
  if Jython[not SoftwareInstallationID or SoftwareInstallationID == ""] then
    SoftwareInstallationID=DCMQuery(/softwareinstallation[@managedsystemid=$ServerID and
@softwareproductid=$SoftwareInstallableID]/@id)
  endif
  var SoftwareModuleID=DCMQuery(/softwareinstallation[@id=$SoftwareInstallationID]/@softwaremoduleid)
  var installationResourceType=DCMQuery(/softwareresourcetype[@name="INSTALLATION"]/@id)
  var instanceResourceType=DCMQuery(/softwareresourcetype[@name="INSTANCE"]/@id)
  var foreignResourceType=DCMQuery(/softwareresourcetype[@name="FOREIGN-CONFIGURATION"]/@id)
  var
installationSRTID=DCMQuery(/softwareinstallation[@id=$SoftwareInstallationID]/softwareresourcetemplate[@so
ftwareresourcetype=$installationResourceType]/@id)
  var foreignSRTID=DCMQuery(/childsoftwareresourcetemplate[@parenttemplateid=$installationSRTID]/@id)

  #######################################################################
  ## 3: Find Supporting Installation for Module DATABASE Reqirements
  #######################################################################
  var requirementType="DATABASE"
  var supportingSoftwareInstallationID
  ITSO_Find_Supporting_Installation_for_Module_Reqirements(requirementType, SoftwareModuleID, ServerID,
supportingSoftwareInstallationID)
```

```
  ##################################################################
  ##  4: Commit the new Installation before creating foreign resources
  ##################################################################
  # update the pending flag of the previously created resources

java:com.ibm.tivoli.orchestrator.datacentermodel.helper.SoftwareHelper#updateResourceFlag(SoftwareInstalla
tionID, “false”)

  ##################################################################
  ## 5: Create the FOREIGN resources
  ##################################################################
  array objects
  ITSO_HostingEnvironment_Host(supportingSoftwareInstallationID, SoftwareModuleID, foreignSRTID, objects)

  ##################################################################
  ## 6: Catalog the trade database
  ##################################################################
  foreach objectID in objects do
    var srtID=DCMQuery(/softwareresource[@id=$objectID]/@resourcetemplateid)
    var targetResourceID=DCMQuery(/softwareresource[@id=$objectID]/@parentresourceid)
    log info Jython[“.....About to call Trade_DBClient_Add_Configuration(“+targetResourceID+”,
“+srtID+”)”]
    Trade_DBClient_Add_Configuration(targetResourceID, srtID)
  done
catchall
  ##################################################################
  ## 7: Handle errors
  ##################################################################
  msg = Jython[message + “problem installing...”]
  log error msg
  ##################################################################
  ## 8: Set the status pending if creation of freign objects failed
  ##################################################################
  # delete foreign resources
  foreach objectID in objects do
    var objID=DCMQuery(/softwareresource[@id=$objectID]/@id)
    if Jython[ objID != None] then
      DCMDelete(/softwareresource[@id=$objectID]/@id)
    endif
  done
  # uncommit the Installation if we failed

java:com.ibm.tivoli.orchestrator.datacentermodel.helper.SoftwareHelper#updateResourceFlag(SoftwareInstalla
tionID, “true”)
  # raise the exception
  rethrow
endtry
```

10.7.3 Trade_DBClient_Module_Installation_Driver

Because of the simple nature of the Trade DBClient Module, the only logical required device operation related to the Software Installation object is the uninstall operation. In our implementation, the name of the device driver that is associated with the Trade DBClient Module Software Installation object is Trade_DBClient_Module_Installation_Driver, and as shown in Figure 10-37, the name of the workflow for uninstallation is Trade_DBClient_Module_Installation_Uninstall.

Figure 10-37 Trade_DBClient_Module_Instation_Driver Overview

The DCM definitions of the Trade_DBClient_Module_Installation_Driver is shown in Figure 10-38.

Figure 10-38 Trade_DBClient_Module_Installation_Driver

Example 10-29 shows the XML statements in the Trade Automation Package XML file used to define the Trade_DBClient_Module_Installation_Driver to the DCM.

Example 10-29 Trade_DBClient_Module_Installation_Driver xml definitions

```
<device-model name="Trade_DBClient_Module_Installation_Driver"
category="Trade">
    <workflow name="Trade_DBClient_Module_Installation_Uninstall" />
</device-model>
```

Archived

Trade_DBClient_Module_Installation_Uninstall

The basic responsibility of the workflow for uninstalling the Trade DBClient Module Installation is to uncatalog the Trade database and the related node definition pointing to the server hosting the database instance. Once the DB2 catalog has been updated, the foreign resources created during installation needs to be removed too, since the underlying DB2 definitions that they represent has been uncataloged.

The outline of the Trade_DBClient_Module_Installation_Uninstall workflow is:

1. Initialize and set up error handling variables.
2. Get base data for control and logging. Most of the data needed was saved in the Software Resource Template by the Trade_DBClient_Add_Configuration workflow (described in “Trade_DBClient_Add_Configuration” on page 408) during installation.
3. Get the configuration templates that have the same source as the one in the installation and reside on the same server. These are the ones that need to be removed. For each object, do the following:
   a. Remove the db2 catalog entries by calling a scriptlet that performs the necessary DB2 commands.
   b. Delete the foreign CONFIGURATION object.
4. Handle errors by returning all exceptions to the caller.

Example 10-30 shows the entire uninstallation workflow.

Example 10-30 Trade_DBClient_Module_Installation_Uninstall

```
workflow Trade_DBClient_Module_Installation_Uninstall(in SoftwareInstallationID) implements
SoftwareInstallation.Uninstall LocaleInsensitive

#######################################################
##  1: Initialize
#######################################################
var simulation
Trade_Get_Simulation(simulation)

var workflowName=”Trade_DBClient_Installation_Uninstall”
var message=Jython[“<b>” + workflowName + “: </b>”]
var msg

try
  #######################################################
  ##  2: Get base data for control and logging
  #######################################################
  var ServerID=DCMQuery(/softwareinstallation[@id=$SoftwareInstallationID]/@managedsystemid)
  var SoftwareModuleID=DCMQuery(/softwareinstallation[@id=$SoftwareInstallationID]/@softwaremoduleid)
  var installationResourceType=DCMQuery(/softwareresourcetype[@name=”INSTALLATION”]/@id)
  var instanceResourceType=DCMQuery(/softwareresourcetype[@name=”INSTANCE”]/@id)
```

```
  var foreignResourceType=DCMQuery(/softwareresourcetype[@name="FOREIGN-CONFIGURATION"]/@id)
  var configurationType=DCMQuery(/softwareresourcetype[@name="CONFIGURATION"]/@id)
  var
installationSRT=DCMQuery(/softwareinstallation[@id=$SoftwareInstallationID]/softwareresourcetemplate[@soft
wareresourcetype=$installationResourceType]/@id)
  var foreignSRT=DCMQuery(/childsoftwareresourcetemplate[@parenttemplateid=$installationSRT]/@id)
  var configSRT=DCMQuery(/softwareresourcetemplate[@parenttemplateid=$foreignSRT]/@name)
  var configSRTID=DCMQuery(/softwareresourcetemplate[@parenttemplateid=$foreignSRT]/@id)

  ##########################################################################
  ##  3: Get the configuration templates that have the same soure as the one in the installation
  ##########################################################################
  #  source template
  var sourceConfigurationSRTID=DCMQuery(/softwareresourcetemplate[@id=$configSRTID]/@sourcetemplateid)
  foreach supportingConfigurationSRTID in
DCMQuery(/softwareresource[@managedsystemid=$ServerID]/softwareresourcetemplate[@sourcetemplateid=$sourceC
onfigurationSRTID and @softwaremoduleid=$SoftwareModuleID and @id!=$configSRTID]/@id) do
    var
supportingConfigurationID=DCMQuery(/softwareresource[@resourcetemplateid=$supportingConfigurationSRTID]/@i
d)
    # get the parameters
    var paramName
    paramName="databaseAliasName"
    var databaseAliasName=DCMQuery(/templateparam[@templateid=$supportingConfigurationSRTID and
@name=$paramName]/@value)
    paramName="databaseNodeName"
    var databaseNodeName=DCMQuery(/templateparam[@templateid=$supportingConfigurationSRTID and
@name=$paramName]/@value)

    ##########################################################################
    ##  a: Remove the db2 catalog entries
    ##########################################################################
    # execute the db2 catalog commands
    if Jython[simulation == "yes"] then
      log info Jython["<b>simulation: </b>... about to execute scriptlet to remove trade3 database catalog
information"]
      log info Jython["      databaseAliasName: " + databaseAliasName]
      log info Jython["      databaseNodeName: " + databaseNodeName]
    else
      scriptlet (databaseAliasName,databaseNodeName ) language = bash target =
DCMQuery(/server[@id=$ServerID]/@id) credentialskey="default"  <<EOFDB2
        function do_db2 {
          db2cmd_prefix="db2cmd -i -w -c "
          cmd2go="${db2cmd_prefix} ${1}"
          msg='${cmd2go}'
          rc=$?
          TIOlog "info" "(${rc})  command:'${cmd2go}'  returned:'${rc}'"
          if [ ${rc} -gt ${2} ]; then
            TIOthrow "databaseCatalogError" "(${rc}) command:${1}"
          fi
          return ${rc}
        }
        accepted_return_code=4
        db2command="db2 uncatalog database ${databaseAliasName}"
```

```
        do_db2 "${db2command}" "${accepted_return_code}"
        db2command="db2 uncatalog node ${databaseNodeName}"
        do_db2 "${db2command}" "${accepted_return_code}"
        return 0
      EOFDB2
    endif
    ##########################################################################
    ##  b: Delete the foreign Configuration object
    ##########################################################################
    log info Jython["..Deleting foreign resource: " + supportingConfigurationID]
    DCMDelete(/softwareconfiguration[@id=$supportingConfigurationID]/@id)
  done
catchall exception
  ##########################################################################
  ##  4: Handle errors
  ##########################################################################
  msg=Jython[message + "A problem oddured during unstallation:" + exception]
  log error msg
  rethrow
endtry
```

10.7.4 Trade DBClient Module Helper workflows

Besides the two workflows implementing logical device operations for installing and removing the Trade DBClient Module software product, we need the following helper workflow, which performs the actual db2 cataloging during the installation:

- Trade_DBClient_Add_Configuration

Trade_DBClient_Add_Configuration

The Trade_DBClient_Add_Confiuration workflow creates the catalog entries defining the Trade database to the DB2 Client in order for the WebSphere Application Server on the same system to define the JDBC resources required to access the database.

This workflow is called from the Trade_DBClient_Module_Installable_Install workflow and could just as well have been included in this workflow instead of being a separate workflow. However, having it as a separate workflow eases development and unit testing.

The main outline of the Trade_DBClient_Add_Configuration workflow is:

1. Initialize and control variables for logging and error handling.
2. Gather data for parameter setup and flow control.
3. Find the database server database instance based on the non-existing APPLICATION requirement (specifying the Trade DBServer Module) using

the utility workflow ITSO_Find_Supporting_Installation_for_Module_Reqirements.

4. Find the configuration of the trade database on the remote installation.
5. Get the configuration parameters to be used for the cataloging from the CONFIGURATION Software Resource Template.
6. Call a scriptlet to issue the required **db2 catalog** commands.
7. Save the parameters in the Software Resource Template of the current software resource.
8. Send any exceptions back to the caller.

For a complete listing of the Trade_DBClient_Add_Configuration workflow, please see Example 10-31.

Example 10-31 Trade_DBClient_Add_Configuration

```
workflow Trade_DBClient_Add_Configuration(in SoftwareResourceID, in
SoftwareResourceTemplateID)LocaleInsensitive

#######################################################
## 1: Initialize
#######################################################
var simulation
Trade_Get_Simulation(simulation)

var workflowName="Trade_DBClient_Add_Configuration"
var message=Jython["<b>" + workflowName + ": </b>"]
var msg

#########################################################
## 2: Gather data, parameters, etc. for control and logging
#########################################################
try
  var ServerID=DCMQuery(/softwareresource[@id=$SoftwareResourceID]/@managedsystemid)
  var
SoftwareModuleID=DCMQuery(/softwareresourcetemplate[@id=$SoftwareResourceTemplateID]/@softwaremoduleid)
  var swInstanceTypeID = DCMQuery(/softwareresourcetype[@name="INSTANCE"]/@id)
  var swConfigurationTypeID = DCMQuery(/softwareresourcetype[@name="CONFIGURATION"]/@id)

  # get the configuration template
  if Jython[ not SoftwareResourceTemplateID ] then
    msg=Jython[message + "no configuration software resource template found for resource " + SoftwareID]
    log error msg
    throw parameterValidationException msg
  endif

  # get the parameters
  var
databaseAliasName=DCMQuery(/softwareresourcetemplate[@id=$SoftwareResourceTemplateID]/templateparam[@name=
"databaseAliasName"]/@value)
```

```
  if Jython[ not databaseAliasName ] then
    msg=Jython[message + “no value found for parameter databaseAliasName in SRT “ +
SoftwareResourceTemplateID]
    log error msg
    throw parameterValidationException msg
  endif

  var
dbServerServerNodeName=DCMQuery(/softwareresourcetemplate[@id=$SoftwareResourceTemplateID]/templateparam[@
name=”databaseNodeName”]/@value)
  if Jython[ not dbServerServerNodeName ] then
    msg=Jython[message + “no value found for parameter databaseNodeName in SRT “ +
SoftwareResourceTemplateID]
    log error msg
    throw parameterValidationException msg
  endif

  # get the name of the instance from which we will read the catalog parameters
  var paramName=”databaseInstanceName”
  var DB2ServerInstanceName = DCMQuery(/templateparam[@templateid=$SoftwareResourceTemplateID and
@name=$paramName]/@value)

  var DB2ServerInstanceID
  foreach InstanceID in DCMQuery(/softwareinstance[@name=$DB2ServerInstanceName]/@id) do
    DB2ServerInstanceID = InstanceID
    # use the first one
    break
  done

  ##############################################################################
  ## 3: Find the DB2 Server Installation object that hosts the trade database object
  ##    based on APPLICATION requirements (non-existing)
  ##############################################################################
  # find DB2Server Instances that comply with the application requirements
  var requirementType=”APPLICATION”
  var dbServerInstallationID
  ITSO_Find_Supporting_Installation_for_Module_Reqirements(requirementType, SoftwareModuleID, <null>,
dbServerInstallationID)


  ######################################################################
  ## 4: Find the Configuration srt on the remote installation
  ######################################################################
  var   templateID=DCMQuery(/softwareinstallation[@id=$dbServerInstallationID]/@resourcetemplateid)

  ## build an array of all templates related to the DBServerInstallation
  array IDs
  IDs[0]=templateID
  var j=arraysize(IDs)
  var i=0
  var end
  while Jython[int(i) < int(j) ] do
    var currentTemplateID=IDs[i]
```

```
    foreach ID in DCMQuery(/softwareresourcetemplate[@parenttemplateid =
$currentTemplateID]/@id[orderBy@id]) do
      end=arraysize(IDs)
      IDs[end] = ID
    done
    j = arraysize(IDs)
    i = Jython[int(i) + 1]
  done

  var instanceTemplateID
  var configurationTemplateID

  foreach templateID in IDs do
    var templateType=DCMQuery(/softwareresourcetemplate[@id=$templateID]/@softwareresourcetype)
    var templateName=DCMQuery(/softwareresourcetemplate[@id=$templateID]/@name)
    array childs = DCMQuery(/softwareresourcetemplate[@parenttemplateid=$templateID]/@id[orderBy@id])

    foreach ID in childs do
      var childType=DCMQuery(/softwareresourcetemplate[@id=$ID]/@softwareresourcetype)
      var childName=DCMQuery(/softwareresourcetemplate[@id=$ID]/@name)
      if Jython[templateType == swInstanceTypeID and childType == swConfigurationTypeID] then
        instanceTemplateID=templateID
        configurationTemplateID=ID
        break
      endif
    done

    if Jython[configurationTemplateID] then
      break
    endif
  done

  var dbServerInstanceResourceSRTID
  var dbServerConfigurationResourceSRTID

  if Jython[configurationTemplateID and configurationTemplateID != ""] then
    dbServerInstanceResourceSRTID=instanceTemplateID
    dbServerConfigurationResourceSRTID=configurationTemplateID
  else
    throw exception "could not find the configuration template as child of instance"
  endif

  ######################################################################
  ## 5: Get the parameters needed to catalog the database and instance
  ######################################################################
  var
databaseInstanceName=DCMQuery(/softwareresourcetemplate[@id=$dbServerInstanceResourceSRTID]/templateparam[
@name="resource-name"]/@value)
  if Jython[ not databaseInstanceName ] then
    msg=Jython[message + "no value found for parameter resource-name in SRT " +
dbServerInstanceResourceSRTID]
    log error msg
    throw parameterValidationException msg
  endif
```

```
  var
databaseInstancePort=DCMQuery(/softwareresourcetemplate[@id=$dbServerInstanceResourceSRTID]/templateparam[
@name="port"]/@value)
  if Jython[ not databaseInstancePort ] then
    msg=Jython[message + "no value found for parameter port in SRT " + dbServerInstanceResourceSRTID]
    log error msg
    throw parameterValidationException msg
  endif

  var dbServerServerID=DCMQuery(/softwareinstallation[@id=$dbServerInstallationID]/@managedsystemid)
  # get the first (managed) ip addess. If no managed addresses exist, use the managementaddress
  var dbServerServerIPAddress
  foreach ip in DCMQuery(/server[@id=$dbServerServerID]/nic/networkinterface/@ipaddress[orderBy@managed])
do
    dbServerServerIPAddress=ip
    break
  done

  var
databaseName=DCMQuery(/softwareresourcetemplate[@id=$dbServerConfigurationResourceSRTID]/templateparam[@na
me="resource-name"]/@value)
  if Jython[ not databaseName ] then
    msg=Jython[message + "no value found for parameter resource-name in SRT " + dbServerResourceSRTID]
    log error msg
    throw parameterValidationException msg
  endif

  var
databaseUserSapDomain=DCMQuery(/softwareresourcetemplate[@id=$dbServerConfigurationResourceSRTID]/template
param[@name="trade_dbuser_SAP_domain"]/@value)
  if Jython[ not databaseUserSapDomain ] then
    msg=Jython[message + "no value found for parameter trade_user_SAP_doamin in SRT " +
dbServerConfigurationResourceSRTID]
    log error msg
    throw parameterValidationException msg
  endif

  var
databaseUserSapContext=DCMQuery(/softwareresourcetemplate[@id=$dbServerConfigurationResourceSRTID]/templat
eparam[@name="trade_dbuser_SAP_context"]/@value)
  if Jython[ not databaseUserSapContext ] then
    msg=Jython[message + "no value found for parameter trade_user_SAP_context in SRT " +
dbServerConfigurationResourceSRTID]
    log error msg
    throw parameterValidationException msg
  endif

  var
databaseUserSapSearchkey=DCMQuery(/softwareresourcetemplate[@id=$dbServerConfigurationResourceSRTID]/templ
ateparam[@name="trade_dbuser_SAP_searchkey"]/@value)
  if Jython[ not databaseUserSapSearchkey ] then
    msg=Jython[message + "no value found for parameter trade_user_SAP_serachkey in SRT " +
dbServerConfigurationResourceSRTID]
```

```
    log error msg
    throw parameterValidationException msg
  endif

  ######################################################################
  ## 6: Call the scriptlet to perform the db2 updates on the DB2 Client
  ######################################################################
  # execute the db2 catalog commands
  if Jython[simulation.lower() == “yes” ] then
    log info Jython[“<b>simulation: </b> ... executing scriptlet to catalog node and database </b>”]
    log info Jython[“     databaseInstanceName: “ + databaseInstanceName]
    log info Jython[“     databaseInstancePort: “ + databaseInstancePort]
    log info Jython[“     dbServerServerNodeName: “ + dbServerServerNodeName]
    log info Jython[“     dbServerServerIPAddress: “ + dbServerServerIPAddress]
    log info Jython[“     databaseName: “ + databaseName]
    log info Jython[“     databaseAliasName: “ + databaseAliasName]
  else
    scriptlet (databaseInstanceName, databaseInstancePort, dbServerServerNodeName,
dbServerServerIPAddress,databaseName, databaseAliasName ) language = bash target =
DCMQuery(/server[@id=$ServerID]/@id) credentialskey=”default”  <<EOFDB2
      function do_db2 {
        db2cmd_prefix=”db2cmd -i -w -c “
        cmd2go=”${db2cmd_prefix} ${1}”
        msg=‘${cmd2go}‘
        rc=$?
        TIOlog “info” “(${rc})  command:’${cmd2go}’  returned:’${rc}’”
        if [ ${rc} -gt ${2} ]; then
          TIOthrow “databaseCatalogError” “(${rc}) command:${1}”
        fi
        return ${rc}
      }
      accepted_return_code=4
      db2command=”db2 catalog tcpip node ${dbServerServerNodeName} remote ${dbServerServerIPAddress}
server ${databaseInstancePort}”
      do_db2 “${db2command}” “${accepted_return_code}”
      db2command=”db2 catalog database ${databaseName} as ${databaseAliasName} at node
${dbServerServerNodeName} authentication server”
      do_db2 “${db2command}” “${accepted_return_code}”
      return 0
    EOFDB2
  endif

  ######################################################################
  ## 7: Save the cataloging parameters in the SRT
  ######################################################################
  var
sourceTemplate=DCMQuery(/softwareresourcetemplate[@id=$SoftwareResourceTemplateID]/sourcesoftwareresourcet
emplate/@id)
  array clonedTemplates =
DCMQuery(/softwareresource[@managedsystemid=$ServerID]/softwareresourcetemplate[@sourcetemplateid=$sourceT
emplate and @id != $SoftwareResourceTemplateID]/@id[orderBy@id])
  # use the last one
  i=arraysize(clonedTemplates)
  i=Jython[int(i) - 1]
```

```
  var jdbcSoftwareResourceTemplateID=clonedTemplates[i]

  DCMInsert parent=DCMQuery(/softwareresourcetemplate[@id=$jdbcSoftwareResourceTemplateID]/@id) <<EOFXML
       <template-param name="trade_dbuser_SAP_searchkey" value="$databaseUserSapSearchkey"
is-changeable="true" />
  EOFXML

  DCMInsert parent=DCMQuery(/softwareresourcetemplate[@id=$jdbcSoftwareResourceTemplateID]/@id) <<EOFXML
       <template-param name="trade_dbuser_SAP_context" value="$databaseUserSapContext"
is-changeable="true" />
  EOFXML

  DCMInsert parent=DCMQuery(/softwareresourcetemplate[@id=$jdbcSoftwareResourceTemplateID]/@id) <<EOFXML
       <template-param name="trade_dbuser_SAP_domain" value="$databaseUserSapDomain" is-changeable="true"
/>
  EOFXML

catchall exception
  ######################################################################
  ## 8: handle errors
  ######################################################################
  msg = Jython[message + exception]
  log error msg
  rethrow
endtry
```

10.8 Installing the Trade application

Now it is time to install the Trade application modules and create the required resources in the servlet engine.

According to the Trade installation instructions (see 10.2.2, “Trade3 installation instructions” on page 284), this involves the successful completion of the following steps:

1. Copy the trade3install.zip file to the target server and unpack it.
2. Define resources by sourcing in the WebSphere Application Server environment and run the command:

   ```
   ws_ant -f Trade3.xml installResources
   ```

3. Install the Trade application by sourcing in the WebSphere Application Server environment and run the command:

   ```
   ws_ant -f Trade3.xml installApp
   ```

4. Restart the WebSphere Application server (server1).
5. Populate the database.
6. Reset the application.

Even though not documented, the Trade3.xml file includes the necessary control statements for stopping and starting the WebSphere Application Server. In our implementation, we will take advantage of this. In addition, the installation instructions specify that application reset and database population should be performed manually, by accessing specific Web pages. In our implementation, we will use the **wget** command to complete these steps automatically from our workflows.

By far, the biggest challenge in this installation is the two install steps. Both prompts the user for input; and naturally this does not fit well with an automated process. To overcome these challenges, we could re-write the trade3.xml to remove the prompts and add our installation specific parameters; however, to demonstrate the use of expect, we have chosen an approach in which the installation is driven by an expect script that answers the prompts. The script will be generated as part of the provisioning process, and will include all the local customization to fit the environment in which Trade is installed.

Since our workflows will generate the expect script, we also chose to generate scripts for all other deployment activities (start, stop, and uninstall) to have a consistent and reusable interface. This approach moves a lot of the work and complexity from individual workflows to the single one, responsible for the installation.

In the following, we make the assumption that the capabilities of the target servers are:

OS	os.family=Windows 2000 Server SP4
SERVLET_ENGINE	servlet.version=2.0

and we will set up requirements in the Software Product and Software Package that reflect these assumptions.

10.8.1 The Trade Application Module Software Product

For the creation of a Trade application instance, we define a Software Product named Trade Application Module. Through the Software Product definition, we will link to a Software Package (named Trade Application Module Installable) that holds information about the installation files, and by including the proper Software Resource Templates in the Software Product, we ensure that the expected hierarchy of DCM objects will be created and named according to our standards.

The properties of the Trade Application Module software product are shown in Figure 10-39.

Figure 10-39 Trade Application Module properties

In addition, we will associate special capabilities with the Software Product. These capabilities will be applied to the target server upon installation, and thus can be used later to ensure that a Trade application instance exists. To ensure that the required servlet server functionality is available on the servers where the Trade Application Module will be installed, we will also add SERVLET_ENGINE related requirements to the Software Product definition.

Capabilities and requirements

The capabilities we assign to the Trade Application Module are:

APPLICATION	software.title	TRADE
APPLICATION	software.name	EAR
APPLICATION	software.vendor	IBM
APPLICATION	software.version	3.1

These capabilities will allow us, at a later point, to reference the Trade Application Module installation as non-existing requirements for other components of the Trade Automation Package. This way, we do not waste time installing modules that rely on the database if the database has not yet been installed.

By specifying the following requirements for our Trade Application Module Software Product, we ensure that all the necessary prerequisites have been installed prior to installation of the Trade Application Module:

SERVLET_ENGINE	servlet.version	2.0
DATABASE	database.family	DB2Client
DATABASE	database.version	8.2
APPLICATION	software.title	TRADE
APPLICATION	software.name	EAR

APPLICATION	software.vendor	IBM
APPLICATION	software.version	3.1

The SERVLET_ENGINE requirement will, in our case, be honored by the WebSphere Application Server installation, and the DATABASE requirements will be resolved to the DB@ Client installation. Both of these components are prerequisites for the Trade application, but in addition, we want to make sure that the Trade database has been cataloged in the DB2 Client environment, which is why we also require the APPLICATION (software.name=JDBC) to be installed on the target server where the Trade Application Module will be installed.

Note: In a real-life implementation, the user should be provided instructions on how to modify the requirements that are set up by default for the Trade Application Module in order to integrate it into a specific environment.

Figure 10-40 shows capabilities and requirements for the Trade Application Module Software Product.

Requirements and Capabilities

Requirement Name	Requirement Type	Requirement Values
software.vendor	APPLICATION	IBM
software.version	APPLICATION	3.1
database.family	DATABASE	DB2Client
database.version	DATABASE	8.2
servlet.version	SERVLET_ENGINE	2.0
software.name	APPLICATION	JDBC
software.title	APPLICATION	TRADE

Capability Type
APPLICATION

Capability Name	Capability Value
software.name	EAR
software.vendor	IBM
software.title	TRADE
software.version	3.1

Figure 10-40 Trade Application Module capabilities and requirements

Software Package(s)

Now, to help determine which files to use during the installation, we created a Software Package named Trade Application Module Installable.

The only file that is referenced by the Software Package is the trade3Install.zip, which is the installation archive for the Trade application. This file is included in the Automation Package (in the repository subdirectory) and will when the Automation package is installed, be placed in the %TIO_HOME%/../SWrepository/trade3 directory. This is achieved by the addition of the XML statements in Example 10-32 to the tc-driver.xml file for the Trade Automation Package.

Example 10-32 Incorporating the trade3Install.zip file in the automation package

```
<item name="repository/trade3Install.html" action="copy-file">
      <param name="editable" value="false" />
      <param name="dest.path"
value="${repository}/trade3/trade3Install.zip" />
      <param name="chmod" value="777" />
</item>
```

Important: The addition of the <item> definition above to the tc-driver.xml file has to be performed manually because the APDE environment has no way of knowing where to copy the files in the repository subdirectory of the automation package.

Even though the trade3Install.zip file can be used to install Trade on any platform, we added, for the sake of the example, a requirement for the Software Package, specifying the following:

OS	os.family	Windows 2000 Server SP4

Figure 10-41 shows the Trade Application Module Installable properties.

Figure 10-41 Trade Application Module Installable software package properties

Variables

For this software package, we will also define a set of variables, which will be used to help find the installation directory of the underlying IBM WebSphere Application Server installation, as described in 10.5.2, “Finding installation directories” on page 324. For the Trade Application Module Installable, these variables are created with the standard values shown in Figure 10-42, and it is expected that they are customized during installation of the Trade Automation Package.

Figure 10-42 Trade Application Module Installable software package variables

The Trade_Application_Module_Installable_Driver Device Driver

To control the installation of the Trade Application Module Installable, we need to assign a device driver that, at a minimum, contains a workflow that implements the SoftwareInstallable.Install logical device operation. In addition, we added a workflow for implementation of the SoftwareInstallable.Distribute LDO, which will be used to transfer and unpack the trade3Install.zip on the target server.

Example 10-33 shows the XML statements embedded in the tc-driver.xml file for the entire Trade Automation Package, which are used to define the device driver to the DCM. The device driver we assign to the Trade Application Module Installable will be named Trade_Application_Module_Installable_Driver. Details regarding the workflows embedded in the device driver are provided in “Trade_Application_Module_Installable_Install” on page 429 and “Trade_Application_Module_Installable_Distribute” on page 432.

Example 10-33 XML code to define the Trade_Application_Module_Installable_Driver

```
<device-model name="Trade_Application_Module_Installable_Driver"
category="Trade">
   <workflow name="Trade_Application_Module_Installable_Install" />
   <workflow name="Trade_Application_Module_Installable_Distribute" />
</device-model>
```

Before we look at the details of the workflows, we need to take a look at the Software Resource Templates of the Trade Application Module Software Product to gain knowledge about the customization parameters that will be available to the installation workflows.

Software Resource Templates

In order to create Trade application instance in the data center model as children of the underlying SERVLET_ENGINE Software Installation object (the WebSphere Application Server installation), we add a FOREIGN_CONFIGURATION Software Resource Template to the Trade Application Module INSTALLATION Software Resource Template, effectively creating a Software Resource Template hierarchy of the Trade Application Module Software Product that looks like this:

INSTALLATION
  FOREIGN_CONFIGURATION
    INSTANCE
      CONFIGURATION

As a result, the INSTALLATION template will be used to create a Software Installation object on the target server, and associate this with the Trade Application Module Software Product. As children of this Software Installation object TIO/TPM will clone the remainder of the SRT hierarchy (the FOREIGN_CONFIGURATION template and all its children) to the new Software Installation, the workflow that implements the SoftwareInstallable.Install LDO is responsible for inspecting the SRT hierarchy and takes appropriate action(s).

Device driver associations

To control the behavior of the resources that are created from the Software Resource Templates, we must associate a device driver to each of them.

For the Trade Application Module Software Package software model, we associate the device drivers and related workflows implementing the required logical operations, as shown in Table 10-9.

Table 10-9 Trade Application Module related device drivers and workflows

SRT	Type	Name	Workflow
INSTALLATION	Device driver	Trade_Application_Module_Intallation_Driver	
	LDOs	SoftwareInstallation .Uninstall	Trade_Application_Module_Installation_Uninstall

SRT	Type	Name	Workflow
INSTANCE	Device driver	Trade_Application_Module_Instance_Driver	
	LDOs	SoftwareInstance .RemoveInstance	Trade_Application_Module_Instance_RemoveInstance
		SoftwareInstance .Start	Trade_Application_Module_Instance_Start
		SoftwareInstance .Stop	Trade_Application_Module_Instance_Stop

Software resource naming

The naming of Software Resources that will be created based on the templates can be controlled by creating a template parameter named *resource-name*, and providing a value that matches your requirements. Figure 10-43 on page 423 shows the final Software Resource Template hierarchy for the Trade Application Module, and here we see that the names that will be assigned to the objects created by the templates are:

Installation Default Trade3 Installation
Foreign_Configuration
Trade Foreign configuration
Instance trade3 ear module instance
Configuration Trade Application configuration

Archived

Software Resource Template parameters

To provide for the configuration of the Apache server created during the installation of the Trade Web Module software package, we will need to provide customization parameters to determine how to configure the Apache Server and update the index.html file. The needed parameters and their meaning are listed in Table 10-10.

Table 10-10 Trade Application Module related SRT parameters

SRT	Name	Value
INSTALLATION	resource-name	Name to be assigned to new installations
FOREIGN_CONFIGURATION	resource-name	Name to be assigned to new foreign configurations
	targetResourceName	Name of the DCM resource to host the instance (server1)
	targetResourceType	Type of DCM resource to host the instance (INSTANCE)
INSTANCE	resource-name	Name to be assigned to new instances
	unpackDir	Directory in which the installation archive will be unpacked
	serverPort	Port number of the application server
CONFIGURATION	resource-name	Name to be assigned to new configurations
	buildDBURL	URL fragment (relative to the server name and port) used to initialize the database
	resetTradeURL	URL fragment (relative to the server name and port) used to reset the Trade application
	goTradeURL	URL fragment (relative to the server name and port) used to run the Trade application

The default values provided in the Software Resource Templates upon installation of the Trade Automation Package are shown in Figure 10-43 on page 423.

Archived

Configuration Templates

Configuration Templates

Default Trade3 Installation

Software Resource Type: Installation Device Model: Trade_Application_Module_Installation_Driver

Parameter	Value
resource-name	Trade Application Module Installation

Trade Foreign configuration

Software Resource Type: Foreign configuration

Parameter	Value
resource-name	Trade Foreign Configuration
targetResourceName	server1
targetResourceType	INSTANCE

trade3 ear module instance

Software Resource Type: Instance Device Model: Trade_Application_Module_Instance_Driver

Parameter	Value
resource-name	Trade Application Instance
unpackDir	/tmp/trade3
serverPort	9080

Trade Application configuration

Software Resource Type: Configuration

Parameter	Value
resource-name	Trade EAR Configuration
buildDBURL	/trade/config?action=buildDB
resetTradeURL	/trade/config?action=resetTrade
goTradeURL	/trade/app

Figure 10-43 Trade Application Module Software Resource Templates

Package overview

Based on the parameters and attributes we have defined for the Trade Application Module Software Product, along with the related Software Package and the embedded Software Resource Templates, we have created the software model for:

- Installation of the Trade application modules and related resources
- Initialization of the Trade application
- Operation (start and stop) of the Trade application

in a way that will be easily customizable for the potential users of the Trade Automation Package. Figure 10-44 shows a depiction of the resources involved and their relationships.

Figure 10-44 Trade Application Module deployment model

Creating the software objects from the Automation Package

The XML deck needed to define our Software Product and Software Package, along with the embedded file, variable, Software Resource Template, and device driver definitions is shown in Example 10-34 on page 425.

To automatically import this XML file as part of the automation package installation, you will have to add an item definition for the XML file using the **import** function on the <dcm> section in the tc-driver.xml file for the automation package:

```
<dcm>
   <item name="xml/Trade_Application_Module.xml" action="import"/>
</dcm>
```

Example 10-34 Trade_Application_Module.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE datacenter SYSTEM "file:../xml/xmlimport.dtd">

<datacenter>

    <software-module name="Trade Application Module"
                     version="3.1"
                     vendor="IBM"
                     title="IBM Trade3 WAS Application"
                     description="Trade sample WAS EAR"
                     is-draft="false">
            <software-capability name="software.name" value="EAR" />
            <software-capability name="software.vendor" value="IBM" />
            <software-capability name="software.title" value="TRADE" />
            <software-capability name="software.version" value="3.1" />
            <supported-requirement-type value="APPLICATION" />
            <software-requirement name="software.version"
                                  type="APPLICATION"
                                  enforcement="MANDATORY"
                                  hosting="true"
                                  accept-non-existing="false">
                     <software-requirement-value value="3.1" />
            </software-requirement>
            <software-requirement name="software.vendor"
                                  type="APPLICATION"
                                  enforcement="MANDATORY"
                                  hosting="true"
                                  accept-non-existing="false">
                     <software-requirement-value value="IBM" />
            </software-requirement>
            <software-requirement name="database.family"
                                  type="DATABASE"
                                  enforcement="MANDATORY"
                                  hosting="true"
                                  accept-non-existing="false">
                     <software-requirement-value value="DB2Client" />
            </software-requirement>
            <software-requirement name="database.version"
                                  type="DATABASE"
                                  enforcement="MANDATORY"
                                  hosting="true"
                                  accept-non-existing="false">
                     <software-requirement-value value="8.2" />
            </software-requirement>
            <software-requirement name="servlet.version"
                                  type="SERVLET_ENGINE"
                                  enforcement="MANDATORY"
                                  hosting="true"
                                  accept-non-existing="false">
                     <software-requirement-value value="2.0" />
            </software-requirement>
            <software-requirement name="software.name"
```

Archived

```
                        type="APPLICATION"
                        enforcement="MANDATORY"
                        hosting="true"
                        accept-non-existing="false">
                <software-requirement-value value="JDBC" />
        </software-requirement>
        <software-requirement name="software.title"
                        type="APPLICATION"
                        enforcement="MANDATORY"
                        hosting="true"
                        accept-non-existing="false">
                <software-requirement-value value="TRADE" />
        </software-requirement>

        <installable-package name="Trade Application Module Installable"
                        is-device-model="Trade_Application_Module_Installable_Driver"
                        locale="en_US"
                        description="Windows installable for Trade3"
                        version="3.1"
                        priority="0"
                        file-repository="LocalFileRepository"
                        status="tested">
                <file name="trade3install.zip" path="/trade3" />

                <property component="DEPLOYMENT_ENGINE"
                        name="install_dir_softwareresource_type"
                        value="INSTANCE" />
                <property component="DEPLOYMENT_ENGINE"
                        name="install_dir_parameter_name"
                        value="install_dir" />
                <property component="DEPLOYMENT_ENGINE"
                        name="install_dir_custom_default"
                        value="c:/ibm/WebSphere" />
                <property component="DEPLOYMENT_ENGINE"
                        name="install_dir_system_default"
                        value="c:\Program Files\WebSphere" />

                <software-requirement name="os.family"
                                type="OS"
                                enforcement="MANDATORY"
                                hosting="true"
                                accept-non-existing="false">
                        <software-requirement-value value="Windows 2000 Server SP4" />
                </software-requirement>
        </installable-package>

        <software-resource-template name="Default Trade3 Installation"
                                software-resource-type="INSTALLATION"

software-resource-device-model="Trade_Application_Module_Installation_Driver"
                                multiplicity-type="Unspecified"
                                software-configuration-type="Regular"
                                is-selected="true">
          <software-resource-template name="Trade Foreign configuration"
```

```
                                    software-resource-type="FOREIGN-CONFIGURATION"
                                    multiplicity-type="Unspecified"
                                    software-configuration-type="Regular"
                                    is-selected="true">
            <software-resource-template name="trade3 ear module instance"
                                        software-resource-type="INSTANCE"

software-resource-device-model="Trade_Application_Module_Instance_Driver"
                                        multiplicity-type="Unspecified"
                                        software-configuration-type="Regular"
                                        is-selected="true">
               <software-resource-template name="Trade Application configuration"
                                            software-resource-type="CONFIGURATION"
                                            multiplicity-type="Unspecified"
                                            software-configuration-type="Regular"
                                            is-selected="true">
                     <template-param name="resource-name"
                                        value="Trade EAR Configuration"
                                        is-changeable="true" />
                     <template-param name="buildDBURL"
                                        value="/trade/config?action=buildDB"
                                        is-changeable="true" />
                     <template-param name="resetTradeURL"
                                        value="/trade/config?action=resetTrade"
                                        is-changeable="true" />
                     <template-param name="goTradeURL"
                                        value="/trade/app"
                                        is-changeable="true" />
               </software-resource-template>
               <template-param name="resource-name"
                              value="Trade Application Instance"
                              is-changeable="true" />
                <template-param name="unpackDir"
                           value="/tmp/trade3"
                           is-changeable="true" />
                <template-param name="serverPort"
                           value="9080"
                           is-changeable="true" />
            </software-resource-template>
            <template-param name="resource-name"
                           value="Trade Foreign Configuration"
                           is-changeable="true" />
            <template-param name="targetResourceName"
                           value="server1"
                           is-changeable="true" />
            <template-param name="targetResourceType"
                           value="INSTANCE"
                           is-changeable="true" />
          </software-resource-template>
          <template-param name="resource-name"
                        value="Trade Application Module Installation"
                        is-changeable="true" />
        </software-resource-template>
```

```
    </software-module>

</datacenter>
```

10.8.2 The Trade_Application_Module_Installable_Driver

By default, the logical device operations for the Software Installable object type are Install and Distribute, and we will support both of them in the Trade_Application_Module_Installable_Driver, as shown in Figure 10-45.

Figure 10-45 Trade_Application_Module_Installable_Driver overview

From Figure 10-45, it should be obvious that the main functionality of generating scripts for installing, uninstalling, and operating the Trade application is handled by the Trade_Application_Add_Instance workflow, and the supporting workflows are used to determine the hosting environment and to copy and unpack the trade3Install.zip installation archive to the target server.

Archived

The definition of the Trade_Application_Module_Installable_Driver is depicted in Figure 10-46.

Figure 10-46 Trade_Application_Module_Installable_Driver

The XML statements included in the tc-driver.xml file used to create the device driver during the installation of the Trade Automation Package is shown in Example 10-35.

Example 10-35 Trade_Application_Module_Installable_Driver xml definitions

```
<device-model name="Trade_Application_Module_Installable_Driver"
category="Trade">
    <workflow name="Trade_Application_Module_Installable_Distribute" />
    <workflow name="Trade_Application_Module_Installable_Install" />
</device-model>
```

Trade_Application_Module_Installable_Install

The main workflow for installation of the Trade EAR module is Trade_Application_Module_Installable_Install. This workflow controls and serializes the activities involved with provisioning the Trade EAR module, and will, by calling other workflows and LDOs, ensure that the specific actions are being performed.

The outline of this workflow is:

1. Initialize.
2. Gather control information for navigation and error handling.
3. Find the Software Installation that implements the SERVLET-ENGINE requirement.
4. Commit the current installation (which has been created by the SoftwareInstallable.Install LDO in a *pending* state. If the implementation workflow fails, the pending state will be used to determine if the software resource should be removed from the DCM). This committal is necessary in order to be able to reference the Software Installation object from the

subsequent workflows called prior to termination of the implementation workflow.

5. Use the ITSO_HostingEnvironment_Host workflow (see 10.10.4, “ITSO_HostingEnvironment_Host” on page 500) to create the foreign resources parented by the FOREIGN-RESOURCE Software Resource Template of the Trade Application Module. During this creation, the Trade Application Module specific device drivers will be assigned to the new resources, as specified in the Software Resource Templates.
6. For each newly created Software Instance, call the Trade_Application_Add_Instance workflow to create the instance and ultimately install the Trade EAR module.

> **Note:** In this situation, we cannot use the SoftwareInstallation.AddInstance LDO, since we are not in control of the implementation provided when the underlying WebSphere Application Server instance was deployed.

7. If the workflow receives an exception, we have to delete the newly created software objects and uncommit the current Software Installation object. In this way, the SoftwareInstallable.Install LDO will take care of the removal of the Software Installation object when the workflow terminates.

For details regarding the Trade_Application_Module_Installable_Install workflow, please see Example 10-36.

Example 10-36 Trade_Application_Module_Installable_Install workflow

```
workflow Trade_Application_Module_Installable_Install(in SoftwareID, in DeviceID, in
SoftwareResourceTemplateID) implements SoftwareInstallable.Install LocaleInsensitive

#####################################################################
## 1: Initialize
#####################################################################
var simulation
Trade_Get_Simulation(simulation)

var workflowName=”Trade_Application_Module_Installable_Install”
var message=Jython[“<b>” + workflowName + “:</b>”]
var msg

var SoftwareInstallationID

try
  #####################################################################
  ## 2: Gather data for control and logging
  #####################################################################
  var SoftwareInstallableID=SoftwareID
  var ServerID=DeviceID
```

```
  SoftwareInstallationID=DCMQuery(/softwareinstallation[@managedsystemid=$ServerID and
@softwareproductid=$SoftwareInstallableID and @pending="Y"]/@id)
  if Jython[not SoftwareInstallationID or SoftwareInstallationID == ""] then
    SoftwareInstallationID=DCMQuery(/softwareinstallation[@managedsystemid=$ServerID and
@softwareproductid=$SoftwareInstallableID]/@id)
  endif
  var SoftwareModuleID=DCMQuery(/softwareinstallation[@id=$SoftwareInstallationID]/@softwaremoduleid)
  var installationResourceType=DCMQuery(/softwareresourcetype[@name="INSTALLATION"]/@id)
  var instanceResourceType=DCMQuery(/softwareresourcetype[@name="INSTANCE"]/@id)
  var foreignResourceType=DCMQuery(/softwareresourcetype[@name="FOREIGN-CONFIGURATION"]/@id)
  var
installationSRTID=DCMQuery(/softwareinstallation[@id=$SoftwareInstallationID]/softwareresourcetemplate[@so
ftwareresourcetype=$installationResourceType]/@id)
  var foreignSRTID=DCMQuery(/childsoftwareresourcetemplate[@parenttemplateid=$installationSRTID]/@id)

  ######################################################################
  ## 3: Find Supporting Installation for Module SERVLET_ENGINE Reqirements
  ######################################################################
  var requirementType="SERVLET_ENGINE"
  var supportingSoftwareInstallationID
  ITSO_Find_Supporting_Installation_for_Module_Reqirements(requirementType, SoftwareModuleID, ServerID,
supportingSoftwareInstallationID)

  ####################################################################
  ## 4: Commit the new Installation before creating foreign resources
  ####################################################################
  # update the pending flag of the previously created resources

java:com.ibm.tivoli.orchestrator.datacentermodel.helper.SoftwareHelper#updateResourceFlag(SoftwareInstalla
tionID, "false")

  ######################################################################
  ## 5: Create the FOREIGN resources
  ######################################################################
  log info Jython["About to call ITSO_HostingEnvironment_Host(" + supportingSoftwareInstallationID + "," +
SoftwareModuleID + "," + foreignSRTID + ")"]
  array objects
  ITSO_HostingEnvironment_Host(supportingSoftwareInstallationID, SoftwareModuleID, foreignSRTID, objects)

  ######################################################################
  ## 6: Install the Trade Application
  ######################################################################
  foreach objectID in objects do
    var srtID=DCMQuery(/softwareresource[@id=$objectID]/@resourcetemplateid)
    var targetResourceID=DCMQuery(/softwareresource[@id=$objectID]/@parentresourceid)
    log info Jython["..About to call: Trade_Application_Add_Instance(" + targetResourceID + ", " + srtID +
")"]
    Trade_Application_Add_Instance(targetResourceID, srtID)
  done

catchall
  ######################################################################
  ## 7: error handling
  ######################################################################
```

```
  msg = Jython[message + “problem installing...”]
  log error msg
  # delete foreign resources
  foreach objectID in objects do
    var objID=DCMQuery(/softwareresource[@id=$objectID]/@id)
    if Jython[ objID != None] then
      DCMDelete(/softwareresource[@id=$objectID]/@id)
    endif
  done
  # uncommit the Installation if we failed

java:com.ibm.tivoli.orchestrator.datacentermodel.helper.SoftwareHelper#updateResourceFlag(SoftwareInstalla
tionID, “true”)
  rethrow
endtry
```

Trade_Application_Module_Installable_Distribute

This is the workflow that implements the SoftwareInstallable.Install logical device operation for the Trade Application Module. This is used to distribute the trade3Install.zip archive to the target system, where it can be unpacked and used to install the Trade EAR module and create the WebSphere resources required by Trade.

The SoftwareInstallable.Install LDO is called by the Trade_Application_Add_Instance workflow (see “Trade_Application_Add_Instance” on page 443), which in turn is called by the Trade_Application_Module_Installable_Install workflow described above.

The outline of the Trade_Application_Module_Installable_Distribute is:

1. Initialize and set up variables for error handling.
2. Gather basic data from the DCM regarding FileRepositories, file names, and so on.
3. Copy and unpack the archive using the ITSO_Copy_And_Unpack_Archive_From_Repository utility workflow. For details regarding this utility workflow, please see 10.10.8, “ITSO_Copy_And_Unpack_Archive_From_Repository” on page 509.
4. Throw any received exceptions back to the caller.

Archived

Example 10-37 shows the content of the Trade_Application_Module_Installable_Distribute workflow.

Example 10-37 Trade_Application_Module_Installable_Distribute workflow

```
workflow Trade_Application_Module_Installable_Distribute(in SoftwareID, in DeviceID) implements
SoftwareInstallable.Distribute LocaleInsensitive

####################################
## 1: Initialize
####################################
var simulation
Trade_Get_Simulation(simulation)

var workflowName="Trade_Application_Module_Installable_Distribute"
var message = Jython["<b> " + workflowName + "</b>: "]
var msg

try
  ####################################
  ## 2: Gather data
  ####################################
  var SoftwareInstallableID=SoftwareID
  var SoftwareModuleID =
DCMQuery(/softwareinstallable[@id=$SoftwareInstallableID]/softwareinstallationmechanism/softwaremodule/@id
)
  var softwareresourcetypeID=DCMQuery(/softwareresourcetype[@name="INSTALLATION"]/@id)
  var
UnpackDir=DCMQuery(/softwareinstallable[@id=$SoftwareInstallableID]/softwareinstallation[@managedsystemid=
$DeviceID and
@pending="Y"]/softwareresourcetemplate[@softwareresourcetype=$softwareresourcetypeID]/templateparam[@name=
"UnpackDir"]/@value)
  UnpackDir= Jython[UnpackDir or "/tmp/trade3"]
  var repository_ID = DCMQuery(/softwareinstallable[@id=$SoftwareInstallableID]/filerepository/@id)

  ####################################
  ## 3: Transfer and unpack the file
  ####################################
  if Jython[simulation == "yes"] then
    log info Jython["<b>simulation:...about to call ITSO_Copy_And_Unpack_Archive_From_Repository(" +
SoftwareInstallableID + ", " + DeviceID + ", " + UnpackDir + ")"]
  else
    ITSO_Copy_And_Unpack_Archive_From_Repository(SoftwareInstallableID, DeviceID, UnpackDir)
  endif
catchall exception
  ####################################
  ## 4: error handling
  ####################################
  msg = Jython[message + " " + exception]
  log error msg
  rethrow
endtry
```

Archived

10.8.3 The Trade_Application_Module_Instance_Driver

During creation of the SoftwareInstance objects that represent the installation of the Trade EAR module installations, all of the instances will be associated with the Trade_Application_Module_Instance device driver. This device driver will support the following logical device operations:

- SoftwareInstance.Start
- SoftwareInstance.Stop
- SoftwareInstance.Remove

The intended functionality of these logical device operations should be self-explanatory from looking at the overview presented in Figure 10-47.

Figure 10-47 Trade_Application_Module_Instance_Driver overview

Figure 10-48 shows the workflow assignments to the LDOs of the Trade_Application_Module_Instance_Driver device driver.

Figure 10-48 Trade_Application_Module_Instance_Driver

In order to automatically create the Trade_Application_Module_Instance_Driver during installation of the Trade Automation Package, we ensure that the xml statements shown in Example 10-38 are present in the <device-models> section of the tcdriver.xml file for the Trade Automation Package.

Example 10-38 Trade_Application_Module_Instance_Driver XML definitions

```
<device-model name="Trade_Application_Module_Instance_Driver"
category="Trade">
    <workflow name="Trade_Application_Module_Instance_RemoveInstance"
/>
    <workflow name="Trade_Application_Module_Instance_Start" />
    <workflow name="Trade_Application_Module_Instance_Stop" />
</device-model>
```

In the following section, we will describe the functionality of the Trade_Application_Module_Instance_Driver workflows, which are:

- Trade_Application_Module_Instance_Start
- Trade_Application_Module_Instance_Stop
- Trade_Application_Module_Instance_RemoveInstance

Trade_Application_Module_Instance_Start

To start the Trade WebSphere Enterprise Application, we need a workflow that implements the SoftwareInstance.Start LDO.

The functionality of this workflow is pretty basic; after our installation procedure (“Trade_Application_Add_Instance” on page 443) has created the scripts, need to invoke the **ws_ant** command. When inspecting the trade3.xml provided in the trade3Install.zip installation archive, we notice that by issuing the **ws_ant -f Trade3.xml startTrade** command, we can start the application independently of the WebSphere server itself.

So, the main outline of the Trade_Application_Module_Instance_Start workflow, which is shown in its entirety in Example 10-39 on page 436, is:

1. Initialize and set up control information for error handling.
2. Retrieve the name of the start script and the working directory, which was saved in the Installation SRT by the Trade_Application_Add_Instance workflow.
3. Run a scriptlet that executes the start script.
4. Parse any exceptions back to the caller.
5. FInally, update the status information in the Data Center Model to reflect the start.

Example 10-39 Trade_Application_Module_Instance_Start workflow

```
workflow Trade_Application_Module_Instance_Start(in SoftwareInstanceID) implements SoftwareInstance.Start
LocaleInsensitive

################################################################
##  1: initialize
################################################################
var simulation
Trade_Get_Simulation(simulation)

var workflowName=”Trade_Application_Module_Instance_Stop”
var message=Jython[“<b>” + workflowName + “: </b>”]
var msg

try
  ################################################################
  ##  2: Get the script name and workdir from the instance srt
  ################################################################
  var ServerID=DCMQuery(/softwareinstance[@id=$SoftwareInstanceID]/@managedsystemid)
  # find the Instance SRT id
  var
SoftwareResourceTemplateID=DCMQuery(/softwareinstance[@id=$SoftwareInstanceID]/softwareresourcetemplate/@i
d)
  ## Get parameters
  var startScript=DCMQuery(/templateparam[@name=”startScript” and
@templateid=$SoftwareResourceTemplateID]/@value)
  var workDir=DCMQuery(/templateparam[@name=”workDir” and @templateid=$SoftwareResourceTemplateID]/@value)

  ################################################################
  ##  3: Start the Trade Application
  ################################################################
  if Jython[simulation == “yes”] then
    log info Jython[“<b>simulation: </b>...about to call scriptlet to stop trade...”]
    log info Jython[“              workDir: “ + workDir]
    log info Jython[“              startScript: “ + startScript]
  else
    scriptlet ( workDir, startScript) language=bash target=DCMQuery(/server[@id=$ServerID]/@id) <<EOF
      [ -z $OSTYPE ] && {
       OSTYPE=‘uname -s‘
      }
      case $OSTYPE in
       cygwin*|CYGWIN*)
        ;;
       AIX*|HP*|hpux*|Linux*|Solaris*|linux*)
       ;;
       *)
         TIOthrow ScriptExitException “${OSTYPE} - Unsupported Operating System”
       ;;
      esac

      cd ${workDir}
      ${startScript}
      return 0
```

```
    EOF
  endif
catchall exception
  ################################################################
  ##  4: handle errors
  ################################################################
  msg=Jython[message + exception]
  log error msg
  rethrow
endtry

################################################################
##  5: get and update the DCM status
################################################################
var status = DCMQuery(/softwareresource[@id=$SoftwareInstanceID]/softwarestate/@name)
var realstatus="Running"
if Jython[status != realstatus] then

java:com.ibm.tivoli.orchestrator.datacentermodel.helper.SoftwareHelper#updateResourceStatus(SoftwareInstan
ceID, realstatus)
endif
```

Trade_Application_Module_Instance_Stop

The workflow used to stop the Trade Application is similar to the one used for start; the only difference being that the parameter to be picked up from the INSTANCE Software Resource Template is the name of the stop script.

Please refer to the previous section for a high-level outline of the workflow, and to Example 10-40 for the details.

Example 10-40 Trade_Application_Module_Instance_Stop workflow

```
workflow Trade_Application_Module_Instance_Stop(in SoftwareInstanceID) implements SoftwareInstance.Stop
LocaleInsensitive

################################################################
##  1: initialize
################################################################
var simulation
Trade_Get_Simulation(simulation)

var workflowName="Trade_Application_Module_Instance_Stop"
var message=Jython["<b>" + workflowName + ": </b>"]
var msg

try
  ################################################################
  ##  2: Find the names of the script and the working directory
  ################################################################

  var ServerID=DCMQuery(/softwareinstance[@id=$SoftwareInstanceID]/@managedsystemid)
```

Archived

```
  # find the instance SRT id
  var
SoftwareResourceTemplateID=DCMQuery(/softwareinstance[@id=$SoftwareInstanceID]/softwareresourcetemplate/@i
d)
  # Get parameters
  var stopScript=DCMQuery(/templateparam[@name="stopScript" and
@templateid=$SoftwareResourceTemplateID]/@value)
  var workDir=DCMQuery(/templateparam[@name="workDir" and @templateid=$SoftwareResourceTemplateID]/@value)

  ################################################################
  ##  3: Stop the Trade application
  ################################################################
  if Jython[simulation == "yes"] then
    log info Jython["<b>simulation:</b> ...about to call scriptlet to stop trade..."]
    log info Jython["     workDir: " + (workDir or "unknown") ]
    log info Jython["     stopScript: " + (stopScript or "unknown")]
  else
    scriptlet ( workDir, stopScript) language=bash target=DCMQuery(/server[@id=$ServerID]/@id) <<EOF
      [ -z $OSTYPE ] && {
       OSTYPE=`uname -s`
      }
      case $OSTYPE in
       cygwin*|CYGWIN*)
       ;;
       AIX*|HP*|hpux*|Linux*|Solaris*|linux*)
       ;;
       *)
         TIOthrow ScriptExitException "${OSTYPE} - Unsupported Operating System"
       ;;
      esac
      cd ${workDir}
      ${stopScript}
      return 0
    EOF
  endif

catchall exception
  ################################################################
  ##  4: handle errors
  ################################################################
  msg=Jython[message + exception]
  log error msg
  rethrow
endtry

################################################################
##  5: get and update the DCM status
################################################################
var status = DCMQuery(/softwareresource[@id=$SoftwareInstanceID]/softwarestate/@name)
var realstatus="Not-Running"
if Jython[status != realstatus] then

java:com.ibm.tivoli.orchestrator.datacentermodel.helper.SoftwareHelper#updateResourceStatus(SoftwareInstan
ceID, realstatus)
```

```
endif
```

Trade_Application_Module_Instance_RemoveInstance

The workflow we need to remove the installation of the Trade application from the WebSphere Application server environment operates similar to the start and stop workflows, that is, it relies on a script provided by the installation procedure, which in turn invokes the **ws_ant** program on the target system.

The outline of the Trade_Application_Module_Instance_RemoveInstance workflow:

1. Initialize.
2. Find the INSTANCE Software Resource Template, and gather parameters for uninstallscript, workdir, and unpackdir.
3. Use the SoftwareInstance.Stop LDO to ensure that the Trade application has been stopped in the WebSphere Application Server environment.
4. Handle errors by returning all exceptions to the caller.

For a full listing of the workflow, please refer to Example 10-41.

Example 10-41 Trade_Application_Module_Instance_RemoveInstance workflow

```
workflow Trade_Application_Module_Instance_RemoveInstance(in SoftwareInstanceID) implements
SoftwareInstance.RemoveInstance LocaleInsensitive

################################################################
##  1: Initialize and gather base data for control and navigation
################################################################
var simulation
Trade_Get_Simulation(simulation)

var workflowName="Trade_Application_Module_Instance_RemoveInstance"
var message = Jython["<b>" + workflowName + ":</b> " ]
var msg
var ServerID=DCMQuery(/softwareinstance[@id=$SoftwareInstanceID]/@managedsystemid)

try
  ################################################################
  ##  2: find the instance template
  ################################################################
  var
SoftwareResourceTemplateID=DCMQuery(/softwareinstance[@id=$SoftwareInstanceID]/softwareresourcetemplate/@i
d)
  ## Get parameters
  var uninstallScript=DCMQuery(/templateparam[@name="uninstallScript" and
@templateid=$SoftwareResourceTemplateID]/@value)
  var workDir=DCMQuery(/templateparam[@name="workDir" and @templateid=$SoftwareResourceTemplateID]/@value)
  var unpackDir=DCMQuery(/templateparam[@name="unpackDir" and
@templateid=$SoftwareResourceTemplateID]/@value)
```

```
  ################################################################
  ##  3: Stop the instance
  ################################################################
  SoftwareInstance.Stop(SoftwareInstanceID)

  ################################################################
  ##  4: Run the uninstall script anad remove the unpack directory
  ################################################################
  if Jython[simulation == “yes”] then
    log info Jython[“<b>simulation: </b>...about to call scriptlet to uninstall trade...”]
    log info Jython[“               workDir: “ + (workDir or “unknown”)]
    log info Jython[“               uninstallScript: “ + (uninstallScript or “unknown”)]
    log info Jython[“               unpackDir: “ + (unpackDir or “unknown”)]
  else
    scriptlet ( workDir, uninstallScript, unpackDir) language=bash
target=DCMQuery(/server[@id=$ServerID]/@id) <<EOF
      [ -z $OSTYPE ] && {
        OSTYPE=‘uname -s‘
      }
      case $OSTYPE in
        cygwin*|CYGWIN*)
        ;;
        AIX*|HP*|hpux*|Linux*|Solaris*|linux*)
        ;;
        *)
            TIOthrow ScriptExitException “${OSTYPE} - Unsupported Operating System”
        ;;
      esac

      # do it
      cd ${workDir}
      ${uninstallScript}
      cd /
      if [ -e ${unpackdir} ]; then
        rm -rf ${unpackDir}
      fi
      return 0
    EOF
  endif
catchall exception
  ################################################################
  ##  4: handle errors
  ################################################################
  msg=Jython[message + exception]
  log error msg
  rethrow
endtry
```

Archived

10.8.4 The Trade_Application_Module_Installation_Driver

Once the Trade Application Module Installation object and the related foreign instances have been created, we need a way to remove it again. This is the responsibility of the SoftwareInstallation.Uninstall LDO, which is the only logical device operation from the Software Installation device model we will implement in the Trade_Application_Module_Installation_Driver device driver, as should be obvious from looking at the driver overview in Figure 10-49.

Figure 10-49 Trade_Application_Module_Installation_Driver overview

Figure 10-50 depicts the driver properties seen in the DCM.

Figure 10-50 Trade_Application_Module_Installation_Driver

The XML statements to be included in the tc-driver.xml file that controls the object creation during installation of the Trade Automation Package are shown in Example 10-42.

Example 10-42 Trade_Application_Module_Installation_Driver xml definitions

```
<device-model name="Trade_Application_Module_Installation_Driver"
category="Trade">
    <workflow name="Trade_Application_Module_Installation_Uninstall" />
</device-model>
```

The following section describes the details of the Trade_Application_Module_Installation_Uninstall workflow.

Archived

Trade_Application_Module_Installation_Uninstall

When removing the Trade Application Module Installation, we have to keep in mind that all the related (foreign) instances on the same systems must be removed along with it. This is not handled automatically by TIO/TPM (as is the case for instances that are natural children of the installation object), so our Trade_Application_Module_Installalation_Uninstall workflow has to have this functionality built in.

The details of the Trade_Application_Module_Installalation_Uninstall workflow are shown in Example 10-43, and the following describes the main outline:

1. Initialize and get base data for error handling.
2. Find the master (source) template and identify all cloned instances.
3. For each instance, call the SoftwareInstance.RemoveInstance LDO. Since our instances are associated with the Trade_Application_Module_Instance_Driver (described in "The Trade_Application_Module_Instance_Driver" on page 434), we are certain that the workflow that will be invoked to implement the LDO is the Trade_Application_Module_Instance_RemoveInstance workflow, which is described in detail in "Trade_Application_Module_Instance_RemoveInstance" on page 439.
4. Handle errors by returning all received exceptions to the caller.

Example 10-43 Trade_Application_Module_Installation_Uninstall workflow

```
workflow Trade_Application_Module_Installation_Uninstall(in SoftwareInstallationID) implements
SoftwareInstallation.Uninstall LocaleInsensitive

################################################################
##  1: initialize
################################################################
var simulation
Trade_Get_Simulation(simulation)

var workflowName="Trade_Application_Module_Instance_Stop"
var message=Jython["<b>" + workflowName + ": </b>"]
var msg

var ServerID = DCMQuery(/softwareinstallation[@id=$SoftwareInstallationID]/@managedsystemid)

try
  ################################################################
  ##  2: Find the source templates and all it's related objects
  ################################################################
  # find the foreignconfiguration srt
  var instanceSwResourceTypeID=DCMQuery(/softwareresourcetype[@name="INSTANCE"]/@id)
  var
foreignSRTID=DCMQuery(/softwareinstallation[@id=$SoftwareInstallationID]/softwareresourcetemplate/childsof
twareresourcetemplate/@id)
```

```
  var instanceSRTID=DCMQuery(/childsoftwareresourcetemplate[@parenttemplateid=$foreignSRTID and
@softwareresourcetype=$instanceSwResourceTypeID]/@id)

  #find the source
  var sourceTemplateID=DCMQuery(/softwareresourcetemplate[@id=$instanceSRTID]/@sourcetemplateid)

  # find all the cloned instances
  foreach srtID in
DCMQuery(/softwareresource[@managedsystemid=$ServerID]/softwareresourcetemplate[@sourcetemplateid=$sourceT
emplateID and @id != $instanceSRTID]/@id) do
    var instanceID=DCMQuery(/softwareresourcetemplate[@id=$srtID]/softwareresource/@id)
    log info Jython["...calling SoftwareInstance.RemoveInstance(" + instanceID + ")"]
    ##############################################################
    ##  3: Remove each instance
    ##############################################################
    SoftwareInstance.RemoveInstance(instanceID)
  done
catchall exception
  ################################################################
  ##  4: handle errors
  ################################################################
  msg=Jython[message + exception]
  log error msg
  rethrow
endtry
```

10.8.5 Trade Application Module Helper workflows

As it should be obvious from the previous section, the only helper workflow needed for the Trade Application Module device drivers is the most critical one, which is responsible for the installation when called from the Trade_Application_Module_Installable_Install workflow. This workflow is called Trade_Application_Add_Instance.

Trade_Application_Add_Instance

The primary responsibility of this workflow is to invoke the **ws_ant** based installation procedures provided in the trade3Install.zip installation archive. During our design process, we decided that this workflow should create the customized scripts needed to perform all the installation, uninstallation, and operational tasks, since the logic needed to generate these scripts is very similar.

During execution, the workflow will look up the actual customization values for the database and application server environments in order to use this workflow when cataloging the JDBC resources in the WebSphere environment.

Once the customized scripts has been placed on the target system, they will remain there until the Trade Application Module is installed.

Archived

The main outline of the Trade_Application_Add_Instance workflow is:

1. Initialize and set up parameters to control error handling.
2. Gather and set up control data to build parameters to be passed to the scriptlet responsible for generating the installation scripts.
3. Find the installation honoring the DATABASE requirement and gather the installation directory (to be able to specify the path to the db2java.zip class).
4. Find the installation honoring the APPLICATION requirement (pointing to the Trade DBClient Module) and read the parameters that were used when the database was cataloged, in order to create the correct datasource in the WebSphere environment.
5. Find the installation honoring the SERVLET_ENGINE in order to find the unpack directory that determines the location in which to store the scripts that will be generated
6. Call the SoftwareInstallable.Distribute LDO to ensure that the trade3Install.zip installation archive is copied to and unpacked at the target system.
7. Run the scriptlet that generates the installation scripts.

   Since we are developing for the Windows platform, all the script invocations that will be used to execute the scripts will be run from the Cygwin environment, and because of dependencies in the WebSphere Application Server implementation on Windows, the **ws_ant** scripts have to execute from the cmd environment. For this reason, we create a bash (.sh) front end and a command environment script (.cmd) for each function: install, start, stop, and uninstall.

   For all the functions, except the install, no parameters are required, so these scripts are very rudimentary. Example 10-44 and Example 10-45 shows the scripts that will be created to uninstall the Trade EAR module.

Example 10-44 trade3uninst.sh bash front-end for removal of the Trade EAR module

```
cmd /c "C:/cygwin/tmp/trade3/t3install\trade3uninst.cmd"
```

The path used in the front-end bash script is added dynamically based on the parameters discovered by the Trade_Application_Add_Instance workflow.

Example 10-45 trade3uninst.sh cmd script for removal of the Trade EAR module

```
set WAS_HOME=c:/ibm/WebSphere/AppServer
  cd C:/cygwin/tmp/trade3/t3install
  echo "initiating WAS environment"
  call %WAS_HOME%\bin\setupcmdline.bat
  echo "Starting WAS server1"
  call %WAS_HOME%\bin\ws_ant -f Trade3.xml startWebSphere
```

```
  echo "Removing Trade3 application"
  call %WAS_HOME%\bin\ws_ant -f Trade3.xml uninstall
  echo "Regenerating http plugin"
  call %WAS_HOME%\bin\GenPluginCfg.bat
  echo "Re-starting WAS server1"
  call %WAS_HOME%\bin\ws_ant -f Trade3.xml restartWebSphere
```

For the installation script, things are a bit more complicated. By default, the Trade3.xml file used to control the installation prompts the user for input. From an automation point of view, this presents a challenge. To overcome this challenge, we can clone the Trade3.xml file at installation time and provide our customized parameters. We chose, however, to create an expect script that will feed input to the prompts as they appear.

To better understand the installation process, please refer to 10.2.2, “Trade3 installation instructions” on page 284.

The installation is now started by calling the expect interpreter using the treade3inst.exp expect script as the input. This will in turn call a bash front end, which starts the command environment and initiates the WebSphere environment and installs the Trade EAR module, which prompts for the input for the various parameters needed.

Example 10-46 shows the trade3inst.exp expect script for the installation of the Trade EAR module.

Example 10-46 trade3inst.exp expect script for installation of the Trade EAR module

```
#!/bin/expect
  # get input arguments:
  set instScript /tmp/trade3/t3install/trade3inst.sh
  set dbtype db2
  set dbuser tradeusr
  set dbpassword smartway
  set db2javapath c:/ibm/sqllib/java/db2java.zip
  set rc 4

# additional work variables
  set oracleDb "none"
  set oracleSid "none"
  set response ""

# expect settings
  log_user 0
  exp_internal 0
  match_max 100000
  set env(TERM) vt100
```

```
set timeout -1 ;# infinite timeout
set send_slow {1 .1}

proc send {arg} { exp_send -- $arg }
proc logit {msg} { puts  "$msg\n" }

logit "Spawning script $instScript"
exp_spawn $instScript

set script_id $spawn_id
set script_pid [exp_pid]
sleep .5

while 1 {
  expect  {

   "Please enter the database type *db2 or oracle]:\r" {
            logit "Received:$expect_out(buffer)"
            set response "$dbtype\r"
            send "$response"
            logit "Send:$response"
            }

   "Please enter the database username:\r" {
            logit "Received:$expect_out(buffer)"
            set response "$dbuser\r"
            send "$response"
            logit "Send:$response"
            }
    "Please enter the database password:\r" {
            logit "Received:$expect_out(buffer)"
            set response "$dbpassword\r"
            send "$response"
            logit "Send:$response"
            }
    "Please enter the Oracle database hostname:\r" {
            logit "Received:$expect_out(buffer)"
            set response "$oracleDb\r"
            send "$response"
            logit "Send:$response"
            }
    "Please enter the Oracle database SID:\r" {
            logit "Received:$expect_out(buffer)"
            set response "$oracleSid\r"
            send "$response"
```

```
                logit "Send:$response"
                }
      "Please enter the fullpath to the JDBC driver db2java.zip or
ojdbc14.jar (e.g. c:/sqllib/java/db2java.zip)\r" {
                logit "Received:$expect_out(buffer)"
                set response "$db2javapath\r"
                send "$response"
                logit "Send:$response"
                }
     "BUILD SUCCESSFUL*" { set rc [expr $rc-1] }

     "*\r" { logit "$expect_out(buffer)"; }

     eof {logit "$expect_out(buffer)";  exit $rc}

  }
  # catch the input
  expect {
                "*$response" { # logit"$expect_out(buffer)" }
                "*" {}
          }
  }
  exit
```

As seen in Figure 10-46 on page 429, the expect script spawns the trade3inst.sh shell script, shown in Figure 10-47 on page 434, which in turn starts the trade3inst.cmd.

Example 10-47 trade3inst.sh bash front end for Trade EAR module installation

```
#!/bin/bash
  cd /tmp/trade3/t3install
  cmd /c "C:/cygwin/tmp/trade3/t3install/trade3inst.cmd"
```

The trade3inst.cmd, shown in Example 10-48, starts the installation by initializing the WebSphere environment, ensuring that the server named server1 (required by the installation procedure) starts, and then calls **ws_ant** command to create WebSphere resources and install the Trade application.

Example 10-48 trade3inst.cmd Trade EAR module installation script

```
set WAS_HOME=c:/ibm/WebSphere/AppServer
  cd C:/cygwin/tmp/trade3/t3install
  echo "initiating WAS environment"
  call %WAS_HOME%\bin\setupcmdline.bat
  echo "Starting WAS server1"
```

```
call %WAS_HOME%\bin\ws_ant -f Trade3.xml startWebSphere
echo "Installing Trade3 resources"
call %WAS_HOME%\bin\ws_ant -f Trade3.xml installResources
echo "Installing Trade3 application"
call %WAS_HOME%\bin\ws_ant -f Trade3.xml installApp
echo "Regenerating http plugin"
call %WAS_HOME%\bin\GenPluginCfg.bat
echo "Re-starting WAS server1"
call %WAS_HOME%\bin\ws_ant -f Trade3.xml restartWebSphere
```

8. When the scripts have been created, the workflow saves all the script names in the Software Resource Template, so they will be available for the logical device operations.
9. At last, the installation is executed (after an uninstallation to ensure a clean system) by executing the scripts through the Device.ExecuteCommand LDO.
10. Before we can start the application, however, we need to initialize the database and reset the performance counters. Both of these functions are available through a Web interface, so we use a scriptlet to execute the **wget** command for the appropriate URLs. Depending on your environment, the database preparation may take a while.
11. Error handling simply passes the exceptions back to the caller after logging the condition.

For a complete listing of the Trade_Application_Add_Instance workflow, please refer to Example 10-49.

Example 10-49 Trade_Application_Add_Instance workflow

```
workflow Trade_Application_Add_Instance(in SoftwareInstallationID, in SoftwareResourceTemplateID)
implements SoftwareInstallation.AddInstance LocaleInsensitive

####################################
## 1: Initialize
####################################
var simulation
Trade_Get_Simulation(simulation)

var workflowName=”Trade_Application_Add_Instance”
var message = Jython[“<b> “ + workflowName + “</b>: “]
var msg

####################################
## 2: Gather and setup control data
####################################
try
  var SoftwareResourceID=SoftwareInstallationID
  var ServerID=DCMQuery(/softwareresource[@id=$SoftwareResourceID]/@managedsystemid)
```

```
  var
SoftwareModuleID=DCMQuery(/softwareresourcetemplate[@id=$SoftwareResourceTemplateID]/@softwaremoduleid)
  var SoftwareModuleName=DCMQuery(/softwaremodule[@id=$SoftwareModuleID]/@name)
  SoftwareInstallationID=DCMQuery(/softwareinstallation[@softwaremoduleid=$SoftwareModuleID and
@managedsystemid=$ServerID]/@id)
  var
SoftwareInstallableID=DCMQuery(/softwareinstallation[@id=$SoftwareInstallationID]/softwareinstallable/@id)

  var
unpackDir=DCMQuery(/softwareresourcetemplate[@id=$SoftwareResourceTemplateID]/templateparam[@name="unpackD
ir"]/@value)
#  var
xmlFile=DCMQuery(/softwareresourcetemplate[@id=$SoftwareResourceTemplateID]/templateparam[@name="xmlFile"]
/@value)
  var tradeInstallDir = Jython(unpackDir + "/t3install")
  var installationObjTypeID = DCMQuery(/dcmobjecttype[@name="SOFTWARE_INSTALLATION"]/@id)
  var instanceObjTypeID = DCMQuery(/dcmobjecttype[@name="SOFTWARE_INSTANCE"]/@id)

  ######################################################
  ## 3: Find Installation honoring DATABASE requirements
  ######################################################
  var requirementType
  var db_inst_id
  requirementType="DATABASE"
  ITSO_Find_Supporting_Installation_for_Module_Reqirements(requirementType, SoftwareInstallableID,
ServerID, db_inst_id)

  log info Jython["module honoring the " + requirementType + " requirements is : " + db_inst_id]
  #  get the parameters for the installation dir
  var db2_instdir =
DCMQuery(/softwareinstallation[@id=$db_inst_id]/softwareresourcetemplate/templateparam[@name="install_dir"
]/@value)
  var db2javapath = Jython[db2_instdir.strip() + "/java/db2java.zip"]

  #######################################################
  ## 4: Find Installation honoring APPLICATION requirements
  #######################################################
  var jdbc_inst_id
  requirementType="APPLICATION"
  ITSO_Find_Supporting_Installation_for_Module_Reqirements(requirementType, SoftwareInstallableID,
ServerID, jdbc_inst_id)
  log info Jython["module honoring the " + requirementType + " requirements is : " + jdbc_inst_id]

  #  get the configuration SRT
  var swConfigurationTypeID=DCMQuery(/softwareresourcetype[@name="CONFIGURATION"]/@id)
  var jdbc_inst_srt_id=DCMQuery(/softwareinstallation[@id=$jdbc_inst_id]/ softwareresourcetemplate/@id)
  var
jdbc_foreign_srt_id=DCMQuery(/childsoftwareresourcetemplate[@parenttemplateid=$jdbc_inst_srt_id]/@id)
  var jdbc_config_srt_id=DCMQuery(/childsoftwareresourcetemplate[@parenttemplateid=$jdbc_foreign_srt_id
and @softwareresourcetype=$swConfigurationTypeID]/@id)

  #  get the db2 jdbc catalog definitions
```

```
  var
jdbc_alias_name=DCMQuery(/softwareresourcetemplate[@id=$jdbc_config_srt_id]/templateparam[@name="databaseA
liasName"]/@value)
  var
dbuser_sap_domain=DCMQuery(/softwareresourcetemplate[@id=$jdbc_config_srt_id]/templateparam[@name="trade_d
buser_SAP_domain"]/@value)
  var
dbuser_sap_context=DCMQuery(/softwareresourcetemplate[@id=$jdbc_config_srt_id]/templateparam[@name="trade_
dbuser_SAP_context"]/@value)
  var
dbuser_sap_searchkey=DCMQuery(/softwareresourcetemplate[@id=$jdbc_config_srt_id]/templateparam[@name="trad
e_dbuser_SAP_searchkey"]/@value)
  var jdbc_credentials_id=DCMQuery(/sap[@domain=$dbuser_sap_domain and
@context=$dbuser_sap_context]/credentials[@searchkey=$dbuser_sap_searchkey]/@id)
  var dbuser
  var dbpassword
  #  get the userid and password values
  java:com.thinkdynamics.kanaha.de.javaplugin.sap.FindCredentialsPassword(jdbc_credentials_id, <null>,
<null>, dbpassword, dbuser)

  ##########################################################################
  ## 5: Find Installation honoring SERVLET_ENGINE requirements (the WAS Server
  ##########################################################################
  var was_inst_id
  requirementType="SERVLET_ENGINE"
  ITSO_Find_Supporting_Installation_for_Module_Reqirements("SERVLET_ENGINE", SoftwareInstallableID,
ServerID, was_inst_id)
  log info Jython["module honoring the " + requirementType + " requirements is : " + was_inst_id]

  var install_dir
  var install_dir_parmName="install_dir"
  Trade_Get_Hosting_SRTParameter_Or_Default(SoftwareInstallableID,  was_inst_id, install_dir_parmName,
install_dir)
  if Jython[not install_dir] then
    msg = Jython[message + " could not find value for install_dir"]
    throw TradeParameterException msg
  endif
  # name the scripts used to manipulate the apache service
  var wasHome=Jython[install_dir + "/AppServer"]
  var dbtype="db2"
  var instScript=Jython[tradeInstallDir + "/trade3inst.sh"]
  var uninstScript=Jython[tradeInstallDir + "/trade3uninst.sh"]
  var startScript=Jython[tradeInstallDir + "/trade3start.sh"]
  var stopScript=Jython[tradeInstallDir + "/trade3stop.sh"]
  var expectScript=Jython[tradeInstallDir + "/trade3inst.exp"]

  ######################################################
  ## 6: Send the index.html file to the target system
  ######################################################
  SoftwareInstallable.Distribute(SoftwareInstallableID, ServerID)

  ######################################################
  ## 7: Run a scriptlet to update the incex.html file
  ######################################################
```

```
  if Jython[simulation == “yes”] then
    log info Jython[“<b>simulation:</b> ... about to call   SoftwareInstallable.Distribute(“ +
SoftwareInstallableID + “, “ + ServerID + “)”]
    log info Jython[“<b>simulation: </b>... calling scriptlet to generate installation scripts”]
    log info Jython[“        tradeInstallDir: “ + tradeInstallDir]
    log info Jython[“        wasHome: “ + wasHome]
    log info Jython[“        dbtype: “ + dbtype]
    log info Jython[“        dbuser: “ + dbuser]
    log info Jython[“        dbpassword: “ + dbpassword]
    log info Jython[“        db2javapath: “ + db2javapath]
  else
    scriptlet (tradeInstallDir, wasHome, dbtype, dbuser, dbpassword, db2javapath) language=bash
target=DCMQuery(/server[@id=$ServerID]/@id) <<EOFSH
      tradeInst=”trade3inst”
      tradeUninst=”trade3uninst”
      tradeStart=”trade3start”
      tradeStop=”trade3stop”
      tradeInstExpect=”${tradeInst}.exp”
      tradeInstScript=”${tradeInst}.sh”
      tradeInstCommand=”${tradeInst}.cmd”
      tradeUninstScript=”${tradeUninst}.sh”
      tradeUninstCommand=”${tradeUninst}.cmd”
      tradeStartScript=”${tradeStart}.sh”
      tradeStartCommand=”${tradeStart}.cmd”
      tradeStopScript=”${tradeStop}.sh”
      tradeStopCommand=”${tradeStop}.cmd”
      # convert installDir to mixed format
      winTradeInstDir=‘cygpath -m ${tradeInstallDir}‘
      # convert installDir to Windows format
      tradeInstDir=‘cygpath -w -s ${tradeInstallDir}‘
      # convert wasHome to mixed format
      wasHome=‘cygpath -m ${wasHome}‘

      TIOlog “info” “Generating trade3 install command ${tradeInstallDir}/${tradeInstCommand}”
      echo “set WAS_HOME=${wasHome}
      cd ${winTradeInstDir}

      # build dos install command
      echo \”initiating WAS environment\”
      call %WAS_HOME%\\bin\\setupcmdline.bat
      echo \”Starting WAS server1\”
      call %WAS_HOME%\\bin\\ws_ant -f Trade3.xml startWebSphere
      echo \”Installing Trade3 resources\”
      call %WAS_HOME%\\bin\\ws_ant -f Trade3.xml installResources
      echo \”Installing Trade3 application\”
      call %WAS_HOME%\\bin\\ws_ant -f Trade3.xml installApp
      echo \”Regenerating http plugin\”
      call %WAS_HOME%\\bin\\GenPluginCfg.bat
      echo \”Re-starting WAS server1\”
      call %WAS_HOME%\\bin\\ws_ant -f Trade3.xml restartWebSphere
      “ > ${tradeInstallDir}/${tradeInstCommand}
      unix2dos ${tradeInstallDir}/${tradeInstCommand}

      # build unix install script
```

```
      TIOlog “info” “Generating trade3 install script ${tradeInstallDir}/${tradeInstScript}”
      echo “#!/bin/bash
      cd ${tradeInstallDir}
      cmd /c \”${winTradeInstDir}/${tradeInstCommand}\”
      “ > ${tradeInstallDir}/${tradeInstScript}
      # make it executable
      chmod +x ${tradeInstallDir}/${tradeInstScript}
      TIOsetVar “instScript” “${tradeInstallDir}/${tradeInstScript}”

      # build DOS uninstall command
      TIOlog “info” “Generating trade3 uninstall command ${tradeInstallDir}/${tradeUninstCommand}”
      echo “set WAS_HOME=${wasHome}
      cd ${winTradeInstDir}
      echo \”initiating WAS environment\”
      call %WAS_HOME%\\bin\\setupcmdline.bat
      echo \”Starting WAS server1\”
      call %WAS_HOME%\\bin\\ws_ant -f Trade3.xml startWebSphere
      echo \”Removing Trade3 application\”
      call %WAS_HOME%\\bin\\ws_ant -f Trade3.xml uninstall
      echo \”Regenerating http plugin\”
      call %WAS_HOME%\\bin\\GenPluginCfg.bat
      echo \”Re-starting WAS server1\”
      call %WAS_HOME%\\bin\\ws_ant -f Trade3.xml restartWebSphere
      “ > ${tradeInstallDir}/${tradeUninstCommand}
      unix2dos ${tradeInstallDir}/${tradeUninstCommand}

      # build unix uninstall script
      TIOlog “info” “Generating trade3 uninstall script ${tradeUninstallDir}/${tradeUninstScript}”
      echo “cmd /c \”${winTradeInstDir}\\${tradeUninstCommand}\”” >
${tradeInstallDir}/${tradeUninstScript}
      # make it executable
      chmod +x ${tradeInstallDir}/${tradeUninstScript}
      TIOsetVar “uninstScript” “${tradeInstallDir}/${tradeUninstScript}”


      # build DOS start command
      TIOlog “info” “Generating trade3 start command ${tradeInstallDir}/${tradeStartCommand}”
      echo “set WAS_HOME=${wasHome}
      cd ${winTradeInstDir}
      echo \”initiating WAS environment\”
      call %WAS_HOME%\\bin\\setupcmdline.bat
      echo \”Starting Trade3 application\”
      call %WAS_HOME%\\bin\\ws_ant -f Trade3.xml startTrade
      “ > ${tradeInstallDir}/${tradeStartCommand}
      unix2dos ${tradeInstallDir}/${tradeStartCommand}

      # build unix start script
      TIOlog “info” “Generating trade3 start script ${tradeInstallDir}/${tradeStartScript}”
      echo “cmd /c \”${winTradeInstDir}\\${tradeStartCommand}\”” > ${tradeInstallDir}/${tradeStartScript}
      # make it executable
      chmod +x ${tradeInstallDir}/${tradeStartScript}
      TIOsetVar “startScript” “${tradeInstallDir}/${tradeStartScript}”

      # build DOS stop command
```

```
TIOlog “info” “Generating trade3 stop command ${tradeInstallDir}/${tradeStopCommand}”
echo “set WAS_HOME=${wasHome}
cd ${winTradeInstDir}
echo \”initiating WAS environment\”
call %WAS_HOME%\\bin\\setupcmdline.bat
echo \”Starting Trade3 application\”
call %WAS_HOME%\\bin\\ws_ant -f Trade3.xml stopTrade
“ > ${tradeInstallDir}/${tradeStopCommand}
unix2dos ${tradeInstallDir}/${tradeStopCommand}

# build unix stop script
TIOlog “info” “Generating trade3 stop script ${tradeInstallDir}/${tradeStopScript}”
echo “cmd /c \”${winTradeInstDir}\\${tradeStopCommand}\”” > ${tradeInstallDir}/${tradeStopScript}
# make it executable
chmod +x ${tradeInstallDir}/${tradeStopScript}
TIOsetVar “stopScript” “${tradeInstallDir}/${tradeStopScript}”

# build expect script to perform installation
TIOlog “info” “Generating trade3 expect installation script ${tradeInstallDir}/${tradeInstExpect}”
echo “#!/bin/expect
# get input arguments:
set instScript ${tradeInstallDir}/${tradeInstScript}
set dbtype ${dbtype}
set dbuser ${dbuser}
set dbpassword ${dbpassword}
set db2javapath ${db2javapath}
set rc 4

# additional work variables
set oracleDb \”none\”
set oracleSid \”none\”
set response \”\”

# expect settings
log_user 0
exp_internal 0
match_max 100000
set env(TERM) vt100
set timeout -1 ;# infinite timeout
set send_slow {1 .1}

proc send {arg} { exp_send -- \$arg }
proc logit {msg} { puts  \”\$msg\n\” }

logit \”Spawning script \$instScript\”
exp_spawn \$instScript

set script_id \$spawn_id
set script_pid [exp_pid]
sleep .5

#  doit
while 1 {
  expect  {
```

```
        \"Please enter the database type *db2 or oracle]:\\r\" {
            logit \"Received:\$expect_out(buffer)\"
            set response \"\$dbtype\\r\"
            send \"\$response\"
            logit \"Send:\$response\"
            }
        \"Please enter the database username:\\r\" {
            logit \"Received:\$expect_out(buffer)\"
            set response \"\$dbuser\\r\"
            send \"\$response\"
            logit \"Send:\$response\"
            }
        \"Please enter the database password:\\r\" {
            logit \"Received:\$expect_out(buffer)\"
            set response \"\$dbpassword\\r\"
            send \"\$response\"
            logit \"Send:\$response\"
            }
        \"Please enter the Oracle database hostname:\\r\" {
            logit \"Received:\$expect_out(buffer)\"
            set response \"\$oracleDb\\r\"
            send \"\$response\"
            logit \"Send:\$response\"
            }
        \"Please enter the Oracle database SID:\\r\" {
            logit \"Received:\$expect_out(buffer)\"
            set response \"\$oracleSid\\r\"
            send \"\$response\"
            logit \"Send:\$response\"
            }
        \"Please enter the fullpath to the JDBC driver db2java.zip or ojdbc14.jar (e.g.
c:/sqllib/java/db2java.zip)\\r\" {
            logit \"Received:\$expect_out(buffer)\"
            set response \"\$db2javapath\\r\"
            send \"\$response\"
            logit \"Send:\$response\"
            }
       \"BUILD SUCCESSFUL*\" { set rc [expr \$rc-1] }
       \"*\\r\" { logit \"\$expect_out(buffer)\"; }
       eof {logit \"\$expect_out(buffer)\";  exit \$rc}
      }
    # catch the input
    expect {
            \"*\$response\" { # logit\"\$expect_out(buffer)\" }
            \"*\" {}
           }
    }
    exit
    " > ${tradeInstallDir}/${tradeInstExpect}

    # make it executable
    chmod +x ${tradeInstallDir}/${tradeInstExpect}
    TIOsetVar "expectScript" "expect ${tradeInstallDir}/${tradeInstExpect}"
  EOFSH
```

```
  endif

  ##################################################
  ## 8: Save the script name s and parameters in SRTs
  ##################################################
  #  save inst and uninst script names in the INSTANCE responsefile template
  var instanceSRTID=SoftwareResourceTemplateID
  var
configurationSRTID=DCMQuery(/childsoftwareresourcetemplate[@parenttemplateid=$SoftwareResourceTemplateID
and @softwareresourcetype=$swConfigurationTypeID]/@id)

  DCMInsert parent=DCMQuery(/softwareresourcetemplate[@id=$instanceSRTID]/@id) <<EOFXML
       <template-param name="installScript" value="$expectScript" is-changeable="true" />
  EOFXML
  DCMInsert parent=DCMQuery(/softwareresourcetemplate[@id=$instanceSRTID]/@id) <<EOFXML
       <template-param name="uninstallScript" value="$uninstScript" is-changeable="true" />
  EOFXML
  DCMInsert parent=DCMQuery(/softwareresourcetemplate[@id=$instanceSRTID]/@id) <<EOFXML
       <template-param name="startScript" value="$startScript" is-changeable="true" />
  EOFXML
  DCMInsert parent=DCMQuery(/softwareresourcetemplate[@id=$instanceSRTID]/@id) <<EOFXML
       <template-param name="stopScript" value="$stopScript" is-changeable="true" />
  EOFXML
  DCMInsert parent=DCMQuery(/softwareresourcetemplate[@id=$instanceSRTID]/@id) <<EOFXML
       <template-param name="workDir" value="$tradeInstallDir" is-changeable="true" />
  EOFXML

  # save the configuration parameters in the CONFIGURATION resource template
  DCMInsert parent=DCMQuery(/softwareresourcetemplate[@id=$configurationSRTID]/@id) <<EOFXML
       <template-param name="dbtype" value="$dbtype" is-changeable="true" />
  EOFXML
  DCMInsert parent=DCMQuery(/softwareresourcetemplate[@id=$configurationSRTID]/@id) <<EOFXML
       <template-param name="dbuser" value="$dbuser" is-changeable="true" />
  EOFXML
  DCMInsert parent=DCMQuery(/softwareresourcetemplate[@id=$configurationSRTID]/@id) <<EOFXML
       <template-param name="dbpassword" value="$dbpassword" is-changeable="true" />
  EOFXML
  DCMInsert parent=DCMQuery(/softwareresourcetemplate[@id=$configurationSRTID]/@id) <<EOFXML
       <template-param name="db2javapath" value="$db2javapath" is-changeable="true" />
  EOFXML

  ##########################################################################
  ## 9: Perform installation - uninsta,,. install, and initialize database
  ##########################################################################
  var ReturnCode
  var ReturnErrorString
  var ReturnResult
  var serverPort=DCMQuery(/templateparam[@templateid=$SoftwareResourceTemplateID and
@name="serverPort"]/@value)
  if Jython[not serverPort] then
    msg = Jython[message + " could not find value for parameter serverPort"]
    log error msg
    throw parameterException msg
  endif
```

```
  var buildDBURL=DCMQuery(/templateparam[@templateid=$configurationSRTID and @name="buildDBURL"]/@value)
  if Jython[not buildDBURL] then
    msg = Jython[message + " could not find value for parameter buildDBURL"]
    log error msg
    throw parameterException msg
  endif

  var resetTradeURL=DCMQuery(/templateparam[@templateid=$configurationSRTID and
@name="resetTradeURL"]/@value)
  if Jython[not resetTradeURL] then
    msg = Jython[message + " could not find value for parameter resetTradeURL"]
    log error msg
    throw parameterException msg
  endif
  var name = DCMQuery(/server[@id=$ServerID]/@name)
  log info Jython["<b> Installing trade on " + name + "</b>"]

  if Jython[simulation == "yes"] then
    log info Jython["<b>simulation: </b>...about the execute: " + uninstScript]
    log info Jython["<b>simulation: </b>...about the execute: " + expectScript]
    log info Jython["<b>simulation: </b>...running scriptlet to build the trade database and reset
counters"]
    log info Jython["                      serverPort: " + serverPort]
    log info Jython["                      buildDBURL: " + buildDBURL]
    log info Jython["                      resetTradeURL: " + resetTradeURL]
  else
    # uninstall
    Device.ExecuteCommand(ServerID,  uninstScript, tradeInstallDir, "default", "600", "ignore",
ReturnCode, ReturnErrorString, ReturnResult)
    # install
    Device.ExecuteCommand(ServerID,  expectScript, tradeInstallDir, "default", "900", "error", ReturnCode,
ReturnErrorString, ReturnResult)
    # initialize the database and reset performance counters
    scriptlet (serverPort, buildDBURL, resetTradeURL) language = bash target =
DCMQuery(/server[@id=$ServerID]/@id) timeout=1200 <<EOF
      wget http://localhost:${serverPort}${buildDBURL}
      wget http://localhost:${serverPort}${resetTradeURL}
    EOF
  endif
catchall exception
  ######################################################################
  ## 10: Error handling
  ######################################################################
  msg = Jython[message + exception]
  log error msg
  rethrow
endtry
```

10.9 Front end of the Trade application

The final task we have to accomplish to deploy the Trade application is to register the application with the front-end Web application in our environment.

This step is not, for obvious reasons, included in the installation procedure provided by the Trade application development team, since they are not aware of the customer environment. Neither are we, but for the sake of the example, we have built a small front-end HTML page, and in this final task, we will demonstrate how to accomplish the following steps:

1. Create a new Apache service for hosting the Trade Application.
2. Deploy the sample front-end HTML page.
3. Customize the HTML page to contain a link the newly deployed Trade Application.
4. Restart the Apache service to activate our changes.

Note: In the following section, we do not take clustering or load balancers into account. However, in a customer environment, there is an additional step for updating the load balancer configuration based on the Virtual IP associations with the Application Tiers to which the Trade Application is deployed. This step is considered the responsibility of the Trade Application user, since it is so environment specific; however, it is considered a standard operation that should be standardized within the Cluster.AddServer logical device operation of any customer.

All other activities relevant to the deployment of the Web front end in Windows based environment will be covered in the following section.

Archived

The sample Web front end we provide in the Trade Automation Package will, when it has been deployed to the target server and customized in accordance with the newly provisioned Trade Application environment, look similar to the one shown in Figure 10-51.

Figure 10-51 Trade Web front end

It is the responsibility of the deployment process to add the link labeled Trade3 Sample Application, and provide an underlying reference that points to http://<wasServer>:9080/trade/app.

As was the case for all other tasks in the deployment of the Trade Application, we are faced with the problem that when implementing in a customer environment, we have absolutely no knowledge of how the underlying infrastructure is defined in the Data Center Model.

In the following, we make the assumption that the capabilities of the target servers are:

OS	os.family=Windows 2000 Server SP4
WEBSERVER	jdk.version=2.0

and we will set up requirements in the Software Product and Software Package that reflect these assumptions.

10.9.1 The Trade Web Module Software Product

For the creation of the Apache server to be the front end of the Trade application, we define a Software Product named Trade Web Module. Through the Software Product definition, we will link to a Software Package, named Trade Web Module Installable, which holds information about the installation files, and by including the proper Software Resource Templates in the Software Product, we ensure that the expected hierarchy of DCM objects will be created and named according to our standards.

Archived

The properties of the Trade Web Module software product are shown in Figure 10-52.

Figure 10-52 Trade Web Module properties

In addition, we will associate special capabilities with the Software Product. These capabilities will be applied to the target server upon installation, and thus can be used later to ensure that the Trade Web Server exists. To ensure that the required Apache Server functionality is available on the servers where the Trade Web Module will be installed, we will also add WEBSERVER related requirements to the Software Product definition.

Capabilities and requirements

The capabilities we assign to the Trade Web Module are:

APPLICATION	software.title	TRADE
APPLICATION	software.name	WEB
APPLICATION	software.vendor	IBM
APPLICATION	software.version	3.1

By specifying the following requirements for our Trade Web Module Software Product:

WEBSERVER	jdk.version	2.0

it is guaranteed that a WebServer that supports our deployment requirements exists on the target system at the time of installation of the Trade Web Module.

In addition, to provide a pointer to the Trade Application (hosted by WebSphere), we add non-hosting APPLICATION requirements for the software.name of EAR, which represents the Trade Application Module installation.

Archived

Note: In a real-life implementation, the user should be provided instructions on how to modify the requirements that are set up by default for the Trade Web Module in order to integrate it into a specific environment.

Figure 10-53 shows the capabilities and requirements for the Trade Web Module Software Product.

Requirements and Capabilities

Requirement Name	Requirement Type	Requirement Values
software.vendor	APPLICATION	IBM
software.version	APPLICATION	3.1
jdk.version	WEBSERVER	2.0
software.name	APPLICATION	EAR
software.title	APPLICATION	TRADE

Capability Type
APPLICATION

Capability Name	Capability Value
software.title	TRADE
software.name	WEB
software.vendor	IBM
software.version	3.1

Figure 10-53 Trade Web Module capabilities and requirements

Software Package(s)

Now, to help determine which files to use during the installation, we created a Software Package named Trade Web Module Installable.

The only file that is referenced by the Software Package is the index.html, which is the front-end HTML page for our environment. This file is included in the Automation Package (in the repository subdirectory) and will, when the Automation package is installed, be placed in the %TIO_HOME%/../SWrepository/trade3 directory. This is achieved by the addition of the XML statements shown in Example 10-50 to the tc-driver.xml file for the Trade Automation Package.

Example 10-50 Incorporating the index.html file in the automation package

```
<item name="repository/index.html" action="copy-file">
      <param name="editable" value="false" />
      <param name="dest.path" value="${repository}/trade3/index.html"
/>
      <param name="chmod" value="777" />
```

```
</item>
```

Important: The addition of the <item> definition above to the tc-driver.xml file has to be performed manually because the APDE environment has no way of knowing where to copy the files in the repository subdirectory of the automation package.

Even though this file can be used to place the Trade Application as a front end on any platform, we added, for the sake of the example, a requirement for the Software Package, specifying the following:

OS	os.family	Windows 2000 Server SP4

Figure 10-54 shows the properties of the Trade Web Module Installable in its final state.

Figure 10-54 Trade Web Module Installable software package properties

Archived

Variables

For this software package, we will also define a set of variables, which will be used to help find the installation directory of the underlying IBM HTTP Server installation, as described in 10.5.2, “Finding installation directories” on page 324. For the Trade Web Module Installable, these variables are created with the standard values shown in Figure 10-55, and it is expected that they are customized during the installation of the Trade Automation Package.

Key	Component	Value
install_dir_custom_default	Deployment engine	/cygdrive/c/ibm/ihs20
install_dir_parameter_name	Deployment engine	install_dir
install_dir_softwareresource_type	Deployment engine	INSTALLATION
install_dir_system_default	Deployment engine	c:\Program Files\ibmhttpserver

Figure 10-55 Trade Web Module Installable software package variables

The Trade_Web_Module_Installable_Driver Device Driver

To control the installation, the Trade Web Module Installable, we need to assign a device driver that, at a minimum, contains a workflow that implements the SoftwareInstallable.Install logical device operation. The transfer of the index.html file will be embedded in this workflow.

Example 10-51 shows the XML statements embedded in the tc-driver.xml file for the entire Trade Automation Package, which are used to define the device driver to the DCM. The device driver we assign to the Trade Web Module Installable will be named Trade_Web_Module_Installable_Driver. Details regarding the workflows embedded in the device driver are provided in “Trade_Web_Module_Installable_Install” on page 472.

Example 10-51 xml to define the Trade_Web_Module_Installable_Driver

```
<device-model name="Trade_Web_Module_Installable_Driver"
category="Trade">
    <workflow name="Trade_Web_Module_Installable_Install" />
</device-model>
```

Before we look at the details of the workflows, we need to take a look at the Software Resource Templates of the Trade Web Module Software Product to gain knowledge about the customization parameters that will be available to the installation workflows.

Software Resource Templates

In order to create the Apache Server and Web front-end resources in the data center model as children of the underlying WEBSERVER Software Installation object, we add a FOREIGN_CONFIGURATION Software Resource Template to the Trade Web Module INSTALLATION Software Resource Template, effectively creating a Software Resource Template hierarchy of the Trade Web Module Software Product that looks like this:

INSTALLATION
  FOREIGN_CONFIGURATION
    INSTANCE
      CONFIGURATION

As a result, the INSTALLATION template will be used to create a Software Installation object on the target server, and associate this with the Trade Web Module Software Product. As children of this Software Installation object TIO/TPM will clone the remainder of the SRT hierarchy (the FOREIGN_CONFIGURATION template and all its children) to the new Software Installation, the workflow that implements the SoftwareInstallable.Install LDO is responsible for inspecting the SRT hierarchy and taking appropriate action(s).

Device driver associations

To control the behavior of the resources that are created from the Software Resource Templates, we can associate a device driver to each of them.

Archived

For the Trade Web Module Software Package software model, we associate the device drivers and related workflows implementing the required logical operations, as shown in Table 10-11.

Table 10-11 Trade Web Module related devise drivers and workflows

SRT	Type	Name	Workflow
INSTALLATION	Device driver	Trade_Web_Module_Intallation_Driver	
	LDOs	SoftwareInstallation .Uninstall	Trade_Web_Module_Installation _Uninstall
INSTANCE	Device driver	Trade_WebServer_Module_Instance_Driver	
	LDOs	SoftwareInstance .RemoveInstance	Trade_Web_Module_Instance_R emoveInstance
		SoftwareInstance .Start	Trade_Web_Module_Instance_S tart
		SoftwareInstance .Stop	Trade_Web_Module_Instance_S top

In addition, these workflows will call additional workflows, which are not associated with any particular device driver, in order to execute specific actions, which are repetitive, or provide a very specific set of functions.

Software Resource naming

The naming of Software Resources that will be created based on the templates can be controlled by creating a template parameter named *resource-name*, and providing a value that matches your requirements. Figure 10-20 on page 340 shows the final Software Resource Template hierarchy for the Trade Web Module, and here we see that the names that will be assigned to the objects created by the templates are:

Installation Trade Web Module Installation
Foreign_Configuration
Trade HTTP Installation
Instance Trade_HTTP_Instance
Configuration Trade HTTP Configuration

Software Resource Template parameters

To provide for configuration of the Apache server to be created during the installation of the Trade Web Module software package, we will need to provide customization parameters to determine how to configure the Apache Server and

update the index.html file. The needed parameters and their meaning are listed in Table 10-11 on page 464.

Table 10-12 Trade Web Module related Software Resource Template parameters

SRT	Name	Description
INSTANCE	resource-name	Name of the Apache Server service to be created.
	port	Port to listen to (AfpaPort).
	config_dir	Subdirectory in which to find the file specified by the config_file parameter (DocumentRoot).
	config_file	Name of the default configuration file that will be cloned during the creation of the new Apache Server.
	documents_dir	Subdirectory (relative to the installation directory) where html documents are stored (DirectoryIndexPath).
	indexFileName	Name of the index.html file on the target server (DirectoryIndexFile).
CONFIGURATION	resource-name	Name of the configuration.
	applicationName	Label to insert into the Web page.

Archived

The default values provided in the Software Resource Templates upon installation of the Trade Automation Package are shown in Figure 10-56.

Configuration Templates

Trade Web Module Installation Template

Software Resource Type: Installation Device Model: Trade_Web_Module_Installation_Driver

Parameter	Value
No records found	

Trade Web Module Foreign Config

Software Resource Type: Foreign configuration

Parameter	Value
application	trade
resource-name	Trade HTTP Installation

Trade Web Module Instance

Software Resource Type: Instance Device Model: Trade_Web_Module_Instance_Driver

Parameter	Value
resource-name	Trade_HTTP_Instance
port	81
config_dir	conf
config_file	httpd.conf
documents_dir	htdocs/en_US
indexFileName	tradeindex.html

Trade Web Module Configuration

Software Resource Type: Configuration

Parameter	Value
resource-name	Trade HTTP Configuration
applicationName	Trade3 Sample Application

Figure 10-56 Trade Web Module Software Resource Templates

Package overview

Based on the parameters and attributes we have defined for the Trade Web Module Software Product, along with the related Software Package and the embedded Software Resource Templates, we have created the software model for:

- Creation of an Apache Server named Trade_HTTP_Instance
- Creation and customization of the tradeindex.html file that is the front end for the Trade application

Archived

in a way that will be easily customizable for the potential users of the Trade Automation Package. Figure 10-57 shows a depiction of the resources involved and their relationships.

Figure 10-57 Trade Web Module deployment model

Creating the software objects from the Automation Package

The XML deck needed to define our Software Product and Software Package, along with the embedded file, variable, Software Resource Template, and device driver definitions, is shown in Example 10-52 on page 468.

To automatically import this XML file as part of the automation package installation, you will have to add an item definition for the XML file using the **import** function on the <dcm> section in the tc-driver.xml file for the automation package:

```
<dcm>
   <item name="xml/Trade_Web_Module.xml" action="import"/>
</dcm>
```

Example 10-52 Trade_Web_Module.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE datacenter SYSTEM "file:../xml/xmlimport.dtd">

<datacenter>

  <software-module name="Trade Web Module" version="3.1"
                   vendor="IBM" title="Trade Web Server configuration"
                   description="Trade HTTP Server configuration"
                   is-draft="false">
        <software-capability name="software.title" value="TRADE" />
        <software-capability name="software.name" value="WEB" />
        <software-capability name="software.vendor" value="IBM" />
        <software-capability name="software.version" value="3.1" />
        <supported-requirement-type value="APPLICATION" />

        <software-requirement name="jdk.version"
                              type="WEBSERVER"
                              enforcement="MANDATORY"
                              hosting="true"
                              accept-non-existing="false">
               <software-requirement-value value="2.0" />
        </software-requirement>

        <software-requirement name="software.title"
                              type="APPLICATION"
                              enforcement="MANDATORY"
                              hosting="false"
                              accept-non-existing="true">
               <software-requirement-value value="TRADE" />
        </software-requirement>
        <software-requirement name="software.name"
                              type="APPLICATION"
                              enforcement="MANDATORY"
                              hosting="false"
                              accept-non-existing="true">
               <software-requirement-value value="EAR" />
        </software-requirement>
        <software-requirement name="software.version"
                              type="APPLICATION"
                              enforcement="MANDATORY"
                              hosting="false"
                              accept-non-existing="true">
               <software-requirement-value value="3.1" />
        </software-requirement>
        <software-requirement name="software.vendor"
                              type="APPLICATION"
                              enforcement="MANDATORY"
                              hosting="false"
                              accept-non-existing="true">
               <software-requirement-value value="IBM" />
        </software-requirement>
```

Archived

```
   <installable-package name="Trade Web Module Installable"
                        is-device-model="Trade_Web_Module_Installable_Driver"
                        locale="en_US"
                        description="Web pages for Trade"
                        version="3.1"
                        priority="0"
                        file-repository="LocalFileRepository"
                        status="tested">
          <file name="index.html" path="/trade3" />
                 <property component="DEPLOYMENT_ENGINE"
                            name="install_dir_softwareresource_type"
                            value="INSTALLATION" />
                 <property component="DEPLOYMENT_ENGINE"
                            name="install_dir_parameter_name"
                            value="install_dir" />
                 <property component="DEPLOYMENT_ENGINE"
                            name="install_dir_custom_default"
                            value="/cygdrive/c/ibm/ihs20" />
                 <property component="DEPLOYMENT_ENGINE"
                            name="install_dir_system_default"
                            value="c:\Program Files\ibmhttpserver" />


       <software-requirement name="os.family"
                             type="OS"
                             enforcement="MANDATORY"
                             hosting="true"
                             accept-non-existing="false">
           <software-requirement-value value="Windows 2000 Server SP4" />
       </software-requirement>
</installable-package>

<software-resource-template name="Trade Web Module Installation Template"
                        software-resource-type="INSTALLATION"
                        software-resource-device-model="Trade_Web_Module_Installation_Driver"
                        multiplicity-type="Unspecified"
                        software-configuration-type="Regular"
                        is-selected="true">
       <software-resource-template name="Trade Web Module Foreign Config"
                                   software-resource-type="FOREIGN-CONFIGURATION"
                                   multiplicity-type="Unspecified"
                                   software-configuration-type="Regular"
                                   is-selected="true">
       <software-resource-template name="Trade Web Module Instance"
                                   software-resource-type="INSTANCE"
                                   software-resource-device-model="Trade_Web_Module_Instance_Driver"
                                   multiplicity-type="Unspecified"
                                   software-configuration-type="Regular"
                                   is-selected="true">
          <software-resource-template name="Trade Web Module Configuration"
                                      software-resource-type="CONFIGURATION"
                                      multiplicity-type="Unspecified"
                                      software-configuration-type="Regular"
                                      is-selected="true">
```

Archived

```
                <template-param name="resource-name"
                                value="Trade HTTP Configuration"
                                is-changeable="true" />
                <template-param name="applicationName"
                                value="Trade3 Sample Application"
                                is-changeable="true" />

            </software-resource-template>
            <template-param name="resource-name"
                            value="Trade_HTTP_Instance"
                            is-changeable="true" />
            <template-param name="port"
                            value="81"
                            is-changeable="true" />
            <template-param name="config_dir"
                            value="conf"
                            is-changeable="true" />
            <template-param name="config_file"
                            value="httpd.conf"
                            is-changeable="true" />
            <template-param name="documents_dir"
                            value="htdocs/en_US"
                            is-changeable="true" />
            <template-param name="indexFileName"
                            value="tradeindex.html"
                            is-changeable="true" />
            </software-resource-template>
            <template-param name="application"
                            value="trade"
                            is-changeable="true" />
        <template-param name="resource-name"
                        value="Trade HTTP Installation"
                        is-changeable="true" />
        </software-resource-template>
      </software-resource-template>

    </software-module>

</datacenter>
```

10.9.2 The Trade_Web_Module_Installable_Driver

By default, the logical device operations for the Software Installable object type are Install and Distribute. Even though we want to distribute our index.html file to the system hosting the Apache server for the Trade application, we choose not to use the Softwareinstallable.Distribute logical operation, primarily because we have gathered most of the necessary data used to control the file transfer in the workflow implementing the SoftwareInstallable.Install logical device operation.

Archived

Because of this, the only LDO supported by the Trade_Web_Module_Installable_Driver is SoftwareInstallable.Install, and we will develop a workflow named Trade_Web_Module_Installable_Install to implement it, as shown in Figure 10-58.

Figure 10-58 Trade_Web_Module_Installable_Driver overview

The main workflow in this driver is the Trade_Web_Module_Installable_Install workflow, which implements the SoftwareInstallable.Install logical device operation. For testing and ease of development reasons, the functionality of the workflow is divided into Trade_Web_Add_Instance and Trade_Web_Add_Configuration.

Archived

When defining the Trade_Web_Module_Installable_Driver to the DCM, it will appear as shown in Figure 10-59.

Figure 10-59 Trade_Web_Module_Installable_Driver

The XML statements included in the tc-driver.xml file used to create the device driver during installation of the Trade Automation Package are shown in Example 10-53.

Example 10-53 Trade_Web_Module_Installable_Driver xml definitions

```
<device-model name="Trade_Web_Module_Installable_Driver"
category="Trade">
    <workflow name="Trade_Web_Module_Installable_Install" />
</device-model>
```

The following section describes the functionality of the Trade_Web_Module_Installable_Install workflow.

Trade_Web_Module_Installable_Install

The intended purpose of this workflow is to:

- Create and customize a new Apache service on the system on which the Trade_Web_Module is installed.
- Distribute the sample index.html file to the DocumentRoot directory of the new Apache instance, and update it according to the current environment.
- Start the new Apache service.

To achieve this, we developed a workflow, shown in detail in Example 10-54 on page 474 with the following main functionality:

1. Initialize and gather data to control navigation and control.
2. Find the Software Installation object that implements the WEBSERVER requirement in order to be able to retrieve Apache server installation directory. The utility workflow ITSO_Find_Supporting_Installation_for_Module_Requirements (described in

details in 10.10.1, “ITSO_Find_Supporting_Installation_for_Module_Requirements” on page 494) is used to identify the underlying IBM HTTP Server v2 installation.

3. Get the installation directory using the Trade_Get_Hosting_SRTParameter_Or_Default workflow, described in 10.5.2, “Finding installation directories” on page 324. This relies on the variables defined for the Trade_Web_Module_Installable, as described in “Variables” on page 462.
4. In order to allow the subsequent workflows to find the new Software Installation object through a DCMQuery, we have to set the value of the pending flag to *N*, thereby committing the object creation.
5. Next, we create the new instance as a child of the supporting Software Installation object, by parsing the ID of the FOREIGN-CONFIGURATION Software Resource Template to the utility workflow ITSO_HostingEnvironment_Host. The details of this process are described in 10.10.4, “ITSO_HostingEnvironment_Host” on page 500.
6. Once the DCM objects have been created, we can physically create the new Apache instance. To ease the development process and increase readability, this is performed by a specialized workflow named Trade_Web_Add_Instance. See the details in “Trade_Web_Add_Instance” on page 483. In addition to creating the Apache instance, this workflow is also responsible for customization, distribution of the index.html file, and starting the Apache server.
7. Should the unlikely situation occur that errors are detected during the instance creation, we have to delete all the foreign objects that were created by this workflow, and set the value of the pending flag back to *Y*, in order to ensure that the SoftwareInstallable.Install logical device operation cleans up after a failing installation.

Archived

The details of the Trade_Web_Module_Installable_Install workflow are listed in Example 10-54.

Example 10-54 Trade_Web_Module_Installable_Install workflow

```
workflow Trade_Web_Module_Installable_Install(in SoftwareID, in DeviceID, in SoftwareResourceTemplateID)
implements SoftwareInstallable.Install LocaleInsensitive

#####################################################
## 1:  initialize and validate input
#####################################################
var simulation
Trade_Get_Simulation(simulation)

var workflowName="Trade_DBServer_Module_Installable_Install"
var message = Jython["<b>" + workflowName + ": </b>"]
var msg

var SoftwareInstallableID=SoftwareID
var ServerID=DeviceID
var ServerName=DCMQuery(/server[@id=$ServerID]/@name)
var DeploymentRequestID
java:com.thinkdynamics.kanaha.de.javaplugin.GetCurrentDeploymentRequestId(DeploymentRequestID)
var SoftwareInstallationID

# get the ID of the SoftwareModule
var
SoftwareModuleID=DCMQuery(/softwareinstallable[@id=$SoftwareInstallableID]/softwareinstallationmechanism/@
moduleid)

try
  ####################################################################
  ## 2: Find the supporting installation
  ####################################################################
  var requirementType="WEBSERVER"
  var supportingSoftwareInstallationID
  ITSO_Find_Supporting_Installation_for_Module_Reqirements(requirementType, SoftwareInstallableID,
ServerID, supportingSoftwareInstallationID)

  ####################################################################
  ## 3: Find install_dir of the HTTP Server
  ####################################################################
  var install_dir_parmname="install_dir"
  var install_dir

  Trade_Get_Hosting_SRTParameter_Or_Default(SoftwareInstallableID, supportingSoftwareInstallationID,
install_dir_parmname, install_dir)
  ## fail if no values for install_dir was found
  if Jython[not install_dir] then
    msg= Jython[message + " Cannot find a value for template parameter " + install_dir_parmname + "in SRT
" + DBServerInstallationTemplateID]
    log error msg
    throw missingParameterException msg
  endif
```

```
  ##########################################################
  ## 4: Commit the creation of this SoftwareInstallation
  ##########################################################
  SoftwareInstallationID=DCMQuery(/softwareinstallation[@managedsystemid=$ServerID and
@softwaremoduleid=$SoftwareModuleID and @pending="Y"]/@id)

java:com.ibm.tivoli.orchestrator.datacentermodel.helper.SoftwareHelper#updateResourceFlag(SoftwareInstalla
tionID, "false")

  ######################################################################
  ## 5: Create the Foreign Instance object
  ######################################################################
  var installationResourceType=DCMQuery(/softwareresourcetype[@name="INSTALLATION"]/@id)
  var instanceResourceType=DCMQuery(/softwareresourcetype[@name="INSTANCE"]/@id)
  var foreignResourceType=DCMQuery(/softwareresourcetype[@name="FOREIGN-CONFIGURATION"]/@id)
  var
installationSRT=DCMQuery(/softwareinstallation[@id=$SoftwareInstallationID]/softwareresourcetemplate[@soft
wareresourcetype=$installationResourceType]/@id)
  var foreignSRTID=DCMQuery(/childsoftwareresourcetemplate[@parenttemplateid=$installationSRT]/@id)
  array objects
  ITSO_HostingEnvironment_Host(supportingSoftwareInstallationID, SoftwareModuleID, foreignSRTID, objects)

  ######################################################################
  ## 6: Create the new Trade Apache Instance
  ######################################################################
  foreach objectID in objects do
    var targetResourceID=DCMQuery(/softwareresource[@id=$objectID]/@parentresourceid)
    var srtID=DCMQuery(/softwareresource[@id=$objectID]/@resourcetemplateid)
   log info Jython["......About to call Trade_Web_Module_Add_Instance("+targetResourceID+", "+srtID+")"]
    Trade_Web_Add_Instance(targetResourceID, srtID)
  done

catchall exception
  ######################################################################
  ## 7: Set the status pending if creation of freign objects failed
  ######################################################################
  msg = Jython[message + exception]
  # delete foreign resources
  foreach objectID in objects do
    log debug Jython["about to delete " + objectID]
    var objID=DCMQuery(/softwareresource[@id=$objectID]/@id)
    if Jython[ objID != None] then
      DCMDelete(/softwareresource[@id=$objectID]/@id)
    endif
  done

  # uncommit the Installation if we failed

java:com.ibm.tivoli.orchestrator.datacentermodel.helper.SoftwareHelper#updateResourceFlag(SoftwareInstalla
tionID, "true")
  log error msg
  rethrow
```

10.9.3 The Trade_Web_Module_Instance_Driver

During the creation of the foreign Apache service instance (performed during installation of the Trade Web Module), all of the instances will be associated with the Trade_Web_Module_Instance device driver. This device driver will support the following logical device operations:

- SoftwareInstance.Start
- SoftwareInstance.Stop
- SoftwareInstance.Remove

The driver overview is shown in Figure 10-60.

Figure 10-60 Trade_Web_Module_Instance_Driver overview

The functionality of the logical operations and the workflows implementing them operations should be self-explanatory.

We could also have decided to implement the Software Resource.UpdateDCM LDO, as was the case for the DB2Server module (see “Trade_DBServer_Module_Instance_UpdateDCM” on page 363); however, for now, we disregard this LDO.

Figure 10-61 on page 477 shows the workflow assignments to the LDOs of the Trade_Web_Module_Instance_Driver device driver.

Archived

Figure 10-61 Trade_Web_Module_Instance_Driver

In order to automatically create the Trade_Web_Module_Instance_Driver during installation of the Trade Automation Package, we ensured that the XML statements shown in Example 10-55 were present in the <device-models> section of the tcdriver.xml file for the Trade Automation Package.

Example 10-55 Trade_Web_Module_Instance_Driver xml definitions

```
<device-model name="Trade_Web_Module_Instance_Driver" category="Trade">
    <workflow name="Trade_Web_Module_Instance_RemoveInstance" />
    <workflow name="Trade_Web_Module_Instance_Start" />
    <workflow name="Trade_Web_Module_Instance_Stop" />
</device-model>
```

In the following sections, we will describe the functionality of the Trade_Web_Module_Instance_Driver workflows, which are:

- Trade_Web_Module_Instance_Start
- Trade_Web_Module_Instance_Stop
- Trade_Web_Module_Instance_RemoveInstance

Trade_Web_Module_Instance_Start

This workflow is used to start the Apache service on the hosting system. Because of the minute differences between the start and the stop operation, the actual interaction with the system for both LDOs is handled by a common workflow named Trade_Web_Instance_StartStop. Please see “Trade_Web_Instance_StartStop” on page 493 for details.

Archived

Example 10-56 shows the simple workflow used to start the Apache service.

Example 10-56 Trade_Web_Module_Instance_Start workflow

```
workflow Trade_Web_Module_Instance_Start(in SoftwareInstanceID) implements SoftwareInstance.Start
LocaleInsensitive

Trade_Web_Instance_StartStop(SoftwareInstanceID, “start”)
```

Trade_Web_Module_Instance_Stop

As described in the previous, we use the common Trade_Web_Instance_StartStop workflow to control the operational state of the Apache service. See “Trade_Web_Instance_StartStop” on page 493 for details.

Example 10-57 shows how we set up the calls to Trade_Web_Instance_StartStop to stop the Apache service from acting as a front end to the Trade application.

Example 10-57 Trade_Web_Module_Instance_Stop workflow

```
workflow Trade_Web_Module_Instance_Stop(in SoftwareInstanceID) implements SoftwareInstance.Stop
LocaleInsensitive

Trade_Web_Instance_StartStop(SoftwareInstanceID, “stop”)
```

Trade_Web_Module_Instance_RemoveInstance

To remove the foreign Apache instances created during installation of the Trade Web Module, they have all been assigned the Trade_Web_Module_Installable_Driver through the specification in the INSTANCE Software Resource Template in the Trade Web Module (see Figure 10-56 on page 466). This implies that the objectID of the Software Instance to be deleted is passed directly to the Trade_Web_Module_Instance_RemoveInstance workflow, and thus, we do not need to identify the foreign instances to be deleted.

To remove the Apache instances, we build the following main functionality into the Trade_Web_Module_Instance_RemoveInstance workflow:

1. Initialize to control error messages and get control and navigation information from the DCM and the Software Resource Templates.
2. Call the SoftwareInstance.Stop logical device operation to ensure that the Apache Service is stopped. This will in turn call our Trade_Web_Module_Instance_Stop workflow described “Trade_Web_Module_Instance_Stop” on page 478.

3. Run a scriptlet to remove the Apache service from the Windows environment using the **apache -n <instancename> -k uninstall** command, and remove the application specific DocumentRoot directory as well as the configuration file.
4. Throw any exceptions that may be raised during execution back the caller.

Example 10-58 shows the Trade_Web_Module_Instance_RemoveInstance workflow.

Example 10-58 Trade_Web_Module_Instance_RemoveInstance workflow

```
workflow Trade_Web_Module_Instance_RemoveInstance(in SoftwareInstanceID) implements
SoftwareInstance.RemoveInstance LocaleInsensitive

################################################################
## 1: initialize and gather base data cor control and navigation
################################################################
var simulation
Trade_Get_Simulation(simulation)

var workflowName="Trade_Web_Module_Instance_RemoveInstance"
var message=Jython["<b>" + workflowName + ": </b>"]
var msg

try

  var ServerID=DCMQuery(/softwareinstance[@id=$SoftwareInstanceID]/@managedsystemid)
  var SoftwareResourceTemplateID=DCMQuery(/softwareinstance[@id=$SoftwareInstanceID]/@resourcetemplateid)
  var httpServerInstallDir=DCMQuery(/templateparam[@templateid=$SoftwareResourceTemplateID and
@name="install_dir"]/@value)
  var httpInstanceName=DCMQuery(/templateparam[@templateid=$SoftwareResourceTemplateID and
@name="resource-name"]/@value)
  var httpServerConfDir=DCMQuery(/templateparam[@templateid=$SoftwareResourceTemplateID and
@name="config_dir"]/@value)
  var httpServerDocsDir=DCMQuery(/templateparam[@templateid=$SoftwareResourceTemplateID and
@name="documents_dir"]/@value)
  var httpServerDocRoot=Jython[httpServerInstallDir + "/" + httpServerDocsDir]
  var httpServerBaseConf = Jython[httpInstanceName + ".conf"]
  httpServerDocRoot=Jython[httpServerDocRoot + "/" + httpInstanceName]

  ################################################################
  ## 2: Stop the appache service
  ################################################################
  SoftwareInstance.Stop(SoftwareInstanceID)

  ################################################################
  ## 3: Run scriptlet to remove the instance and directories
  ################################################################
  if Jython[simulation == "yes"] then
    log info Jython[<b>simulation:</b>...about to call scriplet to remove http instance"]
    log info Jython["         httpServerInstallDir: " + httpServerInstallDir]
    log info Jython["         httpServerConfDir: " + httpServerConfDir]
```

```
    log info Jython["         httpInstanceName: " + httpInstanceName]
    log info Jython["         httpServerBaseConf: " + httpServerBaseConf]
    log info Jython["         httpServerDocRoot: " + httpServerDocRoot]
  else

    scriptlet(httpServerInstallDir, httpServerConfDir, httpInstanceName, httpServerBaseConf,
httpServerDocRoot) language = bash target = DCMQuery(/server[@id=$ServerID]/@id) <<EOF
      confPath=`cygpath -u "${httpServerInstallDir}/${httpServerConfDir}"`
      binPath=`cygpath -u "${httpServerInstallDir}/bin"`
      httpServerDocRoot=`cygpath -u "${httpServerDocRoot}"`

      # remove the service
      ${binPath}/apache -n "${httpInstanceName}" -k uninstall

      # remove the DocumentRoot direcotory
      if [ -d ${httpServerDocRoot} ]; then
        rm -Rf "${httpServerDocRoot}"
      else
        msg="could not delete ${httpServerDocRoot}"
        TIOlog "error" "$msg"
        TIOthrow "scriptExecutionError" "$msg"
      fi

      # remove the conf file C:/ibm/ihs20/trade3.conf
      if [ -e "${confPath}/${httpServerBaseConf}" ]; then
        rm -f "${confPath}/${httpServerBaseConf}"
      else
        msg="could not delete ${httpServerDocRoot}"
        TIOlog "error" "$msg"
        TIOthrow "scriptExecutionError" "$msg"
      fi
    EOF

  endif
catchall exception
  ##############################################################
  ## 4: error handling
  ##############################################################
  msg = Jython[message + exception]
  log error msg
  rethrow
endtry
```

10.9.4 The Trade_Web_Module_Installation_Driver

During the creation of the Trade Web Module Installation objects, the Trade_Web_Module_Installation_Driver will be assigned to these objects. We will not have any Instance objects associated directly with Software Installation objects (remember that all the instances will be created as children of the hosting platform installation through the FOREIGN-CONFIGURATION Software Resource Template). Therefore, the only only logical device operation we need to

Archived

support in the device driver is SoftwareInstallation.Uninstall, as shown in Figure 10-62.

Figure 10-62 Trade_Web_Module_Installation_Driver overview

Figure 10-63 shows that the only supported logical device operation for this driver is SoftwareInstallation.Uninstall.

Figure 10-63 Trade_Web_Module_Installation_Driver

The mission of the Trade_Web_Module_Installation_Uninstall workflow, which implements the SoftwareInstallation.Uninstall LDO, is to remove the Trade Web Module Installation and all related foreign instances created during the installation.

Example 10-59 shows the XML statements included in the tcdriver.xml for defining the Trade_Web_Modle_Installation_Driver during the installation of the Trade Automation Package.

Example 10-59 Trade_Web_Module_Installation_Driver xml definitions

```
<device-model name="Trade_Web_Module_Installation_Driver"
category="Trade">
    <workflow name="Trade_Web_Module_Installation_Uninstall" />
</device-model>
```

The following section contains a detailed description of the workflow used to remove the Trade Web Module installations.

Archived

Trade_Web_Module_Installation_Uninstall

As already stated, the purpose of this workflow is to remove Trade Web Module Installation Software Installation objects and all related SoftwareInstance objects from a server that hosts the Installation.

The main challenge is to identify the foreign instance objects that should be removed as part of the uninstallation. This is achieved by implementing the following high-level functionality:

1. Initialize to get basic information for error control, logging, and navigation.
2. Use the source template of the Installation object to identify all foreign instances and issue the SoftwareInstance.RemoveInstance logical device operation for each one of them. This will in turn invoke the Trade_Web_Module_Instance_RemoveInstance workflow, which is responsible for the physical removal.
3. Handle errors.

The listing of the Trade_Web_Module_Installation_Uninstall is presented in Example 10-60.

Example 10-60 Trade_Web_Module_Installation_Uninstall workflow

```
workflow Trade_Web_Module_Installation_Uninstall(in SoftwareInstallationID) implements
SoftwareInstallation.Uninstall LocaleInsensitive

##############################################################
##  1: Initialize and gather base control data
##############################################################
var simulation
Trade_Get_Simulation(simulation)

var workflowName="Trade_Web_Module_Instance_RemoveInstance"
var message=Jython["<b>" + workflowName + ": </b>"]
var msg

var ServerID = DCMQuery(/softwareinstallation[@id=$SoftwareInstallationID]/@managedsystemid)

##############################################################
##  2: Remove related instances
##############################################################
try
  # find the foreignconfiguration srt
  var instanceSwResourceTypeID=DCMQuery(/softwareresourcetype[@name="INSTANCE"]/@id)
  var
foreignSRTID=DCMQuery(/softwareinstallation[@id=$SoftwareInstallationID]/softwareresourcetemplate/childsof
twareresourcetemplate/@id)
  var instanceSRTID=DCMQuery(/childsoftwareresourcetemplate[@parenttemplateid=$foreignSRTID and
@softwareresourcetype=$instanceSwResourceTypeID]/@id)

  # find the source
```

```
  var sourceTemplateID=DCMQuery(/softwareresourcetemplate[@id=$instanceSRTID]/@sourcetemplateid)

  # find all the cloned instances
  foreach srtID in
DCMQuery(/softwareresource[@managedsystemid=$ServerID]/softwareresourcetemplate[@sourcetemplateid=$sourceT
emplateID and @id != $instanceSRTID]/@id) do
    var instanceID=DCMQuery(/softwareresourcetemplate[@id=$srtID]/softwareresource/@id)
    log info Jython["...calling SoftwareInstance.RemoveInstance(" + instanceID + ")"]
    ##  remove foreign instances
    SoftwareInstance.RemoveInstance(instanceID)
  done
catchall exception
  ################################################################
  ## 3: error handling
  ################################################################
  msg = Jython[message + exception]
  log error msg
  rethrow
endtry
```

10.9.5 Trade Web Module Helper Workflows

To support the provsioning of the Trade Web Module, we developed three helper workflows, with very specific functionality. These workflows are:

- Trade_Web_Add_Instance
- Trade_Web_Add_Configuration
- Trade_Web_Instance_StartStop

The Trade_Web_Instance_StartStop is used to provide similar functionality to the workflows implementing instance related logical device operations, and the first two workflows are used during installation and customization. All three workflows are described in the following sections.

Trade_Web_Add_Instance

This is the main workflow responsible for creating and customizing a new Apache instance. Customization is performed by calling the Trade_Web_Add_Configuration workflow after the creation of the instance through the **apache -n <service-name> -k install** command. This implies that the primary tasks for this workflow are to create the control structures required by the Apache server, and use the platform interfaces to create the service.

Archived

This workflow is called from the Trade_Web_Module_Installable_Install workflow and receives the objectID of the newly created softwareInstance object that represents the Apache service in the DCM. The main functions of the Trade_Web_Add_Instance workflow are:

1. Initialize and gather information for flow control, error handling, and navigation.
2. Get and save the installation directory location from the hosting HTTP Server installation.
3. Get the Apache service configuration parameters from the Instance Software Resource Template.
4. Execute a scriptlet on the target server to:
   a. Clone the default httpd.conf file.
   b. Update the new configuration file to specify customized values for AfpaPort, DocumentRoot, and DirectoryIndex.
   c. Create the service through a scriptlet and issuing the **`apache -n <serviceName> -k install -f <configFile>`** command.
5. Distribute our index.html file to the target server using the FileRepository.GetFile LDO. We chose not to use the SoftwareInstallable.Distribute LDO for this task because of performance reasons. All the necessary parameters has already been collected, so there is no need to do it again, which would have been the case had we chosen to go with the LDO implmementation.
6. Call Trade_Web_Add_Configuration (see “Trade_Web_Add_Configuration” on page 488) to customize the index.html file. The customization was implemented as an individual workflow in order to ease test and development.
7. Start the new Apache service using the SoftwareInstance.Start logical device operation. See “Trade_Web_Module_Instance_Start” on page 477 for details.
8. Return any exceptions to the caller.

Example 10-61 shows the detailled implementation of the Trade_Web_Add_Instance workflow.

Example 10-61 Trade_Web_Add_Instance workflow

```
workflow Trade_Web_Add_Instance(in SoftwareInstallationID, in SoftwareResourceTemplateID)
LocaleInsensitive

####################################
##  1:  initialize and validate imput
####################################
var simulation
Trade_Get_Simulation(simulation)
```

```
var workflowName=”Trade_Web_Add_Instance”
var message=Jython[“<b>” + workflowName + “: </b>”]
var msg

var ServerID=DCMQuery(/softwareinstallation[@id=$SoftwareInstallationID]/@managedsystemid)
var
SoftwareModuleID=DCMQuery(/softwareresourcetemplate[@id=$SoftwareResourceTemplateID]/@softwaremoduleid)
var InstanceID=DCMQuery(/softwareresourcetemplate[@id=$SoftwareResourceTemplateID]/softwareresource/@id)

try
  ##########################################################################
  ##  2: Get and save the installation directory from the hosting Installation
  ##########################################################################
  var install_dir_parmName=”install_dir”
  var install_dir

  var sourceModuleID
  foreach id in
DCMQuery(/softwareinstance[@parentresourceid=$SoftwareInstallationID]/softwareresourcetemplate/sourcesoftw
areresourcetemplate/softwaremodule/@id) do
    # get the first one
    sourceModuleID=id
    break
  done
  var sourceInstallationID
  foreach id in DCMQuery(/softwareinstallation[@managedsystemid=$ServerID and
@softwaremoduleid=$sourceModuleID]/@id) do
    # get the first one
    sourceInstallationID=id
    break
  done

  var sourceInstallationSRT=DCMQuery(/softwareinstallation[@id=$sourceInstallationID]/@resourcetemplateid)
  var sourceInstallableID=DCMQuery(/softwareinstallation[@id=$sourceInstallationID]/@softwareproductid)
  Trade_Get_Hosting_SRTParameter_Or_Default(sourceInstallableID, SoftwareInstallationID,
install_dir_parmName, install_dir)
  if Jython[not install_dir] then
    msg = Jython[message + “ could not find value for install_dir”]
    throw TradeParameterException msg
  endif

  ##  Insert the value for installation directory in the SRT for reuse
  DCMInsert parent=DCMQuery(/softwareresourcetemplate[@id=$SoftwareResourceTemplateID]/@id) <<EOFXML
     <template-param name=”install_dir” value=”$install_dir” is-changeable=”true” />
  EOFXML

  ##########################################################################################
  ##  3: Get the http server configuration parameters from the INSTANCE software resource template
  ##########################################################################################
  var httpServerInstanceName=DCMQuery(/templateparam[@templateid=$SoftwareResourceTemplateID and
@name=”resource-name”]/@value)
  var httpServerConfDir=DCMQuery(/templateparam[@templateid=$SoftwareResourceTemplateID and
@name=”config_dir”]/@value)
```

```
  var httpServerBaseConf=DCMQuery(/templateparam[@templateid=$SoftwareResourceTemplateID and
@name="config_file"]/@value)
  var httpServerDocsDir=DCMQuery(/templateparam[@templateid=$SoftwareResourceTemplateID and
@name="documents_dir"]/@value)
  var httpServerBasePort=DCMQuery(/templateparam[@templateid=$SoftwareResourceTemplateID and
@name="port"]/@value)
  var httpServerDocRoot=Jython[install_dir + "/" + httpServerDocsDir + "/" + httpServerInstanceName]
  var newIndexFileName=DCMQuery(/templateparam[@templateid=$SoftwareResourceTemplateID and
@name="indexFileName"]/@value)

  ######################################################
  ## 4: Create new APACHE instance
  ######################################################
  log info Jython["<b>Creating new apache instance " + httpServerInstanceName + " on port " +
httpServerBasePort + "</b>"]
  if Jython[simulation.lower() == "yes"] then
     log info Jython["       install_dir: " + install_dir]
     log info Jython["       httpServerConfDir: " + httpServerConfDir]
     log info Jython["       httpServerBaseConf: " + httpServerBaseConf]
     log info Jython["       httpServerDocsDir: " + httpServerDocsDir]
     log info Jython["       httpServerBasePort: " + httpServerBasePort]
     log info Jython["       httpServerDocRoot: " + httpServerDocRoot]
     log info Jython["       newIndexFileName: " + newIndexFileName]
     log info Jython["       httpServerInstanceName: " + httpServerInstanceName]
  else
    scriptlet(install_dir, httpServerInstanceName, httpServerConfDir, httpServerBaseConf,
httpServerBasePort, httpServerDocRoot, newIndexFileName) language = bash target =
DCMQuery(/server[@id=$ServerID]/@id) <<EOF
      file=`date -d now +%s`
      file="/tmp/T"$file
      touch $file

      confPath=`cygpath -u "${install_dir}/${httpServerConfDir}"`
      binPath=`cygpath -u "${install_dir}/bin"`

      # make documentRoot
      if [ ! -e ${httpServerDocRoot} ]; then
        mkdir -p `cygpath -u ${httpServerDocRoot}`
      fi

      # copy C:/ibm/ihs20/httpd.conf to C:/ibm/ihs20/trade3.conf
      if [ -e ${confPath}/${httpServerBaseConf} ]; then
        cp -f "${confPath}/${httpServerBaseConf}" "${confPath}/${httpServerInstanceName}.conf"
      else
        msg="cannot find httpd.conf"
        TIOlog "error" "$msg"
        TIOthrow "scriptExecutionError" "$msg"
      fi

      # set new base port
      oldBasePort="AfpaPort.*"
      newBasePort="AfpaPort ${httpServerBasePort}"
      script="s/${oldBasePort}/${newBasePort}/g"
      printf %s "$script" > ${file}
```

```
      sed -i -f ${file} “${confPath}/${httpServerInstanceName}.conf”

      # replace DocumentRoot
      oldDocRoot=”DocumentRoot.*\\\”.*\\\””
      newDocRoot=‘echo ${httpServerDocRoot} | awk -F “/” ‘{ for (i=1;i<NF;i++) { printf “%s\\\/”,$i };
printf “%s”,$i }’‘
      newDocRoot=”DocumentRoot \”${newDocRoot}\””
      script=”s/${oldDocRoot}/${newDocRoot}/g”
      printf %s “$script” > ${file}
      sed -i -f ${file} “${confPath}/${httpServerInstanceName}.conf”

      # replace DirectoryIndex
      oldDirIdx=”DirectoryIndex.*”
      newDirIdx=”DirectoryIndex ${newIndexFileName} index.html.var”
      script=”s/${oldDirIdx}/${newDirIdx}/g”
      printf %s “$script” > ${file}
      sed -i -f ${file} “${confPath}/${httpServerInstanceName}.conf”

      # cleanup
      rm -f ${file}

      # create and start service
      TIOlog “info” “creating apache service named ${httpServerInstanceName}”
      ${binPath}/apache -n “${httpServerInstanceName}” -k install -f
“${install_dir}/${httpServerConfDir}/${httpServerInstanceName}.conf”
    EOF
  endif
  ######################################################
  ## 5: Distribute the files to the new DocumentRoot
  ######################################################
  var ReturnCode
  var ReturnErrorString
  var ReturnResult
  var confiurationTypeID=DCMQuery(/softwareresourcetype[@name=”CONFIGURATION”]/@id)
  var configurationSRTID=DCMQuery(/softwareresourcetemplate[@parenttemplateid=$SoftwareResourceTemplateID
and @softwareresourcetype=$confiurationTypeID]/@id)
  var configurationID=DCMQuery(/softwareresource[@resourcetemplateid=$configurationSRTID]/@id)
  var appServerName=DCMQuery(/templateparam[@templateid=$configurationSRTID and
@name=”WASapplicationName”]/@value)
  var appServerBasePort=DCMQuery(/templateparam[@templateid=$configurationSRTID and
@name=”WASport”]/@value)
  # convert to unix format
  var
unixServerInstallDir=Java[com.thinkdynamics.kanaha.util.PathHelper#convertToCygwinPath(httpServerDocRoot)]
  var sourceSRTID=DCMQuery(/softwareresourcetemplate[@id=$SoftwareResourceTemplateID]/@sourcetemplateid)
  var installableID=DCMQuery(/softwareinstallation[@managedsystemid=$ServerID and
@softwaremoduleid=$sourceModuleID]/@softwareproductid)
  var repositoryID = DCMQuery(/softwareinstallable[@id=$installableID]/filerepository/@id)
  var fileName=DCMQuery(/softwareinstallable[@id=$installableID]/file/@name)
  var filePath=DCMQuery(/softwareinstallable[@id=$installableID]/file/@path)

  log info Jython[“<b>.. copying files..</b>”]
  if Jython[simulation.lower() == “yes”] then
```

```
     log info Jython[“<b>simulation: </b>.. about to distribute “ + filePath + “/” + fileName + “ to “ +
newFileName + “ in “ + unixServerInstallDir + “ on “ + ServerID]
  else
    FileRepository.GetFile(repositoryID, filePath, fileName, ServerID, unixServerInstallDir,
newIndexFileName, “120”)
  endif

  ################################################################
  ## 6: Modify the incdex.html file to reflect current environment
  ################################################################
  Trade_Web_Add_Configuration(configurationID)

  ################################################################
  ## 7: Start the instance
  ################################################################
  log info Jython[“<b>.. starting apache server..</b>”]
  var SoftwareInstanceID=DCMQuery(/softwareinstance[@resourcetemplateid=$SoftwareResourceTemplateID]/@id)
  SoftwareInstance.Start(SoftwareInstanceID)

catchall exception
  ################################################################
  ## 8: Handle errors
  ################################################################
  msg = Jython[message + “......got exception ‘” + exception + “‘”]
  log error msg
  rethrow
endtry
```

Trade_Web_Add_Configuration

The purpose of this workflow is to update the index.html file that acts as a front end to the Trade application to ensure that a link to the actual implementation is added to the file. To ensure this, we have to identify the server hosting the Trade Application Module (using the non-hosting requirements defined for the Trade Web Module) and gather the relevant configuration data, such as the IP address and port number.

The outline of the Trade_Web_Add_Configuration workflow is:

1. Initialize and gather data for control, navigation, and error handling.
2. Get the basic data about the Apache service, and use the convertToCygwinPath Java program to ensure that the DocumentRoot directory name is provided in UNIX format.
3. Find the Software Installation supporting the non-hosting APPLICATION requirements for the Trade Web Module. As described in “Capabilities and requirements” on page 459, these were set up to ensure that an installation of the Trade application, represented by the Trade Application Module, exists prior to installation of the Trade Web Module.

4. Find all resource templates, and identify the INSTANCE and related CONFIGURATION software templates, knowing the resource template hierarchy that was implemented by the Trade Application Module.
5. Gather the necessary configuration data from the Trade Application Module Installation. We need the URL to start the Trade application as well as the port number that the server (instance) is listening to.
6. Now, get the managed IP address from the server hosting the Trade Application Module.
7. Update the index.html file for the Apache service, using a Perl scriptlet, by updating or adding the Trade URL.
8. Return any exceptions to the caller.

The details of the Trade_Web_Add_Configuration workflow are shown in Example 10-62.

Example 10-62 Trade_Web_Add_Configuration workflow

```
workflow Trade_Web_Add_Configuration(in SoftwareConfigurationID) LocaleInsensitive

####################################
##  1:  initialize
####################################
var simulation
Trade_Get_Simulation(simulation)

var workflowName=”Trade_Web_Add_Configuration”
var message=Jython[“<b>” + workflowName + “: </b>”]
var msg

var configurationSRTID=DCMQuery(/softwareresource[@id=$SoftwareConfigurationID]/@resourcetemplateid)
var instanceID=DCMQuery(/softwareresource[@id=$SoftwareConfigurationID]/@parentresourceid)
var installationID=DCMQuery(/softwareinstance[@id=$instanceID]/@parentresourceid)
var sourceTemplateID=DCMQuery(/softwareresourcetemplate[@id=$configurationSRTID]/@sourcetemplateid)
var SoftwareModuleID=DCMQuery(/softwareresourcetemplate[@id=$sourceTemplateID]/@softwaremoduleid)
var ServerID=DCMQuery(/softwareinstance[@id=$instanceID]/@managedsystemid)

try

  ##################################################
  ## 2: get parameters from the HTTP Server Instance
  ##################################################
  var instanceSRTID=DCMQuery(/softwareinstance[@id=$instanceID]/@resourcetemplateid)
  var installDir=DCMQuery(/templateparam[@templateid=$instanceSRTID and @name=”install_dir”]/@value)
  var documentsDir=DCMQuery(/templateparam[@templateid=$instanceSRTID and @name=”documents_dir”]/@value)
  var instanceName=DCMQuery(/templateparam[@templateid=$instanceSRTID and @name=”resource-name”]/@value)
  var documentRoot=Jython[installDir + “/” + documentsDir + “/” + instanceName]
  # transform documentRoot path name to unix format
  documentRoot=Java[com.thinkdynamics.kanaha.util.PathHelper#convertToCygwinPath(documentRoot)]
```

```
  var indexFileName=DCMQuery(/templateparam[@templateid=$instanceSRTID and @name="indexFileName"]/@value)
  var applName=DCMQuery(/templateparam[@templateid=$configurationSRTID and
@name="applicationName"]/@value)
  var indexFile=Jython[documentRoot + "/" + indexFileName]

  ##########################################################################
  ##  3: find the installation honoring the non-hosting APPLICATION requirement
  ##########################################################################
  var requirementType="APPLICATION"
  var earInstallationID
  ITSO_Find_Supporting_Installation_for_Module_Reqirements(requirementType, SoftwareModuleID, <null>,
earInstallationID)
  log info Jython["Installation " + earInstallationID + " honors the " + requirementType + "
requirements"]

  ######################################################################
  ## 4: Find the Instance and Configuration srt on the remote installation
  ######################################################################
  var   templateID=DCMQuery(/softwareinstallation[@id=$earInstallationID]/@resourcetemplateid)
  array IDs
  ## build an array of all templates related to the earInstallation
  IDs[0]=templateID
  var j=arraysize(IDs)
  var i=0
  var end
  while Jython[int(i) < int(j) ] do
    var currentTemplateID=IDs[i]
    foreach ID in DCMQuery(/softwareresourcetemplate[@parenttemplateid =
$currentTemplateID]/@id[orderBy@id]) do
      end=arraysize(IDs)
      IDs[end] = ID
    done
    j = arraysize(IDs)
    i = Jython[int(i) + 1]
  done

  # Identify INSTANCE and CONFIGURATION templats
  var swInstanceTypeID=DCMQuery(/softwareresourcetype[@name="INSTANCE"]/@id)
  var swConfigurationTypeID=DCMQuery(/softwareresourcetype[@name="CONFIGURATION"]/@id)
  var instanceTemplateID
  var configurationTemplateID

  foreach templateID in IDs do
    var templateType=DCMQuery(/softwareresourcetemplate[@id=$templateID]/@softwareresourcetype)
    var templateName=DCMQuery(/softwareresourcetemplate[@id=$templateID]/@name)
    array childs = DCMQuery(/softwareresourcetemplate[@parenttemplateid=$templateID]/@id[orderBy@id])

    foreach ID in childs do
      var childType=DCMQuery(/softwareresourcetemplate[@id=$ID]/@softwareresourcetype)
      var childName=DCMQuery(/softwareresourcetemplate[@id=$ID]/@name)
      if Jython[templateType == swInstanceTypeID and childType == swConfigurationTypeID] then
        instanceTemplateID=templateID
        configurationTemplateID=ID
        break
```

```
      endif
    done

    if Jython[configurationTemplateID] then
      break
    endif
  done

  var earInstanceResourceSRTID
  var earConfigurationResourceSRTID

  if Jython[configurationTemplateID and configurationTemplateID != ""] then
    earInstanceResourceSRTID=instanceTemplateID
    earConfigurationResourceSRTID=configurationTemplateID
  else
    throw exception "could not find the configuration template as child of instance"
  endif

  ######################################################################
  ## 5: Get the parameters needed to catalog the trade application instance
  ######################################################################
  var
applServerPort=DCMQuery(/softwareresourcetemplate[@id=$earInstanceResourceSRTID]/templateparam[@name="serv
erPort"]/@value)
  if Jython[ not applServerPort ] then
    msg=Jython[message + "no value found for parameter serverPort in SRT " + earInstanceResourceSRTID]
    log error msg
    throw parameterValidationException msg
  endif

  var
applGoTradeURL=DCMQuery(/softwareresourcetemplate[@id=$earConfigurationResourceSRTID]/templateparam[@name=
"goTradeURL"]/@value)
  if Jython[ not applGoTradeURL ] then
    msg=Jython[message + "no value found for parameter goTradeURL in SRT " +
earConfigurationResourceSRTID]
    log error msg
    throw parameterValidationException msg
  endif

  var
earServerID=DCMQuery(/softwareinstance[@resourcetemplateid=$earInstanceResourceSRTID]/@managedsystemid)

  ######################################################################
  ## 6: Find the IP address of the system hosting the Trade EAR Application
  ######################################################################
  #build array of managed and non-management IP addresses
  array IPs
  IPs=DCMQuery(/server[@id=$earServerID]/networkinterface[@managed="Y" and @management="N"]/@ipaddress)
  j = arraysize(IPs)
  if Jython[int(j) == 0] then
    # find the management address
    IPs=DCMQuery(/server[@id=$earServerID]/networkinterface[@management="Y"]/@ipaddress)
    j = arraysize(IPs)
```

```
  endif
  # use the first one
  var serverIP = IPs[0]


  ####################################################################
  ## 7: Call the scriptlet to perform the db2 updates on the DB2 Client
  ####################################################################
  if Jython[simulation == "yes"] then
    log info Jython["<b>simulation: </b>.. about to call scriptlet to configure " + indexFile]
    log info Jython["                        serverIP: " + serverIP]
    log info Jython["                        indexFile: " + indexFile]
    log info Jython["                        applServerPort: " + applServerPort]
    log info Jython["                        applName: " + applName]
    log info Jython["                        applGoTradeURL: " + applGoTradeURL]
  else
    scriptlet (serverIP, indexFile, applServerPort, applName, applGoTradeURL) language = perl target =
DCMQuery(/server[@id=$ServerID]) credentialskey = "master" timeout = 400  <<EOF
      $backupFile = $indexFile . "\.BKP";
      system "dos2unix $indexFile";
      system "cp $indexFile $backupFile";
      $done = "";
      $url = "http\:\/\/$serverIP\:$applServerPort$applGoTradeURL";
      $New_Href = "<P><a href\=\"$url\">$applName<\/a><\/P>\n";
      open (OLDINDEX, "< $backupFile");
      open (NEWINDEX, "> $indexFile");
      while (<OLDINDEX>){
        $line = $_;
        if ($line =~ m/href/){
          if ($line =~ m/$applGoTradeURL/){
           print NEWINDEX $New_Href;
           $done = "OK";
          }
          else {
            print NEWINDEX $line;
          }
        }
        else {
          if ($line =~ m/RedBooks/) {
             if ($done !~ m/OK/) {
               print NEWINDEX $New_Href;
               $done = "OK";
             }
          }
          print NEWINDEX $line;
        }
      }
      return 0;
    EOF
  endif
catchall exception
  ##################################
  ## 8: Handle errors
  ##################################
```

```
  msg = Jython[message + " " + exception]
  log error msg
  rethrow
endtry
```

Trade_Web_Instance_StartStop

This workflow is used to start and stop the Trade Apache service using the **apache -n <serviceName> -k <operation>** command.

The outline is:

1. Initialize and gather control data.
2. Determine the operation (start or stop). Use `start` if no value was provided.
3. Execute the Apache command on the server using a scriptlet.
4. Handle errors.

The details are shown in Example 10-63.

Example 10-63 Trade_Web_Instance_StartStop workflow

```
workflow Trade_Web_Instance_StartStop (in SoftwareInstanceID, in operation)  LocaleInsensitive

##############################################
## 1: initialize
##############################################
var simulation
Trade_Get_Simulation(simulation)

var workflowName="Trade_Web_Module_Instance_StartStop"
var message=Jython["<b>" + workflowName + ": </b>"]
var msg

var ServerID=DCMQuery(/softwareinstance[@id=$SoftwareInstanceID]/@managedsystemid)
var SoftwareResourceTemplateID=DCMQuery(/softwareinstance[@id=$SoftwareInstanceID]/@resourcetemplateid)
var httpServerInstallDir=DCMQuery(/templateparam[@templateid=$SoftwareResourceTemplateID and
@name="install_dir"]/@value)
var httpInstanceName=DCMQuery(/templateparam[@templateid=$SoftwareResourceTemplateID and
@name="resource-name"]/@value)

try
  ##############################################
  ## 2: Set default operation to start
  ##############################################
  if Jython[ not operation ] then
     operation = "start"
  endif
  operation=Jython[operation.lower()]

  ##############################################
  ## 3: run the apache command
```

```
  ##############################################
  if Jython[simulation == “yes”] then
    log info Jython[“<b>simulation: </b> ..about to call scriplet to operate http server”]
    log info Jython[“               httpServerInstallDir: “ + httpServerInstallDir]
    log info Jython[“               httpInstanceName: “ + httpInstanceName]
    log info Jython[“               operation: “ + operation]
  else
    scriptlet(httpServerInstallDir, httpInstanceName, operation ) language = bash target =
DCMQuery(/server[@id=$ServerID]/@id) <<EOF
      binPath=‘cygpath -u “${httpServerInstallDir}/bin”‘
      # operate the service
      ${binPath}/apache -n “${httpInstanceName}” -k ${operation}
    EOF
  endif

catchall exception

  ##############################################
  ## 5: Handle erors
  ##############################################
  msg = Jython[message + exception]
  log error msg
  rethrow
endtry
```

10.10 Utility workflows

The workflows presented in the following are generic workflows that provide general functionality supporting the provisioning of the Trade application.

10.10.1 ITSO_Find_Supporting_Installation_for_Module_Requirements

This workflow finds a software installation in the DCM that honors a specific set of requirements for another software installation.

For example, in order to catalog the Trade database at the server hosting the WebSphere Application Server, we need to identify the DB2 server hosting the database in order to pick up configuration parameters needed for the catalog process. To achieve this, we define a database requirement for the Trade_Database_Client_Module and use the ITSO_Find_Supporting_Installation_for_Module_Requirements workflow to identify the DB2 server instance hosting the trade database.

Archived

Note: The implementation of the ITSO_Find_Supporting_Installation_for_Module_Requirements workflow shown in Example 10-64 inspects the entire DCM. Future enhancements may include the ability to limit the search to the same ‘scope’ as the target system to be able to find supporting installations in, for example, the same Application Tier, Application, or Customer as the requesting installation.

Example 10-64 ITSO_Find_Supporting_Installation_for_Module_Reqirements

```
workflow ITSO_Find_Supporting_Installation_for_Module_Reqirements(in requirementType, in SoftwareID, in
ServerID, out supportingSoftwareInstallationID) LocaleInsensitive

  var message
  supportingSoftwareInstallationID = ""

  #  find the software installation that honors the requirement
  var requirementTypeID = DCMQuery(/requirementtype[@value=$requirementType\]/@id)
  if Jython(requirementTypeID == None or requirementTypeID == "") then
    message = Jython("requirementtype " + requirementType + " not known")
    throw InputValidateionException message
  endif

  var installationObjTypeID = DCMQuery(/dcmobjecttype[@name="SOFTWARE_INSTALLATION"\]/@id)
  var installableObjTypeID = DCMQuery(/dcmobjecttype[@name="SOFTWARE_PRODUCT"\]/@id)
  var moduleObjTypeID = DCMQuery(/dcmobjecttype[@name="SOFTWARE_MODULE"\]/@id)

  var objTypeID = DCMQuery(/dcmobject[@id=$SoftwareID\]/@objecttypeid)


  var SoftwareModuleID
  if Jython(objTypeID == installableObjTypeID) then
    SoftwareModuleID =
DCMQuery(/softwareinstallable[@id=$SoftwareID\]/softwareinstallationmechanism/softwaremodule/@id)
  else
    if Jython(objTypeID == installationObjTypeID) then
      SoftwareModuleID = DCMQuery(/softwareinstallation[@id=$SoftwareID\]/@softwaremoduleid)
    else
      if Jython(objTypeID == moduleObjTypeID) then
        SoftwareModuleID = SoftwareID
      endif
    endif
  endif

  var SoftwareModuleName = DCMQuery(/softwaremodule[@id=$SoftwareModuleID\]/@name)
  #
DCMQuery(/supportedrequirementtype[@value="APPLICATION"]/softwaremodule/softwareresource[@objecttypeid="89
"]/@name
  var msg
  array InstallationIDs
  if Jython(ServerID == None or ServerID == "") then
```

```
    msg = Jython("Searching for ANY installation (type:" + installationObjTypeID + "\) honoring the " +
requirementType + " requirement of " + SoftwareModuleName )
    InstallationIDs =
DCMQuery(/supportedrequirementtype[@value=$requirementType\]/softwaremodule/softwareresource[@objecttypeid
=$installationObjTypeID\]/@id)
  else
    msg = Jython("Searching for installation (type:" + installationObjTypeID + "\) honoring the " +
requirementType + " requirement of " + SoftwareModuleName + " on server " + ServerID )
    InstallationIDs =
DCMQuery(/supportedrequirementtype[@value=$requirementType]/softwaremodule/softwareresource[@managedsystem
id=$ServerID and @objecttypeid=$installationObjTypeID]/@id)
  endif
  var i = arraysize(InstallationIDs)
  log info Jython(msg + " among " + i + " installations")

  foreach instID in InstallationIDs do
    var supportingSoftwareModuleID = DCMQuery(/softwareinstallation[@id=$instID]/@softwaremoduleid)
    var supportingSoftwareModuleName = DCMQuery(/softwaremodule[@id=$supportingSoftwareModuleID]/@name)
#log info Jython[("..... checking installation of : " + supportingSoftwareModuleName + "(instID:" +
instID + ")"]
    foreach requirementID in
DCMQuery(/softwaremodule[@id=$SoftwareModuleID]/requirement[@requirementtypeid=$requirementTypeID]/@id) do
      var requirementName = DCMQuery(/requirement[@id=$requirementID]/requirementname/@value)
      var requirementNameID = DCMQuery(/requirement[@id=$requirementID]/requirementname/@id)
#log info Jython["Finding value for: " + supportingSoftwareModuleName + " " + requirementType + "(" +
requirementID + ") " + requirementName + "(" + requirementNameID + ")"]
      var requirementOK="0"
      foreach requirementValue in DCMQuery(/requirement[@id=$requirementID]/requirementvalue/@value) do
#log info Jython("  checking if module " + supportingSoftwareModuleName + " (" +
supportingSoftwareModuleID + "\) is honoring the value for " + requirementID + " (" + requirementName +"\)
of " + requirementValue)
        var supportedValue = DCMQuery(/capability[@nameid=$requirementNameID and
@softwaremoduleid=$supportingSoftwareModuleID]/@value)

        if Jython[ supportedValue] then
#log info Jython("Found capability value: " + requirementName + " of " + supportedValue)

          if Jython(supportedValue == requirementValue) then
            requirementOK="1"
            break
          endif
        endif
       done

        if Jython[requirementOK == "1" ] then
           supportingSoftwareInstallationID = instID
        else
           supportingSoftwareInstallationID = """
           break
       endif
    done
    if Jython(supportingSoftwareInstallationID != """) then
      break
    endif
```

```
  done



  if Jython(supportingSoftwareInstallationID != “”) then
    var supportingSoftwareInstallationName =
DCMQuery(/softwareinstallation[@id=$supportingSoftwareInstallationID]/@name)
    log info Jython[“Installation “ + supportingSoftwareInstallationName + “(“ +
supportingSoftwareInstallationID + “) supports the “ + requirementType + “ requirements of “ +
SoftwareModuleName]
  else
    log info Jython[“No installation found honoring the “ “ supports the “ + requirementType + “
requirements of “ +  SoftwareModuleName]
  endif
```

10.10.2 ITSO_Synchronize_Credentials

This workflow (Example 10-65) is used to ensure that the proper set of matching credentials have been defined on the host and client sides for the specific operation.

For example, when creating the Trade database, we have to authenticate with the database server as the instance user in order to access the database environment, and these credentials must be used for the file transfer and execute command operations required to run scriptlets on the target system. Since the database instance user is created during the provisioning process, these credentials cannot be created in advance.

Besides creating the host credentials for the target system, we also have to create the client credentials for the Deployment Engine by calling the ITSO_Add_Matching_DeploymentEngine_Client_Password_Credential workflow, which is described in 10.10.3, “ITSO_Add_Matching_DeploymentEngine_Client_Password_Credential” on page 499.

Example 10-65 ITSO_Synchronize_Credentials

```
workflow ITSO_Synchronize_Credentials(in CredentialsID, in ServerID, in DeviceID) LocaleInsensitive


  log info Jython(“<b> CredentialsID is: “ +  CredentialsID + “</b>”)
  var credential_userid
  var credential_password
  java:com.thinkdynamics.kanaha.de.javaplugin.sap.FindCredentialsPassword(CredentialsID, <null>, <null>,
credential_password, credential_userid)

  var credential_searchkey = DCMQuery(/credentials[@id=$CredentialsID\]/@searchkey)
  var credential_sap_id = DCMQuery(/credentials[@id=$CredentialsID\]/sap/@id)
```

```
  # does credentials exist for the serverID's default execute command SAP
  var default_execute_sap_id
  java:com.thinkdynamics.kanaha.de.javaplugin.datacentermodel.FindDefaultSAP(ServerID, "execute-command",
default_execute_sap_id)

  # does searchkey already exist in default exec sap ?
  var exec_searchkey_credential_id =
DCMQuery(/sap[@id=$default_execute_sap_id\]/credentials[@searchkey=$credential_searchkey\]/@id)
  if Jython( (credential_sap_id != default_execute_sap_id\) and (exec_searchkey_credential_id == None\))
then
    # copy the credentials to the default execute SAP id
    log info Jython("Updating sap " + default_execute_sap_id + " on server " + ServerID)
    log info Jython("  adding password credential: " + credential_searchkey)
    log info Jython("    user: " + credential_userid)
    log info Jython("    pw  : " + credential_password)

    DCMInsert parent = DCMQuery(/sap[@id=$default_execute_sap_id\]/@id) <<EOFXML
     <credentials search-key="$credential_searchkey" is-default="false">
       <password-credentials username="$credential_userid"
        password="$credential_password" is-encrypted="false" />
     </credentials>
EOFXML
    exec_searchkey_credential_id =
DCMQuery(/sap[@id=$default_execute_sap_id\]/credentials[@searchkey=$credential_searchkey\]/@id)

  else
    log info Jython("credential " + exec_searchkey_credential_id + " has already been created on server "
+ ServerID + " for searchkey " + credential_searchkey)
  endif


  ITSO_Add_Matching_DeploymentEngine_Client_Password_Credential(exec_searchkey_credential_id, DeviceID)


  # does searchkey already exist in default file transfer sap ?
  var default_filetransfer_sap_id
  java:com.thinkdynamics.kanaha.de.javaplugin.datacentermodel.FindDefaultSAP(ServerID, "file-transfer",
default_filetransfer_sap_id)
  var trans_searchkey_credential_id =
DCMQuery(/sap[@id=$default_filetransfer_sap_id\]/credentials[@searchkey=$credential_searchkey\]/@id)
  if Jython( (credential_sap_id != default_filetransfer_sap_id\) and (trans_searchkey_credential_id ==
None\)) then
    # copy the credentials to the default file transfer SAP id
    log info Jython("Updating sap " + default_filetransfer_sap_id)
    log info Jython("  adding password credential: " + credential_searchkey)
    log info Jython("    user: " + credential_userid)
    log info Jython("    pw  : " + credential_password)

    DCMInsert parent = DCMQuery(/sap[@id=$default_filetransfer_sap_id\]/@id) <<EOFXML
     <credentials search-key="$credential_searchkey" is-default="false">
       <password-credentials username="$credential_userid"
        password="$credential_password" is-encrypted="false" />
     </credentials>
```

```
EOFXML
    trans_searchkey_credential_id =
DCMQuery(/sap[@id=$default_filetransfer_sap_id\]/credentials[@searchkey=$credential_searchkey\]/@id)
  else
    log info Jython("credential " + trans_searchkey_credential_id + " has already been created on " +
ServerID + " for searchkey " + credential_searchkey)
  endif

  ITSO_Add_Matching_DeploymentEngine_Client_Password_Credential(trans_searchkey_credential_id, DeviceID)
```

10.10.3 ITSO_Add_Matching_DeploymentEngine_Client_Password_Credential

This workflow (Example 10-66) is used to create client credentials for the default operation SAPs on the Deployment Engine in order to define credentials with search keys matching the target system. The user ID and password to be used will be picked up from the default credentials.

Note: The current implementation only supports password credentials. Future enhancements may include support for RSA based credentials.

Example 10-66 ITSO_Add_Matching_DeploymentEngine_Client_Password_Credential

```
workflow ITSO_Add_Matching_DeploymentEngine_Client_Password_Credential(in CredentialID, in DeviceID)
LocaleInsensitive

  var message
  var exception

  ##  insert password credentials for tioserver for same searchkey

  var credential_searchkey = DCMQuery(/credentials[@id=$CredentialID\]/@searchkey)
  var sap_id = DCMQuery(/credentials[@id=$CredentialID\]/sap/@id)

  var appprotocol
  var Authentication
  var Context
  var Default_Credentials_ID
  var Domain
  var Managed_System_ID
  var Other_Description
  var Port
  var Protocol_Type_ID
  Get_SAP_Attribute(Authentication, Context, Default_Credentials_ID, Domain, Managed_System_ID,
Other_Description, Port, Protocol_Type_ID, sap_id)
  appprotocol = DCMQuery(/sap[@id=$sap_id\]/@appprotocolid)


  # find the client sap
```

```
  var de_client_sap_id = DCMQuery(/sap[@managedsystemid=$DeviceID and @host="N" and @port="0" and
@context=$Context and @domain=$Domain and @protocoltypeid=$Protocol_Type_ID and
@appprotocolid=$appprotocol\]/@id)

  if Jython(de_client_sap_id == None) then
    message = Jython("<b>" + workflowName + ": </b> Cannot find an client credentials on TIOServer for
protocol type " + Protocol_Type_ID)
    log error message
    exception = Jython(workflowName + ":no_client_sap_found")
    throw exception message
  endif

  # find credentials for the searchkey
  var de_client_credential_id =
DCMQuery(/sap[@id=$de_client_sap_id\]/credentials[@searchkey=$credential_searchkey\]/@id)
  if Jython(de_client_credential_id != None) then
    log info Jython("credential " + de_client_credential_id + " has already been created on server " +
DeviceID + " for searchkey " + credential_searchkey)
  else
    var password_cred_id
    var password_type_id = DCMQuery(/credentialstype[@name="password"\]/@id)
    foreach id in
DCMQuery(/sap[@managedsystemid=$DeviceID\]/credentials[@credentialstypeid=$password_type_id\]/@id) do
      password_cred_id = id
      break
    done
    #  get the default userid and password
    var de_client_userid
    var de_client_password
    java:com.thinkdynamics.kanaha.de.javaplugin.sap.FindCredentialsPassword(password_cred_id, <null>,
<null>, de_client_password, de_client_userid)

    DCMInsert parent = DCMQuery(/sap[@id=$de_client_sap_id\]/@id) <<EOFXML
        <credentials search-key="$credential_searchkey" is-default="false">
          <password-credentials username="$de_client_userid"
           password="$de_client_password" is-encrypted="false" />
     </credentials>
EOFXML

  endif
```

10.10.4 ITSO_HostingEnvironment_Host

This workflow will create software resources parented by another object than the one that is currently being deployed. Normally, this functionality will be supported by the workflow implementing the HostingEnvironment.Host LDO for the parenting object; however, in our situation, there is no guarantee that the customer has implemented this LDO for the underlying DB2 installations, so to play it safe, we have to provide it as a part of the Trade automation package.

Archived

The high-level functionality of this workflow is:

1. Initialize and validate input (get the simulation flag using the standard Trade_Get_Simulation workflow, and validate the provided parameters).
2. Find all the templates and for each INSTANCE template use the build-in Java plug-ins to:
   a. Clone the template.
   b. Create the new Instance, and commit it if successful.
   c. Associate the template with the instance.
   d. Create the DB2 instance using the Trade_DB2Server_Add_Instance workflow. Since we are not in control of the SoftwareInstallation.AddInstance workflow associated with the device driver for the custom implemented DB2 Server Installation, we cannot expect the call to this LDO will create the instance according to our parameters for type, owner, and port.
   e. Validate the result of the instance creation and, optionally, uncommit and remove the instance.
   f. Commit the instance.

The full content of the workflow is shown in Example 10-67.

Example 10-67 ITSO_HostingEnvironment_Host

```
workflow ITSO_HostingEnvironment_Host(in SoftwareResourceID, in SoftwareModuleID, in
SoftwareResourceTemplateID, out array objects) LocaleInsensitive


##   this workflow is intended to be called from a workflow that implements a Foreign Configuration SRTs
##   nested off an Instance or Installation SRT in order to apply the imbedded resources to another
Installation

##   Paramaters:
##   -----------
##   SoftwareResourceID   is the ID of the resource to host the new objects
##   SoftwareModuleID     is the ID of the active module (to be able to pass it to the hosting object)
##   SoftwareResourceTemplateID     parent SRT from which all nested resources will be added
##                                  must be of type FOREIGN-CONFIGURATION
#######################################
##  1:  initialize and input validation
#######################################

 var workflowName="ITSO_HostingEnvironment_Host"
 var message=Jython["<b>" + workflowName + ":</b>"]
 var msg

 # Validate SoftwareResourceID
 var softwareResourceID = DCMQuery(/softwareresource[@id=$SoftwareResourceID]/@id)
```

```
if Jython(softwareResourceID == None) then
  throw InvalidSoftwareResourceID “Resource can only be hosted on a specific resource.”
endif

# Validate SoftwareModuleID
var softwareModuleID = DCMQuery(/softwaremodule[@id=$SoftwareModuleID]/@id)
if Jython(softwareModuleID == None) then
  throw InvalidSoftwareModuleID “SoftwareModule not found.”
endif

# Validate SoftwareResourceTemplateID
var softwareResourceTemplateID = DCMQuery(/softwareresourcetemplate[@id=$SoftwareResourceTemplateID]/@id)
if Jython(softwareResourceTemplateID == None) then
  throw InvalidSoftwareResourceTemplateID “SoftwareResourceTemplate not found.”
endif


##########################################################
##  2: Get basic information to control and produce messages
##########################################################
var i
var targetResourceID
var targetModuleID=DCMQuery(/softwareresource[@id=$SoftwareResourceID]/@softwaremoduleid)
var targetServerID=DCMQuery(/softwareresource[@id=$SoftwareResourceID]/@managedsystemid)

var targetResourceName=DCMQuery(/templateparam[@templateid=$softwareResourceTemplateID and
@name=”targetResourceName”]/@value)
var targetResourceType=DCMQuery(/templateparam[@templateid=$softwareResourceTemplateID and
@name=”targetResourceType”]/@value)

if Jython[targetResourceName == None or targetResourceName.strip() == “““ or targetResourceType == None or
targetResourceType.strip() == “““ ] then
   targetResourceID = SoftwareResourceID
else

  ##############################################################
  ##  3: Find all the child resources of the Foreign Resource of targetResourceType
  ##############################################################
  # Determine the target software resource type ID
  var targetSoftwareResourceTypeID=DCMQuery(/softwareresourcetype[@name=$targetResourceType]/@id)

  array
resources=DCMQuery(/softwareresourcetemplate[@softwareresourcetype=$targetSoftwareResourceTypeID]/software
resource[@managedsystemid=$targetServerID and @name=$targetResourceName and
@softwaremoduleid=$targetModuleID]/@id)
  i=arraysize(resources)
  if Jython[ int(i) > 0 ] then
    # use the first one
    targetResourceID = resources[0]
  else
    msg = Jython[message + “no ‘” + targetResourceType + “‘ software resource named ‘” +
targetResourceName + “‘ was found on server “ + ServerID]
    log error msg
    throw objectNotFound msg
```

```
   endif
 endif

 var
targetInstallableID=DCMQuery(/softwareresource[@id=$targetResourceID]/softwaremodule/softwareinstallationm
echanism/@productid)


 ##############################################################
 ##  3: Find all child-tempalates of the FOREIGN-CONFIGURATION
 ##############################################################
 # Determine resulting software resource type
 var name
 var type
 # find all the child templates
 array templates

templates=DCMQuery(/childsoftwareresourcetemplate[@parenttemplateid=$SoftwareResourceTemplateID]/@id[order
By@id])
 i = arraysize(templates)
 if Jython[int(i) == 0] then
   msg = Jython[message + "Foreign-Configuration template " + SoftwareResoureTemplateID + " has no child
templates"]
   log warning msg
   #throw objectNotFound msg
 endif


       ##############################################
       ##  5: Clone the software resource template
       ##############################################

   var newSRTid
   var newSoftwareResourceID


   foreach templateID in templates do

   try
       # clone the template

newSRTid=Java[com.ibm.tivoli.orchestrator.datacentermodel.helper.SoftwareHelper#instantiateSoftwareResourc
eTemplate(targetModuleID,templateID)]
       if Jython[newSRTid != None] then
         log info Jython["Created new SoftwareResourceTemplate : " + newSRTid]
       endif

       # Create the new resource


java:com.ibm.tivoli.orchestrator.datacentermodel.helper.SoftwareHelper#createSoftwareResources(targetInsta
llableID, targetServerID, newSRTid, targetResourceID)
      newSoftwareResourceID =
Java[com.ibm.tivoli.orchestrator.datacentermodel.helper.SoftwareHelper#getSoftwareResource(newSRTid)]
```

Archived

```
      if Jython[newSoftwareResourceID != None] then
        log info Jython[“Created new SoftwareConfiguration object: “ + newSoftwareResourceID + “ as child
of “ + targetInstallableID + “ on server “ + targetServerID + “ using srt “ + newSRTid]
      endif

      ##############################################
      ##  7: Commit the new CONFIGURATION resource
      ##############################################
      # update the pending flag of the previously created resources

java:com.ibm.tivoli.orchestrator.datacentermodel.helper.SoftwareHelper#updateResourceFlag(newSoftwareResou
rceID, “false”)


      i=arraysize(objects)
      objects[i]=newSoftwareResourceID



   catchall

     ##############################################
     ##  9: Cleanup if we failed
     ##############################################
     if Jython[newSoftwareResourceID and newSoftwareResourceID != ““] then
       var id=DCMQuery(/softwareresource[@id=$newSoftwareResourceID]/@id)
       if Jython[ id ] then
         log error Jython[“Deleting new softwareresource: “ + newSoftwareResourceID]
         DCMDelete(/softwareresource[@id=$newSoftwareResourceID]/@id)
       endif
     endif
     if Jython[newSRTid and newSRTid != ““] then
       var id=DCMQuery(/softwareresourcetemplate[@id=$newSRTid]/@id)
       if Jython[ id ] then
         log error Jython[“Deleting new softwareresourcetemplate: “ + newSRTid]
         DCMDelete(/softwareresourcetemplate[@id=$newSRTid]/@id)
       endif
     endif

     msg=Jython[message + “ problems during creation of foreign objects”]
     log error msg
     rethrow
   endtry

  done
```

10.10.5 ITSO_Delete_Credentials

The workflow listed in Example 10-68 on page 505 is intended to be used to delete a specific credential, identified by the searchkey, from the default execute-command service access point of a server.

Important: During our tests, none of the three different ways available to delete a credential worked. Please contact Tivoli support for a resolution.

Example 10-68 ITSO_Delete_Credentials

```
workflow ITSO_Delete_Credentials (in ServerID, in searchkey) LocaleInsensitive

if Jython[not searchkey] then
  throw missingParameterExcepttion “No searchkey provided for deletion of password credentials”
endif

#  find the default SAP for execute-command for the server
var defaultSAPID
java:com.thinkdynamics.kanaha.de.javaplugin.datacentermodel.FindDefaultSAP(ServerID, “execute-command”,
defaultSAPID)

var CredentialsID
java:com.thinkdynamics.kanaha.de.javaplugin.sap.GetSapCredentials(CredentialsID, searchkey, defaultSAPID)

## remove password credential for the searchkey
#
#  -- none of the methods below seem to work !!!
#
## ServiceAccessPoint.RemovePwdCredential(defaultSAPID, CredentialsID)
## java:com.thinkdynamics.kanaha.de.javaplugin.sap.PasswordCredentialsRemove(CredentialsID, defaultSAPID)
## DCMDelete(/credentials[@id=$CredentialsID]/@id)
```

10.10.6 ITSO_Create_User

The ITSO_Create_User workflow (Example 10-69) creates a new user ID in a Windows environment, and makes the user a member of the groups parsed to the workflow in an array. In addition, the home directory is created in the Cygwin environment, and the passwd and groups files are updated.

Example 10-69 ITSO_Create_User

```
workflow ITSO_Create_User(in ServerID, in username, in encrypted password, in array groups)
LocaleInsensitive

  var DE_DeviceID
  java:com.thinkdynamics.kanaha.de.javaplugin.datacentermodel.FindDEDeviceId(DE_DeviceID)
  if Jython( ServerID == DE_DeviceID) then
    throw environmentException “This workflow does not work correctly on the Deployment Engine system”
  endif

  var rcode
  try

    scriptlet(username, password) language = bash target = DCMQuery(/server[@id=$ServerID\]/@id) <<EOFCMD
```

```
    TIOsetVar “rcode” “1”

    OSTYPE=‘uname‘
    echo $OSTYPE | grep -i cygwin 1>/dev/null 2>&1
    if [ $? -ne 0 ]; then
      TIOthrow “unSupportedEnvironment” “${OSTYPE} is not supported”
      return 0
    fi

    #is user defined to cygwin
    cat /etc/passwd | grep ${username} 1>/dev/null 2>&1
    rc=$?
    if [ $rc -eq 0 ]; then
       TIOlog “info” “User ${username} is already defined to cygwin”
#      TIOthrow “userExists” “User ${username} is already defined to cygwin”
#       TIOsetVar “rcode” “0”
#      return 0
    fi

    #is user defined to windows
    net user | grep ${username} 1>/dev/null 2>&1
    rc=$?
    if [ $rc -eq 0 ]; then
      TIOlog “info” “User ${username} is already defined to Windows”
    ##  TIOthrow “userExists” “User ${username} is already defined to Windows”
    ##  return 0
    else
      # create the user in windows
      command=”net user ${username} ${password} /add /active:yes /expires:never /passwordchg:no
/fullname:${username}”
      msg=‘$command‘
      rc=$?
      TIOlog “debug” “(${rc}) ${command} ${msg}”
    fi

    if [ $rc -eq 0 ]; then
      # synchronize windows user unformation with cygwin
      mkpasswd --local > /etc/passwd
      mkgroup --local > /etc/group

      # create the home directory

      homedir=”/home/${username}”
      if [ -e ${homedir} ]; then
       TIOlog “info” “Home directory ${homedir} already exists”
       rc=0
      else
        TIOlog “info” “Creating home directory ${homedir}”
        mkdir -p ${homedir}
        rc=$?
      fi
    fi
```

```
    if [ $rc -eq 0 ]; then
      TIOsetVar “rcode” “0”
      TIOlog “info” “user created successfully”
    else
      TIOlog “error” “${msg}”
      TIOthrow “userCreateError” “${msg}”
    fi


    return 0
EOFCMD


  catch userExists msg
    log info msg
    rcode = 0
  catchall msg
    rethrow
  finally
    noop
  endtry



  if Jython( int(rcode\) == 0 ) then

    foreach groupname in groups do
      rcode = “8”
      scriptlet(username, groupname) language = bash target = DCMQuery(/server[@id=$ServerID\]/@id)
<<EOFCMD
      net localgroup ${groupname} | grep ${username}
      rc=$?
      if [ $rc -ne 0 ]; then
        net localgroup ${groupname} ${username} /ADD
        rc=$?
        # synchronize windows user unformation with cygwin
        mkpasswd --local > /etc/passwd
        mkgroup --local > /etc/group
      fi
      TIOsetVar “rcode” “${rc}”
      return 0
    EOFCMD

    done
  endif
```

Archived

10.10.7 ITSO_Destroy_User

This workflow (Example 10-70) removes a user in a Windows environment and the related Cygwin home directory and synchronizes the Windows and Cygwin settings.

Example 10-70 ITSO_Destroy_User

```
workflow ITSO_Destroy_User(in ServerID, in username) LocaleInsensitive

  var DE_DeviceID
  java:com.thinkdynamics.kanaha.de.javaplugin.datacentermodel.FindDEDeviceId(DE_DeviceID)
  if Jython( ServerID == DE_DeviceID) then
    throw environmentException “This workflow does not work correctly on the Deployment Engine system”
  endif


  scriptlet(username) language = bash target = DCMQuery(/server[@id=$ServerID\]/@id) <<EOFCMD

    OSTYPE=‘uname‘
    echo $OSTYPE | grep -i cygwin 1>/dev/null 2>&1
    if [ $? -ne 0 ]; then
      TIOthrow “unSupportedEnvironment” “${OSTYPE} is not supported”
      return 0
    fi

#   #is user defined to cygwin
    cat /etc/passwd | grep ${username} 1>/dev/null 2>&1
    rc=$?
    if [ $rc -ne 0 ]; then
      TIOthrow “userNotDefined” “User ${username} is not defined to cygwin”
      return 0
    fi

    #is user defined to windows
    net user | grep ${username} 1>/dev/null 2>&1
    rc=$?
    if [ $rc -ne 0 ]; then
      TIOthrow “userNotDefined” “User ${username} is not defined to Windows”
      return 0
    fi


    # delete the user in windows
    command=”net user ${username} /DELETE”
    msg=‘$command‘
    rc=$?
    TIOlog “debug” “(${rc}) ${command} ${msg}”
    if [ $rc -eq 0 ]; then
      # synchronize windows user unformation with cygwin
      mkpasswd --local > /etc/passwd
      mkgroup --local > /etc/group
```

```
      # remove home directory
      homedir="/home/${username}"
      if [ -e ${homedir} ]; then
        rm -r ${homedir}
      fi
      TIOlog "info" "(${rc}) $command  ${msg}"
    else
      TIOlog "error" "(${rc}) $command  ${msg}"
      TIOthrow "userDestroyError" "${msg}"
    fi

    return 0

EOFCMD
```

10.10.8 ITSO_Copy_And_Unpack_Archive_From_Repository

This workflow (Example 10-71) copies a zipped archive from a file repository to the target system, and unpacks the archive in a location parsed to the workflow.

The copying is handled by the ITSO_Copy_File_From_Repository workflow, described in Example 10-72 on page 510.

Example 10-71 ITSO_Copy_And_Unpack_Archive_From_Repository workflow

```
# ------------------------------------------------------------------
# Licensed Materials - Property of IBM
# 5724-F75
# (C) Copyright IBM Corp.  2003 - 2005
# All Rights Reserved
# US Government Users Restricted Rights -Use, duplication or
# disclosure restricted by GSA ADP Schedule Contract with IBM Corp.
# ------------------------------------------------------------------
#
# Install the base software of a DB2 product on a Microsoft Windows platform.
# Please note this workflow will perform a SINGLE PARTITION, TYPICAL install.
# A default instance (named DB2) and a DAS will be created.
#
workflow ITSO_Copy_And_Unpack_Archive_From_Repository(in SoftwareInstallableID, in DeviceID, in UnpackDir)
LocaleInsensitive


  var use_FileRepository_methods = "yes"
  var wkfName = "Copy_And_Unpack_Archive_From_Repository"
  var mgsExecError = ""

  var DestinationPath
  var FileName
  var ReturnCode
  var ReturnErrorString
  var ReturnResult
```

```
  ITSO_Copy_File_From_Repository(SoftwareInstallableID, DeviceID, UnpackDir, DestinationPath, FileName)


  log info Jython("<b>" + " Unpacking " + FileName + " to " + UnpackDir + "</b>")

  var runcmd = Jython(' if [ ! -d "' + UnpackDir + '" \]; then mkdir -p "' + UnpackDir + '"; fi')
  Device.ExecuteCommand(DeviceID, runcmd, "/tmp", <null>, <null>, <null>, ReturnCode, ReturnErrorString,
ReturnResult)


  scriptlet(DestinationPath, UnpackDir, FileName) language = bash target =
DCMQuery(/server[@id=$DeviceID\]) credentialskey = "default" timeout = 300 <<EOFBASH
    TIOlog "debug" "Executing : unzip -o -d ${UnpackDir} ${DestinationPath}/${FileName}"
    rc=$?
    ScriptletExitMsg=`unzip -o -d ${UnpackDir} ${DestinationPath}/${FileName} 2>&1`
    test $rc != 0 && {
      TIOthrow ScriptletExit "RC:${rc} - ${ScriptletExitMsg}; Could not unpack ${FileName}"
      return 1
    }
    TIOlog "info" "Successfully unzipped ${DestinationPath}/${FileName} to ${UnpackDir}"
    if [ -e ${DestinationPath}/${FileName} ]; then
      `rm ${DestinationPath}/${FileName}`
    fi
    chmod -R 777 ${UnpackDir}
  EOFBASH
```

10.10.9 ITSO_Copy_File_From_Repository

This workflow (Example 10-72) copies a file from a file repository to the target server. The functionality offered by this workflow is similar to the FileRepository.GetFile LDO; however, in addition, this workflow ensures that the target directory for the copy operation exists prior to attempting the operation.

Example 10-72 The ITSO_Copy_File_From_Repository utility workflow

```
# ------------------------------------------------------------------
# Licensed Materials - Property of IBM
# 5724-F75
# (C) Copyright IBM Corp.  2003 - 2005
# All Rights Reserved
# US Government Users Restricted Rights -Use, duplication or
# disclosure restricted by GSA ADP Schedule Contract with IBM Corp.
# ------------------------------------------------------------------
#
# Install the base software of a DB2 product on a Microsoft Windows platform.
# Please note this workflow will perform a SINGLE PARTITION, TYPICAL install.
# A default instance (named DB2) and a DAS will be created.
#
```

```
workflow ITSO_Copy_File_From_Repository(in SoftwareInstallableID, in DeviceID, in UnpackDir, inout
DestinationPath, inout FileName) LocaleInsensitive


  var use_FileRepository_methods = “true”
  var wkfName = “Copy_File_From_Repository”
  var mgsExecError = ““
  var ReturnCode
  var ReturnResult
  var ReturnErrorString
  # Get & test file repository related data
  #
  # Ensure repository_name is the host name of the device to be mapped drive for the software executable
  var repository_name = DCMQuery(/softwareinstallable[@id=$SoftwareInstallableID\]/filerepository/@name)
  if Jython(not repository_name or repository_name.isspace(\)) then
    mgsExecError = Jython(wkfName + “: repository_name is null or empty; ensure there is a valid file
repository associated with SoftwarInstallableID”)
    throw parameterError mgsExecError
  endif
  log info Jython(“repository_name = “ + repository_name)

  var repository_ID = DCMQuery(/softwareinstallable[@id=$SoftwareInstallableID\]/filerepository/@id)
  if Jython(not repository_ID or repository_ID.isspace(\)) then
    mgsExecError = Jython(wkfName + “: repository_ID is null or empty; ensure there is a valid file
repository associated with SoftwarInstallableID”)
    throw parameterError mgsExecError
  endif
  log info Jython(“repository_ID = “ + repository_ID)



  var repository_ipaddress =
DCMQuery(/softwareinstallable[@id=$SoftwareInstallableID\]/filerepository/@ipaddress)
  if Jython(not repository_ipaddress or repository_ipaddress.isspace(\)) then
    mgsExecError = Jython(wkfName + “: repository_ipaddress is null or empty; ensure there is a valid
ipaddress associated with file repository”)
    throw parameterError mgsExecError
  endif
  log info Jython(“repository_ipaddress = “ + repository_ipaddress)
  repository_name = repository_ipaddress

  var repository_root =
DCMQuery(/softwareinstallable[@id=$SoftwareInstallableID\]/filerepository/@rootpath)
  var package_path = DCMQuery(/softwareinstallable[@id=$SoftwareInstallableID\]/file/@path)
  FileName = DCMQuery(/softwareinstallable[@id=$SoftwareInstallableID\]/file/@name)

  var SourcePath

  if Jython(not DestinationPath) then
    DestinationPath = Jython(“/tmp”)
  endif

  var runcmd = Jython(‘ if [ ! -d “‘ + DestinationPath + ‘” \]; then mkdir -p “‘ + DestinationPath + ‘”;
fi’)
```

```
  Device.ExecuteCommand(DeviceID, runcmd, “/tmp”, <null>, <null>, <null>, ReturnCode, ReturnErrorString,
ReturnResult)


  if Jython(UnpackDir) then
    UnpackDir = Java[com.thinkdynamics.kanaha.util.PathHelper#convertToCygwinPath(UnpackDir)]
  endif


  if Jython(use_FileRepository_methods.lower(\) == “true”) then
    SourcePath = package_path
    SourcePath = Java[com.thinkdynamics.kanaha.util.PathHelper#convertToCygwinPath(SourcePath)]
    log info Jython(“<b>” + “Copying “ + SourcePath + “/” + FileName + “ to “ + DestinationPath + “/” +
FileName + “</b>”)
    FileRepository.GetFile(repository_ID, SourcePath, FileName, DeviceID, DestinationPath, FileName,
“900”)
  else
    SourcePath = Jython(repository_root + package_path)
    SourcePath = Java[com.thinkdynamics.kanaha.util.PathHelper#convertToCygwinPath(SourcePath)]
    log info Jython(“<b>” + “Copying “ + SourcePath + “:” + FileName + “ to “ + DestinationPath + “:” +
FileName + “</b>”)
    Device.CopyFile(repository_ID, SourcePath, FileName, DeviceID, DestinationPath, FileName, <null>,
<null>, “900”)

endif
```

10.11 Documenting your automation package

Naturally, you need to document the content, implementation activities, and customization points of your package. Traditionally, this is the weak point of any automation package, and greatly impacts the consumability of your work, and possibly impacting your business because your competitors do a better job of helping the implementors.

When creating the automation package in the APDE environment, a default documentation html file is created in the doc directory. We recommend that your follow this template, and document all your workflows, their functionality, required parameters, and so on.

As usual, investing time in providing easy-to-understand, accurate information will benefit you in the long run, avoiding critical situations and unhappy customers. In addition, it will help troubleshooting and debugging.

Archived

10.12 Building the Trade automation package

U se the APDE platform to build your automation package so that it can be easily distributed and installed on customer systems. By default, all the device drivers and workflows that have been created will be automatically defined to the automation package meta file /TC-INF/tc-driver.xml.

In case you need additional files, such as in our case where we want to include both the index.html file for to act as a front end for the Trade application, and the installation archive for the Trade application itself, you need to manually add these to the tc-driver.xml file.

Follow the instructions provided in 8.5, “Generating the automation package using APDE” on page 256 to generate the automation package.

Example 10-73 shows the tc-driver.xml file for the Trade automation package.

Example 10-73 The Trade tc-driver.xml file

```
<?xml version=”1.0” encoding=”UTF-8”?>

<!DOCTYPE tc-driver PUBLIC “-//IBM//DTD TC Driver 2.0//EN”
“http://www.ibm.com/tivoli/orchestrator/dtd/tcdriver.dtd”>

<tc-driver>

  <tc-driver-format>2.0</tc-driver-format>

  <driver-name>Trade</driver-name>

  <version>3.1</version>

  <description />

  <documentation location=”doc/readme.html” />

  <dependencies>

   <dependency name=”ITSO_Utilities” />

   <dependency name=”core” />

  </dependencies>

  <property name=”tc.pkg”
location=”com.thinkdynamics.kanaha.tcdrivermanager.action” />

  <actions>
```

```
  <action name="copy-file" class="${tc.pkg}.CopyFileActions" />
  <action name="java-plugin" class="${tc.pkg}.JavaPluginActions" />
  <action name="workflow" class="${tc.pkg}.TxtWorkflowAction" />
  <action name="import" class="${tc.pkg}.ImportAction" />
</actions>
<items>
  <item name="repository/trade3Install.zip" action="copy-file">
    <param name="editable" value="false" />
    <param name="dest.path" value="${repository}/trade3/trade3install.zip" />
    <param name="chmod" value="777" />
  </item>
  <item name="repository/index.html" action="copy-file">
    <param name="editable" value="false" />
    <param name="dest.path" value="${repository}/trade3/index.html" />
    <param name="chmod" value="777" />
  </item>
  <item name="workflow/Trade_Get_Simulation.wkf" action="workflow">
    <param name="editable" value="false" />
  </item>
  <item name="workflow/Trade_Get_Hosting_SRTParameter_Or_Default.wkf"
action="workflow">
    <param name="editable" value="false" />
  </item>
  <item name="workflow/Trade_Web_Instance_StartStop.wkf"
action="workflow">
    <param name="editable" value="false" />
```

```
    </item>

    <item
name="workflow/Trade_Web_Module_Instance_Driver/Trade_Web_Module_Inst
ance_Start.wkf" action="workflow">

      <param name="editable" value="false" />

    </item>

    <item name="workflow/Trade_DBServer_Create_Database.wkf"
action="workflow">

      <param name="editable" value="false" />

    </item>

    <item name="workflow/Trade_Web_Add_Configuration.wkf"
action="workflow">

      <param name="editable" value="false" />

    </item>

    <item name="workflow/Trade_DBServer_Add_Instance.wkf"
action="workflow">

      <param name="editable" value="false" />

    </item>

    <item name="workflow/Trade_DBClient_Add_Configuration.wkf"
action="workflow">

      <param name="editable" value="false" />

    </item>

    <item name="workflow/Trade_Extract_TableDDL.wkf" action="workflow">

      <param name="editable" value="false" />

    </item>

    <item name="workflow/Trade_Application_Add_Instance.wkf"
action="workflow">

      <param name="editable" value="false" />
```

Archived

```
  </item>

  <item name="workflow/Trade_Create_SAP.wkf" action="workflow">

    <param name="editable" value="false" />

  </item>

  <item name="workflow/Trade_Web_Add_Instance.wkf" action="workflow">

    <param name="editable" value="false" />

  </item>

  <item name="workflow/Trade_DBServer_Delete_Database.wkf"
action="workflow">

    <param name="editable" value="false" />

  </item>

  <item
name="workflow/Trade_Application_Module_Installable_Driver/Trade_Applicatio
n_Module_Installable_Distribute.wkf" action="workflow">

    <param name="editable" value="false" />

  </item>

  <item
name="workflow/Trade_Application_Module_Installable_Driver/Trade_Applicatio
n_Module_Installable_Install.wkf" action="workflow">

    <param name="editable" value="false" />

  </item>

  <item
name="workflow/Trade_Application_Module_Installation_Driver/Trade_Applicatio
n_Module_Installation_Uninstall.wkf" action="workflow">

    <param name="editable" value="false" />

  </item>

  <item
name="workflow/Trade_Application_Module_Instance_Driver/Trade_Application
_Module_Instance_RemoveInstance.wkf" action="workflow">
```


Archived

```
    <param name="editable" value="false" />

  </item>

  <item
name="workflow/Trade_Application_Module_Instance_Driver/Trade_Application
_Module_Instance_Start.wkf" action="workflow">

    <param name="editable" value="false" />

  </item>

  <item
name="workflow/Trade_Application_Module_Instance_Driver/Trade_Application
_Module_Instance_Stop.wkf" action="workflow">

    <param name="editable" value="false" />

  </item>

  <item
name="workflow/Trade_DBClient_Module_Installable_Driver/Trade_DBClient_M
odule_Installable_Install.wkf" action="workflow">

    <param name="editable" value="false" />

  </item>

  <item
name="workflow/Trade_DBClient_Module_Installation_Driver/Trade_DBClient_
Module_Installation_Uninstall.wkf" action="workflow">

    <param name="editable" value="false" />

  </item>

  <item
name="workflow/Trade_DBServer_Module_Installable_Driver/Trade_DBServer_
Module_Installable_Distribute.wkf" action="workflow">

    <param name="editable" value="false" />

  </item>

  <item
name="workflow/Trade_DBServer_Module_Installable_Driver/Trade_DBServer_
Module_Installable_Install.wkf" action="workflow">
```

```
    <param name="editable" value="false" />

  </item>

  <item
name="workflow/Trade_DBServer_Module_Installation_Driver/Trade_DBServer_
Module_Installation_Uninstall.wkf" action="workflow">

    <param name="editable" value="false" />

  </item>

  <item
name="workflow/Trade_DBServer_Module_Instance_Driver/Trade_DBServer_M
odule_Instance_RemoveInstance.wkf" action="workflow">

    <param name="editable" value="false" />

  </item>

  <item
name="workflow/Trade_DBServer_Module_Instance_Driver/Trade_DBServer_M
odule_Instance_Start.wkf" action="workflow">

    <param name="editable" value="false" />

  </item>

  <item
name="workflow/Trade_DBServer_Module_Instance_Driver/Trade_DBServer_M
odule_Instance_Stop.wkf" action="workflow">

    <param name="editable" value="false" />

  </item>

  <item
name="workflow/Trade_DBServer_Module_Instance_Driver/Trade_DBServer_M
odule_Resource_UpdateDCM.wkf" action="workflow">

    <param name="editable" value="false" />

  </item>

  <item
name="workflow/Trade_Web_Module_Installable_Driver/Trade_Web_Module_In
stallable_Install.wkf" action="workflow">
```

```
      <param name="editable" value="false" />

    </item>

    <item
name="workflow/Trade_Web_Module_Installation_Driver/Trade_Web_Module_I
nstallation_Uninstall.wkf" action="workflow">

      <param name="editable" value="false" />

    </item>

    <item
name="workflow/Trade_Web_Module_Instance_Driver/Trade_Web_Module_Inst
ance_RemoveInstance.wkf" action="workflow">

      <param name="editable" value="false" />

    </item>

    <item
name="workflow/Trade_Web_Module_Instance_Driver/Trade_Web_Module_Inst
ance_Stop.wkf" action="workflow">

      <param name="editable" value="false" />

    </item>

    <item name="workflow/Trade_Customize.wkf" action="workflow">

      <param name="editable" value="false" />

    </item>

    <item name="workflow/Trade_Post_Install.wkf" action="workflow">

      <param name="editable" value="false" />

    </item>

  </items>

  <device-models>

    <device-model name="Trade_Application_Module_Installable_Driver"
category="Trade">

      <workflow name="Trade_Application_Module_Installable_Distribute" />
```

```
    <workflow name="Trade_Application_Module_Installable_Install" />

  </device-model>

  <device-model name="Trade_Application_Module_Installation_Driver"
category="Trade">

    <workflow name="Trade_Application_Module_Installation_Uninstall" />

  </device-model>

  <device-model name="Trade_Application_Module_Instance_Driver"
category="Trade">

    <workflow name="Trade_Application_Module_Instance_RemoveInstance" />

    <workflow name="Trade_Application_Module_Instance_Start" />

    <workflow name="Trade_Application_Module_Instance_Stop" />

  </device-model>

  <device-model name="Trade_DBClient_Module_Installable_Driver"
category="Trade">

    <workflow name="Trade_DBClient_Module_Installable_Install" />

  </device-model>

  <device-model name="Trade_DBClient_Module_Installation_Driver"
category="Trade">

    <workflow name="Trade_DBClient_Module_Installation_Uninstall" />

  </device-model>

  <device-model name="Trade_DBServer_Module_Installable_Driver"
category="Trade">

    <workflow name="Trade_DBServer_Module_Installable_Distribute" />

    <workflow name="Trade_DBServer_Module_Installable_Install" />

  </device-model>

  <device-model name="Trade_DBServer_Module_Installation_Driver"
category="Trade">
```

```
    <workflow name="Trade_DBServer_Module_Installation_Uninstall" />

  </device-model>

  <device-model name="Trade_DBServer_Module_Instance_Driver"
category="Trade">

    <workflow name="Trade_DBServer_Module_Instance_RemoveInstance" />

    <workflow name="Trade_DBServer_Module_Instance_Start" />

    <workflow name="Trade_DBServer_Module_Instance_Stop" />

    <workflow name="Trade_DBServer_Module_Instance_UpdateDCM" />

  </device-model>

  <device-model name="Trade_Web_Module_Installable_Driver"
category="Trade">

    <workflow name="Trade_Web_Module_Installable_Install" />

  </device-model>

  <device-model name="Trade_Web_Module_Installation_Driver"
category="Trade">

    <workflow name="Trade_Web_Module_Installation_Uninstall" />

  </device-model>

  <device-model name="Trade_Web_Module_Instance_Driver"
category="Trade">

    <workflow name="Trade_Web_Module_Instance_RemoveInstance" />

    <workflow name="Trade_Web_Module_Instance_Start" />

    <workflow name="Trade_Web_Module_Instance_Stop" />

  </device-model>

 </device-models>

 <dcm>

  <item name="xml/Trade_DeviceDriver_Category.xml" action="import" />
```

Archived

```
    <item name="xml/Trade_DBServer_Module.xml" action="import" />
    <item name="xml/Trade_DBClient_Module.xml" action="import" />
    <item name="xml/Trade_Application_Module.xml" action="import" />
    <item name="xml/Trade_Web_Module.xml" action="import" />
  </dcm>
  <post-install-workflow name="Trade_Post_Install"/>
</tc-driver>
```

Archived

A

Trade3 original installation instructions

This appendix provides the official installation instructions for the Trade3 application, as they appear in the ReadMe.html file that is included in the Trade3 installation archive trade3.0_WAS5.0Install.zip, which can be downloaded from http://www-306.ibm.com/software/webservers/appserv/benchmark3.html.

Please note that compared to the original material, the information in the following has been slightly reformatted to fit the current publishing media.

© Copyright IBM Corp. 2004, 2006. All rights reserved.

Trade3 installation instructions

Product	WebSphere Application Server Advanced Edition - Single Server
Release	Aquila_ASV
Build	WebSphere Application Server 5.0

Note: All text in *italic* must be customized to match your system.

Step 1 - Prepare for installation

1. Download and unzip the Trade3 install package.
2. Open a shell/command window to perform the install and change to the t3install directory.
3. Run the WebSphere Application Server setupCmdLine script to set up your shell environment:
   - UNIX: **. <WAS_HOME>/bin/setupCmdLine.sh**
   - Win32: **<WAS_HOME>\bin\setupCmdLine.bat**
4. Start the WebSphere 5.0 server
   - UNIX: **. /home/db2inst1/sqllib/db2profile**

     ${WAS_HOME}/bin/startServer server1
   - Win32: **%WAS_HOME%\bin\startServer server1**

Step 2 - SOAP enablement

Running Trade3 in the SOAP run time mode requires running the WebSphere SOAPEarEnabler on the Trade3 EAR.

1. Run the SoapEarEnabler script to “SOAPify” your EAR:
   - UNIX: **${WAS_HOME}/bin/SoapEarEnabler.sh trade3.ear 1 Soapdd.xml n 1 trade3EJB.jar n /soap**
   - Win32: **%WAS_HOME%\bin\SoapEarEnabler.bat trade3.ear 1 Soapdd.xml n 1 trade3EJB.jar n /soap**

Archived

Step 3 - DB2

Create the DB2 database for Trade3 and modify DB2 for read stability.

1. Verify that you have run **<*sqllib*>/java12/usejdbc2.[bat | sh]** to configure DB2 to use the JDBC 2.0 driver.
2. Set up a DB2 shell using the following method for UNIX or Win32:
   - UNIX: **su - db2username**
   - Win32: **db2cmd**
3. Execute the following DB2 commands from the Trade3 install directory using the DB2 shell just created:

```
db2 create db trade3db
db2 connect to trade3db
db2 -tvf DB2/Table.ddl
db2 disconnect all
db2set DB2_RR_TO_RS=yes
db2 update db config for trade3db using logfilsiz 1000
db2 update db cfg for trade3db using maxappls 100
db2stop force
db2start
```

4. Execute the following commands to bind DB2 packages for the trade3db.

```
db2 connect to trade3db
cd <sqllib>/bnd
db2 bind @db2cli.lst blocking all grant public
```

Note: Failure to rebind will result in the following run time error:

```
COM.ibm.db2.jdbc.DB2Exception: [IBM][CLI Driver][DB2/NT] SQL0805N
Package "NULLID.SQLLC300" was not found. SQLSTATE=51002
```

Linux Note: Running DB2 on Linux requires the trade3db database to be cataloged to use the TCP/IP communication protocol. Please refer to the WebSphere installation guide for details.

Step 3 - Oracle

As a user with the right Oracle environment variables setup:

1. Set up a database or reuse an existing database.
2. This example assumes a database SID of trade3db.

3. Create the trade user: user name of trade, password of trade.
4. Load the schema for the trade user:

```
sqlplus trade/trade@trade3db @Oracle/Table.ddl
```

Step 3 - Install, configure, and start Trade3 application and resources

1. Install Trade3 JDBC/JMS resources interactively:
   - UNIX: **$WAS_HOME/bin/ws_ant -f Trade3.xml installResources**
   - Win32: **%WAS_HOME%\bin\ws_ant -f Trade3.xml installResources**
2. Install Trade3 Application
   - UNIX: **$WAS_HOME/bin/ws_ant -f Trade3.xml installApp**
   - Win32: **%WAS_HOME%\bin\ws_ant -f Trade3.xml installApp**
3. Restart WebSphere to pick up the newly created resources and Trade3 application:
   - UNIX: **$WAS_HOME/bin/stopServer server1 $WAS_HOME/bin/startServer server1**
   - Win32: **%WAS_HOME%\bin\stopServer server1 %WAS_HOME%\bin\startServer server1**

> **Note:** To later uninstall trade3, you can use the command:
>
> `$WAS_HOME/bin/ws_ant -f Trade3.xml uninstall`

Step 4 - Populate the Trade3 database and go trade

1. Direct your browse to `http://localhost:9080/trade`.
2. Click **Configuration** and then click **(Re-)populate Trade database**.

> **Important:** Run the following db2 command to update the DB2 statistics:
>
> ```
> db2 connect to trade3db
> db2 reorgchk update statistics
> ```
>
> This must be done after database population for performance research.

3. Go trade.

B

The TIO/TPM V3 software model

The following provides a quick introduction to the software model introduced in Tivoli Provisioning Manager V3.1.

Archived

© Copyright IBM Corp. 2004, 2006. All rights reserved.

Software model

To manage software and automate software distribution, TIO/TPM V3 stores several types of information about each software item. Figure B-1 shows the high-level structure of a piece of software.

Figure B-1 Software definition overview

The software definition contains information about the software. When the software is installed, software resources describe the components of the software that are installed on the computer system.

Software definition

A software definition represents a piece of software and the information required to install and configure it. The software definition describes the following information:

The software to install

An installable file is the installation package or image file that is installed on a target system. Examples include a Windows .exe file, a .zip archive, or a Ghost operating system image.

Dependencies

Software prerequisites are stored as requirements. Software definitions can also contain capabilities. Capabilities are attributes that the software makes

available when it is installed. They can fulfill the requirements configured in other software definitions.

Configuration parameters
Configuration parameters are stored in a software configuration template. The software configuration template is similar to a response file for a silent installation of software. It identifies the parameters that are required by workflows for the software.

Software resources

After installation, Tivoli Provisioning Manager represents the configuration of the software on a target server with software resources. Each software resource is created by a software configuration template in the software definition. A software configuration template identifies the parameters and default values for a software resource, and the created software resource describes the actual configuration on a target system.

Software definitions

A software definition acts as a container that brings together all the information about a piece of software.

You cannot install a piece of software without a software definition. When you install a piece of software, the item that you select for installation is a software definition.

A software definition includes the following information:

- Links to the installable file that the software definition describes.
- Requirements for installation of the software.
- Capabilities of the software that can meet the requirements configured in other software definitions.
- Software configuration templates that define configuration options during installation of the software.

There are several types of software definitions that you can create:

Software products
The most general type of software definition. You can define any piece of software that you want to manage as a software product.

Operating system
A software definition for operating systems. This type of software definition is the same as a software product except that the operating system (OS) capability is automatically added to the software definition when you

create it. You can use this type of software definition for operating system software packages or operating system images that you want to use for the base software installation on new systems.

Software patches A software definition for a software maintenance package that updates a software product after the software product is installed. When you add a piece of software as a software patch, you can include additional information about the software in its software definition, such as the patch severity and issue date.

Software stacks A software definition for a software stack. A software stack is a list of software in a specific installation order. A software stack can include software products, software patches, and other software stacks.

Distributed application
A software definition for a composite application. A composite application includes components that are installed on different managed systems. For example, you can use this type of software definition for a Solution Installation package that contains a composite application. For more information about importing Solution Installation packages, refer to the Tivoli Provisioning Manager information center.

Software definitions can also include additional information, such as approval status, vendor name, and category. If you use a wizard to import software into the data center model, Tivoli Provisioning Manager automatically adds a software definition and installable file with the information you provide.

Installable files

An installable file is the actual software package or image file that is distributed and installed on a target system. There are three types of installable files:

- General software packages that you want to install with Tivoli Provisioning Manager. Examples include:
  - Software packages, such as Red Hat Package Manager (.rpm) files, AIX packages for use with installp (.lpp), and Microsoft Installer (.msi) files.
  - A self-extracting file, such as a .exe file.
  - An archived file in a format such as .zip or .tar.
- Vendor-specific software packages that are distributed by a software distribution application, such as Tivoli Configuration Manager.

- System images that are installed by boot servers.

When you define a installable file, you must specify the file name and file repository location. You can also specify requirements that are specific to this installable file, such as the required operating system or locale.

Associating installable files with software definitions

Each software definition can be associated with one or more installable files, but only one installable file is selected for installation. When there are multiple installable files, Tivoli Provisioning Manager identifies the appropriate one to use during installation based on:

Requirements Tivoli Provisioning Manager compares the requirements of the installable files with the hardware and software configured on the target system.

Priority The order of installable files in a software definition determines the priority. During installation, Tivoli Provisioning Manager starts by checking the configuration of a target system against the requirements of the first installable file.

- If the target system meets the requirements of the software definition and this installable file, the installable file is selected for installation. Other installable files are not evaluated.
- If the target system does not meet the requirements, it checks for the requirements of the next installable file.

Archived

Figure B-2 shows a software definition with three installable files.

Figure B-2 Software definition with installable files

The software definition is for a software product called Sample Product. It has an installation requirement for Windows 2000 on the target server. Three software packages are available to install the software on a target server:

package.zip This software package has the highest priority, since it is the first one on the list. It requires WinZip on the target server to install the software.

package.msi This software package is next in the priority order. It requires Windows Installer on the target server to install the software.

package.exe This software package is last in the priority order. It does not have any defined requirements.

In this example, the target server has both Windows 2000 and WinZip installed. When Tivoli Provisioning Manager checks the target server for requirements, it identifies a match for the Windows 2000 requirement of the software definition and the WinZip requirement for the package.zip installable file. Tivoli Provisioning Manager therefore selects package.zip as the installation mechanism for the target server.

If the target server had Windows 2000 but did not have WinZip, Tivoli Provisioning Manager would proceed to check the requirements of each item in the list of installable files until an appropriate match is found.

When to use multiple installable files

Situations where you might want to include multiple installable files in a software definition include:

- A software definition for a software stack that has multiple associated image files. Each image has the same software, but has different hardware dependencies. For each installable image, the requirements list is configured with the appropriate hardware dependencies. During installation, Tivoli Provisioning Manager can select the correct image by matching the image requirements with the hardware configuration of the target system.
- A software definition for a patch that has software packages for different combinations of target system configurations. For example, the software definition can contain software packages for each combination of locale and operating system. For each installable package, the prerequisites list is configured with the appropriate combination of locale and operating system. During installation, Tivoli Provisioning Manager can select the correct installable file by matching the patch requirements with the configuration of the target system.
- A software definition that is associated with different package formats for the same software. For example, a software definition might include a self-extracting .exe file, a archived .zip file, and a Tivoli Configuration Manager software package. During installation Tivoli Provisioning Manager can select the correct package format based on the configuration of the target system.

Software dependencies

Tivoli Provisioning Manager uses information about software dependencies to identify installation prerequisites for software. These dependencies are defined as requirements and capabilities.

Requirements Hardware or software that must be available on a target system to install or run a piece of software.

Capability A feature of a piece of software that can meet the requirements of another piece of software.

Software requirements in one software definition are met by software capabilities associated with a target system.

For example, a software definition for DB2 Universal Database Enterprise Server Version 8.1 for Linux can include the following requirements and capabilities:

- Requirements

  DB2 Universal Database supports Linux Red Hat Advanced Server, Version 3.

  To identify this installation prerequisite, the DB2 Universal Database software definition includes two requirements:

  - Operating system family: Linux
  - Operating system version: 3

- Capabilities

  Since many software applications have DB2 Universal Database as a prerequisite, the software definition also includes capabilities to identify the key attributes that Tivoli Provisioning Manager can use for dependency checking:

  - Software name: DB2
  - Software version: 8.1

When an administrator installs DB2 Universal Database, these capabilities are associated with the target system, and Tivoli Provisioning Manager can check for them when an administrator tries to install a piece of software that requires DB2 Universal Database in the future.

Hardware requirements are for resources such as disk space or memory. For example, a software definition can include a requirement for an Intel® processor. When an administrator initiates installation of the software, Tivoli Provisioning Manager checks the target server for an Intel processor hardware resource. Some hardware resource information can be automatically added to the data center model using discovery. For more information about discovery, refer to the Tivoli Provisioning Manager information center.

Requirements

Requirements describe prerequisites for a piece of software.

When you create a requirement, there are two options that you can specify for validation:

- You can indicate that the requirement is a host requirement. When you enable this option, the requirement should be met by a hosting application, or an application that acts as a containing environment for the software. This type of requirement is more common for multi-tier applications and Java programs. For example, a servlet is a Java program that runs on a Web server and extends the server functionality by generating dynamic content in response to

Web client requests. It requires a stand-alone servlet engine or an application server with servlet support to load the servlet. The servlet engine therefore acts as a container for running the software. If the servlet requires a specific container environment, such as WebSphere Application Server, the software definition for the servlet can include a host requirement for it.

- You can enable the Accept Non Existing Capability option. When this setting is enabled, Tivoli Provisioning Manager rejects a capability with an incompatible value, but accepts the absence of the capability.
  For example, if this setting is enabled for multimedia program that requires a Creative Labs Sound Blaster sound card, the following behavior will occur:
  - A target system with a different type of sound card will be rejected.
  - A target system without a sound card will be accepted.

You can define requirements at the installable file level or at the software definition level. Both types of requirements are useful when you are managing different versions of a software package for the same software product or patch.

Examples include:

- Requirements for different target environments

  You can create a software definition for a software patch that is available for different operating systems and locales. You define the operating system and locale requirements in each installable file, and then add all the installable files to the single software definition for the patch. When an administrator installs the patch, Tivoli Provisioning Manager compares the requirements in each installable with the configuration of the target system to identify the correct patch file to install.

- Requirements for different installation mechanisms

  You can create a software definition that contains the same software in different software package formats.

  Each package contains the same software. For all software packages, Windows 2000 is required to run the software, so this requirement is defined at the software definition level. Two of the software packages have their own requirements that are specific to the software package installation mechanism. package.zip requires WinZip on the target server and package.msi requires Windows Installer on the target server (see Figure B-3 on page 536).

Figure B-3 Software definition with multiple installation mechanisms

If you include multiple installable files in a software definition, but you do not specify requirements at the installable file level, Tivoli Provisioning Manager chooses the first installable file in the list.

In some cases, you might want to use other criteria to determine the installation priority of an installable file, such as the distance of a target from the file repository where the software package or image is stored. This type of alternative decision-making is configured and managed by the workflow developer who creates the automation package for the software.

Tivoli Provisioning Manager includes predefined requirement types. Additional types are typically defined by a workflow developer during automation package development. If the software has requirements that are not predefined, the workflow developer can add them to the XML data that it imports into the data center model.

Capabilities

Capabilities identify attributes of a piece of software that can be used for prerequisite validation. Capabilities are only configured in a software definition, since all associated installable files should share the same software attributes. For example, a software definition for DB2 Universal Database Version 8.1 can contain multiple installable files in different package formats. However, if software contained in each package shares the following capabilities:

- Software name: DB2
- Software version: 8.1

When any of the software packages is used to install DB2 Universal Database Version 8.1 on a target system, these two capabilities are become associated with the target system.

Tivoli Provisioning Manager checks the capabilities of the system whenever it is the target of a software deployment and requirements are specified. The following capability types are predefined:

OS For operating system requirements.

HARDWARE For hardware requirements, such as disk size or available memory. JRE For Java Runtime Engine version requirements.

SERVLET_ENGINE For servlet engine requirements. A servlet engine is a Web server application that helps Java programs (servlets) to process Web browser requests.

DATABASE For database software requirements.

EJB_CONTAINER For Enterprise JavaBean (EJB) container requirements. An EJB container is a run time environment for managing and deploying EJBs.

Additional capability types are typically defined by a workflow developer during automation package development. If the software has requirements that are not predefined, the workflow developer can add them to the XML data that it imports into the data center model.

Software configuration templates

A software configuration template identifies software resources and associated configuration details that you want to represent in the Tivoli Provisioning Manager data center model after the software is installed on a system.

Each software configuration template is used to create a software resource on the target system.

Archived

In Figure B-4, there are three software configuration templates. During the installation process, workflows run for each template to create software resources on the target system.

- The parent template is a software installation template. It creates a software installation resource.
- Each instance template creates a corresponding software instance on the target system.

Figure B-4 Software resources derived from the software definition

You can create one or more software configuration templates in a software definition. For example, an installation of an operating system can include different components and configuration options, depending on the requirements of the managed system and the standards in your organization. If you create a software definition for Windows 2000 workstations, you could include software configuration templates for user workstations and administrator workstations.

Software configuration templates are typically designed by workflow developers. They are included in the software definition when the automation package for a piece of software is created. The software configuration template can include both predefined configuration parameters and user-defined parameters. When administrators install the software, they can make changes to the values of user-defined parameters.

Software configuration template types

The parent template of a software configuration template is typically a software installation.

Child templates then define settings for other components of the software.

You can define the following types of software resources in a software configuration template:

Software installation The parent template for a piece of software. It represents the software definition and is used to create a software installation resource on a target system. This template typically includes installation or run time parameters. Software stacks and composite applications contain multiple software definitions. A software installation template is associated with each software definition in the software stack or composite application.

Configuration Configuration settings for the software installation. The entries in a software configuration template are like parameters in a response file for a silent installation of software.

Foreign configuration Configuration settings for software that hosts the software installation. For example, a software definition for a software patch might include configuration settings for the software product it fixes.

Instance An application instance. This template is useful for providing the ability to start and stop an application that you are running as a service, or control a native application instance in a product such as a DB2 Universal Database.

Application data Software entities that are generated when an application is used, such as data in a database. Application data resources are typically entities that cannot be recreated if they are lost and cannot be recovered through by reinstalling and configuring a product. Application data resources must be backed up so that they can be recovered if they are lost.

Feature An optional installation component of the software. For example, a software product might include optional utilities, plug-ins, or language packs that you define in a configuration template as features that a user can choose during installation.

Archived

Software resources

After you install software on a target system, Tivoli Provisioning Manager adds information to the data center model about the software components and data that are created on the system. These components are called software resources.

Software resources are created based on the settings defined in software configuration templates. The software configuration template defines the default parameters for a software resource. The software resource stores information about the actual parameter values that were used during installation on the target system.

During installation, administrators have the opportunity to change the default values for any template parameters that are configured as editable. For example, Figure B-5 shows that the default installation directory in the software installation template is C:\DB2. If this parameter is editable, an administrator can change the installation directory to another value, such as C:\IBM\DB2, at installation time. Parameter changes made at installation time do not change the template; they only affect the current installation.

Figure B-5 Software resources

Software resource types

The following types of software resources can be created on a managed system:

Software installation The software installation resource represents the associated software definition in the Tivoli Provisioning Manager software catalog.

- For a software product, operating system, or software patch, the software installation is a parent software resource to all other software resource types.
- When software stacks or composite applications are installed, an installation resource is associated with each software definition in the software stack or composite application.

Instance Instances provide the ability to use Tivoli Provisioning Manager to start and stop an application instance or an application that you are running as a service. For example, the default Apache automation package includes a software configuration template with an Apache instance so that you can stop and start Apache from Tivoli Provisioning Manager. An application such as DB2 Universal Database supports multiple instances. To create several database instances whenDB2 Universal Database is installed, the software definition must include one software instance template for each database instance. During installation, each template creates a software resource in the data center model for each instance that is configured on the target system.

Application data Application data that is added to the managed system. For example, if you have saved customer data for an application that you want to copy to a target during installation, you can configure the software configuration template to create this data as an application data resource.

Configuration Software configuration options defined by the software configuration template associated with the installable file. This can include configuration for the software installation itself, or configuration for software that hosts the software installation. For example, a software patch might include configuration for the software product that it fixes.

Generic resource Other types of resources for specific operations. For example, a backup resource defines the backup scope for backing up the associated software installation. These

settings are defined in the software configuration template.

Software stacks

A software stack is a special type of software definition that you can use to identify groups of software that you want to install at the same time and in a specific sequence on target systems.

A software stack can include installable files and software definitions for software products, software patches, and other software stacks.

Software stacks serve several purposes:

- You can consistently install the same software in the correct order and with the same configuration on managed systems.
- You can add software stacks to server templates so that Tivoli Provisioning Manager can identify managed systems that are not compliant with the software stack.
- You can install software stacks on individual systems or add them to server templates that you use with tiers for automatic installation on provisioned servers.

Software definitions

The list of software definitions in a software stack represent each piece of software that you want to install and the required installation order. You can also nest one software stack inside another software stack by adding the child software stack to the list of software definitions in the parent software stack.

To plan software stacks and ways to combine them, consider groups of software that you can install in different environments. You can create software stacks for these reusable groups and then incorporate them into larger software stacks that are for specific deployments.

Figure B-6 on page 543 shows some nesting software stacks in software definitions.

Archived

Figure B-6 Software stacks

For example, if you are setting up Apache Web servers and you want each Web server to include both Apache and two administration tools, you can create the software definitions shown in Table B-1.

Table B-1 Software definitions for software stacks

Software Definition	Contents
Apache for Windows	A software definition for the Apache for Windows software stack.
Monitoring tool	A software definition for one of the administration tools.
File manager	A software definition for one of the administration tools.
Administration Tools Stack	A software stack that contains the software definitions for both administration tools
Apache Server Stack	A software definition that contains the following software definitions: ► Apache for Windows ► Administration Tools Stack

Archived

Installation methods

There are several ways that you can configure the installation of software in a software stack. You can install individual pieces of software using the software definitions in the stack, install all the software in the software stack using an image, or provide both installation methods.

Install each piece of software individually

In this configuration, Tivoli Provisioning Manager uses the software definitions in the software stack to install each piece of software. To use this option, every software definition in the software stack must include at least one installable file. During installation, Tivoli Provisioning Manager starts with the first software definition, and uses the appropriate installable file in that software definition to install the first piece of software. It then proceeds to the next software definition in the list to install the next piece of software, and so on.

The software stack itself requires a special installable file called an iterator to indicate that Tivoli Provisioning Manager should use the software definitions to install each piece of software individually. Table B-2 shows an example of a software stack with software definitions and an iterator.

Table B-2 Example of a software stack with software definitions and an iterator

Configuration	Result
Software Definitions Product 1 Product 2 Product 3	Tivoli Provisioning Manager installs the software in the software definition list in the specified order.
Installable items Iterator	Tivoli Provisioning Manager selects the appropriate installable file in each software definition to install each piece of software.

Software images

A software stack can include one or more software images in its list of installable files. For example, you can create a standard Windows 2000 software stack that has three installable images that you use for three different types of servers. Based on the requirements of each image, Tivoli Provisioning Manager determines the image to deploy to a particular target server.

If you use this option, it is not necessary to include software definitions in the software stack for each piece of software in the image. However, you can include software definitions if you want to model the software in the image.

The installation of an image typically triggers workflows for deploying the image from a boot server defined in Tivoli Provisioning Manager. Since installing an

image overwrites software that is currently installed on the target system, software images should be used for software stacks that you deploy to new systems or systems that you want re-image.

Table B-3 shows an example of a software stack with images only.

Table B-3 Example of a software stack with images only

Configuration	Result
Software Definitions Product 1 Product 2 Product 3	The software definitions Product 1, Product 2, and Product 3 model the software products in both images, but Tivoli Provisioning Manager does not use this information to install the software stack.
Installable items Image 1 Image 2	Image 1 and Image 2 contain all the software described by the software definitions in the software stack. When you start installation of the software stack, Tivoli Provisioning Manager compares the configuration of the target system against the requirements of each image to determine which image to install.

Both options

If you want to make both software stack installation methods available, you must include the following items in the software stack:

- Software definitions for each piece of software in the software stack. Each software definition must include at least one installable file.
- An iterator.
- One or more images that contain the same pieces of software that are represented by the software definitions.

The placement of the iterator in the installable file list determines the priority of the available installation methods. For example, if the iterator is the first item in the list, Tivoli Provisioning Manager will use the list of software definitions to install the software instead of the images in the list. If the iterator is the last item in the list, Tivoli Provisioning Manager will only use list of software definitions to install the software stack if the target system does not match the requirements of all images listed in the software stack.

Since deploying images is generally more efficient than installing individual pieces of software, the iterator is typically placed at the end of the installable file list.

Table B-4 shows an example of a software stack with images and an iterator.

Table B-4 Example of a software stack with images and an iterator

Configuration	Result
Software Definitions Product 1 Product 2 Product 3	When you start installation of the software stack, Tivoli Provisioning Manager tries to use one of the defined images to install the software first by comparing the target system configuration with the requirements of each image.
Installable items Image 1 Image 2 iterator	If the target system does not match the requirements of the images, it checks the configuration of the target system against the requirements and capabilities of the software definitions, taking into consideration dependencies that are matched between software definitions in the software stack.

Software configuration templates

When you add software definitions to a software stack, a copy of the software configuration templates for each software definition are used to build the software configuration template for the entire software stack.

The following example shows the structure of a software configuration template for a software stack:

- Default Template: Unspecified resource installation
  - Apache Web Server for Windows.3019: Software installation
    - Apache Web Server: Software instance
  - WebsiteSample MODULE.3022: Software installation

The parent software configuration template type is “Unspecified resource installation”, since a software resource is not created on the target system to represent the entire software stack. Under the parent software configuration template, a copy of the software configuration template from each software definition in the software stack appears as a child template. The number at the end of the template name is the identifier for the copy of the template.

Since the child templates are copies, changes to the original templates do not affect the templates in the software stack, and changes to the template copies in the software stack do not affect the original templates. This separation provides you with the ability to change the parameter values for installation of the software stack without affecting the original software configuration templates.

Dependencies

Requirements in software stacks are specific to any images that are included in the installable file list of the software stack. If you are using software definitions to install the software stack, the requirements defined in the software definitions are

used to determine if a software stack can be installed on a target system. Tivoli Provisioning Manager takes into consideration that the requirements of one software definition might be met by the capabilities of another software definition that is also in the software stack.

Creation of software definitions

Software definitions are typically created by Tivoli Provisioning Manager or a workflow developer who builds the automation package for a piece of software.

Some parts of the software definition are closely tied to the workflows in the automation package. For example, The workflow developer can ensure that the software configuration template includes the parameters required by workflows in the automation package, and that the correct device models are defined in the automation package.

Tivoli Provisioning Manager automatically creates a software definition when it is provided with enough information to configure it. For example, if you configure Tivoli Provisioning Manager to import patch entries from a Microsoft Software Update Services server or software package entries from Tivoli Configuration Manager, the software can use the available information to automatically create software definition and installable file entries in the Tivoli Provisioning Manager software catalog.

Similarly, if an automation package includes a software definition, device models, and other software elements, the software elements are automatically added to Tivoli Provisioning Manager when you install the automation package.

Software provisioning

Tivoli Provisioning Manager is designed to provide a flexible environment to manage and deploy software.

Each piece of software that Tivoli Provisioning Manager manages must be defined in the Tivoli Provisioning Manager software catalog. The software catalog stores information about software in the environment that Tivoli Provisioning Manager manages. The software catalog can contain software that you deploy from Tivoli Provisioning Manager or entries that simply model software that is already installed in your environment.

Archived

For software that you want to install from Tivoli Provisioning Manager, two items are required:

The software to install
: The software package or software image that you want to install.

An automation package for the software
: The automation package contains the workflows and configuration information required to install and configure the software.

If the automation package that you require exists and is already installed, you can modify the software configuration to match your environment and then deploy the software. Otherwise, you must develop and install an appropriate automation package for the software.

In both situations, it is important to understand how software is represented in Tivoli Provisioning Manager and the various options that are available for software configuration and installation.

> **Note:** Tivoli Provisioning Manager provides additional support for Microsoft patch management. For information about managing patches, refer to the Tivoli Provisioning Manager patch management documentation. The information described in this section focuses on software product provisioning.

Developing software automation packages

When you create an automation package for a piece of software, you must consider how the software will be deployed and managed so that you can design an automation package that meets the needs of software administrators.

Creating an automation package for a piece of software includes the following main steps:

1. Collect information about the software package:
   - What format will you use to distribute the software? For example, will the package be distributed in a native operating system format, in a compressed file, or in a propriety format?
   - What are the procedures for installing the software and any other operations that will be available, such as uninstalling the software or starting instances?
   - What are the hardware and software requirements? What capabilities does the software have that can fulfill software requirements for other pieces of software?

2. Define the software definition, installable file, and the file repository that stores the software package.
3. Create the workflows for the procedures that you collected in step 1.
4. Determine the device models.
5. Create the automation package.

Building a software automation package

The following example demonstrates a simple Web site deployment. You will create automation package that performs the following installation procedure:

- Copies a .tar file with the HTML files to the Apache Web server.
- Extracts the HTML files to the htdocs directory.
- Changes permissions.
- Updates the location of the index.html file.

The automation packages for this example are available starting with Tivoli Provisioning Manager, Version 3.1.0.2.

WebsiteSample.tcdriver
: This automation package is available in the update_installer_dir/installer/samples directory, where update_installer_dir is the directory in which you unzipped the fix pack file.

Apache-Web-Server-Windows.tcdriver
: This automation package is available in the TIO_installdir/drivers directory.

Installing and configuring Apache

Before you can deploy your Web site, the Apache server must be installed. To install and configure the Apache Web server:

1. Install the Apache-Web-Server-Windows.tcdriver automation package. For information about installing automation packages, refer to the Workflow Developer's Guide. The automation package automatically adds a software definition for "Apache Web Server Windows" with a software configuration template, an installable file, and a device driver to the data center model. Review the configuration of these items so that you can better understand how the software will be installed.
2. Configure the Web server according to the automation package documentation. The readme.html file in the automation package describes the parameters to configure for the installable file and the software configuration template.
3. After you have configured the appropriate Apache settings, install Apache using the Install Software wizard. In the navigation pane, select **Tasks** → **Software Provisioning** → **Install Software**. If required, change the

Archived

parameters in the software configuration template to reflect the settings necessary for this installation. The changes you make in the wizard do not affect the original software configuration template. The software configuration template shows that Apache will be configured for with a single instance.

The following actions occur to install Apache:

- The Default_SoftwareModule_ Install workflow implements the SoftwareModule.Install logical device operation and selects the Apache Web Server Installable software package, since it is the only installable file in the software definition.
- The SoftwareInstallable.Install logical device operation calls the Apache_ Server_WIN_SoftwareInstallable_Install workflow to install the software package.
- The SoftwareInstallable.Install logical device operation also identifies an instance to create in the software configuration template and calls the SoftwareInstallation.AddInstance logical device operation to create it.
- The Apache_Server_WIN_SoftwareInstallation_AddInstance.wkf runs to create the instance.

Creating the Web site automation package

After the Apache server is installed and configured, you can develop the automation package for your Web site.

Prerequisites

This example assumes that a file repository is already configured for storing software packages. You can use the default local file repository or create a new one. Copy the WebsiteSample.tar file in the WebsiteSample.tcdriver automation package to the file repository. You will need to know the file repository and path where the file is stored to set up your automation package.

As you build your automation package, you can refer to the contents of the WebsiteSample.tcdriver automation package to view the examples described in this documentation. This automation package includes:

- A compressed file in .tar format that contains the HTML files for your Web site.
- An XML file with the software definition for the .tar file.
- One workflow to copy the HTML files to the Apache server.
- One workflow to remove the HTML files from the Apache server when you no longer require them.
- The manifest file for the automation package.

Creating the software definition

Create the software definition and installable file for the WebsiteSample.tar file. For the purpose of learning the process, you can use the Web interface or an XML file. However, when you create an automation package for use in your organization, include the software definition in XML format. This implementation ensures that the software information is automatically added to Tivoli Provisioning Manager and is ready for software administrators when the automation package is installed.

Define your software definition and installable file with the following information:

Software definition header

Name	The name of the software. For example, Database summary definition.
Description	A description of the software. For example, Database Schema for Tivoli Provisioning Manager V3.1.
Version	The version of the software. For example, V1.1

Installable file

In this example, your software package is the WebsiteSample.tar file in the WebsiteSample.tcdrier automation package. You must define the software package, its location, and the device driver that defines the workflows.

Name	The name of the software package. For example, Database summary package.
Description	A description of the software. For example, Database schema for Tivoli Provisioning Manager V3.1.
Version	The version of the software. For example, V1.1.
File Repository	The file repository where the software is stored.
Package Path	The path where the software is stored. The path is relative to the path defined for the file repository.
File Name	The name of the software package, WebsiteSample.tar.
Device Driver	The device driver for software package. This example uses a device model called WebsiteSample Installable.

Archived

Example B-1 shows the settings in XML format. The software configuration template will be added later, after the workflows to manage the package are created.

Example: B-1 Sample Software Package definition

```
<!DOCTYPE datacenter PUBLIC "-//Think Dynamics//DTD XML Import//EN"
"xmlimport.dtd">
<datacenter>
  <software-module name="Database summary definition"
      description="Database schema for Tivoli Provisioning Manager 3.1"
      version="1.1" locale="en_US" is-draft="false"
module-type="SOFTWARE">
  <installable-package name="WebsiteSample package"
      description= "Database schema for Tivoli Provisioning Manager
3.1"
      version="1.1" file-repository="LocalFileRepository"
locale="en_US"
      priority="1" status="tested" is-device-model="WebsiteSample
Installable">
    <file name="WebsiteSample.tar" path="/Apache" checksum=""
checksum-type=""
        locale="en_US"/>
  </installable-package>
</datacenter>
```

Creating the workflows and device models

After you have defined the software to install, develop the workflows that will automate the installation process and any other operations that are required to manage the software. The WebsiteSample.tcdriver automation package includes two workflows that you can use as examples:

WebsiteSample_Installable_Install
: Copies the WebsiteSample.tar file to a specified target Apache Web server and extracts the contents of the file to a directory defined by the InstallPath parameter.

WebsiteSample_Installation_Uninstall
: Removes the extracted HTML files.

These two workflows are associated with two device models that you will define in the manifest file for the automation package:

WebsiteSample Installable
: This device model is for the installable file that you have defined for the WebsiteSample.tar file. It includes the WebsiteSample_Installable_Install.wkf workflow.

WebsiteSample Installation

This device model is the software installation that is created when you install the WebsiteSample.tar file from Tivoli Provisioning Manager. It includes the WebsiteSample_Installation_Uninstall.wkf workflow.

Example B-2 shows the device model section of the manifest file for the automation package:

Example: B-2 Sample device model definition

```
<device-models>
  <device-model name="WebsiteSample Installable" category="Software">
    <workflow name="WebsiteSample_Installable_Install" />
  </device-model>
  <device-model name="WebsiteSample Installation" category="Software">
    <workflow name="WebsiteSample_Installation_Uninstall" />
  </device-model>
</device-models>
```

Creating the software configuration template

When you create the workflows for your automation package, you define the parameters that are required in the software configuration template. In this example, you only need to create a parent software configuration template for the software installation. The template includes the two workflow parameters.

InstallPath The directory where the HTML files will be extracted.

IndexPath The directory where the index.htm file for the Web site will be located after the HTML files are extracted.

Example B-3 shows the XML for the software definition, updated with the software configuration template.

Example: B-3 Sample Software Resource Template definition

```
<?xml version="1.0" encoding="ISO-8859-1"?> <!DOCTYPE datacenter PUBLIC
"-//Think Dynamics//DTD XML Import//EN" "xmlimport.dtd">
<datacenter>
  <software-module name="Database summary definition"
     description= "Database schema for Tivoli Provisioning Manager
3.1"
     version="1.1" locale="en_US" is-draft="false"
module-type="SOFTWARE">
    <installable-package name="WebsiteSample package"
          description="Database schema for Tivoli Provisioning Manager
3.1"
```

```
          version="1.1" file-repository="LocalFileRepository"
locale="en_US"
          priority="1" status="tested"
          is-device-model="WebsiteSample Installable">
      <file name="WebsiteSample.tar" path="/Apache" checksum=""
          checksum-type="" locale="en_US"/>
    </installable-package>
    <software-resource-template name="WebsiteSample Default Template"
          software-resource-type="INSTALLATION"
          software-resource-device-model= "WebsiteSample Installation">
      <template-param name="installPath"
  value="/cygdrive/c/Program Files/Apache
Group/Apache2/htdocs/WebsiteSample"/>
      <template-param name="indexPath"
          value="/cygdrive/c/Program Files/Apache
Group/Apache2/htdocs"/>
    </software-resource-template>
  </software-module>
</datacenter>
```

Completing the automation package

You now have the main components that you need to build your automation package. You can now create the manifest file for the automation package and package the required files.

> **Note:** The dependencies section of the manifest file specifies automation packages that are required by this automation package. They are independent of any requirements and capabilities that you define in XML files for software definitions. The software definition in this example automation package does not include predefined requirements and capabilities.

The WebsiteSample.tcdriver automation package contains the manifest file shown in Example B-4.

Example: B-4 Sample automation package manifest file

```
<?xml version="1.0" encoding="UTF-8"?> <!DOCTYPE tc-driver PUBLIC
"-//IBM//DTD TC Driver 2.0//EN"
"http://www.ibm.com/tivoli/orchestrator/dtd/tcdriver.dtd"> <tc-driver>
  <tc-driver-format>2.0</tc-driver-format>
  <driver-name>WebsiteSample</driver-name>
  <version>1.0</version>
  <description />
  <documentation location="doc/readme.html" />
```

```
  <dependencies>
    <dependency name="Apache-Web-Server-Windows" />
    <dependency name="core" />
  </dependencies>
  <property name="tc.pkg" location="com.thinkdynamics.kanaha.
tcdrivermanager.action" />
  <actions>
    <action name="copy-file" class="${tc.pkg}.CopyFileActions" />
    <action name="java-plugin" class="${tc.pkg}.JavaPluginActions" />
    <action name="workflow" class="${tc.pkg}.TxtWorkflowAction" />
    <action name="import" class="${tc.pkg}.ImportAction" />
  </actions>
  <items>
    <item name="workflow/WebsiteSample_Installable_Install.wkf"
      action="workflow"> <param name="editable" value="false" />
    </item>
    <item name="workflow/WebsiteSample_Installation_Uninstall.wkf"
      action="workflow"> <param name="editable" value="false" />
    </item>
    <item name="files/WebsiteSample.tar" action="copy-file">
      <param name="dest.path" value="{repository}/WebsiteSample/
WebsiteSample.tar" />
      <param name="chmod" value="755" />
    </item>
  </items>
  <device-models>
    <device-model name="WebsiteSample Installable" category="Software">
      <workflow name="WebsiteSample_Installable_Install" />
    </device-model>
    <device-model name="WebsiteSample Installation"
category="Software">
      <workflow name="WebsiteSample_Installation_Uninstall" />
    </device-model>
  </device-models>
  <dcm>
    <item name="xml/WebsiteSample.xml" action="import" />
  </dcm>
</tc-driver>
```

For more information about packaging an automation package, refer to the Workflow Developer's Guide, found at:

http://www.developer.ibm.com/tivoli/workflow.html

Software logical device operations

The following logical device operations are available for software.

SoftwareModule.DistributeInstallable

Calls the SoftwareInstallable.Distribute logical device operation. Used for Microsoft patch management to distribute patches to target systems without installing them.

Parameters:

SoftwareModuleID The object ID of the software module.

SoftwareInstallationMechanismID
The installation mechanism to use for installation of the software. When it is an installable file or an iterator to a software definition, this ID is created to identify the installation mechanism for the software. When this value is specified, Tivoli Provisioning Manager uses the associated installable file to install the software instead of performing requirements matching to select an installable file.

DeviceID The object ID of the target system.

SoftwareResourceTemplateID
The object ID of the software configuration template.

IgnoreFunctionalValidation
A Boolean flag that determines whether the requirements defined at the software definition level are validated against a target system. If this option is disabled, requirements are validated against the target system. If this option is enabled, Tivoli Provisioning Manager skips requirements checking at the software definition level. Requirements at the installable file level are still validated, since they define prerequisites that must be available for installation.

Access Restrictions None.

SoftwareModule.Install

Runs the workflow to install a piece of software. The default workflow that implements this logical device operation (Default_SoftwareModule_Install) selects the best installable file based on the requirements and capabilities and the priority order of installable files in the software definition. You can create modified versions of the workflow to use different criteria for selecting an installable file, for example, selecting an installable file based on the distance from the target system.

Archived

This logical device operation can be started in the following situations:

- An administrator uses the Install Software wizard to install a software package.
- When the default ServerTemplate.ConfigureServerByTemplate logical device operation runs to make a server compliant with its associated server template. SoftwareModule.Install runs for each piece of software that is missing from the target system.
- When the default Cluster.AddServer logical device operation runs. When a server is added to an application tier, the default Add Server workflow installs software that is required for the application tier.

Parameters

SoftwareModuleID The object ID of the software module.

SoftwareInstallationMechanismID
The installation mechanism to use for the installation of the software. When it is an installable file or an iterator to a software definition, this ID is created to identify the installation mechanism for the software. When this value is specified, the logical device operation uses the associated installable file to install the software instead of performing matching requirements to select an installable file.

DeviceID The object ID of the target system.

SoftwareResourceTemplateID
The object ID of the software configuration template.

IgnoreFunctionalValidation
A Boolean flag that determines whether the requirements defined at the software definition level are validated against a target system. If this option is disabled, requirements are validated against the target system. If this option is enabled, Tivoli Provisioning Manager skips requirements checking at the software definition level. Requirements at the installable file level are still validated since they define prerequisites that must be available for installation.

Access Restrictions DeviceID: Device.ManagerSoftware.

Archived

SoftwareInstallable.Distribute

Runs the workflow for Microsoft patch management that distributes patches to target systems without installing them.

Parameters

SoftwareID The object ID of the installable file.

DeviceID The object ID of the target system.

Access Restrictions None.

SoftwareInstallable.Install

Runs the workflow that copies and installs a software package on a target system and creates the software resources associated with the installation on the target system. Allocates software licenses if defined. Checks for child instance templates, and runs the SoftwareInstance.AddInstance logical device operation for each child template.

Parameters

SoftwareID The object ID of the installable file.

DeviceID The object ID of the target system.

SoftwareResourceTemplateID The object ID of the software configuration template.

Access Restrictions DeviceID:Device.ManagerSoftware.

SoftwareInstance.RemoveInstance

Runs the workflow that removes an instance from the target system. When the workflow is finished, the software resource for the instance is also removed.

Parameters

SoftwareInstanceID The object ID of the instance.

Access Restrictions None.

SoftwareInstance.Start

Runs the workflow that starts an instance on the target system. When the workflow is finished, the status of the software resource changes to Running.

Parameters

SoftwareInstanceID The object ID of the instance.

Access Restrictions None.

Archived

SoftwareInstance.Stop

Runs the workflow that stops an instance on the target system. When the workflow is finished, the status of the software resource changes to Not Running.

Parameters

SoftwareInstanceID The object ID of the instance.

Access Restrictions None.

SoftwareInstallation.AddInstance

Runs the workflow to add an instance to a software installation. Started by the SoftwareInstallable.Install logical device operation or the Create Instance command in the Web interface. Makes a copy of the Software Resource Template if it is already associated with a software resource. Creates the software resource for the software instance on the target system. Updates the software instance status to Not Running.

Parameters

SoftwareInstallationID
The object ID of the instance.

SoftwareResourceTemplateID
The object ID of the software configuration template.

Access Restrictions None.

SoftwareInstallation.Uninstall

Runs the workflow to remove a software installation. Started by the Uninstall command in the Web interface or by workflows that call the Cluster.Remove_Server logical device operation.

Parameters

SoftwareInstallationID
The object ID of the instance.

Access Restrictions None.

Device models for software

Different device models can be associated with the software elements. When you create an automation package for a piece of software, you must decide how you want to organize its device models. Device models or individual workflows can be associated with:

- Software definitions
- Installable files
- Software resources

Archived

Software configuration templates are not data center objects, but represent the resources they create. Since you cannot assign a device model to a software resource until it is created, you can associate it with the software configuration template instead. When the software resource is created, the template automatically assigns a device model to the software resource.

There are two main approaches to organizing device models:

Unified device model

In this approach, a single device model with all the workflows for the software elements is assigned to the software definition. You must use a device model instead of assigning the individual workflows to the software definition. Tivoli Provisioning Manager requires the logical device operations to identify the target software element. You can identify the target software element for a logical device operation by looking at its name.

If you are creating a software definition with multiple installable files, you can assign separate workflows to each installable file that implement the SoftwareInstallable.Install logical device operation.

This approach is the easier to implement but is less flexible than using separate device models for each software element.

Separate device models

In this approach, you create individual device models for each software element associated with a software definition. This approach is more flexible than using a single device model because it can provide more opportunities to reuse device models in other software definitions.

Installing the Web site

When your automation package for the Web site is complete, you can install the automation package and then deploy the software package to the Apache server.

Installing the automation package

1. On the Tivoli Provisioning Manager server, go to the TIO_installdir/tools directory.
2. Run the following command:
   - Windows:

     ```
     tc-driver-manager.cmd installDriver WebsiteSample.tcdriver
     ```

– UNIX or Linux:

```
tc-driver-manager.sh installDriver WebsiteSample.tcdriver
```

When the automation package is installed, verify that the software items are in the data center model. In the navigation pane, expand **Inventory** → **Software Catalog**. Check for the following items:

- A software package entry for the WebsiteSample.tar file under Installables.
- A software definition under Software Definitions that contains:
  - A link to the software package.
  - The software configuration template that was defined in the automation package.

Installing the Web site

Run the Install Software wizard to deploy the Web site to the Apache server:

1. Click **Tasks** → **Software Provisioning** → **Install Software**.
2. Follow the instructions in the wizard. Ensure that you select the following options:

   Target system The Apache server that you are using for this automation package example.

   Software The software definition for the Web site package.

   Software configuration template Verify that the path values are correct.

The files are installed on the Apache server. Verify that the files were correctly deployed by pointing your Web browser to the index.html file.

Extending the automation package example

After you have successfully created and deployed the example Web site automation package, you can try to apply other aspects of the software model to the example.

The following sections provides several ideas for extending the example:

Add software dependencies

Add requirements to the software definitions for Apache or the Web site. For example:

- You can add a requirement to the Apache software definition for the Windows operating system that is installed on the target server. To successfully implement this option, the target server that you are using must have a

Archived

software installation for the Windows operating system with a capability that matches your defined requirement.

- You can add a requirement for Apache to the software definition for the Web site. To successfully implement this option, you must:
  - Add a requirement for Apache to the software definition for the Web site.
  - Add a capability to the Apache software definition.

Create a software stack

Many organizations manage multiple Web servers. To make it easier to deploy both the Apache software and the Web site, you can create a software stack that includes both software definitions and an iterator as the installable file.

Add a software stack to a server template

Server templates are useful for installing a software stack when a new server is added to a resource pool or application tier. If you create a software stack with the Apache server and Web site files, you can test this process by perform the following steps:

1. Assign the software stack to a server template.
2. Assign the server template with a resource pool or application tier for Apache servers.
3. In the workflow that implements Cluster.Add Server or SparePool.InitializeServer, ensure that the workflow calls the ServerTemplate.ConfigureServerByTemplate logical device operation.
4. Add a new server to the application tier or resource pool to test the workflows.

Access control

If you enable access control in Tivoli Provisioning Manager, you must carefully configure and assign the appropriate security roles, access permissions, and access groups. Ensure that administrators have appropriate permissions and have access to all the assets that they need to view or manage. Considerations for access groups include:

- Administrators who manage infrastructure, such as file repositories, must be a member of an access group that contains the file repository.
- Administrators who manage computer systems or install software on them must be a member of access groups that contain the application tiers, resource pools, and systems that they are permitted to view or manage.
- Administrators who configure or install software must be a member of an access group that contains the software definitions that they are permitted to access and all the associated installable files or installable images. Installable files do not inherit the access group configuration for parent software

definitions, so you must ensure that both software definitions and installable files are included in the appropriate access groups.

This software automation package example focused on deployment of software packages. Tivoli Provisioning Manager also supports deployment of software images using a boot server configured in Tivoli Provisioning Manager. For more information about images, refer to the image information in the Tivoli Provisioning Manager information center.

Archived

Archived

C

Popular Tivoli Java classes and methods

This appendix contains listings of the most popular Java classes and methods called from workflows provided out-of-the-box with IBM Tivoli Provisioning Manager V3 and IBM Tivoli Intelligent Orchestrator V3.

The listing is divided into two distinct groups: method calls that return values, shown in Table C-1 on page 566, and method calls that do not return values, shown in Table C-2 on page 576.

© Copyright IBM Corp. 2004, 2006. All rights reserved.

Java methods returning values

Table C-1 lists the most popular Java classes and methods that return values. To call these methods from your workflow, call them using the following syntax:

```
var x =Java[class#method(args)]
```

Table C-1 Polular Java classes and methods returning variables

Class	Method	Arguments	Return value
com.ibm.tivoli.itm.javaplugin.ProfileXMLParser	getProfileList	ResourceID,Profile_XML_File_Name	Prereqer_Results
com.ibm.tivoli.jds.client.xml.JobDistributionService	submitJob	job,targetServers	result
com.ibm.tivoli.orchestrator.apptopo.workflow.AppTopoAgentHelper	findNetworkInterfaceByServer	ServerID	NewNetworkInterfaceId
	findNicByServer	ServerID	NicID
com.ibm.tivoli.orchestrator.cascommon.AgentUtil	getAgentSoftwareModule	agentCapabilityName,os_family,platform_architecture	tcaSoftwareModuleID
	getUIDforJDS	DeviceID,Port	newJdsUID
	pingAgent	DeviceID,Port	newAgentGUID
com.ibm.tivoli.orchestrator.cascommon.AMPasswordHelper	getAgentRegistrationPassPhrase		RegPasswd
	getResourceManagerUserName		AMUserName
com.ibm.tivoli.orchestrator.cascommon.TPMRegistrationManager	getAMRegistrationHost		Agent_Manager_Server_FQN
	pingAgentManagerFromDE		pingAM
com.ibm.tivoli.orchestrator.datacentermodel.helper.ApplicationTopologyHelper	findMatchingNic	ServerID,ServerTemplateID,NicTemplateIds	arrayNicIds

Archived

Class	Method	Arguments	Return value
com.ibm.tivoli.orchestrator.datacentermodel.helper.ClusterDomainHelper	deleteClusterResourceDependency	ClusterResourceID,DependencyID	result
	updateResourceAttributes	ClusterResourceID,"KANAHA",NewResourceAttributeNames,NewResourceAttributeValues	result
com.ibm.tivoli.orchestrator.datacentermodel.helper.HostPlatformHelper	allocateVirtualServer	HostPlatformID,ServerTemplateID,Name	ServerID
	getHostPlatformSoftware	HostPlatformID	hostPlatformSoftwareID
com.ibm.tivoli.orchestrator.datacentermodel.helper.NetworkingHelper	getAllocatedIpAddress	NicID,SubnetworkID	arrayAllocatedIPAddress
	isInTrunkingMode	EndpointID	isInTrunkingMode

Archived

Class	Method	Arguments	Return value
com.ibm.tivoli.orchestrator.datacentermodel.helper.SoftwareHelper	areInstallableHardwareRequirementsSatisfied	ImageID,DestinationDeviceID,"false"	areRequirementsSatisfied
	createSoftwareResources	SoftwareID,DeviceID	softwareInstallationId
	getOperatingSystemFamilyInstalledOnDeviceId	DeviceID	OSFamily
	getOperatingSystemInstalledOnDeviceId	DeviceID	OperatingSystemID
	getOperatingSystemServicePackInstalledOnDeviceId	DeviceID	OSServicePack
	getOperatingSystemVersionInstalledOnDeviceId	DeviceID	OSVersion
	getSoftwareResource	SoftwareResourceTemplateID	softwareInstallationId
	getSoftwareResourceTypeId	INSTANCE	instanceResourceTypeId
	getTemplateParamValueById	paramid	paramvalue
	getTemplateParamValueByName	params	SoftwareResourceID
	instantiateSoftwareResourceTemplate	SoftwareModuleID, SoftwareResourceTemplateID	SoftwareResourceTemplateID
	selectSoftwareInstallable	SoftwareModuleID, DeviceID,IgnoreFunctionalValidation	SoftwareInstallableID

Archived

Class	Method	Arguments	Return value
com.ibm.tivoli.orchestrator.datacentermodel.helper.SoftwareHelper	validateRequirements	SoftwareModuleID,SoftwareInstallableID,DeviceID,IgnoreFunctionalValidation	requirementsValidated
com.ibm.tivoli.orchestrator.datacentermodel.software.JdsPatchEventHandler	processJdsSubmissionFailure	tpmTaskId,targetServers	taskComplete
com.ibm.tivoli.orchestrator.datacentermodel.storage.ESSUtility	discoverStorageVolumes	StorageSubsystemID,vollist_file,volspace_file	result
	discoverVolumeAccess	StorageSubsystemID,volaccess_file,"true"	result
com.ibm.tivoli.orchestrator.datacentermodel.storage.SanFabricUtility	getZoneMapping	ServerID,StorageVolumeIDs,MultiPathSettingsIDs	zoneMapping
com.ibm.tivoli.orchestrator.datacentermodel.storage.StorageAllocationPoolUtility	getStorageVolumes	StoragePoolID,StorageCapabilitiesRequirementIDs	StorageVolumeIDs
com.ibm.tivoli.orchestrator.datacentermodel.storage.StorageSubsystemUtility	getDataPathMapping	ServerID,StorageVolumeIDs,MultiPathSettingsIDs	dataPathMapping
	getFAMapping	StorageVolumeIDs,MultiPathSettingsIDs	faMapping
	getHBAMapping	ServerID,StorageVolumeIDs,MultiPathSettingsIDs	hbaMapping
	getStorageVolumes	StorageSubsystemID,StorageCapabilitiesRequirementIDs	StorageVolumeIDs
com.ibm.tivoli.orchestrator.de.instruction.impl.WorkflowResolver	getWorkflowName	DeviceID,"Device.SoftwareRebootAsync"	workflowName

Archived

Class	Method	Arguments	Return value
com.ibm.tivoli.orchestrator.de.scriptlet.ScriptExecutionAdapter	getCmd	filename,scriptId,scriptParameters	ExecuteCommand
	toFile	scriptId,local_tmp	filename
com.ibm.tivoli.orchestrator.de.scriptlet.ScriptletImpl	extract	instruction_id,sap_id,local_tmp	criptlet_filename
	getExecuteSapId	target_id	ap_id
	getLocalId		local_id
	getLocalTemp		local_tmp
	getTargetId	instruction_id,stack_frame_id	target_id
com.ibm.tivoli.orchestrator.discovery.netview.DiscoveryGetDevices	getDevicesUsingNetviewDumpFile	cygpathDumpFile	xmlFile
com.ibm.tivoli.orchestrator.discovery.tio.CitScannerHelper	getEndDate		expirationDate
	getMBSAEndpoints	windows,"MicrosoftBaselineSecurityAnalyzer%","CitScannerAgent"	arrayservers
	getStartDate		tartDate
com.ibm.tivoli.orchestrator.discovery.tio.TCARunCitScanner	doIt	DeviceID	cit_outPath
com.ibm.tivoli.orchestrator.discovery.util.AddSubnetworkToXml	start	oldWinPath,newWinpath	ch
com.ibm.tivoli.orchestrator.discovery.util.DcmDeviceIntegrator	importDiscoveredData	newWinpath,DiscoveryID	result
com.ibm.tivoli.orchestrator.discovery.util.DcmDriftIntegrator	processDiscoveredData	xmlFile,DiscoveryID,DeviceIDs	result
com.ibm.tivoli.orchestrator.discovery.util.DcmInsertIntegrator	insertNewDevices	xmlFile,DiscoveryID,ConfigDriftIDs	result

Archived

Class	Method	Arguments	Return value
com.ibm.tivoli.orchestrator.discovery.util.DcmUpdateIntegrator	updateDcm	newWinpath,DiscoveryID,DeviceID,ConfigDriftIDs	result
com.ibm.tivoli.orchestrator.itcm.javaplugin.DiscoveryGetThirdPartyPackages	createThirdPartyXML	Host_Name,Host_ID,PackageID,ConfigDriftIDs,Workflow_Execution_ID	xmlFile
com.ibm.tivoli.orchestrator.itcm.javaplugin.GetValue	getFirstSoftwareDistributionAppName	Host_ID	SW_Dist_App_Name
	getFirstThirdPartySoftwarePackageID	SoftwareModuleID	TPackageID
	getRootPolicyRegionName	TCM_Host_Name	Policy_Region_Name
com.ibm.tivoli.orchestrator.itcm.javaplugin.ImportEndpoints	parseAndExecuteFilters	tcmHostName,List_Name,tcmTMALabelPtr,Type,GW_Filter_String_Data,xmlFile	retGW
com.ibm.tivoli.orchestrator.itcm.javaplugin.ImportSoftwarePackages	executeFilters	List_Name,Host_Name,Locale,SW_Dist_App_Name,Device_Model_Name, Workflow_Execution_ID	xmlFile
com.ibm.tivoli.orchestrator.itcm.util.ExceptionThrower	throwNullDiscoveryFilterString	DiscoveryID	temp
	throwNullLocaleServerException	Host_ID	temp

Archived

Class	Method	Arguments	Return value
com.ibm.tivoli.orchestrator.itcm.util.ExceptionThrower	throwNullRepositoryID		temp
	throwNullRepositoryPath	ThirdPartySoftwarePackageID	temp
	throwNullRepositoryPath	tPackageName	temp
	throwNullServerIDFromSoftwareResource	SoftwareInstallationID	temp
	throwNullSoftwareModuleIDFromSoftwareResource	SoftwareInstallationID	temp
	throwNullThirdPartyIDFromSoftwareModule	SoftwareInstallationID	temp
com.ibm.tivoli.orchestrator.itcm.util.TCMConnector	obtainSAPAndConnectToITCM	tcmHostName,tcmServerID	retSAP
com.ibm.tivoli.orchestrator.tca.subagents.installer.SubagentInstallUninstall	getSubagentCopyDir	DeviceID,Port	targetCopyPath
	installSubagent	Subagent_Package_Path,targetCopyPath,Subagent_Filename,DeviceID,Port	temp
	uninstallSubagent	bundleLocation,Subagent_Filename,DeviceID,Port	temp
com.ibm.tivoli.orchestrator.tca.TCAUtil	writeSRTProperties	DeviceID,Port,SoftwareModuleID,Subagent_Package_Path,Subagent_Filename,Workflow_Execution_ID	temp
com.ibm.tivoli.orchestrator.tmf.filter.SoftwarePackageStringParser	parseSoftwarePackageString	Host_Name,List_Name,Filter_String_Data	temp

Archived

Class	Method	Arguments	Return value
com.ibm.tivoli.orchestrator.tmf.proxy.Connector	disconnect	tcmHostName	retdCon
com.ibm.tivoli.tma.javaplugin.TMAInstallConfiguration	createResponseFile	SoftwareResourceTemplateID,SoftwareID,WorkflowExecutionID,TMA_System_IP_Address	ReturnResult
	getCmdLineOptions	ReturnResult	CommandLineOptions
	getInstallationDirectory	ReturnResult	Installation_Directory
	getPolicyRegion	ReturnResult	PolicyRegion
	getResponseFileDirectory	ReturnResult	Response_File_Directory
	getTCMServerIP	ReturnResult	TCM_Server_IP_Address
com.ibm.tivoli.xmlimport.javautility.XMLImporter	importFile	local_filename	result
com.thinkdynamics.kanaha.datacentermodel.inprocess.GroupStatsUCImpl	insertNewMaintenanceTotal		result
com.thinkdynamics.kanaha.de.javaplugin.datacentermodel.DCMHelper	getCurrentRecommendationSizeByServer	ServerID	recommendation
	getOperatingSystemIdForDevice	DeviceID	OperatingSystemID
	getTheOnlyNicIdForDevice	DeviceID	NICID
com.thinkdynamics.kanaha.de.javaplugin.Helper	getIpAddressByName	hostName	ipAddress

Archived

Class	Method	Arguments	Return value
com.thinkdynamics.kanaha.de.sp.ServiceProvisionHelper	getOrderAgreementTermValue	SPOrderID,termName	NumberOfServers
	scheduleDeprovisionTask	SPServiceID,SPServiceInstanceID,SPOrderID	result
	scheduleImmediateDeprovisionTask	SPServiceID,SPServiceInstanceID,SPOrderID	result
	scheduleProvisionTask	SPServiceID,SPServiceInstanceID,SPOrderID	result
	updateServiceInstanceStatus	SPServiceInstanceID,instStatus	result
com.thinkdynamics.kanaha.de.utilities.WorkflowUtility	splitString	FAPortIDs,delimiter	arrayFAPortIDsArray
com.thinkdynamics.kanaha.util.PathHelper	convertToCygwinPath	FilePath	FilePath
	escapeSpaces	local_path_cyg	local_path_cyg_no_spaces
	getDirectoryNameFromPath	cit_outPath	tmpSourcePath
	getFileNameFromPath	cit_outPath	SourceFile
	normalizeUnixFileName	DestinationPath,DestinationFile	oldFileFullPath
	onlyForwardSlash	tmpSourcePath	SourcePath

Archived

Class	Method	Arguments	Return value
java.lang.System	currentTimeMillis		startTime
	getProperty	file.separator	File_Sep
		java.io.tmpdir	Tmp_Dir
		java.io.tmpdir	Tmp_Dir
		tio.home	tio_home
		tio.home.unix	tio_home_unix
java.net.InetAddress	getLocalHost		addr
java.util.Random	nextInt	10000	randomInt

Archived

Java classes not returning values

Table C-2 lists the most popular Java classes that do not return values, which is referenced from workflows provided out-of-the-box with IBM Tivoli Intelligent Orchestrator V3 or IBM Tivoli Provisioning Manager V3.

In order to call the methods of any of these classes from your workflows, use the following syntax:

```
java.class#method(args)
```

Table C-2 Popular Java classes and methods referenced from workflows

CLASS	METHOD	ARGUMENTS
com.ibm.tivoli.orchestrator.apptopo.workflow.AppTopoAgentHelper	addRIPToVirtualIP	vipId,NewNetworkInterfaceId,Out_TCP_Port,Protocol,Weight,RIPMaxConnections
	createVirtualIP	lbl_dcm_id,iterator_cluster_dcm,lbl_vipname,lbl_VIPAddress,lbl_intportfirst,lbl_inportlast,lbl_outport,lbl_argorithm
	deleteNetworkInterface	NewNetworkInterfaceId
	deploymentApplicationInDCM	ApplicationID,dpXml,app_locale
	moveNetworkInterfaceIPAddress	NewNetworkInterfaceId,SubnetworkID,New_IP_Address
	removeVirtualIP	vip_dcm_id
	rmvRIPFromVirtualIP	vipId,NewNetworkInterfaceId
	setNicVlan	NicID,VlanID
	setServerFailed	true,ServerID
	setServerInMaintenance	true,ServerID
	undeploymentApplicationInDCM	ApplicationID

Archived

CLASS	METHOD	ARGUMENTS
com.ibm.tivoli.orchestrator.apptopo.workflow.UpdateApplicationDeployment	removeLogicalClusterByVIP	vip_dcm_id
	removeServerFromCluster	RequestID,ServerID,ClusterID
	updateApplicationDeploymentStatus	ApplicationID,appStatus
	updateApplicationToMaintenance	ApplicationID,"true"
	updateLogicalCluster	logical_cluster_dcmid,vip_dcm_id
	updateServerToCluster	RequestID,ServerID,ClusterID
	updateServerToPool	server_dcmid,pool_dcm_id
com.ibm.tivoli.orchestrator.datacentermodel.helper.HostPlatformHelper	deallocateVirtualResource	ResourceAllocationID
	destroyVirtualServer	ServerID
	expandResourceAllocation	HostPlatformID,ResourceAllocationID,HowMany,Size
	reduceResourceAllocation	HostPlatformID,ResourceAllocationID,HowMany,Size
com.ibm.tivoli.orchestrator.datacentermodel.helper.MonitoringHelper	createMonitoringResourceAssociation	MonitoringApplicationID,ResourceID,MonitoringConfigID
	deleteMonitoringResourceAssociation	MonitoringApplicationID,ResourceID
	updateMonitoringResourceAssociationState	MonitoringApplicationID,ResourceID,DesiredState

Archived

CLASS	METHOD	ARGUMENTS
com.ibm.tivoli.orchestrator.datacentermodel.helper.NetworkingHelper	changeSwitchPortStatus	SwitchPortID,"false"
	checkNetworkInterface	RouterID,NetworkInterfaceID
	deleteVlan	SwitchID,VLANID
	postCarryVLANThroughSwitchEndpoint	EndpointID,VLANID
	postChangeSwitchEndpointAttributes	EndpointID,NativeVLANID,Mode,Encapsulation
	postClearVLANFromSwitchEndpoint	EndpointID,VLANID
	postCreateRoute	NetworkInterfaceID,SubnetworkID,Gateway
	postDisableACL	ACLID,DestinationNetworkInterfaceID
	postEnableACL	ACLID,DestinationNetworkInterfaceID,ACLType
	postRemoveACL	ACLID,FirewallID
	postRemoveRoute	RouteID
	setSwitchVlanId	SwitchPortID,DestinationVLANID
	validateTrunkingEncapsulation	Encapsulation
	validateTrunkingMode	Mode

Archived

CLASS	METHOD	ARGUMENTS
com.ibm.tivoli.orchestrator.datacentermodel.helper.SoftwareHelper	copySourceImagePropertiesToDestImage	ImageID,NewImageID
	copySourceImageReqsToDestImage	ImageID,NewImageID
	copySourceImageSAPsToManagedSystem	ImageID,DestinationDeviceID
	copySourceImageSAPsToManagedSystem	ImageID,NewImageID
	copySourceImageWorkflowsToDestImage	ImageID,NewImageID
	createPendingSoftwareResources	ImageID,softwareModuleID,DestinationDeviceID,SoftwareResourceTemplateID
	createSoftwareResources	ImageID,DestinationDeviceID
	deletePendingSoftwareInstances	softwareInstallationId
	deleteSoftwareInstallation	SW_Installation_ID
	deleteSoftwareInstance	SoftwareResourceID
	updateResourceFlag	softwareInstallationID,"false"
	updateResourceStatus	CVS_Installation,Software_Status
com.ibm.tivoli.orchestrator.datacentermodel.storage.StorageManagerUtility	updateLogicalVolumeLinks	LogicalVolumeId,OutputLogicalVolumeId
com.ibm.tivoli.orchestrator.de.javaplugin.tpmtask.JdsTaskDispatcher	targetWorkflowDispatch	TaskID,jdsJobInstallationWorkflow,jdsJobDistributionWorkflow,jdsAgentSoftware,failurePolicy,priority,executionMode
	targetWorkflowCancellation	TaskID
	targetWorkflowDispatch	TaskID

Archived

CLASS	METHOD	ARGUMENTS
com.ibm.tivoli.orchestrator.de.scriptlet.ScriptletImpl	executeRemote	instruction_id,stack_frame_id,target_tmp,remote_script_filename
	removeLocal	local_tmp,scriptlet_filename
com.ibm.tivoli.orchestrator.de.task.TaskExecutionHelper	checkPermissions	TaskID
	checkTaskSchedule	TaskID
com.ibm.tivoli.orchestrator.itcm.javaplugin.GetValue	validateItcmScriptDirectory	Scripts_Repository
com.ibm.tivoli.orchestrator.itcm.util.ExceptionThrower	throwNullTMALabel	
	throwNullTMALabelKey	TCM_Server_ID
com.ibm.tivoli.tec.javaplugin.TecEventSender	sendTECEvent	TecClassName,AttributeNames,AttributeValues
com.ibm.tivoli.tma.javaplugin.SetProperty	setFilePath	SoftwareInstallableID,ImageDirectory
	setProperty	DCMPropertyKey,TMA_Label,ServerID
	setVersion	TMA_Version,SoftwareInstallableID
com.ibm.tivoli.tma.javaplugin.UpdateSoftwareStatus	setRunning	SoftwareResourceID
	setStopped	SoftwareResourceID
	setUninstalled	SoftwareResourceID
com.thinkdynamics.ejb.deadaptor.DEAdaptorProxy	addServer	deploymentRequestID,ClusterID,ServerID
	releaseServer	deploymentRequestID,ServerID
	removeServer	deploymentRequestID,ServerID

Archived

CLASS	METHOD	ARGUMENTS
com.thinkdynamics.ejb.licensemanagement.LicenseAllocationProxy	allocate	softwareInstallableID,DestinationDeviceID,"true"
	allocateByTemplate	SoftwareResourceTemplateID,DestinationDeviceID,"true"
	deallocate	softwareInstallableID,DestinationDeviceID,"true"
	deallocateByResource	softwareInstallationID,"true"
com.thinkdynamics.kanaha.de.javaplugin.datacentermodel.DCMHelper	deleteServiceAccessPoint	ServiceAccessPointID
	postApplyRoutingTable	DeviceID,RoutingTableID,NetworkInterfaceID,RemoveExistingRoutes
	setDeviceDriverForDCMObject	DeviceDriverID,DCMObjectID
com.thinkdynamics.kanaha.de.javaplugin.datacentermodel.TpmTaskHelper	setTpmTaskStartDate	TaskID
	updateTpmTaskStatus	TaskID,"7"
com.thinkdynamics.kanaha.de.messagetranslator.MessageTranslatorProxy	cancelDeploymentRequest	originalRequestId

Archived

Archived

D

Jython reference

This appendix contains listings of the modified objects used throughout this IBM Redbook.

© Copyright IBM Corp. 2004, 2006. All rights reserved.

Jython string methods

The information in Table D-1 has been extracted from the Python Library Reference which can be accessed online at http://www.python.org/doc/lib/contents.html.

Table D-1 Jython string methods

Method	Description
capitalize( )	Returns a copy of the string with only its first character capitalized.
center(width[, fillchar])	Returns centered in a string of length width. Padding is done using the specified fillchar (default is a space). Changed in version 2.4: Support for the fillchar argument.
count(sub[, start[, end]])	Returns the number of occurrences of substring sub in string S[start:end]. Optional arguments start and end are interpreted as in slice notation.
decode([encoding[, errors]])	Decodes the string using the codec registered for encoding. Encoding defaults to the default string encoding. Errors may be given to set a different error handling scheme. The default is 'strict', meaning that encoding errors raise UnicodeError. Other possible values are 'ignore', 'replace', and any other name registered via codecs.register_error.
encode([encoding[,errors]])	Returns an encoded version of the string. The default encoding is the current default string encoding. Errors may be given to set a different error handling scheme. The default for errors is 'strict', meaning that encoding errors raise a UnicodeError. Other possible values are 'ignore', 'replace', 'xmlcharrefreplace', 'backslashreplace', and any other name registered via codecs.register_error.
endswith(suffix[, start[, end]])	Returns True if the string ends with the specified suffix, otherwise, returns False. With optional start, test beginning at that position. With optional end, stop comparing at that position.
expandtabs([tabsize])	Returns a copy of the string where all tab characters are expanded using spaces. If tabsize is not given, a tab size of eight characters is assumed.
find(sub[, start[, end]])	Returns the lowest index in the string where substring sub is found, such that sub is contained in the range [start, end]. Optional arguments start and end are interpreted as in slice notation. Return -1 if sub is not found.

Archived

Method	Description
index(sub[, start[, end]])	Like find(), but raises ValueError when the substring is not found.
isalnum( )	Returns true if all characters in the string are alphanumeric and there is at least one character. Returns false otherwise.
isalpha( )	Returns true if all characters in the string are alphabetic and there is at least one character. Returns false otherwise.
isdigit( )	Returns true if all characters in the string are digits and there is at least one character. Returns false otherwise.
islower( )	Returns true if all cased characters in the string are lowercase and there is at least one cased character. Returns false otherwise.
isspace( )	Returns true if there are only whitespace characters in the string and there is at least one character. Returns false otherwise.
istitle( )	Returns true if the string is a titlecased string and there is at least one character, for example, uppercase characters may only follow uncased characters and lowercase characters only cased ones. Returns false otherwise.
isupper( )	Return true if all cased characters in the string are uppercase and there is at least one cased character. Returns false otherwise.
join(seq)	Returns a string that is the concatenation of the strings in the sequence seq. The separator between elements is the string providing this method.
ljust(width[, fillchar])	Returns the string left justified in a string of length width. Padding is done using the specified fillchar (default is a space). The original string is returned if width is less than len(s). Changed in Version 2.4: Support for the fillchar argument.
lower( )	Returns a copy of the string converted to lowercase.
lstrip([chars])	Returns a copy of the string with leading characters removed. The chars argument is a string specifying the set of characters to be removed. If omitted or None, the chars argument defaults to removing whitespace. The chars argument is not a prefix; rather, all combinations of its values are stripped.

Archived

Method	Description
replace(old, new[, count])	Returns a copy of the string with all occurrences of substring old replaced by new. If the optional argument count is given, only the first count occurrences are replaced.
rfind(sub [,start [,end]])	Returns the highest index in the string where substring sub is found, such that sub is contained within s[start,end]. Optional arguments start and end are interpreted as in slice notation. Return -1 on failure.
rindex(sub[, start[, end]])	Like rfind(), but raises ValueError when the substring sub is not found.
rjust(width[, fillchar])	Returns the string right justified in a string of length width. Padding is done using the specified fillchar (the default is a space). The original string is returned if width is less than len(s). Changed in Version 2.4: Support for the fillchar argument.
rsplit([sep [,maxsplit]])	Returns a list of the words in the string, using sep as the delimiter string. If maxsplit is given, at most maxsplit splits are done, the rightmost ones. If sep is not specified or None, any whitespace string is a separator. Except for splitting from the right, rsplit() behaves like split(), which is described in detail below. New in Version 2.4.
rstrip([chars])	Returns a copy of the string with trailing characters removed. The chars argument is a string specifying the set of characters to be removed. If omitted or None, the chars argument defaults to removing whitespace. The chars argument is not a suffix; rather, all combinations of its values are stripped.

Archived

Method	Description
split([sep [,maxsplit]])	Returns a list of the words in the string, using sep as the delimiter string. If maxsplit is given, at most maxsplit splits are done. (thus, the list will have at most maxsplit+1 elements). If maxsplit is not specified, then there is no limit on the number of splits (all possible splits are made). Consecutive delimiters are not grouped together and are deemed to delimit empty strings (for example, "'1,,2'.split(',')"returns "['1', '', '2']"). The sep argument may consist of multiple characters (for example, "'1, 2, 3'.split(', ')" returns "['1', '2', '3']"). Splitting an empty string with a specified separator returns "['']". If sep is not specified or is None, a different splitting algorithm is applied. First, whitespace characters (spaces, tabs, newlines, returns, and formfeeds) are stripped from both ends. Then, words are separated by arbitrary length strings of whitespace characters. Consecutive whitespace delimiters are treated as a single delimiter ("'1 2 3'.split()" returns "['1', '2', '3']"). Splitting an empty string or a string consisting of just whitespace returns an empty list.
splitlines([keepends])	Returns a list of the lines in the string, breaking at line boundaries. Line breaks are not included in the resulting list unless keepends is given and true.
startswith(prefix[, start[, end]])	Returns True if the string starts with the prefix, otherwise. return False. With optional start, test string beginning at that position. With optional end, stop comparing string at that position.
strip([chars])	Returns a copy of the string with the leading and trailing characters removed. The chars argument is a string specifying the set of characters to be removed. If omitted or None, the chars argument defaults to removing whitespace. The chars argument is not a prefix or suffix; rather, all combinations of its values are stripped.
swapcase( )	Returns a copy of the string with uppercase characters converted to lowercase and vice versa.
title( )	Returns a titlecased version of the string: words start with uppercase characters, all remaining cased characters are lowercase.

Archived

Method	Description
translate(table[, deletechars])	Returns a copy of the string where all characters occurring in the optional argument deletechars are removed, and the remaining characters have been mapped through the given translation table, which must be a string of length 256. For Unicode objects, the translate() method does not accept the optional deletechars argument. Instead, it returns a copy of the s where all characters have been mapped through the given translation table, which must be a mapping of Unicode ordinals to Unicode ordinals, Unicode strings, or None. Unmapped characters are left untouched. Characters mapped to None are deleted. Note that a more flexible approach is to create a custom character mapping codec using the codecs module (see encodings.cp1251 for an example).
upper( )	Returns a copy of the string converted to uppercase.
zfill(width)	Returns the numeric string left filled with zeros in a string of length width. The original string is returned if width is less than len(s).

Archived

List methods

Table D-2 shows the list methods.

Table D-2 List methods

Method	Description
__len__()	The length of the object.
__findattr__(name)	Returns the value corresponding to name or null if name is not found. *o* - The name to lookup in this namespace must be an interned string.
__imul__(o)	Returns the result of the mul, or null if this operation is not defined. *o* - The object to perform this binary operation with (the right-hand operand).
__add__(genericOther)	Returns the result of the add, or null if this operation is not defined. *genericOther* - The object to perform this binary operation with (the right-hand operand).
toString()	Returns the string representation of the list.
append(o)	Adds a single element to the end of list. *o* - The element to add.
count(o)	Returns the number elements in the list that equals the argument. *o* - the argument to test for. Testing is done with the == operator.
index(o)	Returns the smallest index where an element in the list equals the argument. *o* - the argument to test for. Testing is done with the == operator.
insert(idx, o)	Inserts the argument element into the list at the specified index. idx - The position where the element will be inserted. *o* - The element to insert.
remove(o)	Removes the first occurrence of the argument from the list. The elements are compared with the == operator. *o* - The element to search for and remove.

Archived

Method	Description
reverse()	Reverses the items of s in place. The reverse() methods modifies the list in place for economy of space when reversing a large list. It does not return the reversed list to remind you of this side effect.
pop()	Removes and returns the last element in the list.
pop(n)	Removes and returns the n indexed element in the list. *n* - The index of the element to remove and return.
extend(o)	Appends the elements in the argument sequence to the end of the list. *o* - The sequence of items to append to the list.
__iadd__(o)	Returns the result of the add, or null if this operation is not defined. *o* - The object to perform this binary operation with (the right-hand operand).
sort(compare)	Sorts the items of the list in place. The compare argument is a function of two arguments (list items) that should return -1, 0, or 1, depending on whether the first argument is considered smaller than, equal to, or larger than the second argument. Note that this slows the sorting process down considerably; for example, to sort a list in reverse order, it is much faster to use calls to the methods sort() and reverse() than to use the built-in function sort() with a comparison function that reverses the ordering of the elements. *compare* - The comparison function.
sort()	Sorts the items of the list in place. Items is compared with the normal relative comparison operators.

Archived

Numeric methods

The methods in Table D-3 can be defined to emulate numeric objects. Methods corresponding to operations that are not supported by the particular kind of number implemented (for example, bitwise operations for non-integral numbers) should be left undefined.

Table D-3 Numeric methods

Method	Description
__add__(self, other)	These methods are called to implement the binary arithmetic operations (+, -, *, //, %, divmod(), pow(), **, <<, >>, &, ^, |). For example, to evaluate the expression x+y, where x is an instance of a class that has an __add__() method, x.__add__(y) is called. The __divmod__() method should be the equivalent to using __floordiv__() and __mod__(); it should not be related to __truediv__() (described below). Note that __pow__() should be defined to accept an optional third argument if the ternary version of the built-in pow() function is to be supported.
__sub__(self, other)	
__mul__(self, other)	
__floordiv__(self, other)	
__mod__(self, other)	
__divmod__(self, other)	
__pow__(self, other[, modulo])	
__lshift__(self, other)	
__rshift__(self, other)	
__and__(self, other)	
__xor__(self, other)	
__or__(self, other)	
__div__(self, other)	The division operator (/) is implemented by these methods. The __truediv__() method is used when __future__.division is in effect; otherwise, __div__() is used. If only one of these two methods is defined, the object will not support division in the alternate context; TypeError will be raised instead.
__truediv__(self, other)	

Archived

Method	Description
__radd__(self, other)	These methods are called to implement the binary arithmetic operations (+, -, *, /, %, divmod(), pow(), **, <<, >>, &, ^, |) with reflected (swapped) operands. These functions are only called if the left operand does not support the corresponding operation. For example, to evaluate the expression x-y, where y is an instance of a class that has an __rsub__() method, y.__rsub__(x) is called. Note that ternary pow() will not try calling __rpow__() (the coercion rules would become too complicated).
__rsub__(self, other)	
__rmul__(self, other)	
__rdiv__(self, other)	
__rtruediv__(self, other)	
__rfloordiv__(self, other)	
__rmod__(self, other)	
__rdivmod__(self, other)	
__rpow__(self, other)	
__rlshift__(self, other)	
__rrshift__(self, other)	
__rand__(self, other)	
__rxor__(self, other)	
__ror__(self, other)	
__iadd__(self, other)	These methods are called to implement the augmented arithmetic operations (+=, -=, *=, /=, %=, **=, <<=, >>=, &=, ^=, |=). These methods should attempt to do the operation in-place (modifying self) and return the result (which could be, but does not have to be, self). If a specific method is not defined, the augmented operation falls back to the normal methods. For example, to evaluate the expression x+=y, where x is an instance of a class that has an __iadd__() method, x.__iadd__(y) is called. If x is an instance of a class that does not define a __iadd__() method, x.__add__(y) and y.__radd__(x) are considered, as with the evaluation of x+y.
__isub__(self, other)	
__imul__(self, other)	
__idiv__(self, other)	
__itruediv__(self, other)	
__ifloordiv__(self, other)	
__imod__(self, other)	
__ipow__(self, other[, modulo])	
__ilshift__(self, other)	
__irshift__(self, other)	
__iand__(self, other)	
__ixor__(self, other)	
__ior__(self, other)	

Archived

Method	Description
__neg__(self)	Called to implement the unary arithmetic operations (-, +, abs() and ~).
__pos__(self)	
__abs__(self)	
__invert__(self)	
__complex__(self)	Called to implement the built-in functions complex(), int(), long(), and float(). Should return a value of the appropriate type.
__int__(self)	
__long__(self)	
__float__(self)	
__oct__(self)	Called to implement the built-in functions oct() and hex(). Should return a string value.
__hex__(self)	
__coerce__(self, other)	Called to implement “mixed-mode” numeric arithmetic. Should either return a 2-tuple containing self and other converted to a common numeric type, or None if conversion is impossible. When the common type would be the type of other, it is sufficient to return None, since the interpreter will also ask the other object to attempt a coercion (but sometimes, if the implementation of the other type cannot be changed, it is useful to do the conversion to the other type here). A return value of NotImplemented is equivalent to returning None.

Archived

Archived

E

Developing Java plug-ins

This appendix contains a walk-through on how to develop your own Java plug-ins for IBM Tivoli Intelligent Orchestrator and IBM Tivoli Provisioning Manager.

You should be advised that with the introduction of TIO/TPM V3 the use of Java plug-ins have been replaced by native Java programs, and that you are advised to use Java programs rather than plug-ins when developing new workflows and automation packages.

The following instructions are provided in order to assist you in maintaining existing Java plug-ins.

© Copyright IBM Corp. 2004, 2006. All rights reserved.

Java plug-in development

To provide specialized functions to the workflows in IBM Tivoli Intelligent Orchestrator, Java plug-ins may be developed.

A Java plug-in for IBM Tivoli Intelligent Orchestrator is a Java class file that runs in the context of TIO/TPM. In other words, TIO/TPM provides a container in which the plug-in executes. This is similar to a Java servlet, which runs inside a servlet container, or an applet, which runs inside a Web browser.

Because the plug-in runs inside the TIO/TPM container, the developer must remember two things:

- TIO/TPM as a container provides services to the plug-in, such as providing input values and reading output values.
- TIO/TPM constrains the plug-in, requiring the plug-in to implement certain functions and restricting how it operates.

The following section describes the process we followed for developing a custom Java plug-in. IBM Tivoli Intelligent Orchestrator comes with a standard toolset of plug-ins for string manipulation, one of which is called *String Concatenation*. The String Concatenation plug-in concatenates two strings. We found ourselves repeatedly calling this plug-in for building file paths and commands. We needed a simple way to string together three or four strings and to specify a separator.

For example, given a drive letter “c:”, a home directory of “home”, a path name of “thinkdynamic”, and a file name of “setup.exe”, we want to construct the common c:\home\thinkdynamic\setup.exe. Doing this with the two string functionality of String Concatenation is cumbersome. Our solution is a custom Java plug-in called *String concatenation (multi)*, which allows up to four strings and allows us to specify a separator.

The steps needed to implement a plug-in may be summarized as follows:

1. Install the Java Software Development Kit.
2. Copy the TIO/TPM packages to the development directory.
3. Write the Java source code.
4. Compile the source code.
5. Compress the class file.
6. Add the jar file to the deployment engine classpath.
7. Create and load the Java plug-in XML file.

Install the Java Software Development Kit

The Java Software Development Kit provides libraries and programs for compiling Java source files (with a .java extension) into Java class files (with a .class extension) and then compacting them into a compressed file format called a jar file (with a .jar extension).

The Software Development Kit is referred to in shorthand as the *SDK*, or sometimes mistakenly as *JDK* (for Java Development Kit). There are three separate SDKs: J2SE, J2EE, and J2ME, which are used for developing standard Java components, enterprise Java components, and micro-device Java components. Be sure to download the J2SE SDK. At the time of the writing of this IBM Redbook, the current version for the J2SE SDK is 1.4.2_04.

The download page will offer both the SDK and the *Java Runtime Environment* (JRE), which provides files needed to run Java applications. Be sure to select the J2SE SDK, not the JRE.

You may see a reference to the Java 2 SDK, SE v1.4.2_04; the term Java 2 is often confusing to people. Java 2 is a marketing term used by Sun. The important factor is SE, which is a shorthand for J2SE and v1.4.2_04, which means Java 1.4.2 with a patch level of 4.

The download is available at `http://java.sun.com`. Assuming you wish to download the J2SE SDK for Windows, you will be prompted to save the file j2sdk-1_4_2_04-windows-i586-p.exe. Save it to your temp directory and then launch the download.

Archived

When launched, the wizard will prompt you for an installation directory (see Figure E-1). We chose c:\java\j2sdk1.4.2_04. We also selected all options.

Figure E-1 Java SDK Installation Wizard

After installation is finished, you will find a new directory created in c:\java\sdk1.4.2_04.

Copy the TIO/TPM packages to the development directory

IBM Tivoli Intelligent Orchestrator is commonly installed into the home directory of the tioadmin user ID TC_HOME. On Windows, the %TC_HOME% directory is c:\cygwin\home\thinkcontrol. Within this directory, you will find a c:\cygwin\home\thinkcontrol\lib directory, which contains many files with .jar and .sar extensions. It is the .jar files in this directory that are needed to compile your Java plug-ins. To use them, you must copy them to your development computer.

You can put them wherever you wish on your local computer. Common choices are to copy them to:

- c:\cygwin\home\thinkcontrol\lib
- c:\java\thinkcontrol\lib

For our example, we chose the first option, which allows us to work with the same directory structure on our development computer as we have on our TIO/TPM server. This provides the benefit of being able to repeat the development process on the TIO/TPM server with no modification to the control or set up files.

Write the Java source code

To implement a Java plug-in, you must code at least one .java source file, which will be compiled and packaged. To provide more functionality, additional Java code may be included in the package as needed.

In the following section, we begin by showing the source code for a Java plug-in. Then, we then point out some things to note in the code and show where to find more information.

The Java source file

Java source files may be edited with any text editor, such as Notepad or vi. The only requirement is that they be saved in a text file format with a .java extension.

Example E-1 shows our source file for the ConcatenateMulti Java plug-in. Since we installed the Java SDK into the c:\java directory, we chose to create a c:\java\itito directory to contain our Java plug-in development files. So our source file is placed in c:\java\itito\concatenatemulti.java.

Example: E-1 Java plug-in source file

```
/*****************************************************************
 * (C) Copyright IBM Corp.  2004
 *****************************************************************
 */
package com.ibm.itso.javaplugins;

import com.thinkdynamics.kanaha.de.DeploymentException;
import com.thinkdynamics.kanaha.de.ParameterStack;
import com.thinkdynamics.kanaha.de.javaplugin.CommandDriver;

/**
 * Title:        ConcatenateMulti<br>
 * Description:  Concatenates multiple strings<br>
 *
 *  ------------- DRIVER METADATA START ------------------------
 *  @thinkcontrol:driver
 * driverTypeId="45"
 * name="String Concatenation (multi)"
 * undoable="true"
 * category="2"
 * description="Concatenates multiple strings"
 *
 *  ------------- DRIVER METADATA END ------------------------
 */
```

Archived

```
public class ConcatenateMulti extends CommandDriver {

   // IBM Copyright
   public static final String IBM_COPYRIGHT =
      com.thinkdynamics.util.Constants.COPYRIGHT;

   /** ----DRIVER ATTRIBUTE METADATA -----
   * @thinkcontrol:driver.attribute
   *  name="String 1"
   *  description="First string (manditory)"
   *  required="true"
   *  published="false"
   */
   public static final String STRING1_PROPNAME = "String 1";

   /** ----DRIVER ATTRIBUTE METADATA -----
   * @thinkcontrol:driver.attribute
   *  name="String 2"
   *  description="Second string (leave blank if not used)"
   *  required="true"
   *  published="false"
   */
   public static final String STRING2_PROPNAME = "String 2";

   /** ----DRIVER ATTRIBUTE METADATA -----
   * @thinkcontrol:driver.attribute
   *  name="String 3"
   *  description="Third string (leave blank if not used)"
   *  required="true"
   *  published="false"
   */
   public static final String STRING3_PROPNAME = "String 3";

   /** ----DRIVER ATTRIBUTE METADATA -----
   * @thinkcontrol:driver.attribute
   *  name="String 4"
   *  description="Forth string (leave blank if not used)"
   *  required="true"
   *  published="false"
   */
   public static final String STRING4_PROPNAME = "String 4";

   /** ----DRIVER ATTRIBUTE METADATA -----
   * @thinkcontrol:driver.attribute
   *  name="Result"
```

Archived

```
*  description="Result"
*  required="false"
*  published="true"
*/
public static final String RESULT_PROPNAME = "Result";

/** ----DRIVER ATTRIBUTE METADATA -----
 * @thinkcontrol:driver.attribute
 * name="Separator"
 * description="Separator character (leave blank for no separator)"
 * required="true"
 * published="false"
 */
public static final String SEPARATOR_PROPNAME = "Separator";

public ConcatenateMulti() throws DeploymentException {
}

public void doIt(ParameterStack runtimeParameters)
   throws DeploymentException {

   String string1 =
      runtimeParameters.getVariableNewValue(STRING1_PROPNAME);
   String string2 =
      runtimeParameters.getVariableNewValue(STRING2_PROPNAME);
   String string3 =
      runtimeParameters.getVariableNewValue(STRING3_PROPNAME);
   String string4 =
      runtimeParameters.getVariableNewValue(STRING4_PROPNAME);
   String separator =
      runtimeParameters.getVariableNewValue(SEPARATOR_PROPNAME);
   String noValue = "(no value)";

   StringBuffer separatorBuffer = new StringBuffer();
   if (separator == null || separator.equals(noValue)){
      separatorBuffer.append((Object)"");}
      else{
         separatorBuffer.append((Object)separator);}

   StringBuffer resultBuffer = new StringBuffer();

   if (string1 != null) {
      resultBuffer.append((Object)string1);
   }
   if (string2 != null && !string2.equals(noValue)) {
```

```
            resultBuffer.append((Object)separatorBuffer);
            resultBuffer.append((Object)string2);
         }
         if (string3 != null && !string3.equals(noValue)) {
            resultBuffer.append((Object)separatorBuffer);
            resultBuffer.append((Object)string3);
         }
         if (string4 != null && !string4.equals(noValue)) {
            resultBuffer.append((Object)separatorBuffer);
            resultBuffer.append((Object)string4);
         }
         runtimeParameters.setVariableNewValue(
            RESULT_PROPNAME,
            resultBuffer.toString());
      }

   }
```

TIO/TPM Application Programming Interface documentation

Assuming you installed TIO/TPM in the c:\cygwin directory, your Application Programming Interface (API) documentation home page, shown in Figure E-2 on page 603, will be found in c:\cygwin\home\thinkcontrol\javadoc\index.html. Open this file with your Web browser to view the documentation.

Archived

Figure E-2 TIO/TPM Java API documentation home page

From the left side navigator, you may scroll and select the detailed documentation for the class you are interested in. Note that depending on what your Java plug-in needs to perform, you must import different TIO/TPM classes into your source code. The CommandDriver class may be particularly interesting, since it must be implemented by all Java plug-ins. An example of the method summary for the CommandDriver class is shown in Figure E-3 on page 604.

Archived

Constructor Summary

	CommandDriver()

Method Summary

abstract void	doIt(ParameterStack runtimeParameters)
protected java.lang.String	getDefaultLocalTempDirectory()
protected java.lang.String	getNonEmptyString(ParameterStack runtimeParameters, java.lang.String variableName)
static boolean	isValidString(java.lang.String str) Verify that the string is not null or empty.
protected java.lang.String	replaceVariableString(ParameterStack runtimeParameters, java.lang.String oldStr, java.lang.String delimiter) Treat the string between the specified delimiters as a variable and replace it with the values found from the runtimeParameters
static void	setDelimiter(java.lang.String delimiter)
protected boolean	stringToBoolean(java.lang.String str)
protected int	stringToInt(java.lang.String str)

Figure E-3 TIO/TPM API Interface CommandDriver method documentation

Further information on how to code a Java plug-in is found in this documentation.

Before we continue, note that the Java source code has a method with the following signature:

```
public void doIt(ParameterStack runtimeParameters)
```

You should understand that it is this method that is called by TIO/TPM when the Java plug-in is executed. Note further the parameters of this method and how they are used in the source file to read an input variable:

```
String string1 =
runtimeParameters.getVariableNewValue(STRING1_PROPNAME);
```

and how they are used to write to an output variable:

```
runtimeParameters.setVariableNewValue(RESULT_PROPNAME,resultBuffer.toSt
ring());
```

Archived

The three code lines above illustrate what was said at the start of this section:

- TIO/TPM as a container provides services to the plug-in, such as providing input values and reading output values (the getVariableNewValue and the setVariableNewValue methods).
- TIO/TPM constrains the plug-in, requiring the plug-in to implement certain functions and restricting how it operates (the *doIt* method)

Compile the source code

Java source files are compiled with the javac.exe program, which is located in c:\java\j2sdk1.4.2_04\bin directory, assuming you installed the SDK into that directory.

The javac.exe program requires certain parameters that are lengthy to type from the command line. To make things easier, we created a script (a .bat file in windows) called ConcatenateMulti.bat and placed it in the c:\java\itito directory. The content of this script is shown in Example E-2.

Example: E-2 ConcatenateMulti.bat compilation file

```
c:\java\j2sdk1.4.2_03\bin\javac ^
-classpath ^
c:\cygwin\home\thinkcontrol\lib\de.jar;^
c:\cygwin\home\thinkcontrol\lib\datacentermodel.jar;^
c:\cygwin\home\thinkcontrol\lib\ejbspublic.jar ^
-d c:\java\itito ^
ConcatenateMulti.java
```

To compile the Java plug-in source, do the following:

- Ensure that the source file (concatenatemulti.bat) resides in the c:\java\itito directory.
- Open a command line and change to the c:\java\itito directory.
- Enter `concatenatemulti.bat` and press Enter to run the .bat file.

The first line, `c:\java\j2sdk1.4.2_04\bin\javac`, calls the javac compiler. The last line, `ConcatenateMulti.java`, tells the compiler what source file to compile, which is the concatenatemulti.java file, located in the same directory as the .bat file. The next to last line, `-d c:\java\itito`, tells the compiler where to put our compiled .class file. Finally, the lines beginning with `-classpath` and continuing with a list of .jar files tells the compiler where to find the TIO/TPM packages of interest for this Java plug-in.

Look at the source file in Example E-1 on page 599 and note the third line:

```
package com.ibm.itso.javaplugins;
```

This line tells the compiler what directory structure the compiled .class file should be placed into. Since the .bat file said to place the file into c:\java\itito and the package statement says to use com.ibm.itso.javaplugins, the compiled .class file will be found in a location that is the concatenation of the two:

```
c:\java\itito\com\ibm\itso\javaplugins\concatenatemulti.class
```

To verify that the compilation of the Java plug-in was successful, make sure this directory structure and .class file has been created.

Compress the class file

The Java SDK comes with another program called jar.exe, which is used to compress the newly compiled .class file into a compressed file with a .jar extension. To run the program, do the following:

- Go back to the command line.
- Make sure the active directory is c:\java\itito.
- Execute the following command:

  ```
  c:\java\j2sdk1.4.2_03\bin\jar cf itsoutil.jar com
  ```

  You might notice that the format for the **jar** command is similar to the commonly used **tar** command. The `cf itsoutil` portion tells jar to create a file called itosutil.jar and the `com` portion says to look into the c:\java\itito\com directory structure for our .class file

This results in a new file being created: c:\java\itito\itsoutil.jar.

Make sure this file is created after running the **jar** command.

The jar file is one of two files you need to provide to IBM Tivoli Intelligent Orchestrator for implementing a Java plug-in. The other file is an XML file that is used to inform TIO/TPM about the new plug-in. To make the new plug-in known to TIO/TPM environment, you have to:

1. Add the jar file to the deployment engine classpath.
2. Create and load the Java plug-in XML file.

Add the jar file to the deployment engine classpath

Assuming IBM Tivoli Intelligent Orchestrator was installed in the c:\cygwin\home\thinkcontrol directory, TIO/TPM will look for new Java plug-in classes in the c:\cygwin\home\thinkcontrol\drivers\lib directory.

To add the itsoutil.jar file to the TIO/TPM class path, do the following:

- Copy itsoutil.jar into c:\cygwin\home\thinkcontrol\drivers\lib.
- Stop and restart the TIO/TPM server.

At this point, the new Java plug-in is available to TIO/TPM, but TIO/TPM is not yet aware of it, because the Java plug-in has not yet been defined to TIO/TPM.

Create and load the Java plug-in XML file

The TIO/TPM Data Center Model is informed about the new Java plug-in via an XML file. We will show you the file and how to use it for loading the plug-in into the Data Center Model. Finally, we will explain some of the elements and attributes of the file.

Creating the Java plug-in XML file

Example E-3 shows the XML file used to load the new Java plug-in into the Data Center Model.

Example: E-3 Java plug-in import XML file

```
<?xml version="1.0" encoding="UTF-8" ?>
<!--
 DO NOT EDIT ! THIS FILE IS GENERATED BASED ON XDOCLET TAGS FROM DRIVER JAVA SOURCE FILE.
 -->
<com.thinkdynamics.kanaha.de.command.DriverType>
<driverTypeDeId>ibm.com.itsoutil</driverTypeDeId>
  <description>Concatenates multiple strings</description>
  <driverCategoryDeId>2</driverCategoryDeId>
  <driverJavaClassName>com.ibm.itso.javaplugins.ConcatenateMulti</driverJavaClassName>
  <name>String concatenation (multi)</name>
  <undoableB>true</undoableB>
-
  <driverVariableDescriptors>-
    <com.thinkdynamics.kanaha.de.command.DriverVariableDescriptor>
        <driverTypeDeId>ibm.com.itsoutil</driverTypeDeId>
        <publishedB>false</publishedB>
        <requiredB>true</requiredB>
        <variableName>String 1</variableName>
        <description>First string (mandatory)</description>
    </com.thinkdynamics.kanaha.de.command.DriverVariableDescriptor>
    -
    <com.thinkdynamics.kanaha.de.command.DriverVariableDescriptor>
        <driverTypeDeId>ibm.com.itsoutil</driverTypeDeId>
        <publishedB>false</publishedB>
        <requiredB>true</requiredB>
        <variableName>String 2</variableName>
        <defaultValue>(no value)</defaultValue>
        <description>Second string (optional)</description>
    </com.thinkdynamics.kanaha.de.command.DriverVariableDescriptor>
    -
    <com.thinkdynamics.kanaha.de.command.DriverVariableDescriptor>
        <driverTypeDeId>ibm.com.itsoutil</driverTypeDeId>
        <publishedB>false</publishedB>
        <requiredB>true</requiredB>
        <variableName>String 3</variableName>
        <defaultValue>(no value)</defaultValue>
```

```
        <description>Third string (optional)</description>
    </com.thinkdynamics.kanaha.de.command.DriverVariableDescriptor>
    -
    <com.thinkdynamics.kanaha.de.command.DriverVariableDescriptor>
        <driverTypeDeId>ibm.com.itsoutil</driverTypeDeId>
        <publishedB>false</publishedB>
        <requiredB>true</requiredB>
        <defaultValue>(no value)</defaultValue>
        <variableName>String 4</variableName>
        <description>Fourth string (optional)</description>
    </com.thinkdynamics.kanaha.de.command.DriverVariableDescriptor>
    -
    <com.thinkdynamics.kanaha.de.command.DriverVariableDescriptor>
        <driverTypeDeId>ibm.com.itsoutil</driverTypeDeId>
        <publishedB>true</publishedB>
        <requiredB>false</requiredB>
        <variableName>Result</variableName>
        <description>Result</description>
    </com.thinkdynamics.kanaha.de.command.DriverVariableDescriptor>
    -
    <com.thinkdynamics.kanaha.de.command.DriverVariableDescriptor>
        <driverTypeDeId>ibm.com.itsoutil</driverTypeDeId>
        <publishedB>false</publishedB>
        <requiredB>true</requiredB>
        <variableName>Separator</variableName>
        <defaultValue>(no value)</defaultValue>
        <description>Separator character (leave blank for no separator)</description>
    </com.thinkdynamics.kanaha.de.command.DriverVariableDescriptor>
  </driverVariableDescriptors>
</com.thinkdynamics.kanaha.de.command.DriverType>
```

XML file elements and attributes

The following highlights the elements and attributes that you should concentrate on when developing your XML file. Many of the XML elements are obvious in their use. We just focus on those that may be a bit confusing or misleading.

The <driverTypeDeId> element

This element uniquely identifies the plug-in to the DCM. It is found only once in the XML file. The value in this element may be any unique identifier. If you browse the TIO/TPM Data Center Model database, you will find this information in a table named DRIVER_TYPE_DE. There you will see that various schemes have been used to generate unique values. The best method is to use a common domain name space method, such as *com.ibm.itsoutil*. This is the value we inserted.

The <undoableB> element

This value is also found only once in the file and its value must be either `true` or `false`. It indicates whether the Java class has a method named undoIt(), which is called when the transition fails. However, this method is depreciated, according to the TIO/TPM Application Programming Interface documentation found in cygwin\home\thinkcontrol\javadoc.

Archived

The <requiredB> element

This element is repeated for each required input and output variable. Its value must be `true` or `false`. If `true`, the variable is defined as an input variable. If `false`, it is not an input variable.

The <publishedB> element

Another element with a `true` or `false` value that is repeated for each variable and which indicates whether the variable is an output variable. If `true`, it is included as an output variable. If `false`, it is not included as an output variable.

The <defaultValue> element

This element is used when defining an input variable. If a value is provided, it will appear as the default value when the plug-in is dropped onto a workflow.

We encountered a problem in our example. Our Java plug-in has an input variable named separator. If the value is not blank, then it is used to separate the strings we are concatenating. But what if we do not want a separator? How can we show a blank string value? It is easy enough in XML to provide an empty value for a variable, as shown:

```
<defaultValue></defaultValue>
```

But when we put this in our XML file and then tried to use the Java plug-in, TIO/TPM complained that we had failed to provide a default value. In other words, the current version of TIO/TPM does not recognize an empty string as a value. So our workaround was to define the element as:

```
<defaultValue>(no value)</defaultValue>
```

We then modified the Java code so that whenever it encounters a string with the value of (no value), it treats it as if it were a *blank* string.

Automated XML file generation using the XDoclet engine

You might have noticed that our Java source file has comments that seem to relate to the XML elements in our XML file. For example, the comment shown in Example E-4 seems to describe an input variable.

Example: E-4 Java plug-in source file comment for XDoclet engine

```
/** ----DRIVER ATTRIBUTE METADATA -----
   * @thinkcontrol:driver.attribute
   *  name="String 1"
   *  description="First string (mandatory)"
   *  required="true"
   *  published="false"
   */
```

```
public static final String STRING1_PROPNAME = “String 1”;
```

and the comments seem to relate to the section of the XML file shown in Example E-5.

Example: E-5 Java plug-in import XML file (partial)

```
<com.thinkdynamics.kanaha.de.command.DriverVariableDescriptor>
   <driverTypeDeId>ibm.com.itsoutil</driverTypeDeId>

   <publishedB>false</publishedB>

   <requiredB>true</requiredB>

   <variableName>String 1</variableName>

   <description>First string (manditory)</description>
</com.thinkdynamics.kanaha.de.command.DriverVariableDescriptor>
```

You may then ask if there is a way to set the values for the XML file elements in the source file and have an XML file automatically generated. There is a way to do this. The program used to do this is an open source program called XDoclet, which is found at http://xdoclet.sourceforge.net. XDoclet is described on this site as follows:

> XDoclet is a code generation engine. It enables *Attribute-Oriented Programming* for Java. In short, this means that you can add more significance to your code by adding meta data (attributes) to your Java sources. This is done in special JavaDoc tags.
>
> XDoclet will parse your source files and generate many artifacts, such as XML descriptors and source code from it. These files are generated from templates that use the information provided in the source code and its JavaDoc tags.

Setting up and using XDoclet can be complicated for the novice. If you are already using XDoclet and are developing a large number of Java plug-ins, then you will want to learn how to use XDoclet. If you are developing a small number of Java plug-ins, we recommend creating the XML file manually, as we have shown above.

If you chose to use XDoclet, a template XML file is found in the Java plug-in Toolkit.

Loading the XML file into the TIO/TPM Data Center Model

To import this XML file into IBM Tivoli Intelligent Orchestrator, open the TIO/TPM GUI and navigate to the **System Configuration and Workflow Management** tab. Select **Java plug-ins** and then go to the **Import** tab. The Java plug-in import dialog appears, as shown in Figure E-4.

Figure E-4 Importing a Java plug-in

- Select the **File** radio button.
- Enter the path to the XML file defining the Java plug-in, in our example, c:\javaitito\concatenatemulti.xml.
- Press the **Import** button.

A successful import will be provided with a confirmation box. If you then press F5 to refresh the screen and navigate to the list of Java plug-ins, you will notice that the concatenate multi-Java plug-in was imported, as shown in Figure E-5 on page 612.

Archived

Figure E-5 Imported Java plug-in String concatenation (multi)

Archived

F

Systems Management using the IT Infrastructure Library

In this appendix, we provide an overview of the IT Infrastructure Library (ITIL), followed by an example of how to use it. The purpose of this appendix is to show how to use this model and its processes while developing an automation package.

However, it is important for the ISV to determine the customer's processes and their effect on the operation of the customer's IT environment. Normally, a customer will never ever accept an application and the belonging workflows if they are not compliant to the customer's processes and guidelines.

© Copyright IBM Corp. 2004, 2006. All rights reserved.

Overview

The IT Infrastructure Library is an internationally recognized framework of best practices for IT management. Its origin goes back to the late 1960s when the UK Government initiated an exercise to standardize its diverse IT processes with a focus on efficiency, effectiveness, total quality management, and risk reduction. This exercise produced the first version of the library, which was consolidated in 1999 into two IT Service Management books:

- *Service Support*
- *Service Delivery*

Note: The IT Infrastructure Library is not a standard, although many people refer to it as such. It is a framework of guidelines to which an organization can conform or an individual can become accredited or certified with. The IT Infrastructure Library has been adopted as the basis of the British Standard in IT Service Management, BS 15000, which can be used internationally to seek certification of organizational compliance.

Figure F-1 gives an overview of the capabilities of the IT Infrastructure Library.

Figure F-1 Overview of the IT Infrastructure Library (ITIL)

The original IT Service Management Books have since been supplemented by others, including:

- *Planning to Implement Service Management*
- *Security Management*
- *IT Infrastructure Management*
- *Application Management*

The number of publications continues to grow and all are publicly available. This ready availability from the Office of Government Commerce (OGC), combined with reasonably inexpensive costs for the books themselves, has led to the widespread adoption of the IT Infrastructure Library throughout the UK private sector, into Europe, and across the globe. The best practices contained in the IT Infrastructure Library are independent of tool, vendor, or industry and can be adapted to an organization of any size.

Note: The IT Infrastructure Library is non-proprietary, but it is not free. The Office of Government Commerce owns the intellectual property rights and the copyright.

However, although sorted by discipline and defined to a significant level of detail, the books are not themselves the solution to all IT management issues; the processes require significant work to deploy at an operational level enabling day-to-day use, with dependencies on the three key process-people-tools components of a management system.

For more detailed documentation about the IT Infrastructure Library, please refer to:

http://www.itil.co.uk

The IT Infrastructure Library provides the foundation for quality IT Service Management. The widespread adoption of the ITIL guidance has encouraged organizations worldwide, both commercial and non-proprietary, to develop supporting products as part of a shared “ITIL Philosophy”.

However, from the point of view of the IT Infrastructure Library, we have two different main parts concerning infrastructure management, as shown in Figure F-1 on page 614. They are known as:

- Business perspective
- Technical perspective

Archived

The parts of each of these perspectives are shown in Figure F-2; the business perspective in the top half and the technical perspective in the bottom half.

Figure F-2 Systems management on a High Level View (ITIL)

We give a short overview of each of these major components of the IT Infrastructure Library in the following sections.

Configuration Management

Every enterprise depends on the economical provision of IT services. From the point of view of legal and financial aspects alone, operation and administration of IT assets is becoming ever more problematic. Configuration Management now makes it possible for IT management to be in control of these IT elements and assets (for example, hardware, software, documentation, licences, and so on). Configuration Management also provides fundamental information for calculating costs and invoicing work performed within Service Level Management.

Archived

Business perspective

The next section describes the disciplines of the business perspective, also known as *service delivery*.

Service Level Management

As a major function of the IT services management, Service Level Management is responsible for qualitative and quantitative control of the operating of the IT environment. The quality and quantity of the services are constituted in the service level agreement (SLA) between the IT department and its customers. This includes both qualitative and quantitative specifications, such as performance and availability of these services. It is essential that in assessing service quality, Service Level Management takes into account all factors of service management. Clear and intelligibly defined SLAs, which can be measured in practical application, are an important prerequisite for the assessment and implementation of most activities within the IT organization.

Cost Management

Cost Management is responsible for the identification, calculation, monitoring, and onward allocation of costs for the customer's contracted IT services. By creating cost awareness, financial management can influence the behavior of users and customers alike, determining that the true costs for providing IT services serves as the basis for Cost Management. Cost Management also provides IT management with the basis for budget planning.

Business Continuity

Companies significantly depend on the availability and functionality of the used IT environment. Based on this fact, preparation for most eventualities in combination with Business Continuity management assumes ever greater importance. You want to be sure that you are safeguarding the availability of services, taking preventative measures to reduce the probability of failures, and, if a catastrophic event should occur, restoring services in the required time.

Capacity Management

Capacity Management is responsible for the availability of the quantitative and economic capacities of IT services, especially in regards to:

- Transaction volume
- Processing time
- Response time

The business requirements of IT resources are determined by the Capacity Management. Capacity Management also forecasts the workloads necessary to provide them and carries out planning of the IT resources.

Availability Management

Availability Management provides for reliable access to IT services. Availability means that the customer will always receive the expected services when they are needed. Good availability requires a low error or failure rate. If there is an incident or malfunction, this issue has to be solved quickly. Furthermore, Availability Management ensures that the maximum benefit is gained from the existing IT infrastructure and services. Such a maximum benefit is ensured by reliability of the services, and the ability to service and maintain the IT infrastructure.

Technical perspective

The next section describes the disciplines of the technical perspective, also known as *service support*.

Service Desk

The Service Desk is the central point of contact for the users and the IT department in all matters concerning IT services. Service Desk includes the following disciplines:

- Help desk
- Coordination of change requests
- Service Level Management
- Configuration Management
- All other service management processes

However, the Service Desk is not a process defined by the IT Infrastructure Library, but a function inside of the service organization. To be the first point of contact in its special role, the Service Desk is highly important, because it embodies the image and the quality of service of the IT organization.

Incident Management

Incident Management involves processing enquiries and incidents of all types. This is achieved by a group of specialists who work in virtual unison. In line with the specialist skill levels of their members, these teams are grouped into first-level, second-level, and third-level support units. In this function, Incident Management assumes the particular role of maintaining contact between IT organizations and business.

Problem Management

Problem Management includes the handling of all failures of IT services from the special aspect of identifying the actual underlying cause. This includes recommending changes concerning configuration items to Change Management. Problem Management processes use information provided by

Archived

various other processes (for example, by Incident Management or by Change Management). Moreover, Problem Management takes a proactive approach in which weaknesses are detected early and preventive measures are taken.

Change Management

Experience shows that a high proportion of problems with IT service quality can be traced back to some change. Changes to the IT infrastructure are likely to result in problems. Such problems are enormously expensive, so enterprises and customers are increasingly reluctant to accept them. To be sure, every development of the IT infrastructure, whether relating to Capacity Management or Problem Management, is associated with change measures that in turn pose a degree of risk.

Release Management

The term *release* refers to one or several authorized IT services change measures. Dependencies between a particular software version and the hardware required to run it determine the bundling of software and hardware changes, which, together with other functional requirements, constitute a new release. Release Management ensures successful planning and control of hardware and software installations. The focus is on protection of the productive environment and its services by using formal procedures and checks. Release Management is carried out based on the Configuration Management database so as to ensure that the IT infrastructure is up to date.

Security Management

In addition to the technical and business perspective of systems management, Security Management is another focal point. We prefer to work properly on the customer' security processes when developing an automation package.

Security is a moving target and almost invariably difficult to plan and calculate, because changes often originate from the outside, in particular when understanding a necessary technical innovation. However, the integrity as well as the protection of customer and business data and of IT resources is crucial to the survival of an enterprise. Corporate management must therefore determine and document the security policy of the enterprise or the business sectors to be secured, set objectives, and specify commitments for IT security. Corporate management must further ensure that such objectives and commitments are understood at all levels and are implemented and maintained. In particular, the following must be emphasized:

- Clear determination of security objectives
- Complete definition of applications
- Security processes
- Maintenance of the security process

This should be done by using the disciplines below:

- Identity Management
- Access Management
- Risk Management
- Network Security

For more detailed information about the IT Infrastructure Library, please refer to the related documentation at:

http://www.itil.co.uk

Example of using the IT Infrastructure Library

Here you will find an example of an architectural workflow to show the processes based on the IT Infrastructure Library. For this example, we are using a workflow called Cluster.Addserver.

> **Note:** The workflow Cluster.AddServer is for use as an example and it is independent from the logical operation Cluster.AddServer provided by IBM Tivoli Intelligent Orchestrator.

The workflow Cluster.AddServer inherits a process called *Execute Change Request* with its subprocesses. The content of this main process is to work on:

- Acquisition of hardware
- Installation of the operating system and application
- Customization of the application environment
- If needed, a fallback scenario

This process has to be initiated after the final approval for adding a new server to the application cluster is available.

Figure F-3 on page 621 shows a graphical view about our example workflow *Cluster.AddServer*.

Archived

Figure F-3 Architectural Workflow Cluster.AddServer

The architectural workflow Cluster.AddServer contains several subprocesses:

Acquire Hardware This subprocess initiates the acquisition of the needed hardware either to buy new hardware or to reuse existing hardware.

Bare Metal Installation
During this subprocess, the operating system will be installed on the newly acquired hardware.

Software Installation During this subprocess, the application depending on the cluster will be installed on the new acquired hardware.

Customization of Components
This subprocess initiates the customization of the components belonging to the local application server.

Customization of related Components

As result of this subprocess, all components of the application environment should have been customized:

- Network
- Security
- Middleware
- Application(s)
- Environment
- Monitoring

Cut over

This subprocess integrates the new application server to the application cluster and sets its state to production.

Fallback Procedure

Every subprocess depends on its predecessor; even the subprocess Fallback Procedure should be initiated if an failure occurs during the execution of the process Execute Change Request. Also, that subprocess should have the ability to give back an error code to the main workflow Cluster.AddServer.

Table F-1 shows which disciplines are used by each subprocess.

Table F-1 IT Infrastructure Library disciplines used by Cluster.AddServer

Process/Subprocess	Discipline
Execute Change Request	Service Level Management Cost Management Business Continuity Capacity Management Availability Management Incident Management Problem Management Configuration Management Change Management Release Management
Acquire Hardware	Capacity Management Configuration Management Change Management Release Management
Bare Metal Installation	Configuration Management Change Management Release Management

Archived

Process/Subprocess	Discipline
Software Installation	Configuration Management Change Management Release Management
Customization of Components	Configuration Management Change Management Release Management
Customization of related Components	Configuration Management Change Management Release Management
Cut over	Service Level Management Business Continuity Capacity Management Availability Management Incident Management Problem Management Configuration Management Change Management Release Management
Fallback Procedure	Service Level Management Availability Management Incident Management Problem Management Change Management

Note: Please note that the information provided in Table F-1 should be considered an example only.

Archived

Archived

G

Additional material

This redbook refers to additional material that can be downloaded from the Internet as described below.

Locating the Web material

The Web material associated with this redbook is available in softcopy on the Internet from the IBM Redbooks Web server. Point your Web browser to:

`ftp://www.redbooks.ibm.com/SG246057`

Alternatively, you can go to the IBM Redbooks Web site at:

`ibm.com/redbooks`

Select the **Additional materials** and open the directory that corresponds with the redbook form number, SG246057.

© Copyright IBM Corp. 2004, 2006. All rights reserved.

Using the Web material

The additional Web material that accompanies this redbook includes the following files:

File name	*Description*
SG246057.zip	Zipped samples and templates

System requirements for downloading the Web material

The following system configuration is recommended:

Hard disk space: 10 MB minimum
Operating System: Windows
Processor: Any
Memory: Any

How to use the Web material

Create a subdirectory (folder) on your workstation, and unzip the contents of the Web material zip file into this folder.

In your directory, you will now find the following three zip files, which all are APDE tcdriver files that have been renamed for your convenience:

- ITSO_Utilities.zip
- Trade.zip
- Trade_Simulated.zip

To import these files in your APDE environment, open your APDE workspace, and complete the following steps for each file:

1. Create a new Automation Package Project, providing a name similar to the name of the zip file.

   For example, provide the name Trade for the project in which you will import the contents of the Trade.zip file.

2. Select **File** → **Import** from the APDE toolbar, select the Zip file as the import source, and press **Next**.

3. Now, do the following

   a. Provide the name and location of the zip file to import in the From zip file field.

   b. Supply the name of the newly created project for the Into folder field, for example, Trade.

   c. Check the **Overwrite existing resources without warning** field.

Archived

d. Press **Finish**.

Archived

Archived

Glossary

application A collection of software components associated with a customer account and used to perform a service or user-oriented work on a computer.

application controller A component of Tivoli Intelligent Orchestrator that determines the resource requirements of each application. It also identifies trends and peaks in resource use. Each managed application has an associated application controller.

application deployment template A combination of the logical deployment template and the network topology template.

application topology A model that defines the components of an application and how they interact, independent of their deployment. It can be used with a physical topology of a data center to determine the appropriate installation methodology.

approval status An indicator of whether or not a patch is approved for deployment.

autodiscovery A type of discovery where a program automatically detects the resources that were not previously known.

automatic mode An operating mode in which all deployment requests are automatically generated and approved. This operating mode is for applications or clusters that do not require human review and approval of deployment recommendations.

automation package A collection of commands, shell scripts, workflows, logical operations, and Java plug-ins that applies to the operation of a specific type of software component or a physical device. Also referred to as a tc driver.

breach probability A prediction, based on arrival rate, of the likelihood that an application or resource pool will exceed the maximum defined CPU utilization.

capacity on demand Automatic provisioning capability. Capacity on demand can be enabled for selected applications.

cluster A group of application servers that collaborates for the purposes of workload balancing and failover.

configuration drift discovery The identification of configuration changes to a data center device made outside of Tivoli Intelligent Orchestrator. For example, if the IP address of a server is manually changed at the server, the change is identified during configuration drift discovery.

customer A group or organization that is associated with one or more applications. A customer can be an external organization that accesses a data center, or an internal department within a company.

data acquisition engine A component of Tivoli Provisioning Manager that gathers and pre-processes performance metrics from each managed application.

data center asset A logical or physical resource in a data center. Examples include servers, switches, load balancers, software, VLANs, security policies, and service level agreements.

data center device See data center asset.

data center fragment A list of data center model objects required to support an application deployment template. It is a subset of the data center model.

© Copyright IBM Corp. 2004, 2006. All rights reserved.

data center model An abstract representation of all physical and logical assets in the data center that Tivoli Intelligent Orchestrator manages, and their relationships. The data center model tracks data center devices and associated allocations to applications.

data center object An abstract representation of a data center asset. Data center objects are used to create a data center model.

data mart A subset of a data warehouse that contains data specifically for the reporting needs of a department or team.

deployment The process of reconfiguring and reallocating resources in the data center. Deployment occurs in response to deployment requests, created manually by administrators or automatically by the system resource broker.

deployment engine A component of Tivoli Provisioning Manager that creates, stores, and runs workflows.

deployment plan A list of resources, configurations and implementations required to realize the deployment of an application.

deployment status The status of a deployment task. Deployment states indicate the state of completion of a task and whether any errors occurred when the task was running.

device driver Software that provides an interface between a specific device and the application program that uses the device. In Tivoli Intelligent Orchestrator, a device driver maps logical operations to the specific workflows required to perform changes to the device.

discovery The process of finding resources within an enterprise, including finding the new location of monitored resources that were moved.

File repository An internal or external storage location where software products and software patches are stored.

global operating mode The default operating mode that determines how deployment requests are created and approved for all managed applications. The global operating mode affects all applications that are configured to use the default operating mode.

global resource manager A component of Tivoli Intelligent Orchestrator that determines optimal resource allocation and maintains a stable application infrastructure.

image stack A software stack in an image. An image stack is assigned to a boot server.

infrastructure software module A software module that has the capabilities to meet the requirements of other software modules.

IT services management workflow See workflow.

license pool A collection of shared licenses available for a particular software installation.

locale A language and country specific code that determines formatting conventions such as collation, case conversion, character classification, the language of messages, date and time representation, and numeric representation. In Tivoli Intelligent Orchestrator, you can associate a locale with assets in your data center and run locale-specific workflows for those assets.

logical device A set of input or output operations for a physical or virtual device. For example, the Router logical device has logical device operations for creating a route and removing a route. Logical devices can be an abstraction of a physical device, such as a router, or virtual device, such as a software product.

logical device operation A generic instruction for a logical device that is independent of its implementation. For example, a logical device operation to add an IP address applies to all operating systems, but the steps for performing this instruction on each operating system are defined by device drivers.

Archived

maintenance mode A data center device state in which the device is temporarily removed from Tivoli Intelligent Orchestrator's control for reconfiguration.

manual mode An operating mode in which an administrator must review all deployment recommendations, and then manually create and approve deployment requests as required. This operating mode is for applications or clusters that require human review of deployment recommendations.

monitoring application An application that observes and records the activity of specific applications or systems. It typically monitors information such as available disk space or application errors and compares the information to defined thresholds. When thresholds are exceeded, the monitoring application can either notify an administrator or respond automatically based on predefined rules.

monitoring configuration A monitoring configuration is a set of monitoring options for a particular monitoring application. These options are defined in the monitoring application, and are referenced by Tivoli Intelligent Orchestrator to configure monitoring on devices.

operating mode A setting that determines the degree of automation and manual intervention required in creating and approving deployment requests. There are three modes: automatic, semi-automatic, and manual. An operating mode can be defined at the global, application, and cluster level.

orchestration The process of making real-time decisions about where and when to allocate resources to support business priorities and maintain service levels, based on information collected about the data center environment.

overbooked A resource reservation state that indicates that insufficient resources are available for the reservation.

policy A set of rules or services that are applied to managed resources.

provisioning The process of configuring servers, software, networks, and storage resources.

recommendation A deployment suggestion for an application cluster based on collected data and policies. Recommendations are made by the resource broker.

resource broker A component of Tivoli Intelligent Orchestrator that manages resource pools and attempts to fulfill resource requests made by application controllers.

resource pool A collection of available servers (servers that are not allocated) that support one or more application clusters. Also referred to as a spare pool.

resource reservation A request for an application or for extra servers that is scheduled for a specified period. Reservation requests are not guaranteed, but influence deployment decisions.

SAP See service access point.

script A list of commands embedded in a workflow that can be run without user interaction. Scripts are written in a standard scripting language such as Perl, JavaScript™, or bash.

semi-automatic mode An operating mode in which all deployment requests are automatically generated, and then manually reviewed and approved by an administrator.

service access point (SAP) The protocol and credentials associated with an asset for authentication of remote operations. A data center device can have more than one service access point.

service level agreement (SLA) An agreement or contract between a provider of services and a customer of those services, which sets expectations for the level of service with respect to availability, performance, and other measurable objectives. In the Tivoli Intelligent Orchestrator environment, an organization might define service level objectives that reflect a service level agreement.

service level objective (SLO) In the Tivoli Intelligent Orchestrator environment, a target measurement for applications and clusters that is used to influence resource allocation decisions. For example, a minimum response time might be a defined service level objective for an application.

severity An indicator of the importance of a patch. Patch severity can be used to prioritize patch deployment.

SLA See service level agreement.

SLO See service level objective.

Simple Object Access Protocol (SOAP) A protocol that specifies how to encode information in an HTTP header and an XML file for Web service request and response messages. SOAP provides a way for applications running on different operating systems to communicate.

SOAP See Simple Object Access Protocol.

software catalog The Tivoli Intelligent Orchestrator collection of software patches available for deployment. Any patches that Tivoli Intelligent Orchestrator manages must be added to the software catalog from a software repository or local downloaded files.

software product A piece of software, an operating system, or a software patch, and the associated parameters required for its deployment.

software repository An internal or external server that stores a library of software patches and the associated installation and configuration data. Tivoli Intelligent Orchestrator V3.1 supports the Microsoft Update web site or the Microsoft Software Update Services V1.0 server as a repository.

software stack A list of software products, organized in the required installation order. A software stack can contain individual software products as well as other software stacks.

spare pool See resource pool.

switch fabric A group of related network infrastructure components, such as switches and routers. The switch fabric delimits the scope of a network where devices can be shared. A data center model can include more than one switch fabric.

system A computer managed by Tivoli Intelligent Orchestrator.

system group A group of related systems. An administrator can create system groups to organize systems into meaningful categories, and to facilitate deployment of patches to multiple systems.

task In Tivoli Intelligent Orchestrator, a high-level configuration process that can be applied to individual or multiple target systems. Since a task can include multiple targets and can require a series of subprocesses, it can take a period of time to complete.

tc driver See orchestration package.

transition A step in a workflow that can be another workflow, a logical operation, a simple command, a scriptlet, or a Java plug-in.

Web service A modular application that performs specific tasks and is accessible via open protocols like HTTP and SOAP.

Archived

Web Services Description Language (WSDL) An XML-based specification for describing networked services as a set of endpoints operating on messages containing either document-oriented or procedure-oriented information.

workflow A workflow is a sequential series of steps that the system performs to accomplish a particular deployment task. Workflows are the best practice IT processes and procedures that an organization implements to make IT infrastructure changes.

Archived

Archived

Abbreviations and acronyms

ACL	Access Control List
APDE	Automation Package Development Environment
API	Application Programming Interface
ASCII	American Standard Code for Information Interchange
CD-ROM	Compact Disk - Read Only Memory
COTS	Commercial Off-The-Shelf
CPU	Central Processing Unit
DB	Database
DBA	Database Administrator
DCM	Data Center Model
DE	Deployment Engine
EAR	Enterprise ARchive
EJB	Enterprise Java Bean
EOF	End Of File
ERP	Enterprise Reseorce Planning
FTP	File Transfer Protocol
GUI	Graphical User Interface
HTML	HyperText Markup Language
HTTP	HyperText Transfer Protocol
IBM	International Business Machines Corporation
IIS	Internet Information Server
IP	Internet Protocol
IPX	Internetwork Packet Exchange
ISV	Independent Software Vendor
IT	Information Technology
ITIL	Information Technology Infrastructure Library
ITIO	IBM Tivoli Intelligent Orchestrator
ITM	IBM Tivoli Monitoring
ITSO	International Technical Support Organization
JAR	Java ARchive
JDBC	Java Database Connectivity
JDK	Java Development Kit
JIT	Just In Time
JMS	Java Messaging Service
JRE	Java Runtime Environment
LAN	Local Area Network
LDAP	Lightweight Directory Access Protocol
LDO	Logical Device Operation
MAC	Media Access Control
MB	Megabytes
NIC	Network Interface Card
OGC	Office of Government Commerce
OPAL	Open Process Automation Library
OS	Operating System
PC	Personal Computer
RDBMS	Relational Database Management System
RIP	Real IP
RM	Resource Manager
SAP	Service Access Point
SCM	Supply Chain Management
SCP	Secure Copy
SDK	Software Development Kit
SE	Servlet Engine

Archived

© Copyright IBM Corp. 2004, 2006. All rights reserved.

SI	System Integrators
SLA	Service Level Agreement
SLO	Service Level Objective
SNMP	Simple Network Management Protocol
SOAP	Simple Object Access Protocol
SSH	Secure Shell
TCO	Total Cost of Ownership
TCP/IP	Transmission Control Protocol/Internet Protocol
TEC	Tivoli Enterprise Console®
TIO	Tivoli Intelligent Orchestrator
TMA	Tivoli Management Agent
TMR	Tivoli Management Region
TPM	Tivoli Provisioning Manager
UDDI	Universal Description Discovery and Integration protocol
UI	User Interface
URL	Uniform Resource Locator
VIP	Virtual IP
VLAN	Virtual Local Area Network
VM	Virtual Machine
WAS	WebSphere Application Server
WSDL	Web Services Description Language
XML	eXtensible Markup Language

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 639. Note that some of the documents referenced here may be available in softcopy only.

- *Automated Provisioning Using IBM Tivoli Intelligent Orchestrator and Enterprise Workload Manager*, REDP-4117
- *Composite Application Provisioning with IBM Tivoli Provisioning Manager V3.1*, REDP-4222
- *Deploying Applications Using IBM Rational ClearCase and IBM Tivoli Provisioning Manager*, REDP-4105
- *Dynamic Provisioning of SAP Environments using IBM Dynamic Infrastructure for MySAP and IBM Tivoli Provisioning Manager*, REDP-4096
- *Exploring Storage Management Efficiencies and Provisioning - Understanding IBM TotalStorage Productivity Center and IBM TotalStorage Productivity Center with Advanced Provisioning*, SG24-6373
- *IBM Tivoli Intelligent ThinkDynamic Orchestrator Pre Proof-of-Concept Cookbook for Business Partners*, REDP-3830
- *IBM Web Infrastructure Orchestration*, SG24-7003
- *Introduction to pSeries Provisioning*, SG24-6389
- *An Introduction to Storage Provisioning with Tivoli Provisioning Manager and TotalStorage Productivity Center*, REDP-3900
- *Provisioning On Demand Introducing IBM Tivoli Intelligent ThinkDynamic Orchestrator*, SG24-8888

Archived

© Copyright IBM Corp. 2004, 2006. All rights reserved.

Other publications

These publications are also relevant as further information sources:

- *IBM Tivoli Intelligent Orchestrator Operator's Guide*, SC31-1421
- *IBM Tivoli Intelligent Orchestrator Overview Guide*, SC32-1419
- *IBM Tivoli Intelligent Orchestrator and Tivoli Provisioning Manager Installation Guide*, SC32-1420
- *IBM Tivoli Intelligent Orchestrator and Tivoli Provisioning Manager Release Notes*, SC32-1422
- *IBM Tivoli Intelligent Orchestrator Workflow Developer's Guide Version 3.1*, GC32-1652
- *Workflows Developer's Guide,* found at:

  http://www.developer.ibm.com/tivoli/workflow.html

The following IT Infrastructure Library publications can be obtained at:

http://www.itil.co.uk

- *Application Management*
- *IT Infrastructure Management*
- *Planning to Implement Service Management*
- *Security Management*
- *Service Delivery*
- *Service Support*

Online resources

These Web sites and URLs are also relevant as further information sources:

- Eclipse platform download page

  http://download.eclipse.org/eclipse/downloads
- IBM IT Service Management

  http://www.ibm.com/software/tivoli/features/it-serv-mgmt/index.html
- IBM Software Access Catalog

  http://www.developer.ibm.com/welcome/softmall.html
- IBM Tivoli Open Process Automation Library

  http://www.ibm.com/software/ondemandcatalog

Archived

- IBM Tivoli TIO Support Homepage

  http://www.ibm.com/software/sysmgmt/products/support/IBMTivoliIntelligentOrchestrator.html

- IT Infrastructure Library

  http://www.itil.co.uk

- Java home page

  http://java.sun.com

- Jython API documentation

  http://www.jython.org/docs/api

- Product Documentation

  http://publib.boulder.ibm.com/infocenter/tivihelp/v3r1/index.jsp?toc=/com.ibm.tivoli.tio.doc/tio_nav.xml

- Software support for IBM Tivoli Intelligent Orchestrator

  http://www.ibm.com/software/sysmgmt/products/support/IBMTivoliIntelligentThinkDynamicOrchestrator.html

- TIO APDE Forum

  http://www.ibm.com/developerworks/forums/dw_forum.jsp?forum=373&cat=15

- WebSphere Application Server: Trade3

  http://www.ibm.com/software/webservers/appserv/benchmark3.html

- XDoclet: Attributed-Oriented Programming

  http://xdoclet.sourceforge.net

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:

ibm.com/redbooks

Archived

Help from IBM

IBM Support and downloads

ibm.com/support

IBM Global Services

ibm.com/services

Archived

Index

Symbols
${tc.pkg} 240
) credentialskey 155, 212
.class file 606
.tcdriver file 42
.tcdriver jar file 244
bash target
    =DCMQuery(/server 155, 212

Numerics
1.4.2_04 Java SDK 244

A
access control
    global 109
access groups 106–107, 110, 113, 562
access permission 106, 108–109, 111
    Locical Device Operations 108
access privileges 106
accessRule 152, 204
acl-1 New1 152, 204
action name 168, 172, 240, 248, 555
action tag 237
adding a workflow manually 188
allocation decisions 29
Apache 233–235, 294, 463, 479, 541, 543, 546
Apache automation package 236–238
    example 243
    main section 238
    manifest file 249
Apache instance 472–473, 541
    DocumentRoot directory 472
Apache Server 294, 422, 459, 543
    application tier 562
    HTML files 550
    service 465
Apache Service 290, 297, 457, 478
    basic data 488
    index.html file 489
    operational state 478
Apache Web Server
    Installable software package 550
    Windows 549
Apache Web server
    automation package 234
apache.tcdr iver 234, 238, 245
APDE 116
APDE environment 113, 119, 132, 188, 258, 270, 418, 461, 512
    automation package 512
    long response times 113
APDE installation
    CD-ROM 122
    medium 118
APDE platform 347, 355, 358
APDE project 132–133, 341, 397
    additional configuration 135
    dependancy 135
    directory structure 136
    specific configuration 132
APDE workstation 118–119, 122
    command line environment 136
API 31
application 26
application cluster 26
application controller 29, 42
application environment 28–29, 64, 68, 620, 622
    different server requirements 29
    physical view 68
application instrumentation 15, 31
application manipulation 31
application pool 64
application priority 24
application programming interface 31
Application Programming Interface (API) 602, 608
application server 82, 282–284, 535, 622
    Port number 422
application tier 55, 57, 66, 289, 457, 557, 562
    VIrtual IP associations 457
appropiate line 192, 194
architecting best practices 63
arg value 86–87
ariableNewValue 601, 604
Attribute-Oriented Programming 610
attributes 179
authorization internals 76

© Copyright IBM Corp. 2004, 2006. All rights reserved.

automated best practices 16
automated provisioning 24
Automation 2
automation blueprint 5
automation capabilities 6
automation feasibility 56
automation implementation 6
automation package 18, 25, 37, 40, 45–46, 48–49, 90, 103, 106, 116, 125, 137–140, 156, 234–236, 282, 286–291, 302, 341, 397, 424, 467, 512, 536, 538, 595, 613, 619
  builder 256
  component 138, 144, 242
  components 160
  comprehensive affiliation 88
  creation process 244–245
  current implementation 234
  data center devices 41
  definition 49, 91
  definition file 91
  design 236
  developer 324
  development 68, 138, 537
  different elements 236
  directory structure 245, 247, 249–250
  doc directory 167
  documentation 91, 99, 156, 167, 236, 238, 247, 549
  documentation file 62
  documentation files 162
  Documentation tab 247
  example 561, 563
  file 49, 54, 60, 62, 234, 249, 252
  file name 238
  file repository 332
  increased size 301
  index.html file 460
  installation procedure 323
  installation process 242, 253
  major elements 234
  manifest file 237, 247, 255, 347, 354, 358
  meta file 513
  multiple workflows 39
  name 43, 91, 159, 164, 166, 238
  need 169
  new set 41, 49
  other content 159
  other workflows 144
  other XML files 315
  Overview 48
  perspective 125
  potential users 341, 396
  provision 162
  readme file 243
  readme.html file 549
  Reference Documentation 92
  repository directory 302
  repository subdirectory 418, 461
  short description 238
  simple commands 246
  software objects 341, 397, 424
  supporting documentation 156
  tcdriver 140
  trade3Install.zip file 418
  trouble-shooting problems 165
  user 160
  users guide 171
  weak point 512
automation package definition 40, 49
Automation Package Development Environment (APDE) 104, 116, 119
automation package directories 234
automation package documentation 247
automation package installation 253
automation package name 245
automation package packaging 243
automation packages 24
automation packages basics 40
automationlibrary 18
autonomic technology 10

B

back-end service 64
balancing techniques 22
bare metal server 68
bash target 279
beans 37–38
benefits of on demand 16
best practices 16
BigIP 41
billing for system usage 7
branching 61
break down the operations 58
British Standard (BS) 614
build-in Java 37, 501
business policies 6
business policy 2, 9

Archived

business requirements 3
Business Services Management 7

C

capacity fluctuations 13
Capacity Management 13
case study 281
  overview 282
Change Management 89, 91–92, 618–619, 622
changing demand 2
Cisco CSS 41
Citrix 59
class 177
class file 214–215, 225, 232, 596, 605–606
classpath 605
cloning a workflow 189
cluster manager 22
cluster visualizing 24
Cluster.Add Server 562
code libraries 51
collection of clusters 24
command line
  interface 43, 63, 253
CommandDriver class 603
command-line interface 31
compiled .class file 606
compiled Java plug-in 605
component relationships 38
compressed Java archives 244
compressed Java jar file 42, 234
Configuration Management 616, 618–619
consistent on demand 15
container 40, 49
control of workflows 47
controlling workflows 35
core tcdriver 135
core.tcdr iver 238–239
core.tcdriver 239, 248
COTS 27
creating automation packages 245
customer 26
customer environment 50, 63, 181, 291, 302
cygwin 215, 222, 230, 236, 249–250, 365, 371, 444, 598, 602, 605

D

Data Acquisition Engine 42
Data Center
  manual process 176
  manual processes 179
  other processes 183
  product definition 145
data center 10, 13, 22, 25, 29, 46–48, 104, 140, 145, 156, 176, 179, 234–236, 328, 333, 335–336, 393
  Assigned object 100
  load balancer 35
  logical model 336, 393
  modular solutions 37
  network device 234
  network devices 34
  resource management solutions 13
Data Center Model 25
  database 42
  interactions 42, 49
data center operations 35
database client 118, 129–130, 288–290
database definition 311
database instance 290, 293, 295, 541
  software instance template 541
database SID 285, 525
database.fami ly 299
database.family 299, 307
database.vers ion 299
database.version 299, 307
datacenter 31, 282
db cfg 114, 285, 525
db config 285, 525
DB2 Client 117, 120, 288, 292–293, 387
  catalog entries 387
  directory 403
  installation 120, 311, 392
  installation executables 402
  instance 402
  software 305, 310
  software installation object 393
  software instance 311
  software package 311
  Trade database 387
  V8.2.1 120
DB2 instance 120, 292, 336, 339, 370
DB2 node 390, 402
DB2 Server
  environment 300
  installation 301, 345, 370
  Installation object 293
  v8.2 288

Archived

DB2 Server installation
object 347
DB2 shell 284, 525
DB2 UDB 304, 306–307
DB2 Universal Database 534, 537, 539
Enterprise Server 534
software definition 534
db2 update 285
db2stop force 114, 285
DBA 113, 121
DCM 51, 55–56, 106, 108–109, 145–146, 148, 178, 195, 203, 263, 270, 288, 290–291, 608
DCM definition 401, 405
DCM object 50, 60, 62, 106, 140, 154, 163, 179, 202, 206, 255, 302, 330, 387, 415
expected hierarchy 330, 387, 415
parameter 147
Property 145
software product definition 171
DCM query 149, 203–204
expression 151, 203
language 154, 206
DCM resource 110, 422
DCMInsert parent 153, 205, 273
DCMQuery 117, 140, 149–150, 196, 202, 263, 265, 333, 473
debug message 165
default value 121, 130, 171, 190, 301, 540, 609
defaultValue element 609
definition templates 50
dependencies tag 237
dependency name 167, 171, 239, 248, 555
deployment cost 16
deployment engine 29, 35, 37, 42, 47, 49, 84, 323, 347–348, 497
class path 158, 161
client credentials 497
Device ID 260
log 155, 211
system 499
deployment engine classpath 606
design preparation 70
DestinationDeviceID 568, 579, 581
developer's perspective 176
developing Java plug-ins 596
development challenges 181
Development Kit 244
device driver 24, 40, 48, 62, 89, 108, 159–160, 292, 304–305, 335, 549, 551
device instrumentation 31
device model 26, 36, 49, 141, 166, 169, 177, 241–243, 547
device models 237
Device.ExecuteCommand 76, 144, 146, 149, 198, 208, 275, 327, 352, 448
DeviceID 566, 568–569
device-model name 169–170, 172, 242, 249, 334, 347, 354, 553, 555
device-model tag 241
device-models XML tags 248
devices and logical operation 56
Directory Server 65
directory structure 161, 231, 234, 244, 247, 249, 598, 606
Distribute 345, 556, 558
Distribute LDO 334
doc directory 42, 167, 236, 247, 512
DOCTYPE tc-driver (DTD) 552–554
domain name space 608
download 160, 215–216, 244, 597
driver 24, 40, 390
driver-name >
and 159, 238
tag 159
driver-name tag 237
driverTypeDeId element 608
DriverVariableDescriptor 607–608, 610
dynamic business 3
dynamically provisioned 6

E

e.g. c 86, 447
Eclipse environment 104, 117, 123, 231
Eclipse SDK 122
Economical 2
EJB 25, 37
EJB methods 42, 49
ejb.version 299, 313, 388
enabling application 15
endif 328
Enterprise Java Beans 25, 37–38
error handling 148, 255, 278, 448
base data 442
control information 435
error message 149–150, 154, 212, 363, 384, 478
esac function 342, 363, 398
excess capacity 13

exe file 528, 530, 533
executable 160, 162
executable chmod 302, 418, 460
executable workflow 29
exit-points 58
expect scripts 246
exporting workflows 245
expression x 591–592
external fault management 43

F

feasibility of workflows 47
file name 157, 162
file repository 55, 302, 531, 536, 549
  specific location 302
  Table.ddl file 333
  zipped archive 509
File Transfer Protocol (FTP) 78, 81
Flexible 2
flexible business processes 3
fluctuating business demands 11
fluctuations 13
following logical device operation
  Trade specific implementations 353
foreign resource 406, 430
foundation of IBM Automation blueprint 6

G

Get Current Request ID 187
Global Access Control 109
global ITITO variables 237
Global Resource Manager 42
Global Resource Manager component 29
global visibility 17
grouping workflows 31
groups
  access 110
GUI 35, 37, 42, 56, 138

H

hardcoding 182
hardware object 77
higher service levels 16
higher-level abstraction 41, 49
high-level operations 53
Horizontal scaling 23
host name 263–264, 266
HTML 549–550, 553
HTML file 549–550, 552
html format 160

I

IBM AIX 40, 48
IBM DB2 118–119
  client 118–119
  system 54
  UDB 65
  V8.2.1 UDB Database Server 120
  Version 8.2 119
IBM HTTP Server
  V2 287–288, 304
  V2 software product 317
IBM JRE 119, 123
IBM market share 17
IBM On Demand Automation Catalog 234
IBM Orchestration library 18
IBM Tivoli 17–19, 34, 47, 50, 53, 156
  Configuration Manager 59
  Intelligent Orchestrater 4, 8, 11, 21–23, 45–46, 48, 175–176, 179, 214, 233–235, 282, 595–596, 598, 620
  Intelligent Orchestrator v3 57, 61, 202, 565, 576
  Intelligent ThinkDynamic Orchestrater 55–56, 59, 181
  Monitoring 54
  Software validation 19
IBM Tivoli Enterprise Console 51
IBM Tivoli Monitoring 51
IBM WebSphere Application Server
  V5.1 304
ibm.com.itso util 607–608, 610
ICMP 78
identify transitions 53
if.. then.. else 61
immediate impact on business 11
implementor 57, 59–60
index.html file 422, 460, 462, 549, 561
info Jython 146, 149, 198, 200, 202
information-gathering task 29
infrastructure component 27, 287, 291, 298
  access credentials 300
  installation directories 324
infrastructure cost 16
Infrastructure Library 69, 77, 613–615
  discipline 622

Archived

infrastructure provisioning 13
input parameter 148, 261–263, 270, 273, 301–302, 327
input variable 38, 47, 97, 145, 148, 190–191, 604, 609
installable file 157, 528–530
  entry 547
  installation priority 536
  level 535–536, 556
  list 545–546
  locale requirements 535
  object ID 558
  priority order 556
  software definition 532
installable-package name 552–553
installation directory 123–124, 216, 300, 540, 598
installation mechanism 532, 535, 556
installation process 40, 85, 123, 242, 244, 253, 292, 345, 538, 552
installation script 146, 198, 208, 444
  Trade3Provision.xml 86
Installation SRT 310, 435
  direct children 354
Instance 329, 345, 353, 550, 559
instance user 322, 357, 365
instantiation 292, 294, 338
instrumenting an application 31
integration 2
integration interface 43
inter-networking device 41
IP address 22, 46, 67, 126, 153, 186–187, 205, 489
IP addresses 203
IP configuration 261
ISO9000 69
ISO9001 69
ISV 16, 32, 35, 54, 57, 613
IT infrastructure 2
IT Infrastructure Library 69
items tag 237
iterator 194, 544–546
ITIL 69
ITIO server 263
ITITO benefits 14
ITM 51
ITPM benefits 14
ITSO_Create_User 347, 371, 505
ITSO_Destroy_User 365, 508

J

J2EE 215, 597
J2ME 215, 597
J2SE 215, 244, 597
J2SE SDK 215–216, 597
  current version 597
jar command 250
JAR file
  checkboxes 225
  ItsoPathHelper.jar 225
jar file 42, 136, 215, 224, 231, 234, 239, 244, 246, 597, 606
  final step 227
jar utility 244
Java 2 Standard Edition 244
Java Beans 37–38
Java class 34, 47, 57, 96, 117, 156–157, 177, 208, 214–215, 228, 283, 596–597, 608
  single method 228
Java code 37, 52, 63, 104, 177, 180, 223, 599
Java Development Kit (JDK) 215, 244, 597
Java drivers 235, 240
Java extension 215
java extension 597, 599
Java method 177
Java object 177
Java plug-in 34, 47
Java plug-in development 596
Java plug-in xml file 607
Java plug-ins 34–36, 46–47, 49, 144, 180–181, 202, 214, 228, 234–236, 239, 241, 595–596, 598
  huge set 181
  information 245
  XML 246
  XML definition file 246
Java program 24, 34–35, 46–47, 62, 144, 213, 534, 537, 595
java program 34
Java Runtime Environment (JRE) 118–119, 123, 215, 597
Java SDK 69
Java source
  code 604
  file 215, 597, 599, 605, 609
Java source file 599
java.lang.Syst em 156, 208, 274–275, 575
java.lang.System 230
javac compiler 605
javac.exe 605

Archived

JavaDoc tags 610
JDBC resource 408, 443
JDK 244
JDK level 244
jdk.version 300, 458–459
just-in-case provisioning 10
just-in-time provisioning 10

K

kanaha 240
key automation capabilities 6
key components of on demand 4
key data center processes 13
key elements of automation 4
Kit 1.4.2_01 244
ksh 78

L

LDAP 65
LDOs 289, 327, 329
lines of business 12
linked access permission
  access groups 108
linking business and IT 3
load balancer 22, 41, 67
  configuration 289, 457
  cycle 22
  detailed understanding 22
  technology 289
Local variables 38
log error
  Jython 208, 210, 229, 279
  message exception 349, 430
  msg 279
log info Jython 76, 146, 149–150, 198, 200, 202, 230, 269, 272, 275
logical device 41, 49, 140, 144
  operation 106, 108, 111, 144, 188, 268–269, 289, 295–297, 550, 556
  operation SoftwareInstallable.Dist ribute 352
logical device operation (LDO) 295–297, 550, 556, 558
Logical operation 46
  naming convention 89
logical operation 24–25, 33, 35–36, 42, 46, 49, 88–89, 93, 95–96, 140, 143–144, 176–177, 180, 240, 242–243, 292, 304, 345, 620
  core implementation 144
  full suite 41
  underlying defaults 144
lower support costs 16
low-level operations 53

M

macro substitution 240
managed data center 42
Management Interface 42
Management Interface component 30
Manager 3.1 551, 553
managing hardware objects 77
managing software objects 82
manifest file 160, 162, 167, 227, 236–238, 550, 552, 554
  attribute string 240
  dependencies section 554
  device model section 553
  string references 240
manifest file sections 238
manifest file tags 237
manual installation
  procedure 49
manual process 50, 70, 176, 183, 186, 256
manual provisioning 10, 183
market share 17
metadata information 234
metering usage 7
method 177
method implementation 177
middleware product 54
modularity 58
modularity of the design 73
Morten Moeller 199, 207
multi-application environment 13

N

named driver
  installation status 254
naming automation package 245
naming convention 140, 157, 159, 303, 333
naming convention for logical operation 89
naming convention for workflows 89
naming conventions 53, 88
naming conventions for automation packages 90
native interfaces 31
ne 0 155, 212
network device 14, 29, 34–35, 47, 64, 67, 142, 234

Archived

Network Management 64
network-interface name 153
none 445, 505
number of requests 24

O

object 177
object ID 556–558
object oriented development 177
Office of Government Commerce (OGC) 615
on demand benefits 8
on demand operating environment 2
on demand strategy 17
OPAL 18, 75
Open Process Automation Library (OPAL) 75, 119
open standards 2
operating environment 2
operating system (OS) 25, 29, 40, 68, 143, 240, 299, 305, 528–529, 531, 620–621
    bare metal installation 40
    required operations 143
operational model 13
operational processes 33
operations break down 58
optimal allocation decisions 29
Optimization 7
optional argument
    count 586
    deletechars 588
Oracle database
    environment 118
    host name 86, 446
    SID 86, 446
orchestration 7
Orchestration and Provisioning Automation Library (OPAL) 18
orchestration capabilities 9
orchestration of the automation 9
Orchestration Solution Package 50, 53, 55–56, 247
    setup tasks 55
orchestration workflows 31
Orchestrator and Provisioning Automation Library (OPAL) 139
os.family 329, 331, 387
output parameter 58, 260, 262, 264
output variable 34, 47, 97, 145, 190, 196, 604, 609
Output variables 38
over-provisioning 23

P

package documentation 51
Package Overview 340, 396, 423
package verification task 249
Packaged Intelligence 10
PackageRepositoryDir 237
PackageRepositoryHost 237
packaging the automation package 243
param name 168, 171–172, 241–242, 248, 302, 418, 460, 554–555
parameter name 170, 173, 243, 300, 324
password credential 79, 81, 270
PathHelper 228–230
Perl scripts 246
permission
    access 109, 111
physical device 24, 40–41, 49, 61, 160, 167
plug-in run 214, 596
Policy Region 76
Policy-based orchestration 4, 9
post-installation workflow 62, 237
post-install-workflow >
    tag 242
post-install-workflow name 242
post-install-workflow>
    definition 242
pre-package verification 249
preparing to write a workflow 185
primary line of business 12
privileges
    access 106
product families 19
properties tag 237
property component 170–171, 173
protocols and credentials 59
provisioning 4, 7–8
Provisioning Automation Library 18
provisioning cycle 10
provisioning interfaces 16
provisioning practices 10
provisioning workflows 31
publishedB element 609

Q

quality 7
quoted string references 240

Archived

R

README file 92, 156, 243, 247
Ready for IBM Tivoli software 19
real-life implementation 331–332, 389–390, 417, 460
real-time access 30, 42
real-time correlation 13
recovery action steps 32
recovery policies 53
recovery script 191
Red Hat
  Package Manager 530
Red Hat.xml 236, 241, 246
Redbooks Web site 639
  Contact us xxviii
Re-initiate process 70–72
Release Management 619, 622–623
repeatable workflows 29
replicas 22
replicated servers 22
repository server 185
required middleware product
  installation images 54
requiredB element 609
Reserve systems 23
resilient 2
resource consumption 24
Resource Manager 35
Resource Manager (RM) 29, 42, 178, 187
Resource Models 51
resource pool 26, 65, 178, 562
resultBuffer.appe nd 601
rethrow 149–150, 184, 210–211, 279
Return Code (RC) 145, 212, 279
reusable workflows 75
reuse 58
reuseability 58
reusing transitions 75
RiversEdge Web 152–153, 203, 205
roles 108
  security 106, 109
round robin 22
RSA credential 81
RSA credentials 80
rule files 51
Running DB2 285, 525

S

SAP 27
SCP 79
SCP protocol 79–80
scriptlet 155, 195, 211, 275, 333, 358, 363
scriptlets 33
SDK SE v1.4.2_04 215, 597
SDK version and level 244
SDK1.4.2_04 244
second workflow 37
Secure Shell 78
security 6
security exposure 16
security roles 106, 109
self-configuring 7
self-managing 2
Server Name 126, 151, 153, 203, 205, 422
Server NICID 158
Server1 152, 204, 284, 286–287
servers definition 26
Service Access Point
  password credentials 78
  RSA credentials 79
  SNMP credentials 80
service access point (SAP) 27, 76–78, 81, 147, 188, 273, 323
service level
  agreement 617
  objective 13
service level delivery 14
Service Level Management 13
service level objective 11, 13, 27
service level priority 26
service levels 7
servlet 276, 534, 537
servlet engine 414, 535, 537
  required resources 414
SI 17
simple command 47
Simple Network Management Protocol (SNMP) 56, 80–81
simulated infrastructure component
  software definitions 304
single-purpose servers 12
SLO 27
SNMP 78
SNMP credential 80
SOAP 35
SOAP calls 31

Archived

SOAP commands 35
soapcli command 31
software architect 53–55, 58, 63
software component 24, 40, 48–49, 56, 83, 160, 234, 298, 540
  data center model 540
  specific type 24, 40
software configuration 292, 529, 537, 539
software configuration template 302, 529, 537, 539
  basic purpose 302
  object ID 556–558
  parent template 539
software definition 51, 295–297, 528–530
  appropriate installable file 544
  device models 560
  general type 529
  installable files 531
  level 535, 556
  list 544
  multiple installable files 533, 536
  only installable file 550
  several types 529
  software configuration template 538, 541
  software definition 561
  Software requirements 533
  special type 542
  Windows 2000 requirement 532
  XML files 554
Software Development
  Kit 215, 244, 597
Software Development Kit (SDK) 597
software element 547
  individual device models 560
  separate device models 560
software installation 51, 139, 179, 185, 288, 291–293, 530, 538–539, 619
  device driver 329
  device model 441
  implementation details 336, 393
  object 292, 307
  object TIO/TPM 336, 393, 420, 463
  parent software configuration template 553
  resource 292, 402, 538–539, 541
  template 539–540
  wizard 304
software installation object 293, 329, 335–336
software instance 292, 311, 335, 546
  device driver 355
  software resource 559
software model 293, 298, 335, 392, 527
  other aspects 561
software object 82, 140, 160, 170, 176, 301, 341, 392, 397, 424
software package 35, 68, 84, 87, 242, 292, 295, 301, 333–334, 345, 390–391, 418, 530
  definition 242, 288, 291, 294
  device driver 551
  different versions 535
  entry 547
  format 535
  logical operation 329
  path names 333
software patch 141, 530
Software product 10, 27, 41, 56, 59, 140–141, 143, 175, 182, 186, 235, 288, 291–292, 294, 329, 529
  configuration settings 539
  software definitions 542
software product 27
  associate special capabilities 330, 388, 416
  customer customizable parameter 171
  DCM object definition 170
  DCM object ID 145
  default variables 171
  definition 84, 170, 304, 306, 330
  Installation Software Resource Template 329
  local variables 83
  ordered list 27
  proper software resource templates 330, 387, 415
  stanza 170
  variable 84
software resource 291–292, 335, 528–529, 537
  anchor point 291
  change 558–559
  default parameters 540
  default values 529
  following types 539
  hierarchy 292
  model 328, 335–336, 392
  software resource template 409
  stores information 540
  template 84, 179, 292, 295–296, 327, 329, 335, 340, 559
  template definition 83
  template hierarchy 336, 393, 420, 463
  template structure 336, 393
software resource template 292, 295, 322, 325, 335–336, 338

software resource type (SRT) 292–293, 301, 541
default behavior 335, 392
software signatures 54
software solution 13, 54, 57, 140, 143–144
software stack 27, 56–57, 176, 530, 539, 542
software value 16
Software.Start 141, 143, 163, 180
software.title 299, 307
Software.Uninstall 141, 143, 163, 180
software.version 307, 311, 313
software-module name 552–553
software-products >
tag 242
tag definition 243
software-products tag 237
software-resource-template name 554
source code 136, 222, 231, 599, 603–604
SRT hierarchy 336, 393, 420
SSH 56, 78–80, 274
SSH communications 65
stable application control 29
stall LDO 441, 481
storage device 64
storing application specific password (SAP) 60, 82–83
structured documentation 91
subnetwork 149, 152, 204
subprocess 621–622
Switch Fabric 27
Switch.Move Port to VLAN 41
system internal properties 43
system usage 7
systems integrators 17

T

Table.ddl file 322, 332
target server 145, 185, 327, 329, 529, 532, 534
DB2 executables 370
index.html file 465
installation mechanism 532
software installation object 336, 393, 420
Trade DBClient Module installation 392
Trade DBServer Module installation 335
trade3 DB2 Instance 346
trade3db DB2 database 346
Windows Installer 532, 535
target system 140, 144, 146, 160, 198, 237, 246, 276, 307, 322, 328, 331, 333, 336, 528–531, 533
actual configuration 529
configuration 533
corresponding software instance 538
DB2 Client 393
DB2 UDB Server 336
host credentials 497
object ID 556–558
software installation resource 539
software package 558
software resource 537
software resources 538
Table.ddl file 377
temporary directory 352
Trade Module DBClient Installable 402
Version 8.1 537
ws_ant program 439
TC driver 24, 40
TC driver package 40
tc.pkg 240
tcdriver 24
tcdriver extension 40, 62, 234, 252–253
tc-driver manifest 41, 49
TC-Driver.xml file 92, 136, 138, 140, 157
tc-driver.xml file
dcm> section 342, 398, 424
referenced items 136
tc-driver.xml manifest file 237
tc-driver-manager 43
tcdrivermanager 240
tc-driver-manager command 43, 105, 238, 244, 252
TC-INF 234
TC-INF directory 166, 236–237, 302
TCO 16
TEC 51
TEC rules 54
technical workshops 18
template-param name 554
thinkdynamics 240
thrash 24
three-tier architecture 63
TIO and TPM differences 15
TIO overview 13
TIO server 76, 157, 162, 166, 260–261, 264
automation package 166
DCMQuery workflow 265
remote system 275
TIO/TPM 138–140, 179, 202–203, 214, 220, 224, 292, 298, 302, 596, 598, 605

Archived

TIO/TPM API 602
TIO/TPM architecture 28
TIO/TPM GUI 324
TIO/TPM objects 176
TIO/TPM server 160, 302, 332–333
  system 302, 348
TIO/TPM v3 199, 211, 214
TIOsetVar 363
Tivoli endpoint 76
Tivoli Intelligent Orchestrator
  configuration 144
  deployment engine 155, 211
  server 126
Tivoli Intelligent Orchestrator (TIO) 137–139
Tivoli Management Region 76
Tivoli Provisioning Manager (TPM) 137–139, 258
TMR server 76
top-level object 153–154, 205–206
top-level path 60
Total Cost of Ownership (TCO) 16
total cost of ownership (TCO) 16
TPM overview 13
Trade application 282, 286–287, 414, 457
  Apache server 470
  DB2 database 377
  installation archive 418, 513
Trade Automation Package 288–289, 294, 513
  integral part 333, 390
  other components 330, 388, 416
  potential users 424, 467
  software objects 301
  successful implementation 300
  tc-driver.xml file 334, 391, 419
  xml files 341, 397
Trade database 286, 289, 293, 387, 526
  catalog entries 388
  existing instance 396
Trade DBClient Module 293, 296, 325, 387
  completed software resource template hierarchy 396
  final software resource template hierarchy 394
  Helper 408
  Installable 387, 389
  installation 388, 392
  property 388
  simple nature 405
  software installation object 405
  software model 394
  software product 293, 387–388
  software product look 393
  software resource template 392
Trade DBServer Module 293, 295, 307, 329
  completed software resource template hierarchy 340
  INSTALLATION Software Resource Template 354
  software model 338
Trade Web Module 293, 325, 422, 458
  final software resource template hierarchy 464
  INSTANCE software resource template 478
  non-hosting APPLICATION requirements 488
Trade3 application 54–55, 59, 282–284, 523, 526 286, 445, 448
  example 85, 99
  executables 54
  installation script Trade3Provision.xml 86
  main architecture 283
  official installation instructions 523
  official installations instructions 282
  provider 54
trade3 db2 instance
  operational status 363
Trade3.xml installApp 448
Trade3.xml installResources 448
Trade3.xml restartWebSphere 445, 448
Trade3.xml startWebSphere 444, 448
Trade3.xml uninstall 445
trade3db database 285, 331, 335, 340, 371, 392, 525
  catalog entries 392
  database tables 331
traffic demands 13
Transitions 180
transitions 25, 33, 46
transitions adding 189
transitions reuse 75
try-catch-endtry sequence 209–210

U

undoableB element 608
UNIX format 488
user ID 82–83, 85
  Encrypted coding 88
user-defined parameter 538
use-workflow workflow-id 170, 173
utility workflow 432
  ITSO_Find_Supporting_Installation_for_Module

_Reqirements 409
ITSO_Find_Supporting_Installation_for_Module _Requirements 472
ITSO_HostingEnvironment_Host 402, 473

V

var i 196, 200, 202
var j 202
var message 269
var msg 184, 229, 279
var MSG1 146, 149, 198, 208
var name 199, 271
var result 148, 197, 267, 269
var ServerID 196, 269
var simulation 327
var simulation Trade_Get_Simulation 303, 327, 346
variableName >
Result 608
Separator 608
String 1 607, 610
String 2 607
String 3 607
String 4 608
variables 179
veInstance 337, 355–356, 558
verify automation package install 253
verifying deployment 253
Version 8.1 534, 537
version tag 237
virtual device 41, 49
virtual IP address 22
virtualization 2, 6
VLAN 41, 57, 67
VMWare 59

W

wareHelper 348, 568–569, 579
Web cluster 22–24
Web clusters 22
Web front end 291, 457, 463
Web site 549–550, 553
index.htm file 553
software definition 562
Web UI
page 110
workflow interaction 109
Web-based interface 30, 42, 56, 255
WebsiteSample Installable 551–552, 554
WebsiteSample Installation 553–555
WebsiteSample.tar file 550–552
installable file 551
software package entry 561
WebSphere Application
Server 82, 284, 287–289, 535
Server 5.0 524
Server Administrator 300
Server implementation 444
Server installation 417, 419–420
Server Installation object 293
Server node 292
Server setupCmdLine script 524
WebSphere Application Server 282–284
Administrator 300
environment 439
implementation 430
WebSphere environment 82, 181, 300, 302, 443–444
correct datasource 444
JDBC resources 443
WebSphere resource 432, 447
workflow 15, 18, 24–25, 29, 34, 46–48, 104–106, 138–140, 175–177, 214–215, 228, 230, 234–236, 259–261, 263, 301–303, 323, 536, 538, 620–622
complete listing 377
detailled implementation 484
entire content 366
following main functionality 478
full listing 372
Installation SRT 435
main functionality 365
main functions 352
main outline 408
sample invocation 327
software resource template 406
wrapper workflow 230
Workflow assembly component 29
workflow coding
practical details 175
workflow component 33, 38, 47, 69, 75
Workflow Composer 103–105, 116, 138, 192, 194
dialogue 116
Workflow browsing 116
workflow definition 24
workflow developer 18, 34, 108–109, 117, 183, 185, 536, 547, 638
focal point 18

Archived

workflow development 39, 76, 115, 136, 138, 212, 215
basic platform 215
workflow development challenges 181
workflow execution 30, 117, 260, 262, 264
controller 30
history 118, 260, 262, 264
log 146, 149, 198, 208
workflow execution component 29
workflow file 132, 162, 169
workflow name 89–90, 169, 171–173, 242, 249, 303, 323, 334, 347, 354, 553, 555
different parts 90
workflow objects 185
workflow scope 185
Workflow Transitions 35
workflow transitions
Java plug-in 34
java program 34
logical operation 33
workflow 34
Workflow Variables 35
workflow variables 38
workflows 10, 15–16, 24, 26, 29, 33, 45–47, 103–104, 106, 137–139, 175–176, 179, 213–214, 225, 234–236, 259, 263, 266, 286, 290, 295, 529, 538, 544, 565, 576, 595
workflows and the DCM 179
workflows design 70
workflows for provisioning 31
workflows functionality 46
workflows tasks 34
workflows templates 32
working threads 30
workload model 29

X

XDoclet engine 609
XML data 152, 204–205, 536
xml deck 342, 398, 424
xml definition 158, 347, 402, 405
XML element 608–609
XML elements 608
XML file 127–128, 131, 166, 168, 226, 235–236, 248, 302, 305, 308, 312, 550, 606–608
item definition 342, 398, 424
XML files 51
XML format 551–552
software definition 551
XML statement 166, 334, 391, 402
XML tags 237
XML version 128, 131, 158, 167, 171, 238, 248, 553–554, 607

Archived

Archived

Redbooks

Developing Workflows and Automation Packages for IBM Tivoli Intelligent Orchestrator V3.1

(2.0” spine)
2.0” <-> 2.498”
1052 <-> 1314 pages

Archived

Archived

Developing Workflows and Automation Packages for IBM Tivoli Intelligent Orchestrator V3.1

Reduces infrastructure and deployment costs through automated best practices

Enables your solution for on demand provisioning

Gives practical aspects for workflows and automation packages design

IBM Tivoli Intelligent ThinkDynamic Orchestrator and IBM Tivoli Provisioning Manager help automate the provisioning and orchestration tasks by invoking pre-prepared workflows that perform most of the provisioning and orchestration tasks automatically. This frees up resources to focus on more productive issues, and helps allocate CPU capacity to where it is needed and when it is needed.

Workflow and Automation Package management are the single most important success factor in any IBM Tivoli Intelligent ThinkDynamic Orchestrator and IBM Tivoli Provisioning Manager implementation. Understanding and mastering this issue is critical for Independent Software Vendors (ISV) who wish to allow their products to be an integrated part of any IBM Tivoli Intelligent ThinkDynamic Orchestrator and IBM Tivoli Provisioning Manager environment.

The primary goal of this IBM Redbook is to support ISVs, Business Partners, Clients, and IBMers in developing and implementing Workflows and Automation Packages to be used for automation purposes by IBM Tivoli Intelligent ThinkDynamic Orchestrator and IBM Tivoli Provisioning Manager. This IBM Redbook is focused on effectively planning, developing, testing, and implementing product, device, or customer specific best practices into the IBM Tivoli Intelligent ThinkDynamic Orchestrator and IBM Tivoli Provisioning Manager engines, in order to automate management processes related to specific system and device implementations.

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

SG24-6057-01 ISBN 0738496693