World Wide Web Programming: VisualAge for C++ and Smalltalk
The VisualAge Series

Bitterer, Brassard, Nadal, and Wong
*VisualAge and Transaction Processing in a Client/Server Environment*

Bitterer, Hamada, Oosthuizen, Porciello, and Rambek
*AS/400 Application Development with VisualAge for Smalltalk*

Bitterer and Carrel-Billiard
*World Wide Web Programming: VisualAge for C++ and Smalltalk*

Carrel-Billiard, Jakab, Mauny, and Vetter
*Object-Oriented Application Development with VisualAge for C++ for OS/2*

Carrel-Billiard, Friess, and Mauny
*Programming with VisualAge for C++ for Windows*

Fang, Chu, and Weyerhäuser
*VisualAge for Smalltalk and SOMobjects: Developing Distributed Object Applications*

Fang, Guyet, Haven, Vilmi, and Eckmann
*VisualAge for Smalltalk Distributed: Developing Distributed Object Applications*

Nilsson and Jakab
*VisualAge for C++: Visual Programmer’s Handbook*
World Wide Web Programming: VisualAge for C++ and Smalltalk

Andreas Bitterer
Marc Carrel-Billiard
To the members of **Take Five**: Didi (Pharaoh), Hägar (Mountaineer), Mathew (Detective), and HaPe (Captain), for the great time on and off the stage. Watch out for those falling cactuses!

A.B.

To Tammy and Neil and our unfailing friendship. To the philosophers, their chopsticks, and all the people with whom I had fun teaching around the world. If you visit Tokyo, steer clear of 納豆！

M.C.
Foreword

Application development has dramatically changed in the last several years. In just a few evolutionary steps, programming has shifted from host-based green-screen applications, to workstations and PCs with graphical user interfaces and a client/server architecture, to the Internet, the World Wide Web, and its browser interface. Applications are no longer slated to run only on the desktop; they now run on Web servers throughout the Internet and company intranets. As an application user, you do not install an application. Instead, you simply point your favorite Web browser to a Web site and launch the application from there.

In this book, you will learn how to leverage IBM application development products such as VisualAge for Smalltalk and VisualAge for C++, along with their Web Connection technology, and how to rapidly deploy sophisticated Web applications from prefabricated parts. The Web Travel Agent sample application is built through visual programming and includes relational database access, dynamic Web pages using the Common Gateway Interface, multimedia, and Java.

The authors walk you through the full application development cycle, from object-oriented analysis and design to implementation and packaging. They present the sample application in two flavors, which are based on the same design and access the same DB2 database. So, whether you are a C++ or Smalltalk programmer, this book will help you come closer to your own first application on the Web.

The CD-ROMs that accompany the book contain Windows trial versions of IBM’s VisualAge for Smalltalk, VisualAge for C++, Database 2, Internet Connection Server, and the Web Travel Agent sample application. Install them on your own system and give them a try.

Welcome to the world of Web programming!

John Patrick
Vice President – Internet Technology
IBM Corporation
Preface

This book demonstrates how to use VisualAge for C++ and VisualAge for Smalltalk for the creation of object-oriented World Wide Web (WWW) applications. The reader learns to visually develop parts to build web pages that interact with underlying business logic and databases. A case study of the (fictional) Web Travel Agency (WTA) guides the reader through the steps of analysis, design and implementation of a sample web application that uses the Common Gateway Interface to communicate between HTML pages and VisualAge. All necessary parts and connections between them are explained, and all design considerations are covered in detail.

To demonstrate portability and to show different application platforms, the Smalltalk version of the WTA was implemented in OS/2, the corresponding C++ version was implemented under Windows 95.

This book explains how to use DB2 for OS/2 or DB2 for Windows 95/NT as database for application data through the VisualAge database parts, and also as server for the DB2 WWW Connection gateway.

The Web Travel Agent provides the business logic, interfaces with a DB2 relational database, and generates HTML pages dynamically through the Common Gateway Interface. The VisualAge Web Connection parts are exploited to develop a Smalltalk application, that is based on the Visual Modeling Technique (VMT).

World Wide Web Programming: VisualAge for C++ and Smalltalk was written for webmasters and web application programmers who want to extend their WWW presence through the use of object-oriented applications, and for VisualAge for C++ or Smalltalk developers looking for ways to add a WWW interface to their DB2 databases and object-oriented applications.

Some knowledge of object-oriented programming with C++ or Smalltalk, the Structured Query Language (SQL), and HTML scripting is assumed.

On the enclosed CDROMs we provide you with Windows trial versions of VisualAge for Smalltalk, VisualAge for C++, IBM Internet Connection Server, and DB2, along with the sample Web Travel Agent application. This application may serve as a base for testing your own setup and can easily be expanded and adapted to match your environment.
How This Book Is Organized

This book consists of two parts. In Part 1, we set the stage for Web application development using IBM VisualAge products. We introduce the Web programming environment and the used development tools. In Part 2, we describe a real-world case study by going through the development steps of analysis, design, prototyping, and implementation. We also cover installation, configuration, and packaging issues. In some more detail, the book is organized as follows:

- Chapter 1, “Introduction and Overview”
  This chapter introduces the World Wide Web development environment and describes the Common Gateway Interface.
- Chapter 2, “VisualAge Web Development Directions”
  This chapter contains a description of the various Web models and the VisualAge support for the models. We explain how the VisualAge product family can be leveraged for Web application development.
- Chapter 3, “VisualAge for Smalltalk Web Connection”
  In this chapter, we describe the Web Connection Feature of the VisualAge for Smalltalk product. We demonstrate how VisualAge interacts with the Web server through CGI Link. We also show examples of using visual and nonvisual parts to create Web pages.
- Chapter 4, “VisualAge for C++ Web Parts”
  In this chapter, we introduce the VisualAge for C++ Web Parts beta code also known as Waikiki. We describe the extensions of the User Interface Classes Library which supports CGI programming. We demonstrate different techniques to create static and dynamic Web pages.
- Chapter 5, “Introduction to DB2 World Wide Web Connection”
  In this chapter, we describe the DB2 World Wide Web Gateway. We explain how regular HTML can contain SQL statements to quickly create Web pages with database access.

Part 2, “Case Study: Web Travel Agent”
- Chapter 6, “Analysis and Design”
  In this chapter, we describe the analysis and design steps for the Web Travel Agent application. We demonstrate how to use the Visual Modeling Technique with CRC cards, use cases, and various diagrams.
How This Book Is Organized

❑ Chapter 7, “Application in Action”
In this chapter, we navigate through some sample threads of the Web Travel Agent application. We show how a user could book a travel package from the Web browser.

❑ Chapter 8, “Implementation with VisualAge for Smalltalk”
In this chapter, we describe the implementation of the Web Travel Agent application using VisualAge for Smalltalk. We explain all major parts and their interactions with each other. All Smalltalk scripts are investigated and the generated HTML is shown.

❑ Chapter 9, “Implementation with VisualAge for C++”
This chapter describes the C++ implementation of the Web Travel Agent application using the VisualAge for C++ Web parts. An original approach is proposed for supporting session data and all the parts and their interactions are fully described.

❑ Chapter 10, “Implementation Considerations”
In this chapter, we describe some implementation issues we came across when we used the VisualAge for Smalltalk Web Connection feature.

❑ Chapter 11, “Implementation of Administrative Functions”
In this chapter, we describe the implementation of the Web Travel Agent administration functions through the use of the DB2 World Wide Web Gateway.

❑ Chapter 12, “Installation and Setup”
In this chapter, we show the necessary installation and setup steps for Web Connection, Waikiki, IBM Internet Connection Server, DB2 WWW Gateway, and the TCP/IP configuration.

❑ Chapter 13, “Packaging a Web Application”
In this chapter, we examine the VisualAge for Smalltalk and C++ packaging function for a Web application, and how to set up a runtime system for the application.

Appendixes
❑ Appendix A, “CRC Cards”
❑ Appendix B, “Database Definition”
❑ Appendix C, “Java Applet”
❑ Appendix D, “Windows 95/NT”
❑ Appendix E, “Web Connection Classes”
❑ Appendix F, “Related Publications”
ITSO on the Internet

Internet users may find additional material about new redbooks on the ITSO World Wide Web home page. Point your Web browser to the following URL:

http://www.redbooks.ibm.com

IBM internal users may also download redbooks or scan through redbook abstracts. Point your Web browser to the internal IBM Redbooks home page:

http://w3.itso.ibm.com

If you do not have World Wide Web access, you can obtain the list of all current redbooks through the Internet by anonymous FTP to:

ftp.almaden.ibm.com
cd /redbooks
get itsopub.txt

The FTP server, ftp.almaden.ibm.com, also stores the sample application from the accompanying CDROM. To retrieve the sample files, issue the following commands from the /redbooks directory:

cd SG244734
binary
get VAWWW.EXE
ascii
get VAWWW.TXT

All users of ITSO publications are encouraged to provide feedback to improve quality over time. Send questions about and feedback on redbooks to:

redbook@vnet.ibm.com or
REDBOOK at WTSCPOK or
USIB5FWN at IBMMAIL

If you find an error in the software that is supplied with this book, please send a bug report with a description of the problem to

bugs@vnet.ibm.com

To receive regular updates on new redbooks and general IBM announcements, you can subscribe to the IBM Announcement Listserver. It automatically supplies an Internet e-mail user with timely new announcement information (titles and optionally the letter or abstract) from selected categories. To get started, send an e-mail to

announce@webster.ibmlink.ibm.com
The keyword SUBSCRIBE must be the only word in the body of the e-mail; leave the subject line blank. You will receive a category form and listserv details. To immediately start your subscription to, for example, AS/400 announcements, put the words SELECT HW120 in the body of the note. On the afternoon of an announcement day, you will receive e-mail with the announcement, along with a list of newly available redbook titles. To obtain a full abstract of a particular redbook, use GET SG24xxxx in the note.

VisualAge for Smalltalk Support

VisualAge users are encouraged to use CompuServe® to ask questions about VisualAge and its features. When logged on to your CompuServe account, type GO VISUALAGE to get into the IBM Workstation Rapid Application Development (WRAD) conference. You will find sections to discuss the following and other VisualAge topics:

- VisualAge Communication/Languages
- VisualAge Database
- VisualAge AS/400 Connection
- VisualAge Web Connection

VisualAge for C++ Support

VisualAge for C++ Service and Support is staffed by developers who handle everything from how-to’s to complex technical problems. The resolution may take the form of education, a workaround, or a fix to the product (Corrective Service Diskette, CSDs). There are several ways to contact the VisualAge for C++ Service and Support department electronically:

- **CompuServe**: GO VACPP
- **Internet**
  - Anonymous FTP:
    - site: ftp.software.ibm.com
    - directory: ps/products/visualagecpp/fixtures/V35
  - World Wide Web:
    - http://www.software.ibm.com/ad/visualage_c++
- **TalkLink (Windows Selected Fixes Area)**
  - 1-800-992-4777 for information (USA and Canada)
  - VACPP CFORUM
- **Developer Connection (DevCon) CD**
  - Ordering information: 1-800-6DEVCON (USA and Canada)
About the Authors

Andi Bitterer works as a consultant for the International Technical Support Organization at the Almaden Research Center in San Jose, California. He graduated with a degree in computer science from the Technical University in Darmstadt, Germany. Andi joined IBM in 1987 and worked as an application development specialist in large customer projects for IBM Integration Systems Services and at the German AS/400 Field Support Center. Since joining the ITSO in 1994, he has taught workshops worldwide on object-oriented application development and the Internet. Andi is the author of two other VisualAge for Smalltalk books published by Prentice Hall. You can reach him by e-mail at bit@acm.org.

Marc Carrel-Billiard works as consultant for the International Support Organization in San Jose, California. He holds an engineer’s degree in computer science from the Institut d’Informatique d’Entreprise of Evry, France, and a master degree in computer science from Ecole Superieure des Telecommunications of Paris, France. Marc has 15 years of experience in the computer industry as an architect and project manager mostly in the object-oriented area. He joined IBM in 1989 and worked as an object-oriented application development specialist at the Integration System Division. In 1995, Marc joined the ITSO San Jose and has been teaching around the world workshops on C++, Java and object-oriented methodology. Marc is the author of two other VisualAge for C++ books published by Prentice Hall. His e-mail address is carrel@vnet.ibm.com.

Acknowledgments

This book would not have been possible without the unlimited efforts and support of the VisualAge product developers in Raleigh, North Carolina. Many thanks to Pat Mueller, Dave Bowman, Jenny Chow, Paul Blizniak, and Morgan Kinne, for developing the VisualAge for Smalltalk Web Connection feature; to Lou Mauget, for developing the VisualAge for C++ Web feature; to Peter Spung, VisualAge Development Manager, John Kellerman, VisualAge for Smalltalk Development Manager, Dibbe Edwards, VisualAge for Smalltalk Feature Development Manager, Sandra Thomas, Visual Builder for C++ and Java Development Manager, Ed Lynch, VisualAge for C++ Workstation Planner, Mary Anne Buttigieg, VisualAge for C++ Market Manager, and Dave Allison, VisualAge Family Market Manager, for their commitment to helping us during our project; and to Chris Bosman-Clark, IBM Milwaukee, Joe Bianco, Lockheed Martin Federal Systems, Gaithersburg, Bruno Georges, IBM EMEA ADTC, La Gaude, France, and Bernard Tsang, IBM Hong Kong, for their excellent contributions to the design and implementation of the Web Travel Agency application.
Thanks also to Tam Nguyen, IBM Santa Teresa Lab, for his support with the DB2 WWW Gateway; to Paul Buck, IBM Toronto Lab, for providing the VisualAge Web Development Directions white paper; and to Robert Berry, IBM Research Triangle Park, for letting us “reuse” sections of the WebConnection User’s Guide.

Many thanks go to the following people of the ITSO San Jose Center: Elsa Barron, Mary Comianos, Alan Tippett, and David Wray, for the great administrative support, Carmine Di Grande, ITSO San Jose Manager, for supporting the publishing efforts, Shirley Hentzell and Maggie Cutler, for the excellent editing, and Liz Rice and Christine Dorr, for the editorial assistance. Thanks also to Lou Evart, Doris Chen, and Satkartar Khalsa, Softline International, for creating the FrameMaker version of this book, and to the folks at Prentice Hall—Mike Meehan, Kerry Reardon, Lisa Iarkowski, and Jerry Votta—for publishing this book.
Contents

Foreword ................................................................. vi

Preface ................................................................. vii

How This Book Is Organized ................................. viii
ITSO on the Internet ............................................. x
VisualAge for Smalltalk Support ............................ xi
VisualAge for C++ Support .................................. xi
About the Authors ............................................... xii
Acknowledgments ............................................... xii

Figures ................................................................. xxv

Tables .................................................................. xxxiii


Chapter 1.
Introduction and Overview ................................. 3

Overview ............................................................... 4
Programming on the Web ..................................... 6
Uniform Resource Locators ................................. 7
Hypertext Transfer Protocol Header Information .... 9
Common Gateway Interface ................................. 11
Overview ............................................................... 11
CGI and Dynamic Documents ............................ 12
Uses for CGI .......................................................... 13
CGI Process .......................................................... 13

Chapter 2.
VisualAge Web Development Directions ................ 17

The Opportunity ....................................................... 17
The Challenge ....................................................... 18
Application Scenarios .......................................... 19
VisualAge Support for Web Application Models ... 21
VisualAge Support for Lotus Notes ....................... 24
Productivity and Leveraging of Assets .................. 25
## Chapter 3.
**VisualAge for Smalltalk Web Connection**  
Introduction ............................................ 30  
Web Connection Components ................................. 30  
  HTML Parts ............................................. 31  
  Common Gateway Interface Link .......................... 31  
Why Web Connection Parts Technology? .................... 32  
Web Connection Architecture ............................... 33  
  Application Flow ....................................... 33  
  URL Format ............................................ 34  
Visual Parts ............................................. 34  
  Working with Visual Parts ............................... 37  
  Building Basic Pages .................................. 38  
  Building Forms ........................................ 42  
  Coding HTML Directly .................................. 44  
  Defining Links ........................................ 44  
Nonvisual Parts ........................................ 45  
  Retrieving Form Data .................................. 45  
  Preserving Session Data ................................. 47  
  Connecting to Nonvisual Parts ........................... 49  
Common Gateway Interface Link ............................ 49  
  Running the CGI Link Server ............................ 50  
  Processing CGI Requests ............................... 52  
  Accessing Request Data ................................ 52  
  Routing Requests ...................................... 53  
  Using Multiple Pages .................................. 56  
Web Application Development Process Template .......... 57  
Considerations ........................................ 59  
  Tables with Graphics ................................... 59  
  HTML Lists ........................................... 59

## Chapter 4.
**VisualAge for C++ Web Parts**  
Design Paradigm ......................................... 63  
Parts Supported .......................................... 63  
New Parts ............................................. 65  
Parts Settings .......................................... 65  
Parts Behavior .......................................... 66  
  IApplication ........................................... 66  
  ICanvas ............................................... 66  
  ICheckBox ............................................. 66  
  ICGI .................................................. 66  
  IComboBox ............................................. 66  
  IContentViewListbox ................................. 67  
 .IContainerControl ................................. 67  
  IEntryField .......................................... 67
Chapter 5.
Introduction to DB2 WWW Connection ..................... 91

Overview ............................................... 92
How it Works ........................................... 94
DB2 World Wide Web Connection Macros ................ 94
Define Section ............................................ 95
HTML Input Section ...................................... 96
SQL Section ............................................. 96
HTML Report Section ..................................... 99
Security Considerations .................................. 100
Authentication ............................................ 100
Encryption ............................................... 100
Firewall .................................................. 101
Hidden Variables ....................................... 101

Part 2. Case Study: Web Travel Agent ..................... 103

Chapter 6.
Analysis and Design ..................................... 105

Introduction ............................................. 106
Customer Interactions .................................. 106
Travel Agent Interactions ............................... 109
VMT Overview .......................................... 110
Object-Oriented Analysis ........................................ 110
Use Cases Description ........................................ 112
CRC Cards Description ........................................ 112
Object Model Description ................................... 113
Summary of VMT Modeling Process ......................... 115
Use Cases ....................................................... 116
  Hotel Room Selection ..................................... 117
CRC Cards ....................................................... 120
User Interface Analysis .................................... 120
Dynamic Object Model ....................................... 123
  Event Trace Diagrams .................................... 123
  State Transition Diagrams ................................. 125
Static Object Model .......................................... 126
  Analysis Phase ........................................... 126
  Hotel Room Reservation Design Phase ................. 127
Database Design ............................................. 129
  Object Persistence ....................................... 130
  Mapping Object Classes to Tables .................. 131
  Defining Keys ........................................... 132
  Referential Integrity ................................... 133

Chapter 7.
Application in Action ........................................ 135
  Using the Web Travel Agent .............................. 135
  Startup ..................................................... 135
  Booking a Travel Package ................................ 137
  List Existing Packages .................................. 149
  Change Personal Information ............................ 152
  View Attractions ........................................ 154
  Exception .................................................. 157

Chapter 8.
Implementation with VisualAge for Smalltalk ............ 159
  Application Architecture ................................ 159
  VisualAge Parts .......................................... 161
    WtaWelcomeView ........................................ 161
    WtaWelcomeProcessor .................................. 167
    WtaPackageCreationController ...................... 168
    WtaPackageCreationView ................................ 170
    WtaPackageCreationProcessor ....................... 172
    WtaComponentSelectionView ............................ 175
    WtaComponentSelectionProcessor .................... 179
    WtaHotelSelectionView ................................ 182
    WtaHotelSelectionProcessor ......................... 184
    WtaFlightSelectionView ................................ 187
    WtaFlightSelectionProcessor ......................... 189
Chapter 10.
Implementation Considerations ........................................... 347

Part Naming ............................................................................ 347
Part Types ................................................................................ 349
Reuse of Web Parts .................................................................... 351
Build Composite ........................................................................ 351
Reuse Composite ....................................................................... 356
Single Row Query ...................................................................... 359

Chapter 11.
Implementation of Administrative Functions ......................... 365

Administrator Authentication .................................................. 366
Home Page for Administrative Functions ................................. 368
Complete Old Packages ............................................................ 368
Search for Customer Information ............................................. 370
Search for Package Information ................................................ 375
Reset the Whole Database ......................................................... 376
Display Database Statistics ....................................................... 376
Chapter 12.
Installation and Setup ................................................. 377

Web Connection ......................................................... 378
  Setting Up the CGI Link ............................................ 378
  Testing the Installation ............................................ 382
  Troubleshooting ...................................................... 383
  Using the CGI Link Server Interface .............................. 384
Waikiki ................................................................. 385
  Installing VisualAge for C++ for Windows .................... 385
  Installing Waikiki ................................................... 386
  Testing the Waikiki Installation .................................. 389
  Uninstalling Waikiki ............................................... 392
IBM Internet Connection Server ................................. 392
  Hardware and Software Requirements ......................... 392
  Installation .......................................................... 393
  Starting the Server ................................................ 395
  Configuring the Server ............................................. 396
  Protecting the Server .............................................. 399
DB2 World Wide Web Connection ............................... 399
  Hardware and Software Requirements ......................... 400
  Installation .......................................................... 400
  Configuration ......................................................... 402
TCP/IP Configuration ................................................ 403
  Loopback ............................................................ 404
  Loopback and Nonloopback ........................................ 405
  Web Connection Loopback Setup ................................. 405
Web Travel Agent ....................................................... 410
  Application .......................................................... 410
  VisualAge for Smalltalk .......................................... 416
  VisualAge for C++ and Waikiki ................................. 416
  Database 2 .......................................................... 416
  Internet Connection Server ....................................... 418

Chapter 13.
Packaging a Web Application ................................. 419

VisualAge for Smalltalk .......................................... 419
  Creating a Run-time Image ....................................... 420
  Setting up the Run-time System ................................. 424
VisualAge for C++ .................................................... 426
  Setting up the Run-time System ................................. 426
  Running Environment ............................................... 427
Appendix A. CRC Cards ............................................. 429

WelcomeView .................................................. 430
WelcomeProcessor ............................................. 430
PackageCreationController ................................... 430
PackageCreationView .......................................... 431
PackageCreationProcessor .................................... 431
ComponentSelectionView ...................................... 432
ComponentSelectionProcessor ................................ 432
FlightSelectionView ........................................... 433
FlightSelectionProcessor ...................................... 433
Flight .......................................................... 434
RequestedFlightSeats ........................................ 434
HotelSelectionView ............................................ 434
HotelSelectionProcessor ..................................... 435
Hotel .......................................................... 435
RequestedHotelRooms .......................................... 435
CarSelectionView ................................................ 436
CarSelectionProcessor ......................................... 436
CarClassView .................................................... 437
RequestedCar .................................................... 437
PackagePricer .................................................... 438
PackageFailedView ............................................ 438
PackagePurchaseView ......................................... 439
PackagePurchaseProcessor ................................... 439
RequestedPackage ............................................. 440
CustomerIdView ................................................ 440
CustomerIdProcessor .......................................... 440
NewCustomerIdView ............................................ 440
NewCustomerIdProcessor ...................................... 441
Committer ....................................................... 441
CreditCardInfoView ........................................... 443
PackageConfirmationView ..................................... 443
CustomerAuthenticationView .................................. 444
CustomerAuthenticationProcessor ............................ 444
PersonalInfoView ............................................... 444
PersonalInfoProcessor ........................................ 444
Customer ........................................................ 445
PackageListView ................................................ 446
PackageListProcessor ......................................... 446
PackageDetailsView ........................................... 447
AttractionView .................................................. 448
AttractionProcessor ........................................... 448
Location ......................................................... 449
AboutView ........................................................ 449
DeveloperView .................................................. 449
Appendix B. Database Definition ........................................ 451

Appendix C. Java Applet .................................................. 455

Appendix D. Windows 95/NT ............................................. 459
  Database 2 (DB2) ...................................................... 459
  Internet Connection Server ........................................ 460
  VisualAge .............................................................. 460

Appendix E. Web Connection Classes .................................. 463
  Accessing CGI Request Information ............................... 463
    Instance Methods for AbtAppBldrHtmlPage .................... 464
    Instance Methods for AbtCgiLinkTransaction .................. 464
    Instance Methods for AbtCgiLinkRequest ...................... 464
    Instance Methods for AbtCgiLinkResponse ...................... 464
  Session Data .......................................................... 464
  Generating HTML from Smalltalk .................................. 465
  Error Handling with AbtCgiLinkServer Class Methods ......... 465
  Codepage and Execution Control ................................. 466
    Class Methods for AbtCgiLinkServer ............................ 466
    Class Methods for AbtAbtBldrHtmlPage ......................... 467

Appendix F. Related Publications ..................................... 469
  International Technical Support Organization Publications .......... 469
  Redbooks on CD-ROMs .............................................. 470
  Other Publications .................................................. 470

How To Get ITSO Redbooks ........................................... 473
  How IBM Employees Can Get ITSO Redbooks .................... 473
  How Customers Can Get ITSO Redbooks .......................... 474

Special Notices ........................................................ 475

Glossary ........................................................................ 477

Abbreviations .............................................................. 495

Index ............................................................................ 497
### Figures

1. Uniform Resource Locator .................................................. 7
2. Interactions in the Application Models .................................. 20
3. Enterprise Distributed Web Model ......................................... 21
4. VisualAge Solution for Interactive and Distributed Web .......... 23
5. VisualAge Solution for Web Application Models .................... 25
6. Web Parts Placement .......................................................... 38
7. Settings: Link Specifications ............................................... 45
8. Session Data ........................................................................ 48
9. CGI Link: Menu Option .......................................................... 50
10. CGI Link: Start Server ............................................................ 50
11. CGI Link: Running (Initial State) ........................................... 51
12. CGI Link: Running (Usage Snapshot) ...................................... 51
13. AbtWebSampleChooser .......................................................... 54
14. AbtWebSampleRouter ........................................................... 56
15. New Part ........................................................................... 58
16. ExampleWebView ................................................................. 58
17. IFrameWindow Settings Notebook .......................................... 64
18. Creating the Hello World Web Page .................................... 71
19. Adding an IImgControl to the IFrameWindow ....................... 72
20. IHeadControl Settings Notebook .......................................... 73
21. The Final Hello World Web Page ........................................ 74
22. Hello World Web Page ........................................................... 75
23. CGI Program Output in a DOS Session ................................ 76
24. HelloCGI Web Page ............................................................... 77
25. Specifying Form Action and Form Method ............................ 78
26. HelloCGI Web Page Revisited ............................................. 79
27. HelloCGI Web Page Completed .......................................... 80
28. Hello World Web Page in Action ......................................... 81
29. Testing the Form from a DOS Session ................................. 82
30. Hello World Web Page Revisited ........................................ 83
31. Web Page A ....................................................................... 84
32. Router Part in Action ............................................................. 85
33. DB2 WWW Connection Product Overview ........................... 92
34. DB2 WWW Run-Time Flow Control .................................... 93
35. VMT Modeling Process .......................................................... 111
36. OMT Notation Subset ........................................................... 114
37. Sample Event Trace Diagram .............................................. 114
38. Sample State Transition Diagram ........................................ 115
39. Prototype Page for Flight Reservation ................................. 120
40. Web Travel Agent Part Hierarchy .......................................... 121
41. Event Trace Diagram for Hotel Room Selection – Analysis Phase 123
42. Event Trace Diagram for Hotel Room Selection – Design Phase 124
43. Event Trace Diagram for Hotel Room Reservation – Design Phase 125
45. State Transition Diagram of Hotel Object .............................................. 126
46. Analysis Object Model ................................................................. 127
47. Design Object Model for Hotel Room Selection .................................. 129
48. Database Design .............................................................................. 132
49. Web Travel Agent Settings ............................................................. 136
50. Local Logon ...................................................................................... 136
51. Welcome View .................................................................................. 137
52. Create a Travel Package ................................................................... 138
53. Select Package Components ............................................................ 139
54. Specify Hotel Preferences ............................................................... 140
55. Specify Flight Preferences ............................................................... 141
56. Specify Rental Car Preferences ......................................................... 142
57. Select Car Class .............................................................................. 143
58. Select Package Components (Package Complete) ....................... 144
59. Package Summary and Purchase Option ........................................... 145
60. Customer Identification ................................................................. 146
61. New Customer Personal Information .............................................. 147
62. Credit Card Information ................................................................. 148
63. Package Confirmation Details .......................................................... 149
64. Customer Authentication ............................................................... 150
65. Select Existing Package .................................................................. 151
66. Existing Package Details ................................................................ 152
67. Customer Authentication ............................................................... 153
68. Modify Personal Information ........................................................... 154
69. View Attractions ............................................................................ 155
70. Tahiti Video ..................................................................................... 156
71. Chicago Home Page ........................................................................ 156
72. Exception ......................................................................................... 157
73. WTA Application Architecture ....................................................... 160
74. Schematic Diagram of the WTA Application .................................... 162
75. Welcome to the Web Travel Agency ................................................. 163
76. WtaWelcomeView ........................................................................... 165
77. WtaWelcomeProcessor ................................................................. 167
78. WtaPackageCreationController ..................................................... 169
79. WtaPackageCreationView ............................................................... 170
80. WtaPackageCreationProcessor ...................................................... 172
81. WtaComponentSelectionView ......................................................... 176
82. WtaComponentSelectionProcessor (First Implementation) ........... 179
83. WtaComponentSelectionProcessor .................................................. 180
84. WtaHotelSelectionView .................................................................. 182
85. WtaHotelSelectionProcessor .......................................................... 184
86. WtaFlightSelectionView .................................................................. 187
87. WtaFlightSelectionProcessor .......................................................... 189
88. WtaCarSelectionView ...................................................................... 192
89. WtaCarClassView ............................................................................ 195
90. Car Image Specifications ............................................................... 197
91. WtaCarSelectionProcessor ............................................................. 198
92. WtaPackagePricer ........................................................................... 199
93. WtaPackagePurchaseView ............................................................. 203
94. WtaPackagePurchaseProcessor .......................................... 207
95. WtaPackageFailedView .................................................. 208
96. WtaCustomerIdView ....................................................... 210
97. WtaCustomerIdProcessor .................................................. 211
98. WtaCustomerAuthenticationView ....................................... 212
99. WtaCustomerAuthenticationProcessor .................................. 213
100. WtaNewCustomerIdView .................................................. 216
101. WtaNewCustomerIdProcessor ............................................. 218
102. WtaCreditCardInfoView .................................................. 221
103. WtaCommitter ............................................................ 223
104. WtaPackageConfirmationView .......................................... 227
105. WtaPackageListView ..................................................... 228
106. WtaPackageListProcessor ................................................. 230
107. WtaPackageDetailsView .................................................. 234
108. WtaPersonalInfoView ..................................................... 236
109. WtaPersonalInfoProcessor ............................................... 237
110. WtaAttractionView ....................................................... 239
111. WtaAttractionProcessor .................................................. 241
112. WtaAboutView ........................................................... 243
113. WtaDeveloperView ....................................................... 244
114. WtaDeveloperView Applet Settings .................................... 245
115. WtaExceptionView ....................................................... 245
116. Reusable Footer .......................................................... 246
117. Customer (Public Interface) .............................................. 247
118. Location (Public Interface) .............................................. 248
119. Flight (Public Interface) .................................................. 248
120. Hotel (Public Interface) .................................................. 249
121. CGI Program Flow Diagram ............................................ 253
122. Session Data Stored in the Windows Registry ....................... 256
123. WtaWelcomeView Page .................................................. 259
124. IFrameWindow Settings Notebook ..................................... 259
125. IImgcontrol Settings Notebook ....................................... 260
126. IHeadControl Settings Notebook ..................................... 261
127. IMultiCellCanvas Settings Notebook ................................ 261
128. WtaWelcomeProcessor .................................................. 263
129. Custom Logic Source Code for WtaWelcomeProcessor ............ 264
130. WtaPackageCreationView ............................................... 265
131. Session Key Promoted Feature ........................................ 266
132. ICollectionViewListBox Settings Notebook ....................... 266
133. CityLocation Settings Notebook ...................................... 267
134. WtaPackageCreationProcessor ........................................ 268
135. WtaPackageCreationProcessor-Part Interface Editor ............. 269
136. ProcessDuration Source Code ......................................... 269
137. CheckIfDataIsPassed Source Code ................................... 270
138. ProcessDateStrings Source Code ..................................... 270
139. ProcessLocationName Source Code ................................... 270
140. CheckEntry Source Code ................................................. 271
141. CheckIfDateIsValid Source Code ...................................... 271
142. CheckDaysInMonth Source Code ....................................... 272
242. WtaPackageDetailsView ................................................................. 334
243. WtaPersonalInfoView .................................................................. 336
244. WtaPersonalInfoProcessor ............................................................... 336
245. Part Interface Editor ...................................................................... 337
246. CheckInput Source Code ................................................................. 337
248. UpdateDb Source Code ................................................................. 339
249. Create Classes Window ................................................................. 340
250. Imported Table and Database Parts ................................................. 342
251. Customer Part Settings ................................................................. 343
252. HOTEL Part Settings ..................................................................... 344
253. Part Name Settings ....................................................................... 348
254. Form Data Settings ...................................................................... 348
255. Form Data Part Connection Menu ................................................... 349
256. Part Types and their Relationships ................................................ 350
257. Router Part ................................................................................. 351
258. HTML Composite ......................................................................... 352
259. Settings: Query Specifications ...................................................... 352
260. SELECT Details ........................................................................... 353
261. SQL Details .................................................................................. 353
262. Reusable Query ........................................................................... 354
263. Promote Features from PackageQuery ........................................ 355
264. Promote Features from LocationHitListTable ................................ 355
265. Add Part ....................................................................................... 356
266. Location Hit List Settings .............................................................. 357
267. HTML Page With Reused Composite ............................................. 357
268. Location Hit List ........................................................................... 357
269. New Part for Single Row Query .................................................... 359
270. SELECT Details ........................................................................... 360
271. WHERE Details (Incomplete) ........................................................ 360
272. Subquery for WHERE Clause ........................................................ 361
273. WHERE Details (Complete) ........................................................... 361
274. SQL Statement ............................................................................. 361
275. LastPackageView - Composition Editor ........................................ 362
276. LastPackageView - Web Browser .................................................. 363
277. Administrative Functions Home Page ............................................. 368
278. Complete Old Packages .............................................................. 369
279. Search for Customers .................................................................. 371
280. Search for Customers (List Found) ................................................ 372
281. Detail Information of Customer ..................................................... 373
282. Select Package ............................................................................ 375
283. Package Details ............................................................................ 376
284. Welcome Panel of Waikiki Installation Program .......................... 387
285. Choose Destination Location Panel .............................................. 387
286. Select Program Folder Panel ......................................................... 388
287. Waikiki Folder .............................................................................. 388
288. Accessing the Waikiki Folder from the Start Menu ....................... 389
289. Using Chkmod32 to List DLLs ....................................................... 390

XXX World Wide Web Programming with VisualAge for C++ and Smalltalk
290. CInput.html Form ......................................................... 390
291. Web Page Generated by COutput.exe ................................. 391
292. IBM Internet Connection Server Installation ......................... 394
293. Install Directories – Internet Connection Server ..................... 394
294. Internet Connection Server Install Configuration .................... 395
295. DB2 WWW Connection Gateway Installation ......................... 400
296. Install Directories – DB2 WWW ....................................... 401
297. TCP/IP Configuration for Loopback Interface ....................... 406
298. TCP/IP Configuration for Loopback Router .......................... 407
299. TCP/IP Configuration for Loopback Nameserver .................... 407
300. TCP/IP Configuration for LAN Interface ............................. 408
301. TCP/IP Configuration for Loopback and LAN Router .............. 408
302. TCP/IP Configuration for Loopback and LAN Nameserver .......... 409
303. IBM WebExplorer Server Configuration for Loopback ............. 409
304. IBM WebExplorer Server Configuration for Internet ............... 410
305. Welcome Panel of Web Travel Agent Installation Program ........ 411
306. Installation Type Information Panel ................................ 411
307. Destination Location Window ....................................... 412
308. Select the Components to Install .................................. 412
309. Select Program Folder Dialog Window ............................... 413
310. Web Travel Agent Folder ........................................... 413
311. WTA Directory Structure ........................................... 414
312. Database Builder .................................................. 417
313. Prompter: No Visual Parts .......................................... 420
314. Make Executable .................................................... 420
315. Prompter: Packaging Time ......................................... 421
316. Save Packaged Image .............................................. 421
317. Prompter: Existing Files ............................................ 421
318. Packaging Problems ............................................... 422
319. Packaging Messages ............................................... 422
320. Packaging Log ..................................................... 423
321. Windows NT Control Panel ........................................ 460
322. Windows NT Services ................................................ 460
323. Windows NT Database Connection Specifications .................. 461
Tables

1. HTTP Request Header Information .................................................. 9
2. Information Available in HTTP Response Headers .......................... 10
3. HTTP Application Flow ................................................................. 33
4. Visual Parts ..................................................................................... 34
5. Building the Sample Chooser ......................................................... 54
6. Building the Sample Router .......................................................... 55
7. Testing the Sample Router ............................................................. 56
8. View Descriptions .......................................................................... 122
9. Matching Table Columns and Attributes ...................................... 247
10. Visual Parts ...................................................................................... 257
11. Nonvisual Parts ............................................................................... 257
12. Push Button Names and TextAttributes ...................................... 260
13. Relational Table Identifiers ........................................................... 344
14. WTA Directory Details ................................................................. 414
15. WelcomeView ................................................................................ 430
16. WelcomeProcessor ......................................................................... 430
17. PackageCreationController ........................................................ 430
18. PackageCreationView ................................................................. 431
19. PackageCreationProcessor ........................................................ 431
20. ComponentSelectionView ............................................................. 432
21. ComponentSelectionProcessor ..................................................... 432
22. FlightSelectionView ....................................................................... 433
23. FlightSelectionProcessor ............................................................. 433
24. Flight .............................................................................................. 434
25. RequestedFlightSeats ................................................................. 434
26. HotelSelectionView ........................................................................ 434
27. HotelSelectionProcessor ............................................................. 435
28. Hotel ............................................................................................... 435
29. RequestedHotelRooms ............................................................... 436
30. CarSelectionView ........................................................................... 436
31. CarSelectionProcessor ............................................................... 436
32. CarClassView ................................................................................. 437
33. RequestedCar ............................................................................... 437
34. PackagePricer ................................................................................ 438
35. PackageFailedView ...................................................................... 438
36. PackagePurchaseView ............................................................... 439
37. PackagePurchaseProcessor ........................................................ 439
38. RequestedPackage ........................................................................ 440
39. CustomerIDView .......................................................................... 440
40. CustomerIDProcessor ............................................................... 440
41. NewCustomerIdView ...................................................................... 441
42. NewCustomerIdProcessor ........................................................... 441
43. Committer ....................................................................................... 441
44. CreditCardInfoView ...................................................................... 443
45. PackageConfirmationView .......................................................... 443
<table>
<thead>
<tr>
<th></th>
<th>Class</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>46</td>
<td>CustomerAuthenticationView</td>
<td>444</td>
</tr>
<tr>
<td>47</td>
<td>CustomerAuthenticationProcessor</td>
<td>444</td>
</tr>
<tr>
<td>48</td>
<td>PersonalInfoView</td>
<td>444</td>
</tr>
<tr>
<td>49</td>
<td>PersonalInfoProcessor</td>
<td>445</td>
</tr>
<tr>
<td>50</td>
<td>Customer</td>
<td>445</td>
</tr>
<tr>
<td>51</td>
<td>PackageListView</td>
<td>446</td>
</tr>
<tr>
<td>52</td>
<td>PackageListProcessor</td>
<td>446</td>
</tr>
<tr>
<td>53</td>
<td>PackageDetailsView</td>
<td>447</td>
</tr>
<tr>
<td>54</td>
<td>AttractionView</td>
<td>448</td>
</tr>
<tr>
<td>55</td>
<td>AttractionProcessor</td>
<td>448</td>
</tr>
<tr>
<td>56</td>
<td>Location</td>
<td>449</td>
</tr>
<tr>
<td>57</td>
<td>AboutView</td>
<td>449</td>
</tr>
<tr>
<td>58</td>
<td>DeveloperView</td>
<td>449</td>
</tr>
</tbody>
</table>
Part 1

VisualAge and the World Wide Web
Introduction and Overview

Computers are not the topic of this book. Computers are the enablers for the topics in this book: Object-oriented programming and the World Wide Web. Computers have changed the world in which they were invented. Communicator is a more appropriate name for the machines we are discussing in this book, because will be doing scarcely any computation.

This book explains how object-oriented programming makes people and the behavior modeled by computers more efficient. It focuses on the widespread and robust communication between people provided by the World Wide Web and illustrates the potential of these two technologies by bringing them together in an example of a very simple on-line travel agency. This example provides a starting point for your development of more interesting and sophisticated applications.
Overview

Object-oriented (OO) programming, like all computer programming, is an abstraction connecting how a computer works to how people think about the world. The circuitry inside the computer is programmed to simulate how people think about the workings of the world. Object-oriented programming is an explicit and consistent way of modeling the world. Data structures and routines are built by the OO application development environment; using them, you can build applications that approach what your organization expects of the system. More important, when the organization changes and the system needs to accommodate the change, you can make modification quickly and effectively. In this book, we use VisualAge for Smalltalk. Your systems can also be modified to be reused by different yet similar part of the organization.

The World Wide Web (WWW) is the shibboleth of the mid 1990s. It is seen everywhere; billboards, television commercials, and business cards give Web addresses. It is the talisman that is turning the computer into the communicator. People and machines across the planet are taking advantage of the Web, which combines many threaded connected systems to form a fabric for instantaneous global communication.

To many people, WWW is a simple computer interface that provides access to a wide variety of information. The Web’s apparent simplicity is a tribute to the work done to mask its complexities. These characteristics allow us to use the Web effectively:

- **Connectionless transport mechanism**
  
  The Transmission Control Protocol/Internet Protocol (TCP/IP) works behind the scene to take data from one machine and move it across the network to another machine. It is connectionless in that no fixed path is needed between the machines. TCP/IP allows you the equivalent of leaving a message on an answering machine rather than having to wait for a telephone conversation, in which the recipient must be on the line before you can transmit the message. The answering machine lets you talk and holds the message until the other party is ready to listen.

- **Browser**
  
  Browser programs take a well defined set of information (a file) from a computer and display it in a consistent and often esthetically pleasing fashion.

- **Tagged text**
  
  The consistent structure to the files sent to the browser program allows the storage of information in a file without knowing the kind of device on which it will be displayed.
Overview

- **Stateless connection**
  The connection to the server is stateless in that the server drops any connection to the browser after responding to a browser request. The server is also stateless, in that it remembers nothing between repeated browser requests. (The Web Connection feature extensions provided by VisualAge for Smalltalk take into account the need to store state information.)

In addition, we use a number of WWW terms with these definitions:

- **Web server**
  Web servers are responsible for servicing requests for information from Web browsers. The information can be a file retrieved from the server’s local disk, or information generated by a program called by the server to perform a specific application function.

  A number of public-domain Web servers are available for a variety of platforms including most UNIX variants, as well as personal computer environments such as OS/2 Warp and Windows NT. Some well-known servers are the CERN and NCSA servers (available on a variety of UNIX and non-UNIX platforms), and GoServe from IBM, which runs on OS/2 Warp.

  In addition, commercial Web servers are available that offer higher levels of vendor support and additional functionality. IBM has released the Internet Connection Server products, a development of the CERN implementation, for various platforms, such as AIX, OS/2 Warp, or Microsoft Windows.

- **HTTP**
  The Hypertext Transfer Protocol (HTTP) is the protocol that a Web browser uses to communicate with a Web server.

- **HTML**
  The Hypertext Markup Language (HTML) defines the format and contents of documents sent from the Web server to a Web browser for formatting and display. HTML uses tags to specify formatting and document structure and identify hypertext links. It is similar to the Standard Generalized Markup Language (SGML), the ISO standard for specifying document format and structure.

- **URL**
  The Uniform Resource Locator, or URL, is a standard naming convention for identifying a resource such as an HTML document, Gopher menu, or FTP file transfer. You use URLs within HTML documents to define hypertext links to other documents or resources.

- **Forms**
  The initial goal of the Web designers was to provide a mechanism to allow you to find and display information on the Internet. In other words, the information flow was essentially one-way (from Web
server to Web browser). Subsequently, a forms function has been added to HTML that allows you to specify input fields that enable Web browsers to be used to send data to the server. This function allows the Web browser to be used to access many different types of applications, including those that run on online transaction processing (OLTP) systems such as CICS.

- **CGI**

  The Common Gateway Interface (CGI) requests the Web server to execute a program specified by the Web browser. This differs from the more typical Web browser behavior of requesting a file. All popular Web servers today support the CGI. Web servers use CGI, for example, when displaying information from a database, it requires you to do more than simply retrieve an HTML document from a disk and send it to the Web browser. For those applications, the Web server needs to call a program to generate the HTML to be displayed.

### Programming on the Web

The World Wide Web was initially used for static information only: pre-formatted HTML files were sent to the browser for reading. While still useful, this is not the only possibility.

Most Web servers provide an interface that allows the Web server to execute a program and return the results of that program (if required) to the Web browser. The most commonly used interface is the Common Gateway Interface, an architected way of invoking programs from a Web server and returning any output from that program to the Web browser.

Allowing Web servers to invoke programs expands the WWW from data retrieval and display to a powerful and flexible method of client/server transaction processing. The Web browser user acts as a client within the programs and options provided by the browser. The Web server now works as more than a file reference. By using Web server programs and parameters in this way, you can perform such tasks as:

- Dynamically creating HTML pages in response to user input
- Performing administration functions on your Web server remotely
- Creating, updating, and deleting files on your Web server
- Executing traditional-style short-running transactions
- Issuing Structured Query Language (SQL) queries to a relational database manager (RDBM) such as DB2 or Oracle
- Accessing any application that manages shared data, such as Lotus Notes
- Using an object-oriented application through the facilities of VisualAge for Smalltalk
The documentation of VisualAge for Smalltalk, on-line and printed, explains and illustrates how to get started with object-oriented programming. We expect that you have done some of the samples provided with the product you are using. We review the Web parts for VisualAge for Smalltalk in Chapter 3, “VisualAge for Smalltalk Web Connection,” on page 29.

### Uniform Resource Locators

Uniform Resource Locators (URL) are what makes the Web special. Tagged text, connectionless protocols, and everything else associated with the Web is not as important to WWW success as the URL. A browser user begins accessing information sources with the URL; for a page of Web information other URLs provide:

- Branches into other parts of the same document
- Branches to other Web servers
- Passing of data and running of programs (our biggest interest in this book)

A URL can be divided into three distinct sections, as shown in Figure 1.

![Figure 1. Uniform Resource Locator](http://www.almaden.ibm.com/redbooks/homepage.html)

In this example,

- The scheme is `http`.
- The port is not used, as it is in the majority of URLs.
- The path is `redbooks/homepage.html`.

This example uses `http` for HTML documents, but the Web also supports other Internet protocols including `ftp`, `telnet`, `news`, and `gopher`. You can also specify an access method of `file` to allow your Web browser to access a file directly from a disk on your personal computer.

The second part, `www.almaden.ibm.com`, is the Internet address of the computer on which the data or service is located. If your domain name server (DNS) cannot resolve the name, you can also specify the TCP/IP address directly, for example `9.20.2.35`. Provision is also made to append the TCP/IP port to which the request is to be sent, for example...
www.ibm.com:80 or 9.20.2.35:80. You need to do this only if the server is not using the standard or well-known TCP/IP port assigned to that service. The standard port for HTTP is 80.

The third part, redbooks/homepage.html, specifies the name of the file or service that is being requested.

In object-oriented programming terms, the service is the beginning of the class hierarchy, the host is the class and the path is the method.

Sites that run World Wide Web servers often include www as the first part of their Internet address. Here are some examples of URLs:

  Retrieves a sound file and plays it.

  Retrieves a picture and displays it, either in a separate program or within a hypermedia document.

- file://c:/www/html
  Displays the contents of the C:\www\html directory on your own workstation.

  Connects to an HTTP server and retrieves an HTML file.

  Opens a File Transfer Protocol (FTP) connection to www.xerox.com and retrieves a text file.

- gopher://www.hcc.hawaii.edu
  Connects to the gopher at www.hcc.hawaii.edu.

- telnet://www.hcc.hawaii.edu:1234
  Opens a telnet session to www.hcc.hawaii.edu at port 1234.

- news:alt.hypertext
  Reads the latest Usenet news by connecting to a user-specified news host and returns the articles in the alt.hypertext newsgroup in hypermedia format.

Most Web browsers allow you to specify a URL and connect to that document or service. When selecting a hypertext link in an HTML document, you are actually sending a request to open a URL. In this way, hyperlinks can be made not only to other texts and media, but also to other network services. Web browsers are not simply Web clients, but can also be FTP, Gopher, and News clients.

Note

URLs can be case sensitive, depending on the file system being used.
Hypertext Transfer Protocol Header Information

Any request received from a Web browser has HTTP header information that can be very useful both for logging and auditing purposes, and for making your Web programming easier and more user-friendly. Table 1 shows the current HTTP architected headers.

Table 1. HTTP Request Header Information

<table>
<thead>
<tr>
<th>Header</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>From</td>
<td>The IP address of the submitter of this request (need not be the address of any proxy or socks server used).</td>
</tr>
<tr>
<td>Accept</td>
<td>The MIME-types accepted by the Web browser.</td>
</tr>
<tr>
<td>Accept-Encoding</td>
<td>Used to indicate whether the response is encoded in any way, for example if it is a .ZIP or compressed file.</td>
</tr>
<tr>
<td>Accept-Language</td>
<td>Indicates the preferred language.</td>
</tr>
<tr>
<td>User-Agent</td>
<td>The Web browser that issued this request (to be used in the response).</td>
</tr>
<tr>
<td>Referer</td>
<td>The URL of the document from which this request originated. Can be used to generate backward links. Very useful.</td>
</tr>
<tr>
<td>Authorization</td>
<td>Extensible header containing the name of the authorization scheme.</td>
</tr>
<tr>
<td>ChargeTo</td>
<td>As yet undefined extensible header intended to be used to carry charging information for the requested service.</td>
</tr>
<tr>
<td>If-Modified-Since</td>
<td>Used with the GET method to make it conditional. The Web server returns the requested document only if it has been updated since the date supplied.</td>
</tr>
<tr>
<td>Pragma</td>
<td>Used to return directives to the Web server. The only current architected use is the “no-cache” parameter, which tells a proxy server not to cache this document, but always get a fresh one.</td>
</tr>
</tbody>
</table>

For the full specifications for the HTTP headers, see the following URL: http://www.w3.org/hypertext/WWW/Protocols/HTTP/HTRQ_Headers.html

Table 2 shows a set of HTTP headers that can be used by the Web server when building a response, to provide information about the document being returned to the Web browser.
### Table 2. (Part 1 of 2) Information Available in HTTP Response Headers

<table>
<thead>
<tr>
<th>Header</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Allowed</td>
<td>The set of HTTP requests that the requesting user can specify for this URL, for example, GET, POST, or HEAD.</td>
</tr>
<tr>
<td>Public</td>
<td>The set of HTTP requests that any user can specify for this URL, for example, GET, POST, or HEAD.</td>
</tr>
<tr>
<td>Content-Length</td>
<td>Indicates the length of the body of the document.</td>
</tr>
<tr>
<td>Content-Type</td>
<td>Indicates the MIME-type of the body of the document.</td>
</tr>
<tr>
<td>Content-Transfer Encoding</td>
<td>Related to MIME-type. Not currently implemented.</td>
</tr>
<tr>
<td>Content-Encoding</td>
<td>Any encoding scheme used for the body of the document. The only currently recognized ones are x-compress and x=gzip.</td>
</tr>
<tr>
<td>Content-Language</td>
<td>Language in which the document is written.</td>
</tr>
<tr>
<td>Date</td>
<td>Creation date of the document.</td>
</tr>
<tr>
<td>Expires</td>
<td>Gives the date after which the document is invalid, and after which it should be retrieved again. Useful for Web servers that have caching enabled.</td>
</tr>
<tr>
<td>Last-Modified</td>
<td>Useful for living documents. The last time the document was modified.</td>
</tr>
<tr>
<td>Uniform Resource Identifier (URI)</td>
<td>This URI should resolve to a URL that can be used to retrieve the document.</td>
</tr>
<tr>
<td>Message-ID</td>
<td>Unique identifier for this document for all time. Not related to the unique identifier discussed elsewhere in this book in the context of state data.</td>
</tr>
<tr>
<td>Version</td>
<td>Indicates the version of a living document that is being sent. The format of this header has yet to be defined.</td>
</tr>
<tr>
<td>Derived-From</td>
<td>Related to Version. Indicates the source of a particular version of a document.</td>
</tr>
<tr>
<td>Cost</td>
<td>Indicates the cost of access to a copyrighted work. Format of this header is still to be decided. Currently used to implement charging schemes that are agreed upon outside the context of HTTP.</td>
</tr>
</tbody>
</table>
In this section, we explain the Common Gateway Interface, that plays an important role for all Internet application development purposes.

Overview

The Common Gateway Interface (CGI) is a standard interface, supported by almost all Web servers, that defines how information is exchanged between a Web server and an external program (CGI program).

CGI programs can be written in any language supported by the operating system on which the server is run. The language can be a programming language, like C++, or it can be a scripting language, like Perl or REXX. Programs written in programming languages need to be compiled, and typically run faster than uncompiled programs. On the other hand, those written in scripting languages tend to be easier to write, maintain, and debug.

The functions and tasks that CGI programs can perform range from the simple to the very advanced. In general, those that perform the simple tasks are called CGI scripts (because they are not compiled). Those that perform more complex tasks are often called gateway programs (because they tend to be written in a programming language). In this section, we refer to both types as CGI programs.

Given the wide choice of languages and the variety of functions, the possibilities for CGI programs seem almost endless. How you use them is up to you. Once you understand the CGI specification, you will know how servers pass input to CGI programs and how servers expect output.

<table>
<thead>
<tr>
<th>Table 2. (Part 2 of 2) Information Available in HTTP Response Headers</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Header</strong></td>
</tr>
<tr>
<td>----------------</td>
</tr>
<tr>
<td>Link</td>
</tr>
<tr>
<td>Title</td>
</tr>
</tbody>
</table>

For full details of the HTTP response header specification, see the following document:

http://www.w3.org/hypertext/WWW/Protocols/HTTP/Object_Headers.html
There are many uses for CGI programs. Basically, they are designed to handle dynamic information. Dynamic in this context refers to temporary information that is created for a one-time use and not stored anywhere on the Web. This information may be a document, an e-mail message, or the results of a conversion program.

**CGI and Dynamic Documents**

There are many types of files that exist on the Web. Primarily they fall into one of the following categories:

- Images
- Multimedia
- Programs
- HTML documents

Servers break HTML documents into two distinct types:

- Static
- Dynamic

*Static documents* exist in source form on the Web server. *Dynamic documents* are created as temporary documents to satisfy a specific, individual request.

Consider the process of *serving* these two types of documents. Responding to requests for static documents is fairly simple. For example, you access the Travel Web server to get information on hotel availability on the Hawaiian island of Maui. You click on Hotels, then on Hawaii, and finally on Maui. Each time you click on a link, the Web browser uses the URL attached to the link to request a specific document from the Web server and the server responds by sending a copy of the document to your browser.

What if you decide you want to search through the information on the Travel Web server for all documents that contain information on Maui, such as temperatures, diving spots, flight schedules, hotel listings, and restaurants. This is a more difficult request to process. This is not a request for an existing document. Instead, it is a request for a dynamically generated list of documents that meet certain criteria. This is where CGI comes in.

You can use a CGI program to parse the request, search through the documents on your Web server, and create a list with hypertext links to each of the documents that contain the specified word or string.
Uses for CGI

HTML allows you to access resources on the Internet using other protocols by specifying the protocol in the URL. One such protocol is mailto. If you code a link with mailto followed by an e-mail address, the link will result in a generic mail form.

What if you wanted your customers to provide specific information, such as how often they use the Web or how they heard about your company? Rather than using the generic mailto form, you can create a form that asks these questions and more. You can then use a CGI program to interpret the information, include it in an e-mail message, and send it to the appropriate person.

CGI programs are not limited to processing search requests and e-mail. They can be used for a wide variety of purposes. Basically, anytime you want to take input from the reader and generate a response, you can use a CGI program. The input may even be apparent to the reader. For example, many people are interested in how many other people have visited their home page. As a fun way to keep count of this, you can create a CGI program that keeps track of the number of requests for your home page and displays the new total each time someone links to your home page.

CGI Process

CGI programs are referenced from within HTML documents. In general, the HTML document defines the environment variables that specify how information is passed. When you design the layout of your document, keep in mind how a CGI program that helps the user search for data, set preferences, or add information affects the look your document. Developing the CGI program along with the HTML document will help you avoid many design mistakes.

When you fill out a form or enter a phrase in a search field and click on the submission button, the Web browser sends the request to the server in a format described as URL-encoded. In URL-encoded information:

- The URL starts with the URL of the processing program.
- Attached data, such as name=value information from a form, is appended to the URL and preceded by a question mark.
- Fields are separated by an ampersand (&).
- Space are represented by a plus sign (+).
- Special characters, such as a period or slash, are represented by a percent sign (%) followed by the ASCII hexadecimal equivalent of the symbol.
- Multiple values for a field, such as check boxes, are sent as a string separated by an ampersand (&).
When the CGI program is finished, it passes the resulting response to the Web server using standard output (stdout). The Web server interprets the response and sends it to the Web browser. If the program encounters errors, it writes error information to standard error (stderr). The Internet Connection Server writes the error information to the cgi_error log.

The response contains Multipurpose Internet Mail Extensions (MIME) headers that tell the browser how to display the returned information. The response must contain at least one MIME header (CONTENT-TYPE) and a blank line. The blank line after the MIME headers separates the headers from the content of the response. For example:

```
HTTP/1.0 200 OK
MIME-Version:1.0
Date: Monday, 25 Oct 96 13:14:15 GMT
CONTENT-TYPE:text/html
DOCUMENT_NAME:c:/tcpip/cgi-bin/form.cmd
Expires: Wednesday, 27 Oct 96 13:14:15 GMT
```

If the response is a static document, the program returns the URL of the document using the HTTP location header, followed by a blank line. For example:

```
http://www.acme.com/products.html
```

Upon receiving this information from the CGI program, the Web server will retrieve the specified document and send a copy of it to the Web browser. If the response is a dynamic document, such as a list of hypertext links to documents that meet specified criteria, the program should indicate that the response is an HTML document, using the Content-type header followed by a blank line, and then include links to the documents in HTML format. If the response is an HTML file, the program should indicate that the response is an HTML file, using the content-type header followed by a blank line, and then the document body, for example:

```
<html>
<head>
<title>Test HTML Page</title>
</head>
<body>
<h1>Variable Information</h1>
<p>Variable "var1" = 123</p>
<p>Variable "var2" = XYZ</p>
<p>Variable "var3" = 3</p>
<p>Variable "var4" = TEST value</p>
<p>Variable "pword" =</p>
<p>Variable "hidden" = Text not shown on form...</p>
<p>Variable "pushbutton" = Apply</p>
</body>
</html>
```
Most CGI programs include the following three stages:

- Parsing
- Data manipulation
- Response generation

**Parsing**

Parsing is the first stage of a CGI program. In this stage, the program takes the data in one or more of the possible formats (environment variables, command-line arguments, or standard input devices), breaks it into components, and decodes the information in the components.

For example, the following could be received using the environment variable QUERY_STRING:

```
NAME=Joe+S%E+Fox&ADDR=jfox%7Cibm.net&INTEREST=RCO
```

Parsing breaks the fields at the ampersands and decodes the ASCII hexadecimal characters. The results look like this:

```
NAME=Joe S. Fox
ADDR=jfox@ibm.net
INTEREST=RCO
```

**Data Manipulation**

Data manipulation is the second stage of a CGI program. In this stage, the program takes the parsed data and performs the appropriate action. For example, a CGI program designed to process an application form might:

1. Take the input from the parsing stage.
2. Convert abbreviations into more meaningful information.
3. Plug the information into an e-mail template.
4. Call the sendmail program.
5. Send the filled-in template to a specified e-mail address.

**Response Generation**

Response generation is the final stage of a CGI program. In this stage, the program formulates its response to the Web server, which forwards it to the Web browser. The response contains MIME headers that vary depending on the type of response. With a search, the response might be the URLs of all the documents that met the search criteria. With a request that results in e-mail, the response might be a message confirming that the e-mail was sent.
VisualAge Web Development Directions

In this chapter, we outline the current situation and future steps of the VisualAge family of products in regard to development directions for the Internet and the World Wide Web.

The Opportunity

Millions of people are now connected to each other through the Internet, creating a magnificent spectrum of new application possibilities. This paper discusses the IBM Web directions in response to the opportunity as projected by independent industry analysts, such as IDC and the META group.
IDC estimates that the 8-10 million current Web users will grow to 125-150 million by 1999. Information on almost any subject is becoming easily accessible. Dramatic changes are possible in the way people handle daily activities, such as making travel plans, making buying decisions, banking, learning new information, and accessing entertainment. This is known as electronic commerce. Likewise, businesses will benefit from an environment in which all computers can exchange information. It is the Internet that will finally give rise to information at your fingertips through an open, network computing approach—something that no single person or company can do.

With this explosion in the number of people becoming connected to the Internet, many businesses have responded immediately with a Web site to offer information about their products and services. The Gartner Group reports that the number of Web sites is doubling every 50 days. Businesses consider the Internet an opportunity to:

- Improve the way they disseminate information
- Sell their products and services.

The META Group projects that, by the year 2000, approximately 90% of the Global 2000 companies will not only be connected to the Internet but will have a Web site by the year 2000. Although businesses view the Internet as an opportunity, various problems must be solved before enterprises can fully exploit this technology.

**The Challenge**

Today there are problems conducting business over the Web. For example, most customers are reluctant to send their credit card numbers over the Internet, and businesses must have Internet security to send confidential data. They also need to be able to scale servers and bandwidth to meet potential customer demand.

Internet applications currently have unpredictable performance as a result of variations in transmission speed, network load, and server load.

The majority of Web pages are static in nature, capturing information about a business’ products and services at a particular point in time. Web page owners need to be able to update these pages in real time for such items as new prices, services, or products. Businesses must be able to solve this updating problem by deploying Web applications that are more interactive.
The Opportunity

Application Scenarios

Many businesses would benefit if prices could be updated automatically from the central database of prices, so that hourly specials could be advertised over the Internet. All would benefit if customers could be assured of getting the latest information from their inquiries. Business managers are impatient for the day when host systems with known and proven security, transaction processing, and scalability can be designed to conduct business over the Internet.

That is, businesses want solutions that:

❑ Leverage investments in their existing systems
❑ Exploit the Internet as another way to increase opportunities
❑ Quickly extend access to their current systems, such as order entry, over the Internet. This is a critical requirement.

The META Group identifies three current application models for deploying Internet applications and for connecting them into their information technology infrastructure (see Figure 2). IBM will provide solutions for all of the models identified by the META Group.

Simple Web Model

The simple Web model consists of publishing static Web pages using the Hypertext Markup Language (HTML). The Web browser opens up a connection to the Web server, and the server returns a page and closes the connection. This model is suitable for providing access to business information that is relatively stable. It originally provided the foundation for the Web. Although later enriched by extensions to HTML, such as tables and frames, the simple Web model remains a static model. The user’s only interactive options are limited to clicking on a link to visit another page or requesting a page reload.
Interactive Web Model

In the interactive Web model, pages can contain forms, fields, and buttons that allow users to enter data and choices. When the user completes the form and makes the choices, the browser opens a connection with the server to allow the data and choices to be transmitted. The Web server passes the information to a custom server program or script, which makes an inquiry or does calculations, and then passes back a new page for the browser to display. The connection is then closed.

The interactive Web model supports a simple form of client/server computing using HTTP as the middleware and the Web server’s Common Gateway Interface (CGI), which calls a custom server program or script. This model works well for simple interactions that do not require high interactivity between the client and server. However, the method is quite expensive in terms of server resources and time when establishing a connection between the browser and the server for each interaction.

Distributed Web Model

The distributed Web, also known as Internet PC (Meta Group), introduces Java™. With Java, a small application called an applet can be transmitted to the browser along with the Web page. The Java applet runs within the browser as users view the page. It can provide lively animation or sound as well as rich graphical user interface components, such as the ones usually found on windowed PC applications. The applet can display a window or be displayed in a Web browser; it can also process user input. If
server interaction is needed, the applet can open a connection to the Web server to access a server or script via the Web server’s CGI. The distributed Web model enriches the user interface and off-loads tasks from the Web server (such as parameter checking in the browser), but it is still tied to HTTP for communications. Another model, the enterprise distributed Web model, supports additional communication protocols.

**Enterprise Distributed Web Model**

IBM intends to support this additional application model (Figure 3). Enterprise distributed Web applications are true open client/server applications. The client is coded in downloadable Java, which runs in any Java-capable browser. A page containing the Java applet is downloaded by a Web server to a browser. While the applet runs in the browser, it opens its own communications session with the server. The applet also communicates with a server to provide access to databases (such as IBM’s DB2* and IMS*), CICS*, C++*, Java application servers, or SOM objects, in addition to supporting a variety of middleware, including HTTP, TCP/IP, Secure Sockets Layer (SSL), DSOM, and MQ.

![Enterprise Distributed Web Model](image)

**VisualAge Support for Web Application Models**

To support building applications for the open Web or for a company Intranet, VisualAge intends to focus on the following models:

- Interactive Web
- Distributed Web
- Enterprise distributed Web

This will allow businesses to take advantage of existing applications and enable these applications to take advantage of the Intranet and Internet to disseminate information and conduct electronic commerce.
Although VisualAge is not targeted for creating static Web pages, you can enter HTML through the VisualAge editor or any text editor, and can make HTML pages with Lotus** Inter Notes Web Publisher 4.0 or any specialized HTML page-authoring tool.

Interactive Web Support

VisualAge support is envisioned to begin with the interactive Web model. This is the first model where actual programming logic occurs, in the form of a custom program or script that is known to the Web server’s CGI. The interaction starts when a page is displayed in a browser that can accept input values. These values, in turn, are passed to the CGI program or script. The CGI program or script handles the name-value pairs passed to it by the Web server and returns an HTML page to the browser to be displayed.

With VisualAge, you are able to create CGI programs quickly. The HTML pages that are to be returned by these CGI programs are designed and created visually. Rather than hand-coding HTML, you select visual controls from a palette and then drag and drop the selected controls onto a visual composition surface. These controls will paint the page that the CGI program returns when the Web server calls it.

You can further leverage the full capability of VisualAge to connect a CGI program to enterprise data, business logic, or transactions. This reduces the need to develop HTML skills or to learn additional scripting languages, such as Perl or Tcl.

Distributed Web Support

A key advantage of the distributed Web is that you have additional application distribution and user-interface functions at your disposal. The interactive Web is primarily limited by the forms and controls that you can describe with HTML. With the distributed Web, you can introduce Java applets. You can thus use Java to present such elements as text, animation, audio, and “live” graphs in the browser. Java can also handle calculations and check values on forms running in the browser, which avoids the overhead of going back to the Web server for each calculation or check.

IBM intends to add a new set of VisualAge tools for Java development to the VisualAge family of application development solutions. These tools will let you visually construct the Java client, both visual and nonvisual aspects, and connect the client to an existing CGI program. The Java client is meant to be downloaded to run in a Web browser, such as IBM WebExplorer* or Netscape** Navigator.
Just as for C++ and Smalltalk, a Java visual builder will have palettes of visual and nonvisual parts for visually constructing a Java applet or an application. You can select, drag and drop and connect these parts with other Java parts to construct visual and nonvisual aspects of the Java program. When you complete the program, the visual builder can generate the Java code to create the Java applet.

After you create a Java applet, the next step is to connect it to a CGI program that is known to the Web server. To do this, you need an enhanced distributed builder capability to create a proxy Java part that corresponds to the CGI program. With the distributed builder, you specify the name and the input/output interfaces for the CGI program. The distributed builder, in turn, generates a proxy Java part. This proxy part is connected to the applet using the visual builder to handle communications to the CGI program. In this case, the middleware is HTTP.

The distributed Web model spans the gap between the interactive Web and the enterprise distributed Web. The optimal distributed model for a Java client does not always use the Web server for communications and uses protocols other than HTTP. This is the essence of the enterprise distributed model. As a result, the interactive Web is becoming the legacy of Web applications. The distributed Web, a rich and flexible model, is evolving into the enterprise distributed Web.

Enterprise Distributed Web Support

Just as for the distributed Web, IBM will extend the VisualAge visual construction from parts programming paradigm to visually construct the Java applet or client (both visual and nonvisual aspects) and connect that client to the enterprise. Unlike the distributed Web, the enterprise distribu-
The Opportunity

The Web will include communications using protocols other than HTTP. Using the new Java VisualAge builder technology, you will create Java data-access parts and proxy parts for distributed services. On the visual composition surface, you will connect these data-access parts and distributed proxy parts with the Java client to complete an Enterprise Distributed Web application.

The VisualAge Data Access Builder integrates an existing database scheme and automatically create Java data-access classes and parts. These classes will use Java Database Connection** (JDBC) to access remote data in DB2 databases (or any other vendor’s database that provides drivers for JDBC). Currently, Oracle** and Sybase** as well as IBM intend to support JDBC.

For the distributed Web model, the distributed builder will create proxy parts for CGI programs. For the enterprise distributed Web model, the distributed builder will create Java proxy parts for application servers, CICS and IMS transactions, or SOM objects. The distributed builder will generate the code to marshal (convert local arguments into network data and package the network data for transmission) and unmarshal data and to issue calls to the appropriate middleware for handling communications. All required code to marshal data and to handle communications will be encapsulated in the proxy.

VisualAge Support for Lotus Notes

IBM also added new components to VisualAge for Smalltalk to access Lotus Notes** databases. These components allow your Smalltalk programmers to develop applications that interoperate with Lotus Notes’ Internet applications. You can use Lotus Notes to disseminate information inside or outside of your enterprise, and you also have the additional data manipulation and team collaboration capabilities that Lotus’ SmartSuite** provides with its set of tools. Figure 5 depicts the runtime view of the VisualAge solution for the various Web application models.
The Opportunity

Productivity and Leveraging of Assets

With the VisualAge family of products, you can leverage your investment in existing production systems that are procedural applications. These products provide a safety net while distributed-object computing moves from the prototype phase to the production phase.

In summary, IBM’s VisualAge family Web directions will help you:

- Move to distributed, object-oriented (OO) applications for the Web
- Leverage your existing business systems
- Run your applications across a range of operating systems and platforms of your choice.
- Make OO programming easier and faster through VisualAge’s programming paradigm of visual construction from parts.

Figure 5. VisualAge Solution for Web Application Models
IBM is currently concentrating its efforts on delivering open, cross-platform, standards-compliant solutions to you, its customers. These industry standards include the Java language and byte-code format, as well as HTML and HTTP.

VisualAge also offers an integrated development environment of power tools. These tools include:

- An integrated, OO, program development environment
- IBM Open Class, a comprehensive library of robust reusable classes
- Visual construction from both visual and nonvisual parts
- Team programming support via TeamConnection
- Data Access Builder
- Support for building distributed-object applications

When you develop enterprise distributed Web applications with the new VisualAge for Java tools, you will use licensed Java technology to build and deploy your applications. With this Java technology, you will also be able to build downloadable Web clients that run in a browser. The Web clients will:

- Distribute workload effectively from the server to the client.
- Develop better-performing Internet applications.
- Access enterprise data and services.

These additional Java components and associated tools will enable you to rapidly build Java applications to extend and interoperate with your existing VisualAge for C++, VisualAge for COBOL, VisualAge for Smalltalk, or VisualGen, applications. As a result, you will leverage your assets by delivering applications that will help give you a competitive advantage. Not only will you be able to provide better service to your target customer markets, but you will also be able to conduct electronic commerce over the Internet.

New Internet capabilities will let your VisualAge programmers develop client/server applications that have easy-to-use Internet HTML user interfaces. These interfaces will allow authorized internal users or your customers to use their favorite Internet browsers to efficiently access data that can reside anywhere within the enterprise. For example, your VisualAge programmers will be able to create product electronic-catalog applications with rich content, such as multimedia, to market your products directly to your customers. In turn, your customers will be able to order products or request services easily.

These capabilities, along with support for Lotus Notes, will make the VisualAge family solution a premier set of products for developing Web applications. These products will provide unparalleled technology for:

- Rapid application development on multiple platforms
- Designing and delivering network-computing applications
- Using enterprise development tools.
The IBM strategy for the VisualAge family addresses customer requirements in the emerging environment of network computing. For the latest detailed, product-specific information, visit the Web sites at the following URLs:

IBM VisualAge for Smalltalk
   http://www.software.ibm.com/ad/smalltalk

IBM VisualAge for C++
   http://www.software.ibm.com/ad/visualage_c++

IBM Internet Connection Server
   http://www.ics.raleigh.ibm.com

IBM Java
   http://www.ibm.com/java

Lotus products
   http://www.lotus.com
This chapter describes the new VisualAge for Smalltalk Web Connection feature. The Web Connection joins other VisualAge features such as Reports, Communication/Transaction, native Oracle access, and Distributed Objects as a separately installable component of VisualAge available on a purchase or try-and-buy basis.

The VisualAge Web Connection feature enables you to use VisualAge for Smalltalk to build applications that are accessible through the World Wide Web. With the Web Connection feature, you use the VisualAge Composition Editor to build dynamic Web pages, much as you would build windows for an ordinary graphical user interface (GUI). The Web pages you build provide the user interface for your application, and you implement program logic by making connections and writing Smalltalk scripts, just as you would with any VisualAge application.
We first give an overview of the advantages of the Web Connection parts, followed by a description of the Web Connection parts architecture, and finally an overview of a development process using Web Connection parts, and a peek at what it would be like to book a travel package over the net with an application using this technology.

**Introduction**

The Web Connection feature expands the functionality of VisualAge for Smalltalk, already an industrial-strength, object-oriented application development environment, to include the development of WWW applications. The Web Connection parts technology consists of a set of Smalltalk classes and tools that are designed to support the development of applications that can be accessed through a WWW server, allowing client interaction though standard Web browsers.

The Web Connection parts provide:

- The capability to easily create WWW pages that utilize logic.
- Construction from parts, using visual programming and object-oriented programming techniques.
- Compatibility with existing VisualAge for Smalltalk nonvisual parts and class libraries.
- New visual and nonvisual parts that can be used to develop and process Web pages and forms.

With the Web Connection parts, any VisualAge for Smalltalk programmer can, after learning a few additional concepts, create applications that run on the Web.

**Web Connection Components**

The Web Connection feature provides two main components that make it possible to build Web-based applications with VisualAge:

- A set of new visual and nonvisual parts designed for building Web applications. The new parts are in the Web Connection category, which is added to the parts palette when you install the Web Connection feature.
- A stand-alone executable program called CGI Link. This program acts as a relay between your Web server and your VisualAge application. CGI Link must be on your HTTP server machine, but it can either be on the same machine as the VisualAge application or be on a different machine.
HTML Parts

The visual parts in the Web Connection category are replacements for the standard VisualAge visual parts. They represent the various entities and controls that can be displayed by a Web browser. You cannot use any other visual parts in the user interface of a Web application. However, you can still use any VisualAge nonvisual part in your application logic. All of the features of VisualAge, such as database access and communications, are available to your Web application.

The parts in the Web Connection category are special VisualAge parts that have the ability to translate themselves into HTML code. At run-time, instead of creating windows and user-interface controls on your screen, the parts generate HTML code that recreates your user interface in a Web browser. Incoming requests from Web browsers trigger the generation of HTML code by the application.

Because HTML and Web browsers support only certain functions, the Web Connection category contains only a limited set of user interface controls. Furthermore, when you build a Web page, you will find that the editing behavior of the Composition Editor is somewhat different from the behavior when you build an ordinary VisualAge application. Again, this is because the architecture of HTML does not support as many options for customization and precise placement. The Web Connection feature supports only what HTML supports, and it will not let you build an application that won’t work on the Web.

Common Gateway Interface Link

In order to run your Web application, you must also be running an HTTP server that supports the CGI specification. CGI is a standard protocol through which Web clients can run programs that reside on the server. Applications developed with the Web Connection feature are accessed through the CGI protocol.

CGI Link is a stand-alone executable program that is shipped along with the Web Connection feature. On your Web server, you set up the CGI Link program to be accessible through the CGI protocol. Web clients can then access your VisualAge application by submitting CGI requests, specifying CGI Link as the process to run. CGI Link then relays the client request to the VisualAge application, which processes the request and returns any results, through CGI Link, to the client.

CGI Link runs each time a new client request comes in, but the VisualAge application stays up and running all the time. This arrangement provides two advantages:
Why Web Connection Parts Technology?

- It optimizes performance. Without CGI Link, the Web server would have to start the VisualAge application with each request. This would result in unacceptable performance, because of the overhead involved in loading a Smalltalk image.

- It makes it possible to preserve state information within the VisualAge application. Because the application stays running all the time, it can store information between one client request and the next.

CGI Link communicates with the VisualAge application by way of the CGI Link Server, a process running within the VisualAge image.

Why Web Connection Parts Technology?

Many companies find that by placing pages of information on the World Wide Web that they can provide information in a more timely manner to their customers. This information may include current product specifications, marketing or support information. Additionally, some organizations have found it beneficial to use interactive technology to customize information to the needs of a specific customer, including such features as search, request, or order-processing capabilities.

Web application builders today face a number of hurdles in creating quality WWW sites. Today, application builders usually implement their Web applications on UNIX platforms. They must learn the HTML in order to develop their pages, or use an editor that can generate HTML. Their finished applications generally consist of multiple HTML files and programs that need to be installed correctly on the Web server.

Today’s Webmaster who wants to go beyond linked pages and provide an interactive application is faced with a steep learning curve. Typically, application builders must learn CGI programming concepts, and tools such as Perl or Tcl.

With the Web Connection parts, the application developer works completely within the VisualAge for Smalltalk development environment to create the Web pages, process client input, and interact with other application components such as Smalltalk objects, databases, and other existing applications. Once the application has been developed, it can be packaged into a single run-time executable which can be easily installed. Building WWW server applications is now possible without having to become an expert in HTML, CGI, Perl, or UNIX.
Web Connection Architecture

The architecture of the Web Connection parts includes a set of new visual and nonvisual parts for the VisualAge for Smalltalk parts palette, a CGI program, and a VisualAge for Smalltalk server that links to the CGI program.

Application Flow

A single request from a Web client, and its results, follow a multistep process such as that shown in Table 3.

<table>
<thead>
<tr>
<th>Step</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>The client Web browser submits a HTTP request that specifies your VisualAge Web Connection application.</td>
</tr>
<tr>
<td>2</td>
<td>The Web server (a daemon running on the specified host) receives the HTTP request from the client. The server parses the HTTP request and runs the specified program (CGI Link), passing it the additional information contained in the URL.</td>
</tr>
<tr>
<td>3</td>
<td>The CGI Link program runs. CGI Link passes the client request to the running VisualAge image, communicating through a TCP/IP socket. The VisualAge image can be on the same machine as CGI Link or on a different machine.</td>
</tr>
<tr>
<td>4</td>
<td>The CGI Link server, a process running in the VisualAge image, receives the request from CGI Link and creates a new instance of the specified VisualAge HTML part. CGI Link also parses any other data fields from the request so that they are available to the new part. Each HTML part can have a nonvisual subpart called HTML Form Data, which contains the parsed data from the request. By connecting to the attributes of the HTML Form Data part, you can access any of the data from the fields in the form.</td>
</tr>
<tr>
<td>5</td>
<td>The part generates HTML code, which is relayed back through CGI Link and the HTTP server to the client browser, which renders the page.</td>
</tr>
</tbody>
</table>

Most of this process is completely invisible to the client Web browser. From the client’s perspective, the pages generated by the VisualAge application do not appear any different from any other Web pages.
URL Format

To access your VisualAge application, the client Web browser sends an HTTP request. A user can generate such a request in any of three ways:

- Directly specifying the URL for the CGI program
- Selecting a link in an HTML document that specifies the URL for the CGI program
- Selecting the **Submit** button on an HTML fill-out form that specifies the CGI program as the action to take

In any case, the request sent by the Web browser takes the form of a URL containing the HTTP request and any data that goes along with it. The complete URL to access a VisualAge Web Connection application takes the following form:

```http://server/cgipath/cgilink/PartName```

The parts of the URL are as follows:

- **server**: The TCP/IP host name of the HTTP server.
- **cgipath**: The path to the CGI directory on the server machine (typically `cgi-bin`).
- **cgilink**: The name of the CGI Link executable program on the server (usually `abtcgil.exe` or `abtcgil`).
- **PartName**: The name of the VisualAge part to open in response.

Visual Parts

When the VisualAge for Smalltalk Web Connection feature is installed, a Web Connection category with new visual and nonvisual parts is added to the VisualAge for Smalltalk parts palette. Table 4 lists all visual parts and an HTML sample that each of them generates when it is dropped onto a Web part in the Composition Editor.

<table>
<thead>
<tr>
<th>Icon</th>
<th>Part Name</th>
<th>Generated HTML (Example)</th>
</tr>
</thead>
<tbody>
<tr>
<td>![Category Icon]</td>
<td>Category</td>
<td>The Web Connection category on the Composition Editor</td>
</tr>
</tbody>
</table>
Table 4. (Part 2 of 3) Visual Parts

<table>
<thead>
<tr>
<th>Icon</th>
<th>Part Name</th>
<th>Generated HTML (Example)</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image" alt="Text" /></td>
<td>Text</td>
<td>Plain text</td>
</tr>
<tr>
<td><img src="image" alt="Image" /></td>
<td>Image</td>
<td><code>&lt;img src=&quot;/images/mypic.gif&quot; align=&quot;MIDDLE&quot; alt=&quot;Image here&quot;&gt;</code></td>
</tr>
<tr>
<td><img src="image" alt="Line Break" /></td>
<td>Line Break</td>
<td><code>&lt;br&gt;</code></td>
</tr>
<tr>
<td><img src="image" alt="Paragraph" /></td>
<td>Paragraph</td>
<td><code>&lt;p&gt;</code></td>
</tr>
</tbody>
</table>
| ![Applet](image) | Applet | `<applet code="myapplet.class" width="200" height="10" codebase="/myhome/mydir/applets" name="HTML Applet1" align="TOP">`<br>  
`<param name="parm1" value="100" >`<br>  
`<param name="parm2" value="some text" >`<br>  
`Applet to be shown here.`<br>  
`</applet>` |
| ![Embed](image) | Embed | `<embed src="/myjava/plugin1" width="200" height="10" parm1="55">` |
| ![Table](image) | Table | `<table border="1" cellspacing="4" cellpadding="3">`<br>  
`<tr> </tr>`<br>  
`</table>` |
| ![Table Column](image) | Table Column | `<th valign="MIDDLE" align="CENTER">My Heading</th>` |
| ![Form](image) | Form | `<form method="POST" action="MyProcessorPart" target="MyFrame">`<br>  
`<input type="hidden" name="_ABT_FROM_PART" value="MyPart">`<br>  
`<input type="hidden" name="_ABT_SESSION_KEY" value="582C...8129">`<br>  
`</form>` |
| ![Push Button](image) | Push Button | `<input type="submit" name="HTML Push Button1" value="OK" >` |
| ![Check Box](image) | Check Box | `<input type="checkbox" name="HTML Check Box1" checked >` |
This section takes a look at the Web Connection visual parts and how they work, as well as how they behave differently in the Composition Editor from other VisualAge parts. With these parts, you can build application user interfaces that appear as Web pages in client browsers.

### Table 4. (Part 3 of 3) Visual Parts

<table>
<thead>
<tr>
<th>Icon</th>
<th>Part Name</th>
<th>Generated HTML (Example)</th>
</tr>
</thead>
</table>
| ![Radio Button Set Icon](image) | Radio Button Set | `<input type="radio" name="HTML Radio Button Set1" value="Apple" checked>Apple  
<input type="radio" name="HTML Radio Button Set1" value="Garlic" checked>Garlic` |
| ![Entry Field Icon](image) | Entry Field | `<input type="text" name="HTML Entry Field1" size="20" max_length="15" value="Hello world.">` |
| ![Multi-Line Entry Field Icon](image) | Multi-Line Entry Field | `<textarea name="HTML Multi-Line Entry Field1" rows="3" cols="40" wrap="VIRTUAL">The quick brown fox jumps over the lazy dog.</textarea>` |
| ![List Icon](image) | List | `<select size="3" name="HTML List1">  
<option>Ant</option>  
<option>Bee</option>  
<option>Cat</option>  
<option>Dog</option>  
</select>` |
| ![Multiple Select List Icon](image) | Multiple Select List | `<select multiple size="4" name="HTML Multiple Select List1">  
<option>John</option>  
<option>Paul</option>  
<option>George</option>  
<option>Ringo</option>  
</select>` |
| ![Drop-Down List Icon](image) | Drop-Down List | `<select name="HTML Drop Down List1">  
<option>Jazz</option>  
<option>Musical</option>  
<option>Opera</option>  
</select>` |
| ![Page Icon](image) | Page | `<html>  
<head>  
<title>Html Page</title>  
</head>  
<body>  
</body>  
</html>` |
Similarly, the next chapter explains in detail how the Web Connection nonvisual parts work, and how you use them to implement application logic.

Working with Visual Parts

Building an application with the Web Connection feature follows the same basic process as building any VisualAge application, but with some differences. Chiefly, these differences fall into the following categories:

- **A smaller set of visual parts.** In a Web application, you can use only the visual parts in the Web Connection category. These parts represent the user-interface controls and other page elements that are supported by most Web browsers.

- **Different placement behavior in the Composition Editor.** The placement and alignment of the elements on a Web page are handled by the Web browser and are not under the control of the developer. The visual parts in the Web Connection category reflect this by arranging themselves in approximately the way they would appear when rendered in a Web browser. Most of the part placement and alignment functions of the Composition Editor do not function when you edit a Web page. In addition, most of the visual parts are not resizable; generally, size is determined by the browser based upon the element content.

  Placement of the parts within the page adheres to a scheme called *flowed layout*, which mirrors the linear way in which HTML is transmitted to the browser for rendering. In flowed layout, visual parts are always arranged from upper left to lower right, as they would be rendered in a Web browser (Figure 6). This arrangement provides an approximation of how your page will look, but remember that what you see in the Composition Editor may not be the same as what your users see in their browsers.

- **Differences in application flow.** In a typical VisualAge application, events driving application execution (such as button clicks or other user actions) can happen at any time and can dynamically change what appears on the screen. A Web application, however, consists of a series of static windows, each processing the input from the previous window.

- **Differences in testing.** Because Web Connection parts do not appear on the local screen, you cannot test your application with the VisualAge **Test** button. The only way to test a Web Connection application is to access it with a Web browser. This also means that you must save the part you are editing before you can test it.
Building Basic Pages

The simplest kind of Web page—referred to here as a basic page—uses text and images to convey information. A basic page cannot have any user-interface controls; in order to use these, you must use a form. Form pages are covered in “Building Forms” on page 42. However, you might use a basic page as an introductory page with instructions, or as a menu of links that can take users to the other pages in your application.

The basic parts—parts that do not require a form—are these:

- HTML Text
- HTML Image
- HTML Line Break
- HTML Paragraph
- HTML Rule
- HTML Applet
- HTML Embed
- HTML Table
- HTML Table Column
Creating a Page

A single Web page is represented in the Composition Editor by an HTML Page part. When you create a new Web Connection part from the VisualAge Organizer, the new part automatically contains a single HTML page. You can add additional pages by selecting the HTML Page part and adding it to an empty area of the free-form surface. (See “Using Multiple Pages” on page 56 for more information about using multiple pages in a single Web Connection part.)

When a Web browser connects to your application, the HTML Page part provides all of the basic tagging required to define a page. At a minimum, this includes the following HTML elements:

```
<html>
<head>
<title>
</title>
</head>
<body>
</body>
</html>
```

This tagging by itself does not result in any visible content in the Web browser, except for the <title> element, which defines the title that appears in the browser’s title bar. To change the title of the page, edit the settings of the HTML page or directly edit the title bar.

Adding Text

Once you have created a Web page with the HTML Page part, you can add content using the other visual parts in the Web Connection category.

You can add text to the page with the HTML Text part. HTML Text represents a stream of text of any length; you can use it for just a few words (such as a heading) or for an entire paragraph. By setting attributes in the settings of HTML Text, you can specify special formatting for the text, for example:

- Heading levels
- Highlighting (italics or bold)
- Font size
- Hypertext linking
- Color

Any settings you change in the HTML Text settings apply to all of the text displayed by the part. Therefore, if you want to have a single word or a group of words that appear in a different font or have a hypertext link, those words need to be displayed by a separate HTML Text part.
For example, consider the following sentence (underlining represents a hypertext link):

The **humuhumunukunukuapuaa** is the Hawaiian state fish.

This sentence would require three separate HTML Text parts, each with different attributes:

<table>
<thead>
<tr>
<th>Text</th>
<th>Attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>The</td>
<td>Normal text</td>
</tr>
<tr>
<td>humuhumunukunukuapuaa</td>
<td>Hypertext link</td>
</tr>
<tr>
<td>is the Hawaiian state fish.</td>
<td>Normal text</td>
</tr>
</tbody>
</table>

**Note this!**

The fonts that you see in the Composition Editor at edit time do not necessarily resemble the fonts users will see at run-time. Run-time fonts are entirely under the control of the Web browser.

You can change the fonts used in the Composition Editor at edit time on the Web Connection page of the VisualAge Preferences notebook.

**Defining hypertext links.** When you specify in the settings that a text item (represented by an HTML Text part) has a link to another page, you must also specify the location of the link. You can do this in two ways:

**By part name**

You can specify the name of a Web Connection part. Use this method if the page you want to link to is another Web Connection part in the same VisualAge image.

**By URL**

You can specify the URL of the page you want to link to. You can use this method to specify any Web page, whether or not it was built with Web Connection.

**Adding Images**

In addition to text, a basic Web page can include inline graphic images. To add an image to a page, use the HTML Image part. This part represents the HTML `<img>` element.

In the settings of the HTML Image part, you must specify where the graphic you want to use for the image is located. You actually configure two different graphics or locations, one for edit-time and one for run-time:
visual parts

- **edit-time graphic**
  The graphic to display in the Composition Editor during development. This must be a graphic file located on your local machine.

  For the edit-time graphic, you can specify a file in any of the following formats:
  - PCX
  - TIF
  - BMP
  - GIF

  Specifying an edit-time image is optional and does not affect what will appear in client browsers.

- **run-time graphic**
  The graphic to specify in the generated HTML, which will be loaded by the browser. You can specify the run-time graphic in either of two ways:
  - A URL specifying the location of the graphic. If you specify a URL, the browser must have access to the location the URL describes, and the graphic file must be in a format supported by the browser.
  - A VisualAge part containing the graphic. This option makes it possible to build a VisualAge part that dynamically generates an image at run time.

  If the run-time graphic is in a file, it can be in any format supported by your client browsers. Most browsers support GIF and JPG (JPEG) formats.

  As with *HTML Text*, you can place hypertext links on an *HTML Image*. In the settings of the *HTML Image* part, you must specify a VisualAge part name or URL for the link.

**Building Reusable Composites**

The *HTML Composite* part enables you to build reusable pieces of a Web page made up of multiple Web Connection parts. Once you have built a composite, you can reuse it repeatedly on different pages, or even multiple times on the same page. A typical use for a composite would be to build a standard header or footer that you want to use on every page in order to provide a common appearance, or a common set of navigational links.

Building a composite is essentially the same as building a page. To create a new composite, follow these steps:

1. Create a new Web Connection part to contain the composite and open it for editing.
2. In the Composition Editor, delete the default *HTML Page* from the part.
3. Select *HTML Composite* from the Web Connection category. Add the *HTML Composite* to the free-form surface.
4. Use the Web Connection visual parts to build the composite just as you would build a page. You can use any of the Web Connection visual parts within a composite.
5. Save the part.

In order to use a composite in a page, select **Add Part** from the **Options** menu and specify the name of the part containing the composite. You can then place the composite on an HTML page just as you would any other Web Connection visual part.

**Building Forms**

In order to build an application interface with buttons, lists, and entry fields, you must use a form. In a Web Connection application, you build a form with the *HTML Form* part, which represents the HTML `<form>` element.

The form parts, which can only be used inside a form, are:

- HTML Push Button
- HTML Entry Field
- HTML Multiline Edit
- HTML List
- HTML Multiple Select List
- HTML Drop-down List
- HTML Check Box
- HTML Radio Button Set

In addition, you can use any of the basic parts inside a form.

**Creating a Form**

To create a new form, select the *HTML Form* part and place it within the *HTML Page* part. Remember that all Web Connection parts use flowed layout. You can have other Web Connection visual parts, and even other forms, above or below the *HTML Form* part.

When you first add the *HTML Form*, it appears as a small, empty rectangle in the page. As you add form parts to the form, the form itself will automatically expand to accommodate the parts it contains.

**Setting up the form action.** For each form, you must open the settings and define an action to take when the data on the form is submitted. To define the action to take, you must specify the location of the program or
Visual Parts

page that will process the form data. As with hypertext links, you can specify either the name of a VisualAge Web Connection part, or a fully specified URL. For a form action, the URL should point to a CGI program.

In addition to the location, you must specify the method to use when submitting the form data: GET or POST. The default is POST.

**Adding Parts to a Form**

To add a part to a form, place the part within the rectangle that defines the form’s boundary. You can place any Web Connection visual part inside a form, except for another form or page. Parts within a form observe the same flowed layout as parts outside a form. To see a visual indicator of where a part will go, hold down the left mouse button as you move the loaded pointer over the HTML Page part. When you release the mouse pointer, the part is placed in the indicated location.

The form parts represent different user-interface controls, and generally, they function like the corresponding standard VisualAge parts. However, there are some differences in the way the Web Connection parts work, and different browsers may display them differently as well.

Most of the form parts represent data entry controls, which enable users of the application to select items or type information in fields. These parts work in essentially the same way as the corresponding base VisualAge parts. When the user submits the form data for processing, each data entry part provides a single value, based upon the user’s input. Because of the nature of the Web, these values are string-based.

*Adding push buttons.* Although most of the form parts are similar to their base VisualAge counterparts, the HTML Push Button part is a special case. Within an HTML form, there are two distinct kinds of push buttons:

**Submit**  
A submit button executes the action defined for the form: it connects to the specified server program, passing in the data from all of the fields in the form. Typically, each form should have at least one submit button. You can put multiple submit buttons on the same form, but they all perform the same action; it is up to the server program to determine which button was pressed and to process the input fields accordingly.

**Reset**  
A reset button clears any information entered by the user and restores the fields to their default values. A reset button does not open a connection to the server.
In the settings of an *HTML Push Button* part, you can specify whether the push button is a submit button or a reset button. When first created at edit time, a submit button has the default label *Submit*, and a reset button has the default label *Reset*. You can change these labels by specifying a new label in the settings for the part or by directly editing the label.

**Note!**

The default labels *Submit* and *Reset* are edit-time defaults only. At run-time, the default labels for submit and reset buttons are determined by the browser. If you want to be sure of the label on a push button, explicitly define it in the settings.

**Coding HTML Directly**

In some situations, you might want your page to use HTML extensions that are not directly supported by the Web Connection parts. You can do this by using *literal text*. To use this feature, check the *Literal* check box in the settings of the *HTML Text* part.

Any text that you mark as *literal* will be passed to the client browser exactly as you enter it. This means that you can enter HTML tags into the text part, and these will be sent in line with the HTML generated by the rest of the page. In this way, you can directly code any HTML extensions that you want to use; you can also implement paragraphs of text with complex formatting in a single *HTML Text* part.

Remember, however, that any HTML tags that you code directly with literal text will not be rendered graphically in the Composition Editor. Instead, you will see the HTML tags as text, exactly as you entered them. As always, to see the final result, you must test your application with a Web browser.

**Defining Links**

Defining a link for a part is common to several different part types. You can, for example, define a text link or an image link. Both settings (see Figure 7) work the same way, that is, you can click on either object.

If you want to define a link to another part in your image, select the *Part name* radio button and enter the part’s name. Alternatively, you can link to another URL.

Select *Use session key* to add the session key to the URL specification so that the session data is maintained within the Web application. *Use session key* is available after you select *Link* and *Part name*. If you have two separate Web applications and want to link between the two applications, do not select the *Use session key* check box.
Chapter 3. VisualAge for Smalltalk Web Connection

Nonvisual Parts

Generally speaking, the program logic of a Web application consists of receiving and processing the data from a form, generating a result, and then returning an HTML data stream to the browser. The browser renders the HTML in order to display the result to the user.

The Web Connection feature provides nonvisual parts that you use to implement the function of your application. These parts make it possible for you to retrieve the data from a form, and also to route conditional requests to different pages.

Retrieving Form Data

The HTML Form Data part is a nonvisual part that holds the values entered by the user in an HTML form and submitted as part of a CGI query. These fields appear as attributes on the HTML Form Data part. When a Web Connection part is accessed by a CGI query that includes form data, these attributes provide your application with a way of accessing the data for processing.

Figure 7. Settings: Link Specifications
Form Data Attributes

In order for the HTML Form Data part to have the correct attributes, you must indicate what form the CGI query will come from. To do this, you specify the VisualAge part that defines the form. The HTML Form Data part uses the specified part as a template for its attributes.

When a user fills in a form on a page and presses a Submit button, the browser sends all of the form data to the server. On the server, the program specified by the form receives the form data by way of the CGI specification. In the case of Web Connection, CGI Link is the CGI program; instead of processing the form data itself, it sends it to the VisualAge CGI Link Server process.

The CGI Link Server receives several pieces of information from CGI Link:

- The name of the part to open in order to process this request
- The name of the part containing the form that generated this request
- The contents of the fields in the form when the request was submitted

When the CGI Link Server receives the request, it opens the part specified in the request. (This part will usually generate HTML to be sent back to the browser, but not until after the request has been fully processed.) If the part contains an HTML Form Data part that matches the incoming form data, the CGI Link Server makes the incoming form data available through the attributes and events of the HTML Form Data part.

By connecting to these attributes and events, you have access to all of the values that the user specified when submitting the request. In addition, the HTML Form Data part provides several events that you can use to trigger actions in your part.

Setting up HTML Form Data

When you build a page that processes input from a previous page, you must include an HTML Form Data part. Add the part to any empty area of the free-form surface. Then open the settings of the part and specify the name of the Web Connection part containing the form that generates requests for this page.

In a typical Web application, users move through a series of pages one after another. Each page depends upon the data from the previous page. Therefore, in the settings for HTML Form Data, you should specify whichever page is the previous page in your application design.
In some cases, a simple Web application might consist of a single page that is loaded over and over, processing new input each time. In this situation, you would configure the HTML Form Data part to point to the Web Connection part that it is a subpart of. But this is an unusual case, because in fact the previous page is the same page.

Preserving Session Data

Each time a request comes in to your application from a Web browser, VisualAge creates a new instance of your HTML page in response. This new instance exists only long enough to process the request and generate HTML in response; once the HTML has been sent back to the browser, the part goes away. This means that the subparts of a Web Connection part are not persistent; any information you store in a subpart of a Web Connection part is lost between one client request and the next.

For many applications, though, you may need to preserve some information across multiple CGI requests. For example, if your application requires users to log in, you might want to keep track of the user’s name or account information for the duration of the session. You might also want users to be able to interactively build a “shopping cart” of items selected during a session, all of which are to be processed at the end.

The CGI Link Session Data part provides a persistent location where you can store session-specific information or state data related to your application. Although the CGI Link Session Data part appears as a subpart of your HTML Page part, its value attribute persists across multiple CGI requests, for the duration of a session. The value attribute can contain any kind of Smalltalk object; by tearing off this attribute, you can access all of the attributes, actions, and events of that object.

What Defines a Session

A session is a series of requests or transactions that come from the same client, and are part of the same logical sequence.

The first time a client accesses a Web Connection part, VisualAge generates a random session key that identifies the session that has just started. The session key is then passed to the client Web browser in a hidden HTML field. When the user finishes entering data and presses a Submit button, the session key is posted back to the VisualAge application. Web Connection then uses the key to identify the latest request as belonging to the same session as the previous request (see Figure 8). In this way, Web Connection can keep track of multiple sessions from different clients at the same time.
If your page includes a CGI Link Session Data part, VisualAge creates a new object of the appropriate class each time a new session begins. The object remains valid for the duration of the session. Any subsequent requests with the same session key will have access to the same object (provided a CGI Link Session Data part is available).

A session ends when a specified amount of time has elapsed since the last request in that session. You can set the time out value in the settings of the CGI Link Session Data part.

**Setting up CGI Link Session Data**

To use session data in your application, add a CGI Link Session Data part to any empty area of the free-form surface. Add a CGI Link Session Data part to any page in your application that needs access to the session data.

After you add the CGI Link Session Data part, you must open the settings of the part in order to specify what Smalltalk class you want to use for the value attribute of the CGI Link Session Data part. For example, if you want to store a user name in the session data, you would probably specify a String. After you have specified the class, you can tear off the value attribute of CGI Link Session Data in order to get full access to the object.
Each application can have only one session data object, so you must specify the same CGI Link Session Data value class in every page that uses the session data. If you need multiple persistent Smalltalk objects, define a composite class with whatever instance variables you need. You can then use your composite class for the value of CGI Link Session Data.

Connecting to Nonvisual Parts

To implement your application’s program logic, you can make connections between the visual and nonvisual parts just as you would in any VisualAge application. However, in a Web Connection application, the connections are different from what they would be in an ordinary VisualAge application. Most notably, most of the connections will be to attributes and actions of HTML Form Data, and not to the fields and controls in your user interface.

The explanation for this lies in the way a Web application works. Each transaction in a Web application is triggered by a request from a client, but once a page is loaded, it remains static until it is replaced by the next page. Each page builds its contents based upon the information submitted on the previous page. The HTML Form Data part is the gateway through which that information is made available.

Common Gateway Interface Link

Most Web servers available today use the CGI to allow end-user applications to run on the server. When an incoming request is received from a Web server that needs service from one of these end-user applications, the Web server does the following:

- Starts the end-user application, passing the request information in environment variables or as standard input.
- Waits for the end-user application to terminate, and sends the data the application generated back to the originator of the request.

The IBM technology called CGI Link, allows end-user applications, developed with VisualAge for Smalltalk and designed for the Web, to remain constantly running. A small CGI program gathers the request information, sends it to the running program, and then waits for the application to send the generated data back. This technology allows one VisualAge for Smalltalk application to remain running on a server, instead of having to start up for each request, eliminating VisualAge for Smalltalk image and database initialization startup time.

Each Web server request that is destined for the VisualAge for Smalltalk application is handled by a separate Smalltalk process, allowing multiple requests to be processed simultaneously.
Running the CGI Link Server

The only way to test your VisualAge web parts is to start the Internet connection server and the CGI Link server program and then use a Web browser to link to your Web server system. To start the VisualAge CGI Link Server, use the menu option as it is shown in Figure 9. Alternatively, you can use the Smalltalk Tools pull-down option from the VisualAge Transcript to start the CGI Link Server.

You have to decide which TCP/IP port number the CGI Link Server should respond to (see Figure 10). The port number has to match the number in the ABTCGIL.CNF configuration file. See “Web Connection” on page 378 for instructions on setting up the CGI Link Server.

As soon as the CGI Link Server is running, you see a window that will eventually contain a list of parts that the server processed (Figure 11). For every part the server counts the hits and calculates the minimum, maxi-
mum, and average response times. To reset the server, click with the right mouse button anywhere on the list to show the context menu, and select Reset Statistics.

![Figure 11. CGI Link: Running (Initial State)](image1)

After running for a while, the CGI Link Server contains a list of processed parts from your Smalltalk image (Figure 12). The list reveals information about, for example, the most popular Web part, the part with the best or worse response time, and the total number of requests received.

![Figure 12. CGI Link: Running (Usage Snapshot)](image2)
Processing CGI Requests

In a standard Web Connection part, incoming CGI requests are processed automatically. Each request causes the part to process the incoming form data (if any) and generate HTML to be sent back to the client browser.

Not all application designs can be satisfied with this default behavior, however. The Web Connection feature provides several nonvisual parts that you can use to handle incoming CGI requests in ways other than the default. The ways you might use these parts include:

- Using incoming requests to trigger processing that takes place with each request
- Conditionally routing requests to different pages
- Accessing the CGI variables passed from the client
- Accessing data from forms not built with Web Connection

Accessing Request Data

The *CGI Link Request* part represents a CGI request from a client browser. It provides a means of directly accessing an incoming request. A Web Connection part does not have to include a *CGI Link Request* part; but if it does, the CGI Link Server makes the information from the incoming request available through the *CGI Link Request* part.

By connecting to the events and attributes of the *CGI Link Request* part, you can do the following:

- You can retrieve the CGI variables sent from the browser from the *cgi-Vars* attribute. These variables provide information such as the client and server software being used, the client host and user names, and authentication information (when available).

- You can access all of the form data sent with the request, stored in a *LookupTable* part, from the *formData* attribute. This provides an alternative to the *HTML Form Data* part for getting access to the form data. You might want to use this method if you need access to all the form data in a *LookupTable* part, or if the form submitting the request was not built with Web Connection.

- You can use the *requestReceived* event to trigger actions that you want to execute each time a request comes in. For example, you might want to write an entry to a log file when a request comes in. The *requestReceived* event fires with each request, regardless of whether the request results in HTML being generated.
Routing Requests

In some applications, you might want to conditionally route an incoming request to different pages in different situations, for example:

- If a user enters invalid data into a form, you can route the request to a page displaying an error message.
- If your application requires users to log in, you can route first-time connections to a login page.
- You can have requests routed to different pages according to which push button the user clicked on, or which selection was made in a list.

The HTML Page Wrapper part provides a way for you to implement the routing mechanism. An HTML Page Wrapper part represents another Web Connection part. If you want to route incoming requests to a different part, add an HTML Page Wrapper part for each Web Connection part you want to route requests to. In the settings for the HTML Page Wrapper, specify the VisualAge part name of the Web Connection part represented by the wrapper.

Once you have set up the HTML Page Wrapper part to represent another page, you can route the current CGI request to that page by connecting to the transferRequest action of HTML Page Wrapper. When transferRequest executes, the current CGI request is immediately transferred to the specified part. The part containing the HTML Page Wrapper does not generate HTML if the request is transferred to a different part.

You can also use the transferRequestWith: action to transfer an object along with the request. You can specify any Smalltalk object as the parameter for transferRequestWith:. This makes it possible to supply additional information about the request as you transfer it. For example, if you are transferring the request to an error page, you might also want to specify an error string that describes the error. The page receiving the request can access the transferred object through the transferredObject attribute of the CGI Link Request part.

If you transfer an incoming request to a different page through an HTML Page Wrapper, the current page generates no HTML.

Example: Page Router

This example shows how you can build a front-end application from which you can select from several Web Connection sample applications. This application consists of two parts:

- A part containing a simple form. This part displays a menu page from which users can select which example they want to see.
A router part, which does not generate any HTML itself. Instead, it receives the request from the menu part and routes the request to the appropriate sample.

To build the menu page, carry out the steps in Table 5.

<table>
<thead>
<tr>
<th>Step</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Create a new Web Connection part for the menu page. Call the part <code>AbtWebSampleChooser</code> (Figure 13).</td>
</tr>
<tr>
<td>2</td>
<td>The menu page is a straightforward Web Connection form part. In the Composition Editor, build the user interface as shown in Figure 13, using <code>HTML Form</code>, <code>HTML Text</code>, <code>HTML Rule</code>, and <code>HTML Push Button</code>.</td>
</tr>
</tbody>
</table>
| 3 | In addition to changing the labels, also change the names of the push button parts to the following:  
   - `ToDoListButton`  
   - `AddingMachineButton`  
   - `ChatButton` |
| 4 | Save the part and close the Composition Editor. |

Figure 13. `AbtWebSampleChooser`
To build the router page, follow the steps described in Table 6.

Table 6. Building the Sample Router

<table>
<thead>
<tr>
<th>Step</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Create a new Web Connection part for the router page. Call the new part <em>AbtWebSampleRouter</em>.</td>
</tr>
<tr>
<td>2</td>
<td>The router page will not generate HTML, so it does not require any Web Connection visual parts. (You can choose to include an HTML page with an error message, to be used only if a user tries to connect directly to <em>AbtWebSampleRouter</em> by URL.) In the Composition Editor, delete the default HTML Page from the free-form surface.</td>
</tr>
<tr>
<td>3</td>
<td>Add an <em>HTML Form Data</em> part to the free-form surface.</td>
</tr>
<tr>
<td>4</td>
<td>Open the settings of <em>HTML Form Data</em> and specify <em>AbtWebSampleChooser</em> as the part containing the form.</td>
</tr>
</tbody>
</table>
| 5    | Add three *HTML Page Wrapper* parts to the free-form surface. These parts represent the three samples available from the menu. Change the names of the *HTML Page Wrapper* parts to the following:  
  - *To-do List Wrapper*  
  - *Adding Machine Wrapper*  
  - *Chat Wrapper* |
| 6    | In the settings of *To-do List Wrapper*, specify *AbtWebSampleToDoList* as the part containing the page. |
| 7    | In the settings of *Adding Machine Wrapper*, specify *AbtWebSampleAddingMachine* as the part containing the page. |
| 8    | In the settings of *Chat List Wrapper*, specify *AbtWebChatList* as the part containing the page. |
| 9    | Connect the *ToDoListButton clicked* event of the *HTML Form Data* part to the *transferRequest* action of *To-do List Wrapper*. |
| 10   | Connect the *AddingMachineButton clicked* event of the *HTML Form Data* part to the *transferRequest* action of *Adding Machine Wrapper*. |
| 11   | Connect the *ChatButton clicked* event of the *HTML Form Data* part to the *transferRequest* action of *Chat Wrapper*. |
| 12   | Save the part. |

When you have finished, the *WebSampleRouter* part should look something like Figure 14 in the Composition Editor. Finally, to test your new application, follow the steps in Table 7.
Another way to conditionally return different pages is to use multiple HTML Page parts within one Web Connection part. You might consider this approach when the pages are closely related to one another and do not need to be accessed from multiple different places.

### Using Multiple Pages

Another way to conditionally return different pages is to use multiple HTML Page parts within one Web Connection part. You might consider this approach when the pages are closely related to one another and do not need to be accessed from multiple different places.
Ordinarily, a Web Connection part contains no more than one HTML Page part. Because there is only one HTML page, VisualAge automatically generates the HTML for that page when a request comes in (unless the request is transferred to another page with HTML Page Wrapper or intercepted with a CGI Link Request part).

A Web Connection part, however, can contain multiple HTML Page parts, just as a VisualAge part can have multiple windows. In this situation, you can conditionally control which of the HTML Page parts will generate HTML when the part is accessed from a browser. This makes it possible for a single Web Connection part to generate different pages in different situations.

When you build a Web Connection part with multiple pages, you must designate one of the pages as the primary part. (Again, this is comparable to the primary part of an ordinary VisualAge part.) By default, the first HTML Page added to a Web Connection part is automatically designated as the primary part. You can choose another part by selecting Make primary part from the pop-up menu of another HTML Page part.

When a request comes in for a part containing multiple pages, VisualAge automatically generates HTML for the primary part unless you specifically trigger the generateHTML action of a different part. You can do this by making connections from the events of HTML Form Data or from a script that is triggered by a CGI Link Request.

Web Application Development Process Template

The following lists shows a typical sequence of events when developing a Web application with the VisualAge for Smalltalk Web Connection feature.

- Analysis and design
  This phase proceeds with the usual object-oriented analysis and design. See Chapter 6, “Analysis and Design,” on page 105 for an overview and example using our sample application, the Web Travel Agent.

- Creating parts for each page
  Each HTML page is implemented as a separate VisualAge for Smalltalk part. Make sure you select a part type of Web Connection part when creating a new part (Figure 15).
When you create a new page, you can drop other Web parts on that page (Figure 16). You can also use existing nonvisual parts such as databases and communication protocols to provide your business logic.

Figure 16 shows the default new Web Connection part as it appears in the Composition Editor. Notice that most of the icons that control the visual appearance of parts are grayed out. This is consistent with HTML architecture, which provides limited control over how a page is displayed. The Web Connection visual parts such as HTML Line Break, HTML Paragraph, and HTML Rule allow for visual placement. The Distribute Horizontally, Distribute Vertically, and Snap To Grid icons are active, but apply only to placement of the page within the Composition Editor.
Considerations

❑ Debugging

Pages can be debugged during development. Changes made to the HTML pages or associated business logic are immediately available for testing by saving the part.

❑ Packaging

Once the application is fully developed, the one-step packager can create a minimally sized application image. This application can be placed on the Web server for immediate deployment.

Considerations

The following sections describe various topics that we discovered while deploying the application. If we hit a wall because some parts were not implemented in the feature, we describe a workaround.

Tables with Graphics

The standard HTML Table part allows the user to drop only HTML Table Column parts on it. Table columns again can contain only regular text. This is suitable for connecting a table to a torn-off result table of a Query part. This is also what the Quick HTML function of the result table generates for you when dropped on a HTML page. If you need to include parts in a table that are not text, such as buttons, check boxes or images, you need to create the table environment yourself, by assembling text parts with the Literal flag turned on. For example, use regular text parts to build the \texttt{<TABLE> <TR> <TD> </TD> </TABLE>} structure, then place the images between the appropriate text parts. Because the literal text parts will be interpreted at run time, what you see in the Composition Editor is not really what will be shown on the browser. Also, if you have fairly large graphics to include in the table, we suggest that you create thumbnails of the images and use those as the edit-time graphic, otherwise your Web page will be difficult to handle.

If you need to include buttons or check boxes in the table, make sure that all table-generating parts are placed within an HTML form.

HTML Lists

The current version of the VisualAge for Smalltalk Web Connection parts does not support ordered and unordered lists nor definition lists. If you need to include lists of type \texttt{<OL> \<UL> or \<DL>}, create the list structure again with literal text parts, then use whatever part you need to build the list.
This chapter describes the VisualAge for C++ Web parts beta code known as Waikiki. The packaging and the final functionality of this VisualAge for C++ extension has not yet been determined, but IBM’s intend is to distribute it with the new version of VisualAge for C++ for Windows and for OS/2.

Like the VisualAge for Smalltalk Web Connection feature, Waikiki enables you to use VisualAge for C++ to build applications that are accessible through the World Wide Web. With Waikiki, you build static or dynamic Web pages by using the Visual Builder, in the same way you build windows for a standard GUI. The Visual Builder can generate both C++ source code and HTML code for the pages you design. The C++ source code can be compiled into an executable Common Gateway Interface (CGI) program. Of course, you still can use the “building from parts” paradigm to construct your program logic by wiring visual and nonvisual parts with connections, just as you would with any VisualAge for C++ application.
In the sections that follow, we explain the advantages of using Waikiki to program your Web applications. We describe the Waikiki components and the different visual and nonvisual parts for building your CGI programs. We conclude with an overview of the development process using Waikiki, focusing on the differences between it and the VisualAge Web Connection feature.

**Disclaimer**

Waikiki is provided as a technology preview only. It is highly recommended not to develop production applications with Waikiki, since its features may be different at shipment date.

**Introduction**

Waikiki, an update of the IBM Open Class user interface classes and the Visual Builder, enables you to build CGI applications that run on a Web server. A selection of GUI parts emits C++ source code that, when compiled to an executable file, emits HTML to be displayed on the Web browser of a Web client. Because Waikiki relies on an extension of the Open Class user interface classes, you can easily transform a standard OS/2 or Windows application to a Web application without too many changes: The GUI presentation is shifted in time and space, but the nonvisual parts are unchanged. Connections work as they do with a GUI application.

Server-side Web applications connect a browser HTML page to a heavy-duty data action such as a database query. The browser sends a form filled with query parameters to the server application. The server sends back the query result to the browser as a custom HTML page generated on-the-fly. The most common API for these applications is the CGI. Most serious servers support it on a variety of platforms.

The Visual Builder enables you to design and maintain both the custom HTML logic and the application logic. Web applications typically “change with the wind.” The Visual Builder component, which uses custom user interface parts and the building for parts paradigm, is a definite solution to this problem.

The user interface parts typically are the same parts that are used in the GUI version of VisualAge for C++. Some novel mappings to HTML occur, such as that of the INoteBook part, which has no direct equivalent in HTML. Waikiki targets Windows NT and Windows 95. An OS/2 version is also planned.
Design Paradigm

You drop part icons on the surface of the Visual Builder Composition Editor to compose a new visual part. The part is instantiated and runs while visible on the composition surface. The same Open Class library code that will be invoked in the application is run by the Visual Builder in the Composition Editor. When you generate the part, the Visual Builder stores C++ source code that will instantiate the Open Class parts as members of the new composite part.

The same action is carried out in the Waikiki version of the tool. However, the resultant C++ source code must produce HTML instead of windows on the screen. The C++ source code may be exactly the same. The difference is in the library code that is used at run time. Thus, a given part can be compiled as a GUI program and then recompiled as a CGI program that generates HTML. The VBWEB environment variable causes the build process to incorporate the CGI version of the Open Class library.

When the Visual Builder generates the C++ source code, it also generates a make file to compile your CGI program. Currently, you must manually remove the /PMTYPE:PM flag from the make file to prevent the linker from building a GUI executable.

All available nonvisual parts work as they do with any VisualAge for C++ application. Visual Builder connections fire as they do with a standard GUI application.

The Waikiki design paradigm maps a subset of visual part behaviors to corresponding behaviors in HTML. Therefore some HTML tags are not mapped, and others are mapped in a richer way. Some of the missing tags have new Open Class parts created for them so they are covered. Other tags turn out to be less important within the framework of Waikiki. For these tags, escape mechanisms are provided to enable you to supply HTML tag information.

Parts Supported

The following nonvisual and visual parts have been extended to support an HTML mapping:

- IApplication
- ICanvas
- ICheckBox
- IComboBox
- ICollectionViewListbox
- IContainerControl
- IEntryField
- IFrameWindow
- IGroupBox
Extending these parts implies extending the Visual Builder to manipulate them and extending the Open Class library itself to generate HTML flow at run time instead of GUI code. Basically, the extensions for these parts stand out when you look at their settings notebook. Figure 17 shows the settings notebook for an IFrameWindow part. The two extra Form and Web tabs in the notebook let you specify specific parameters for your Web application.

In this example, the IFrameWindow is mapped to an HTML page, and we provide a background image to this page by filling in the User HTML Attributes entry field. In “Parts Behavior” on page 66, we detail the mapping of each visual part.
New Parts

Some HTML tags cannot be associated with any visual parts, so three new parts are provided:

- ICGI
- IHeadControl
- IImgControl

The ICGI part parses a CGI command line invocation, the IHeadControl part provides an equivalent to the <HEAD> HTML tag, and the IImgControl part provides an equivalent to the <IMG> HTML tag. We describe each of these parts in the next section.

Parts Settings

All supported visual controls contain a Web tab on the settings notebook that you can use to set up Web common attributes:

- **HREF URL** defines a hyperlink to a URL.
- **Anchor Name** controls a hyperlink target name. Defaults if blank.
- **HTML Name** defines the value put into the HTML Name attribute.
- **HTML Value** defines the value put into the HTML Value attribute.
- **HTML user attributes** enables you to define any additional HTML that makes sense for the tag, for example, BGCOLOR="#FFFF80" on an IFrameWindow.

The list controls have the **By Value** attribute, which returns the value of the selection instead of the row number.

The list controls are:

- IComboBox
- ICollectionViewListbox
- IListBox

The table-generating controls have the **Has border** attribute, which enables an HTML border.

The table-generating controls are:

- IMultiCellCanvas
- ISetCanvas
- INotebook
Parts Behavior

In this section we detail the mapping of each visual and nonvisual parts.

IApplication

The IApplication nonvisual part is a singleton used only in the generated main function of your CGI program. When its run method is called in the main function, HTML is emitted by walking the tree child windows anchored by the instance of IFrameWindow in the main function.

ICanvas

The ICanvas part covers the client area when a new IFrameWindow-based part is built in the Composition Editor. It acts as a container for other parts. You can drop other supported HTML parts on an ICanvas, but they will not keep their positions in the generated HTML flow. The browser will reformat them to appear in one line unless you supply a <BR> tag through an IStaticText part with the asIs attribute checked. In short, ICanvas is useful only in simple HTML pages or as a cell in some other canvas.

ICheckBox

The ICheckBox part maps to an HTML <INPUT TYPE=CHECKBOX ...> tag. Because this tag has no label, the text attribute of ICheckBox is emitted as open HTML text after the tag. If the text attribute is empty, the label text is not emitted.

ICGI

The CGI specification defines an interface between the programs you develop and the Web server. Parameter information from the requesting browser flows into the interface. HTML and response headers flow the other way. The ICGI nonvisual part allows visually built parts to talk through this interface. It internally uses Taligent WebRunner Server Works© classes to carry out its task.

IComboBox

The IComboBox part emits <INPUT TYPE=SELECT SIZE=1> and <OPTION...> HTML tags. Data can be placed into the combo box at design time or by program logic. The data items map to HTML <OPTION...> tags.
**ICollectionViewListbox**

The `ICollectionViewListbox` part emits `<INPUT TYPE=SELECT>` and `<OPTION..>` HTML tags populated with a collection.

**IContainerControl**

Only the details view of the `IContainerControl` part is supported. The details view is simulated with an HTML table with radio buttons if it is on a form or with static texts if it is not in a form.

**IEntryField**

An `IEntryField` part maps to an HTML `<INPUT TYPE=TEXT>` or `<INPUT TYPE=HIDDEN>` tag according to the visibility setting of the part.

**IFrameWindow**

The `IFrameWindow` is the root of the tree of windows. It is usually instantiated by the generated main function. Be aware that frame extensions such as menus or toolbars are not supported in HTML.

**IGroupBox**

The `IGroupBox` part has no direct counterpart in HTML. Group boxes are used to visually frame a group of window controls and provide a label. The HTML version of this part only provides the label. It maps to an HTML `<H4>` tag pair.

**IHeadControl**

There was nothing in the Open Class libraries to map to the HTML `<Hx>` tag pairs, so `IHeadControl` was created. It has a header level attribute on the settings page to allow production of HTML heading tags at the corresponding levels ranging from H1 to H6.

**IImgControl**

There was nothing in the Open Class libraries to map an HTML `<IMG ...>` tag, which names a file or URL in HTML. The `IImgControl` part was created to show a visual icon for dropping on the composition surface. In its CGI guise, it emits the correct HTML tag, naming the file or URL where
Parts Behavior

the image resides. Any file (usually a GIF or JPEG file) supported by the browser can be used. Be aware that the image is not rendered on the composition surface at this time. An icon placeholder appears instead.

IListBox

An IListBox part emits HTML <INPUT TYPE=SELECT> and <OPTION..> tags. Operation is much like that of the IComboBox part.

IMessageBox

The IMessageBox part is used to report exceptions or errors in HTML. It is the only part that emits its own complete HTML page. When a connection fires this part, the IMessageBox HTML is sent to the browser instead of the page that was being built.

IMle

The IMle part emits an HTML <TEXTAREA> tag pair. The width and height in pixels are translated to approximate characters and rows for HTML. If one or both of these attributes is missing, a default value is provided, and it is the same for either attribute.

IMultiCellCanvas

The IMultiCellCanvas part is the workhorse composition control for HTML. The HTML tag language is a document markup language, not a page description language. Therefore, without using cascading style sheets, there is not a precise way to position document elements on the browser screen. A Web author’s workaround is to use HTML tables. IMultiCellCanvas maps to an HTML table. The spanning attributes can be used to modify the arrangement. Borders are an optional setting. The resultant sizes are not WYSIWYG, but everything appears in the proper relative positions. An IMultiCellCanvas part can contain other canvases or other controls. The FORM and the ACTION tags must be specified to generate an HTML FORM tag pair. The FORM tag pair enables any contained HTML input controls in the browser.

INotebook

An INotebook part emits a complex hyperlinked set of tables. Each page of the notebook is a separate table. Borders can be made optional by setting the border attribute in the Web tab of the settings notebook page. Each tab of the notebook becomes a hyperlink to the previous or next table.
**IPushButton**

An *IPushButton* part is mapped to an HTML `<INPUT TYPE=SUBMIT>` or `<INPUT TYPE=RESET>` tag. The latter results when the button is set to be a cancel button.

**IRadioButton**

An *IRadioButton* part maps to an HTML `<INPUT TYPE=RADIO...>` tag. Because this tag has no label, the `text` attribute of *ICheckBox* is emitted as open HTML text after the tag. If the `text` attribute is empty, the label text is not emitted.

**ISetCanvas**

The *ISetCanvas* part is used to lay out contained controls in a one- or two-dimensional pattern. A common use is to align a set of *IPushButton* controls in a row or column. The HTML version of *ISetCanvas* uses an HTML table to provide the constraints. An *ISetCanvas* usually is placed into a cell of an *IMultiCellCanvas*. If an *ISetCanvas* contains any HTML input controls, such as *IPushButton*, it must have either the FORM or ACTION tags specified on the outer canvas.

**ISplitCanvas**

The *ISplitCanvas* part is the same as an *ICanvas* control. There is no split bar. This behavior is “sane degradation” so that the part will work.

**IStaticText**

The *IStaticText* part maps to HTML open text. The Taligent WebRunner Server Works® takes care of encoding special characters as required by the CGI.

**Working with the Web Parts for C++**

In the sections that follow, we provide a sneak preview of how to use the Web parts for C++, before you start building the Web Travel Agent Web application in Chapter 9, “Implementation with VisualAge for C++,” on page 251. Refer to Chapter 12, “Installation and Setup,” on page 377 if you need information about installing VisualAge for C++ or Waikiki, which are both provided on the CDROMs that accompany this book.
First we show you how to use visual parts to build static and dynamic Web pages. We also show you how to build forms that can accept input from Web surfers. We then show you how to use the ICGL nonvisual part to retrieve the Web surfer input and explain how to build a router page that generates different Web pages upon a Web surfer request.

Building Basic Pages

You build a Web application using the Web parts for C++ in the same way you build a VisualAge for C++ application. However consider these differences:

- **A smaller set of visual parts.** When you design your Web pages, you cannot use all of the visual parts available from the palette. You can only use the parts described in "Parts Supported" on page 63 and "New Parts" on page 65 can be used. You cannot use window frame extensions, for example.

- **Different placement behavior in the Composition Editor.** When you place parts on an ICanvas part to compose a Web page, you cannot expect to get the same placement when the Web page is generated either statically or dynamically. The reason lies in the way a browser renders an HTML page. Unless you use an IMultiCellCanvas to place parts in cells, the browser always displays your parts in a flowed layout. In a flowed layout, the parts are arranged from upper left to lower right. Therefore, do not expect WYSIWYG behavior from the Composition Editor regarding the parts you place in your Web page. Also the order in which you place your parts on your page controls the order in which they are displayed in the Web browser. Hence, you can place an IEntryField on an ICanvas and then place an IStaticText above the IEntryField, but the Web browser will display the IEntryField first and then the IStaticText underneath. To rearrange the order in which you placed your parts on the ICanvas, you must use the Visual Parts List option of the Composition Editor and drag and drop the parts.

- **Differences in application flow.** A Web application consists of a set of static windows that process input from previous windows. This rather static flow, which can be compared to usual standard processing applications, differs from event-driven VisualAge for C++ applications where Web surfers can access simultaneously several windows at the same time and switch from one to another in the same application.

- **Differences in testing.** Because an executable file compiled from the C++ source code generated by Waikiki emits an HTML flow on your computer standard output, you have two ways to test your CGI program:
➤ You can redirect the HTML flow to a file and load the file in your favorite Web browser.

➤ You can execute your program directly from your Web browser as long as you have set up a Web server. In effect you need a Web server to support the CGI protocol.

Now that you are aware of the differences between developing a Web application and developing a standard VisualAge for C++ application let us start to create our first Web page.

**Designing the Hello World Page**

The page you are about to create is simple. It consists of a logo and a simple text. In fact, you can consider this page as the famous Hello World program adapted to Internet programming.

To build our Hello World Web page, start the Waikiki Composition Editor by accessing the program from the Windows start menu.

**Note:** Do not start the Composition Editor from the VisualAge for C++ folder. Waikiki uses a different Composition Editor that supports the extended Open Class Library parts.

Create a new visual part by selecting Part→New from the Visual Builder main window and fill in the window options as shown in Figure 18.

![Figure 18. Creating the Hello World Web Page](image)

When you click the Open push button, the Composition Editor displays the IFrameWindow you just created. This IFrameWindow is mapped to the Web page when the code is generated.

To edit the IFrameWindow title, hold down the Alt key while selecting the title of the IFrameWindow. You can modify the title by typing: Hello World. Click on the free-form surface to commit your change.
To add a graphic to your Web page, select **Options**→**Add Part**... from the Composition Editor menu bar. Fill in the Part class entry field with *IImgControl* and provide a name. Click on the **Add** push button and drop the *IImgControl* part on the upper left of the *IFrameWindow* (Figure 19).

![Figure 19. Adding an IImgControl to the IFrameWindow](image)

Double-click the *IImgControl* part to display its settings notebook and select the Web tab. In the Image URL entry field, enter the URL of your image: `/vbbase.gif` (Figure 20). Validate the modification and close the notebook by clicking the **OK** push button. Note that the *vbbase.gif* file must be located in the default directory where your Web server access its Web pages. Refer to Chapter 12, “Installation and Setup,” on page 377 for more information about installing and configuring your Web server.
Let us now add a header to our page by selecting **Options → Add Part...** from the Composition Editor menu bar. Fill in the Part class entry field with **IHeadControl** and provide a name. Click the **Add** push button and drop the **IHeadControl** part underneath the **IImgControl** part. Double-click the **IHeadControl** to open its settings notebook. Change its **Text** attribute to “Hello World!” by changing the content of the Text entry field (Figure 21). Validate your change by clicking the **OK** push button.

---

**Figure 20. IImgControl Settings Notebook**

**Figure 21. IHeadControl Settings Notebook**
Once you modify the Text attribute of the IHeadControl part and commit your change by closing the notebook, your Web page should look like the Web page in Figure 22.

![Figure 22. The Final Hello World Web Page](image)

**Generating the Source Code**

It is time to generate the source code for your Web page. From the Composition Editor menu bar select File→Save and generate→Part source. Select also File→Save and generate→main() for part to produce the main entry point and the make file of the CGI program. Different source codes are generated:

- **HTML documentation source code** is generated if the VBGENDOC environment variable is set to ALL. By default this variable is set when you run the Visual Builder. You can check it out by opening the Windows property sheet of the Waikiki Visual Builder. In our case an `hllopage.htm` is generating in your working directory. It contains a reference list to the part you are using when designing the Hello World Web page.

- **HTML page source code** is generated if the VBGENCODE environment variable is set to ALL. By default this variable is set when you run the Visual Builder. A `hllopage.html` file is produced in your working directory. This file is the HTML source code of your page generated by mapping the different parts to their HTML tag equivalents.

- **C++ source code** is always generated. When compiled with the generated make file, it produces an executable file that generates dynamically the HTML page source code for the Hello World Web page.
compile your C++ source code, access the Waikiki submenu from the Windows start menu. Select the DOS command line session. In the DOS session change to the directory where your source code is located and issue this command: \texttt{nmake hillopage.mak}. After compiling and linking your code, you should get your CGI program: \texttt{hillopage.exe}.

\textbf{Testing the Hello World Web Page}

To test our Hello World Web page, two options are possible depending on whether or not you are running a Web server on your machine.

\textit{Loading the page in a Web browser.} You can directly load your HTML Web page in your favorite browser by simply opening the \texttt{hillopage.html} created by the Visual Builder (Figure 23).

\textit{Running the CGI program.} To run the \texttt{hillopage.exe} CGI program from your browser, access a Web server and place the CGI program in the appropriate server’s directory. For more information, refer to Chapter 12, “Installation and Setup,” on page 377 where we describe how to set up a Web server. When your Web server is set up, start your browser and enter the URL to access your program (for example, \texttt{http://hostname/cgi-bin/hillopage.exe}). The browser should display the Hello World page.

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{hillopage.png}
\caption{Hello World Web Page}
\end{figure}
If you want to check out the HTML page generated by your CGI program without setting up a Web server, you can run the program directly in a DOS session and redirect the standard output to a file that you can later load in your browser. Figure 24 shows the HTML flow produced by *hellopage.exe* run from a DOS session.

![Figure 24. CGI Program Output in a DOS Session](image)

In the next section you are going to beef up your simple Hello world Web page. You modify the Web page to accept input from a Web surfer and you provide a submit button to display in a second page the data the Web surfer entered.

### Building Forms

Let us modify the Hello World Web page to let the Web surfer enter his name. Make sure the *hello.vbb* file is loaded in your Visual Builder. If the file is not loaded, select the **File**→**Load...** option from the menu bar of the Visual Builder main window. Once the file is loaded, open the HelloPage visual part by double-clicking its name in the Visual Parts list box of the Visual Builder main window. Modify the *HelloPage* part in the Composition Editor as follows:

1. Add an *IStaticText* part underneath the *IHeadControl* and change its *text* attribute to *name*.
2. Add an *IEntryField* part next to the *IEntryField* and change its name to *name*.
3. Add an *IPushButton* part underneath the *IEntryField* and change its label to *Submit*.

Once you have applied these changes, your Hello World page should look like that shown in Figure 25.
Now that your page defines an HTML form, you have to specify its form action and its form method. In short, the action method is the CGI program that the Web server calls when the user submits the form by clicking the Submit push button. The CGI specification defines two methods of accessing the form: GET and POST. The method used depends on how the encoded data retrieved from the form is transmitted to the CGI program. If the GET method is used, the CGI program receives the encoded form input in the QUERY_STRING environment variable. If the POST method is used, the CGI program receives the encoded form input on the standard input. Although Waikiki supports both methods, it is preferable to use the POST method. For more information about CGI programming, and specifying form action and form method, refer to the bibliography.

Double-click the ICanvas inside the IFrameWindow to open its settings notebook. Click the Form tab to select the Form page. Fill in the Form Action entry field with /cgi-bin/hellocgi.exe and the Form Method with POST (Figure 26). Click on the OK push button to validate your entry and close the notebook.

The form action refers to the CGI program you build in the next section to generate a Web page that displays the user input.
Figure 26. Specifying Form Action and Form Method

You can save and generate the source code for your Web page. Only the HTML source is used for our sample because this page does not change dynamically. The new page you are about to build must be generated dynamically by a CGI program to display the user’s name.

Using CGI to Read a Form

In this section, you build another Web page, which is generated dynamically by a CGI program. This page displays the name the user types in the Hello World page before submitting the form. This time you use the C++ generated source code, which you compile to produce a CGI program. The CGI programs receives from the standard input the form content and dynamically generates an HTML flow, using this input, on the standard output.

Designing the HelloCGI Page

To build the HelloCGI page create a visual part, HelloCGI, which inherits from an IFrameWindow. This time you are going to use an IMultiCellCanvas to lay out your parts:

1. Change the window title to your liking.
2. Click inside the IFrameWindow and press the Delete key to delete the standard ICanvas provided by default.
3. Select an IMultiCellCanvas from the parts palette and drop it inside the IFrameWindow.
4. Drop an IStaticText part in the second line, first column of the IMultiCellCanvas. Change its text attribute to Hello.
5. Drop an IHeadControl part in the fourth line, third column of the IMultiCellCanvas. This IHeadControl will later receive the name retrieved from the form.

After you have completed Steps 1 through 5, your Web page should look like that shown in Figure 27.

![Figure 27. HelloCGI Web Page](image)

### Reading the Encoded Form Input

To read the encoded form input from the standard input, you use the ICGI part. This part sports many methods, but the method you will use most is IString get(IString key). This method takes a string key as a parameter and returns the key value. The key name is the name of the input parameter sent by the form when the user clicks the Submit push button of your Hello World page. Because the input parameter name is also the name of the input control (the IEntryField) that receives the user’s name, it is important that you pay attention to the naming convention of all your input controls. In short, you must use the name of each input control as the parameter of the get method to read its content.

When the form containing the input controls is part of a static Web page, you can assign names to the controls by changing the names of the parts in the Composition Editor. When the form is part of a dynamically generated Web page, however, you must set the control names by changing their HTML Name attribute from their settings notebook.
To complete the part, follow these step-by-step instructions (see Figure 28):

1. Add an ICGI part on the free-form surface of the Composition Editor.
2. Create an event-to-action connection by connecting the ready event from the free-form surface to the get action of the ICGI part.
3. Double-click the connection to open its settings window.
4. Click the Set parameters button of the settings window to provide the key name. A notebook with a single page is displayed.
5. Enter name in the key entry field of the notebook page and click the OK push button to close the notebook.
6. Click the OK push button of the settings window to close it.
7. Create an attribute-to-attribute connection by connecting the action-Result attribute from the connection to the text attribute of the IHeadControl part.

Figure 28. HelloCGI Web Page Completed

Save and generate the source code and the make file. The compile and link edit your code to create your CGI program.

When the user enters his or her name in the Hello World Web page and clicks the Submit push button, a call to the HelloCGI program is sent to the Web server with the input parameter name and its value. When the CGI program starts, the event-to-action connection is fired, and the get method of the ICGI part is called with the “name” parameter as the key name. The value entered by the user is retrieved from the encoded form input and transmitted by the attribute-to-attribute connection to the IHeadControl part.
Testing the HelloWorld Form Page

Two options are available for testing your form. The option you choose depends on whether or not you are running a Web server on your machine.

Testing the Form in a Web Browser. If you can access a Web server, copy your Hello World Web page to its appropriate directory (typically the HTML directory). Copy the CGI program to its appropriate directory (typically the CGI-BIN directory. Refer to Chapter 12, “Installation and Setup,” on page 377 for setting up your Web server). Start your favorite Web browser and access your Hello World Web page. Enter a user’s name and submit the form by pressing the Submit push button. After a little while, the browser should display a new page with the name you just entered (Figure 30).

Testing the CGI Program in a DOS Session. If you do not have access to a Web server, you can run the hellocgi.exe CGI program from a DOS session. You can provide the user input using either the standard input directly from the command line or an intermediate file. To test your program from a DOS session, you must set the REQUEST_METHOD environment variable to POST or GET depending on which form method you use in your form. You also must set the CONTENT_LENGTH to the approximate length of the data written on the standard input (256 characters suits our needs for this program).
To run the program, open a DOS session and enter the following commands:

```
SET REQUEST_METHOD=POST
SET CONTENT_LENGTH=256
hellocgi
```

The program should be waiting for your input. Type `name=Bob` and terminate the entry with Ctrl+Z for end-of-input. The program should resume and generate the proper HTML flow (Figure 30). Instead of typing the user input, you can provide it using an input text file that you edit and add the line: `name=Bob`. Then you can redirect the standard output to a file to collect the HTML flow, which you can display later on in your Web browser. Assuming you use `input.txt` as your input file and you redirect the standard output to `output.html`, you should enter the following command line: `output.html < hellocgi < input.txt`

![Figure 30. Testing the Form from a DOS Session](image)

If you look at the Hello World Page, you will notice that although you placed the **Submit** button underneath the entry field, the button appears beside the entry field. This is because you used an **ICanvas** part to hold each subpart. In contrast, the page produced by the CGI program displays the control one under the other because you are using an **IMultiCellCanvas** part that maps to an HTML `<TABLE>` tag.

The HTML parameter-producing controls must be placed within an HTML `<FORM>` tag pair. There can be multiple forms on a page, but forms cannot be nested. The form is generated by an enclosing composition control that contains either a form action or form method attribute. The controls that accept these attributes are:

- ICanvas
- IMultiCellCanvas
- ISetCanvas
- ISplitCanvas
- INotebook
Because any button you place on your form is associated with a single CGI program (the one which is defined in the form action attribute), you might wonder how you can build a page with several buttons that would generate a different page according to the button that the Web surfer clicks. In the next section, we show you a technique for doing just that.

**Building a Router Part**

In this section you build a CGI program that serves as a router and generates different Web pages according to which button the surfer selects on a previous page. This time you create the CGI program as a nonvisual part.

**Modifying the Hello World Page**

Load in the HelloPage part in the Composition Editor and make the following changes:

1. Delete the IStaticText and IEntryField parts.
2. Add an extra IPushButton next to the first one.
3. Change the push buttons labels to Page A and Page B.
4. Change the push button name to PBA for the Page A button and PBB for the Page B button.
5. Open the settings notebook of the ICanvas part inside the IFrameWindow and change the Form Action attribute to /cgi-bin/router.exe.

Once you have applied all of these changes, your Web page should look like that shown in Figure 31.

![Figure 31. Hello World Web Page Revisited](image_url)
Generate the HTML code for your Web page by selecting **File→Save and generate→Part source** from the Composition Editor menu bar. Copy the generated Web page in the appropriate Web server HTML directory.

---

**Create Web Page A and Web Page B**

Let us now create two distinctive parts, **PageA** and **PageB** as follows:

1. Create a new visual part, **PageA** that inherits from **IFrameWindow**.
2. Change the title window to **Page A**.
3. Add an **IHeadControl** in the middle of the window and change its text to “Hello, I’m Page A!” (Figure 32).

Repeat steps 1 through 3 but this time replace **A** with **B**.

---

![Figure 32. Web Page A](image)

---

Generate the C++ source code for each page by selecting **File→Save and generate→Part source** from the Composition Editor menu bar. The source code produced will be compiled and linked with the **Router** part source code you build in the next section.

---

**Building the Router Part**

To build the **Router** part from the Composition Editor, follow these step-by-step instructions:

1. Create a nonvisual **Router** part that inherits from **IStandardNotifier**. The Part Interface Editor is displayed.
2. Click on the left pie in the lower right of the Part Interface Editor to switch to the Composition Editor.

3. Add two *IVBFactory* parts on the free-form surface and change their names to *FactoryPageA* and *FactoryPageB*.

4. Change the type of the *IVBFactory* parts to *PageA* and *PageB*.

5. Draw a custom logic connection from the *ready* event of the free-form surface to the *ICGI* part and add the following code:

```c++
if (target->get("PBA") == "Page A")
    source->iFactoryPageA->create();
else
    source->iFactoryPageB->create();
```

Save and generate the router C++ source code and the router’s make file. Then compile and link edit the code to produce the *router.exe* CGI program. Copy your *router.exe* CGI program and the *hillopage.html* Web page in your Web server.

**Testing Your Application**

You have two ways of testing your *Router* part: from a browser that connects to your Web server or from a DOS session.

**Testing from a Browser.** Start your favorite Web browser and load the Hello World Web page. Then click on either the *Page A* or *Page B* push button to call the router CGI program. If you click the *Page A* button, the *PBA* parameter is transmitted to the CGI program with the value *Page A*. The custom logic connection fires, and the test executes the *create* method of the *FactoryPageA* object, which generates the Page A Web page (Figure 33). If the test fails, the user has clicked on *Page B*, the *create* method of the *FactoryPageB* object is called to generate the Page B Web page.
Testing from a DOS Session. You can test your router program even if you cannot access a Web server. Just set the REQUEST_METHOD to POST and CONTENT_LENGTH to 256, and open a DOS session to start the router. While the programs waits for your input, enter \texttt{PBA=Page A} and terminate the entry with Ctrl+Z for end-of-input. The program should generate the HTML corresponding to Web Page A.

Congratulations! You now know the basic techniques you can use with the VisualAge for C++ Web parts to create a Web application. These techniques are used throughout the development of the Web Travel Agent application.

Frequently Asked Questions

In this section we present some frequently asked questions about using Waikiki.

How do I create an HTML form?

The simplest way of creating a form is to place a push button on a canvas part and then set the \textit{Form Action} and \textit{Form Method} attributes of the canvas. By default, the push button becomes the submit button for the form. Typically, you will also place input controls, such as entry fields, radio buttons, and list boxes on the form. When a Web surfer clicks the submit button, a response is sent to the server program identified in the \textit{Form Action} attribute. The response takes on the form of \texttt{NAME=VALUE} parameters concatenated together with ampersand characters.

Can I create a static HTML page?

Yes! Set \texttt{VBGENCODE=ALL} in the environment. Use the system applet in Windows NT, or enter the command in your \texttt{AUTOEXEC.BAT} if using Windows 95. At present, this HTML is not guaranteed to compare byte-for-byte with the HTML generated by the code. Use this feature to generate any static HTML pages, such as a query form, for your application.

Is there a documentation tool for my CGI parts?

Yes! Set \texttt{VBGENDOC=ALL} in the environment. Use the system applet in Windows NT, or enter the command in your \texttt{AUTOEXEC.BAT} if you are using Windows 95. An HTML page is generated for your CGI part that describes its part interface.

How do I debug my program?

This is a thorny question that CGI programmers frequently ask. The program is run within the Web server, which only reports that an error occurred. You can use \texttt{iddebug} to debug many problems with your program.
or even run your program from a command line to observe the HTML that it emits. Set REQUEST_METHOD=GET or REQUEST_METHOD=POST and then run your executable from the command line or the debugger. You may also have to set CONTENT_LENGTH or some other standard CGI environment variable if your program connects to the corresponding attributes in ICGL. The ICGL part also supports logging actions and has assertion actions.

If you want a debug trace of connections firing and member functions being called, try tracing to a file. Build with the VBDEBUG environment variable set to 1. Run the CGI program in a server that has the following environment variables set:

- SET ICLUI TRACE=ON
- SET ICLUI TRACEFILE=C:\WWW\LOG\TRACE.LOG
  (substitute path and file name of your choice)
- SET ICLUI TRACETO=FILE

Make sure your server actually sees the above settings:

If the Web server is running on Windows 95, write these settings in AUTOEXEC.BAT or from a process (command line or batch file) from which you start your server.

If your server executes on Windows NT, issue the above settings from the system applet Environment tab. Set them in the System section. You have to restart the system because most servers run as NT services and service processes capture the environment settings only when the system boots.

The trace file will build over time, so you may want to delete it and rebuild with VBDEBUG undefined, after you get your application in shape. The major advantage of the trace file is in showing connections firing in the real server environment.

**How do I control the NAME and VALUE used for each input element?**

By default the name of each input control defaults to the control’s name. To change the name, set the htmlName attribute. For radio buttons, check boxes, and push buttons, the htmlValue attribute will override the default return value. For entry fields and multiline edit controls, the text attribute is used to set the initial contents of the HTML element; the final contents are returned as the VALUE. For listboxes and combo boxes, the selected item is returned as the VALUE.

**How do I implement an HTML reset button?**

Create a push button part and set the enableReset attribute.
How does my server application receive input?

The new nonvisual part, ICGI, is used for requesting the NAME=VALUE parameters from the input stream. The get action is used to retrieve the IString value associated with the parameter identified in the name parameter. For multiple selection lists, you can retrieve the paramCount attribute. Use the getNameAtIndex and getValueAtIndex actions to filter the input parameters until the count is reached.

How do I create an HTML HREF?

Place the text for the HTML reference into the text attribute of an IStatic-Text part and then set the anchorRef attribute to the reference URL.

How do I create a reference to some other location within the same HTML page?

Identify the control that is being referenced by setting the anchorName attribute to a name that is unique to that HTML page. Then create an HTML HREF, as described above, except that the URL should be the name that was used for the reference, prefixed by a pound sign (#).

Since there is no HTML notebook, what does INotebook emit?

The INotebook part is simulated in HTML through <TABLE> tags. Each page of the notebook is laid out in the outer table. Three inner tables are used to lay out the tabs, the page contents, and the status area. The status area contains the page’s status text and page buttons that appear as < >. The page contents are laid out as they would be without being in the notebook. The tabs appear as same-page (that’s HTML page) hyperlinks. Clicking on a tab or page button will hyperlink to that page. Major and minor tabs are supported. Note that all of the notebook pages are emitted into a single HTML page.

How do I use the IContainerControl part?

The IContainerControl part is supported by simulating the container’s details view in an HTML page. Each visible column becomes a column of the table, and each visible object becomes a row. Titles and details view headings are placed at the top of the table if their corresponding attributes are enabled. Object selection is supported with HTML INPUT elements. If the container selectionType is set to single, a radio button column appears at the beginning of the container table. Multiple or extended selection yields a column of check boxes.

Use the container element to gather form input by placing the container on a canvas. Add an IPushbutton submit button. Set the canvas’ formAction. An end-user selection is submitted to the server using CNR-NAME=value. CNRNAME is the name attribute of the container. The value
is either an index of the object in the container, for example 1, 2, or 50, or a string value corresponding to the selected object’s iconText attribute. To return value instead of index, enable the container’s byValue attribute. Connect the ready event to custom logic in the container: target→enableByValue(0); for index values, or target→enableByValue(); for iconText.

❑ **Hint 1:** Select text as the container item text attribute, or your program may return debug strings for values.

❑ **Hint 2:** If you forget to place the container inside one of the canvas parts, HTML FORM will not be generated for the INPUT tags. Fill in the canvas Action and Method settings to cause an HTML form to generate. In that way, your action routine can process the container selections with the browser showing the input tags that represent the container.

❑ **Hint 3:** If you change the container item type after a C++ build, erase the tempinc directory or the type change will not be picked up by the C++ templates. The tempinc directory is specified in the /Ft switch in the make file.

**How can I insert raw HTML into a Web page?**

The easiest way to place HTML into the page is to place an IStaticText control at the desired location in your window layout, place the raw HTML into the text attribute, and set the textAsIs attribute on.

**Can I hide state information in an HTML page?**

Yes. Place the state information into an IEntryField part’s text attribute and set the show attribute to false.

**Why does my program build as a GUI application?**

The symbol VBWEB=1 must be defined in the environment or the make file to compile and link a program as a Web server application. If you try to compile a program as a GUI application that contains a form method or form action attribute on a canvas, you will get a compile error.

**How do I prevent the Builder from showing a warning when I start it?**

Set VBCDM=OFF in the environment. Use the system applet of the control panel to set it in the user environment of Windows NT. If you are using Windows 95, set it in the AUTOEXEC.BAT file.
Why does my program just “go away” when I run it on the command line?

The linker step probably still has the /pmttype:pm option set. Remove it or substitute /map. The generate main emits a make file set for GUI builds. This problem is being addressed.

Another possibility is that the program is throwing an exception. Build it with VBDEBUG=1 set on the command line or in the environment. Run it with ICLUI TRACE=ON and ICLUI TRACETO=ERR set in the environment. Any exception will show up on stderr.

Why can’t my link step find IWEB.LIB?

The link editor is trying to build your CGI program as a GUI program. Set VBWEB=1.

How do I test on Windows 95 without a network card?

You still must have a Web server. First make sure you have dial-up networking TCP installed from Windows 95. Ensure that you have a localhost 127.0.0.1 in your TCP hosts file. During installation, tell the server not to dial to establish a connection. Specify localhost as the server name when prompted.

Where are my IImgControl, IHeadControl, and ICgi parts?

They are in VBBASE.VBB along with the existing user interface parts.
Introduction to DB2 World Wide Web Connection

Imagine that you are going to create some Web pages that query and update your database, and that you are quite familiar with HTML and SQL. You cannot solely depend on simple static HTML pages, as your pages demand access to your database. You need some CGI programs instead.

You know that you are not going to have complicated conditionals and decision-making logics in your page flow, nor do you need to generate completely different screen layouts dynamically according to the data obtained from the database. You may ask, “Do I need to write my own CGI programs?” With the DB2 World Wide Web Connection gateway, the answer is “No.”
Overview

DB2 World Wide Web Connection enables Web access to DB2 databases. It delivers the Web gateway and application development models you need to build powerful Internet applications.

Web applications for DB2 built with DB2 World Wide Web Connection allow any Internet user with an HTML-standard Web browser to access your DB2 data without your having to make any changes to the existing data structure. Using standard HTML and SQL, you can build applications that query your DB2 data with standard SQL statements. Figure 34 gives an overview for the DB2 World Wide Web Connection.

DB2 World Wide Web Connection not only links the WWW and DB2 together, it also links HTML and SQL together. It gives you the full power of HTML and the versatility of SQL to access data on DB2 databases from anywhere on the Internet, working with Internet servers via the CGI.

An application developer writes macros, and stores them on the Web server. These macros enable customers to query the databases using HTML forms. The results of the query are displayed on the Web browser. Since all the codes and macro files reside on the Internet server, that means what the end user needs is only a Web browser. Also, an application developer can use DB2 World Wide Web Connection with DataJoiner to access data in non-IBM databases and nonrelational databases.
The power of DB2 World Wide Web Connection is its ability to pass variables from HTML to SQL and back. Variables may be specified by you in the macro or by a customer through your HTML form. Figure 35 shows how the run-time flow is controlled when we are calling a DB2 WWW macro.

From the developer’s point of view, using DB2 World Wide Web Connection has another big advantage. Developers can run a text editor and a Web browser and put them side by side. When they make some modifications to a DB2 WWW macro in the editor and save it, they can immediately go to the browser and test for the result. They do not need to do any compilation or code generation every time they make even minor changes, which helps to shorten the development cycle significantly.

DB2 World Wide Web Connection delivers open Internet access to DB2 data, a simple yet powerful development environment, and the capability of fitting into both a two-tier and three-tier Internet-to-data client/server environment.

![Figure 35. DB2 WWW Run-Time Flow Control](image)

Using the DB2 World Wide Web Connection gateway also eliminates the need to customize support for multiple, simultaneous clients. Any of the currently popular Web browsers supporting HTML on OS/2, UNIX, Macintosh, and Windows platforms can access DB2 data through applications built with DB2 World Wide Web Connection.
How it Works

The key feature of DB2 World Wide Web Connection is a CGI run-time engine, which processes the input from HTML forms on the Web and sends SQL commands to a DB2 system specified in a DB2 World Wide Web Connection application. This application consists of a macro file containing HTML input and report form definitions, SQL commands and variable definitions. DB2 World Wide Web Connection uses the Web page paradigm for application development, which means that the application user on the Web sees only a familiar Web page form that may prompt for user input and then launch the application transaction, returning the DB2 query results in another familiar-looking Web page report.

DB2 World Wide Web Connection applications use native HTML and SQL, exploiting the expressive power of those languages without proprietary limitations. Both HTML input and report forms can support various designs. The run-time engine supports SQL commands for SELECT, INSERT, UPDATE, and DELETE. DB2 World Wide Web Connection applications can use data linking, the capability to use data returned by an SQL query as input to one or more subsequent SQL commands that can drill down further into DB2 data.

DB2 World Wide Web Connection is a CGI program that you can invoke from an HTML anchor reference or form. The file name is db2www.exe, which is located in the \CGI-BIN subdirectory of your Internet server.

The format for invoking DB2 WWW from an HTML anchor reference is like this:

```html
<A HREF=http://{web-server}/cgi-bin/db2www(.exe)/{macro-file}/ {cmd}[?name1=val1&...]>any text here</A>
```

And the format for invoking DB2 WWW from an HTML form is like this:

```html
<FORM METHOD={method} ACTION=http://{web-server}/cgi-bin/db2www(.exe)/{macro-file}/{cmd}[?name1=val1&...]>any text here</FORM>
```

A typical URL without parameters looks like this:

`http://www.abc.com/cgi-bin/db2www/admin2.d2w/input`

DB2 World Wide Web Connection Macros

A DB2 WWW macro is a file containing HTML and SQL. A macro usually contains four sections:

- **Define**
  
  This section is used to define variables used in the macro.
DB2 World Wide Web Connection Macros

- **HTML Input**
  This section receives input from the client Web browser to place in the SQL query.

- **SQL**
  This section is used to define the query to send to the database.

- **HTML Report**
  This section invokes the SQL query and returns the results to the client Web browser.

The SQL query is dynamically created by using variables specified in the HTML form in the SQL section. That means the SQL query and the HTML form share the same set of variables.

To write a macro, you need to know HTML, SQL, and the few specifics of macro design discussed below.

### Define Section

This section is for defining the database name and other variables used in the macro. You must always define the variable `DATABASE` for each macro that accesses a database. You can define all the different types of variables supported by DB2 WWW, such as List variables, Conditional variables, and so on. Refer to the *DB2 World Wide Web Connection Application Developer’s Guide* for more details on DB2 WWW variables.

There is a one-line and multiple-line syntax for the Define section:

```%
DEFINE varname="value"
%
```

and

```%
DEFINE{
    varname1="value1"
    varname2="value2"
%}
%
```

The value specified can be a numeric value or a character string. The simplest variable statement takes this form:

```%
DEFINE DATABASE="wtadb"
%
```
**HTML Input Section**

The HTML Input section contains the HTML form where customers can specify information they want from the database by entering values in the form using their Web browsers. Input is dynamically placed in the SQL query. DB2 WWW Connection macros do not require an HTML input section for simple queries.

The syntax of HTML Input section is like this:

```
%HTML_INPUT{
  any valid html text
  on multiple lines
%
```

Here is an example of a HTML Input section:

```
%HTML_INPUT{
 <html>
  <head><title>Search for Customers</title></head>
  <body>
    <h1>Search for Customers</h1>
    <form method=POST action="/wta-bin/db2www/admin2.d2w/report">
      Please enter the customer’s name<br>
      <input type=TEXT name=custname size=30><p>
      <input type=SUBMIT value="Search">
    </form>
  </body>
%}
```

**SQL Section**

SQL section contains a SQL command. You can have multiple SQL sections, but each section can contain only one SQL command with optional SQL_REPORT and SQL_MESSAGE subsections. If you have more than one SQL section in a macro, you must name each of the sections so that you can call them from anywhere in the HTML Report section using the section names. If you have only one SQL section in a macro, however, you need not name it.

There are two kinds of syntax for the SQL section, a one-line and multiple-line syntax:

```
%SQL (sql-section-name) any SQL on one line
```

and
Chapter 5. Introduction to DB2 World Wide Web Connection

DB2 World Wide Web Connection Macros

The following example of an SQL section returns a list of customer IDs and customer names in the customer table, and orders them by customer name:

```sql
%SQL{
    SELECT custid, cust_name FROM custtable
    WHERE cust_name LIKE '$(custname)%'
    ORDER BY cust_name
}
```

The SQL in a section is executed when it is called by the `%EXEC_SQL` command in the HTML Report section. See “HTML Report Section” on page 99 for more details.

If an error or a warning occurs in a SQL command, the execution terminates and a return code is given. You must decide if you want the application to continue after receiving a warning message from a SQL command. Refer to “SQL Message Subsection” on page 98 for information dealing with these issues.

**SQL Report Subsection**

This subsection gives you the ability to customize the query output using HTML formatting. If you have no SQL Report subsection, a default table is displayed with column names at the top.

The syntax of a SQL Report subsection is as follows:

```sql
%SQL_REPORT{
    Any valid header HTML or column variable names returned from the query.
    %ROW{
        Any valid HTML with special variables to display once for each row returned.
    }%}
    Any valid HTML footer HTML.
}%
```

The following example of an SQL section returns a list of customer IDs and customer names in the customer table, and orders them by customer name:

```sql
%SQL{
    SELECT custid, cust_name FROM custtable
    WHERE cust_name LIKE '$(custname)%'
    ORDER BY cust_name
}
```
All text and graphics before the %ROW declaration constitute header
information and are displayed before any information from the SQL
query. Following the SQL query processing, the column names are placed
in special variables \textit{Ni}, \textit{N\_column-name}, and \textit{NLIST}.

The ROW subsection contains information displayed once for each row
returned by the SQL query.

Information, including text and graphics, following the %ROW subsection
is footer information and is displayed once after all rows are displayed.

Here is the SQL Report subsection for the previous example of the SQL
section. This subsection puts all the records returned from the SQL query
into a list box of an HTML form:

\begin{verbatim}
%SQL_REPORT{
  <form method=POST action="/wta-bin/db2www/admin2a.d2w/report">
    <select name=cid size=3>
      %ROW{
        <option value="$(V1)">$(V2)
      }
    </select>
    <input type=SUBMIT value="Detail information">
  </form>
%
\end{verbatim}

**SQL Message Subsection**

This subsection allows you to customize error and warning messages
from SQL commands. If you place this declaration inside a SQL section, it
is local only to the SQL command in that section. If it is placed outside of
all SQL sections, it is global to the entire macro.

You create a table of SQL codes and specify the information to display fol-
lowing each SQL code. The syntax of a SQL Message subsection is as fol-
lows:

\begin{verbatim}
%SQL_MESSAGE{
  +SQLCODE: "warning message" : exit or continue
  -SQLCODE: "error message"
  default: "default message"
%
\end{verbatim}
The default error message is shown when a SQL code not in the declaration is returned by the special variable SQL_CODE. For warning messages with positive SQL codes, you have the option to choose whether to exit or to continue. However, the macro terminates whenever it encounters an error message.

The following SQL Message subsection of the example for the SQL section displays an error message when no customers were found.

```sql
%SQL_MESSAGE{
100 : {
<b>No customers were found</b><p>
<a href=/wta-bin/db2www/admin0.d2w/input>Go back to main menu</a><p>
}%}
```

**HTML Report Section**

This section is where you call the SQL query. The section is executed when DB2 World Wide Web is started in the report mode, often from the HTML Input section of the same macro.

The syntax is like this:

```sql
%HTML_REPORT{
    any valid HTML text
    %EXEC_SQL(SQL section name1)
    any valid HTML text
    %EXEC_SQL(SQL section name2)
    any valid HTML text
    .......
%}
```

You can specify any HTML and include any variables from the Define section in the HTML code. You can also use input from the HTML form to override variables in the Define section.

When an `%EXEC_SQL` line is encountered, the SQL section matching the name or defined variable is called. Using a variable for the SQL section name is an easy way to allow customers to select a query to perform. If you do not specify a section name, all unnamed SQL sections are executed in the order they appear in the macro.

Here is a simple example of what an HTML report section might look like:
Security Considerations

You can keep your assets secure with the measures you are already using to protect your system and data. However, you must decide what level of security is appropriate for your assets.

Authentication

A DB2 WWW Connection gateway supports two types of authentication, one protecting certain directories on your server and one protecting your database:

- Most Web servers, such as IBM Internet Connection Server, allow you to specify directories on the server to protect. You can also have your system require a user ID and password for people accessing files in the directories you specify. In our Web Travel Agency application, we use this method to protect all our administrative functions macros. Refer to Chapter 11, “Implementation of Administrative Functions,” on page 365 for how to implement it.

- DB2 has another authentication system for database access that can restrict access to tables and columns to certain users. You can use DB2 WWW Connection’s special variables, LOGIN and PASSWORD, to link to DB2’s authentication routine.

Encryption

You can encrypt all data sent between a client system and your Web server when you use a Web server with Secure Sockets Layer (SSL) or Secured Hypertext Transfer Protocol (SHTTP). These security measures encrypt login IDs, passwords, and all data input through HTML forms from the client and all data sent from the Web server.
Security Considerations

Firewall

We can use DB2 WWW Connection with IBM’s NetSP Firewall and most other available firewall products, which protect both the DB2 WWW Connection server and the network from external probes or attacks.

Hidden Variables

For additional security, you can consider using hidden variables to conceal the internal structure of the database from people who choose to view your HTML source with their Web browser. Here are the steps for setting up hidden variables in your DB2 WWW macros:

1. Define a variable for each string you want to hide. Put the Define section for these variables after the HTML Input section where you reference the variables and before the HTML Report section where they are used.

2. In the HTML Input section where the variables are referenced, use double dollar signs instead of a single dollar sign to reference the variables. For example, $$X$$ instead of $X$.

The following is an example showing how hidden variables work:

```
%HTML_INPUT{
  <form ...>
  Select fields to view:<p>
  <select name="Field">
    <option value="$$\{\text{name}\}">Name
    <option value="$$\{\text{addr}\}">Address
  </select>
  .......
  </form>
%}

%DEFINE{
  name="custtable.cust_name"
  addr="custtable.address"
%}

%SQL SELECT ${Field} FROM custtable
  .......

%HTML_REPORT{
  %EXEC_SQL
%}
```
When the HTML form is displayed on the client’s Web browser, \$\{(name)\} and \$\{(addr)\} are replaced with \$(name) and \$(addr) respectively, so the actual table and column names never appear on the HTML form, and there is no way for the client user to know the actual table and column names of the database.

When the client user submits the form, the HTML Report section is called. When \%EXEC\_SQL calls the SQL section, \$\{(Field)\} is replaced in the SQL section with custtable.name or custtable.address in the SQL query, depending on what the client user has chosen in the HTML Input form.
Part 2

Case Study: Web Travel Agent
In this chapter, we show how to build a real-life application, the Web Travel Agency, to illustrate IBM Internet technology.

The Web Travel Agent enables a customer to create a travel package directly from a Web browser. It also enables the travel agent to keep track of the package that was sold and to do some database clean up. Our goal here is not to build a complete application that would solve all the problems of a travel agent. Instead, we want to show how it is possible, after applying an analysis and design methodology, to use the tools we describe in Chapters 1 - 5, and build a good object-oriented application. We use the Visual Modeling Technique (VMT), which is discussed in “VMT Overview” on page 110.

We begin by describing our application, together with the assumptions we made. We then proceed to the analysis and design phase and to the effective implementation.
Introduction

Let us assume for a moment that you are a customer of a travel agency. You are surfing from your favorite Web browser, and want to book all of your travel arrangements for your next business trip or vacation. You also want to use the trip you are going to make to Miami next Friday to learn about the Miami area so that you can better prepare for the move your company plans.

You will also probably want to connect directly to the home page of the Miami Chamber of Commerce. Furthermore, you want to plan a fabulous trip to Europe with your spouse, but you are still not sure which cities you would enjoy more, London? Paris? Berlin? A movie of the main attractions each city has to offer, would probably help you in making your final decision.

Once you have decided where to go, you need to decide when, and make the necessary reservations. For your selected destination city, you can book one or more of the following:

- Airplane tickets
- Hotel rooms
- Rental car

To reserve your tickets, accommodations, and car, you must provide the information you would have to provide at any travel agency counter, beginning with your preferred departure date and the length of your stay. Then you need to reserve a convenient flight on the airline with your preferred mileage program, your favorite hotel, and a car that suits your style and budget.

Customer Interactions

The following sections describe the interactions you are asked to perform when putting together a travel package. These interactions are typically down in a travel agency or over the telephone. As you will see later, the same interactions can be mapped to Internet Web pages.

Flight Reservation

To book seats on a flight, you must provide the following information:

- Your departure city (the Web Travel Agent cannot know whether you are on a connecting flight or leaving from your home city).
- Number of seats you want
- Whether you want to fly economy, business, or first class
- Whether you prefer the smoking or nonsmoking section
Notice that we do not ask for the departure date, or offer the option of choosing one non-business-class seat, smoking, and one first-class seat, nonsmoking. For the sake of simplicity in the interface and application design, we omit whatever does not contribute much to the demonstration of WWW application development.

You are asked only once for departure date and length of stay and this information remains valid for the entire reservation process. The end date of the package can be computed from the departure date and length of stay. The return trip is from the destination city back to the departure city, on the end date of the package. Only one airport per city is assumed, and one flight per day between any two cities of our list. We consider that the flight always arrives on the same day it leaves, so the booking of a hotel can still be taken from the departure date. In our application, we do not consider flight overbooking; but we do consider overbooking of hotel rooms. Pricing is computed from a base price for each flight, to which we apply a multiplier according to the seat class.

**Hotel Reservation**

Now that you know where you want to go, you may want to reserve your hotel. So the application displays the names of hotels in your destination city and its surroundings. If two hotels of the same chain exist in the same city, they are distinguished by their full name. You can choose to book one or more rooms, all with same room category (standard, deluxe, suite) and either smoking or nonsmoking.

For this type of reservation, we have assumed that some hotels are full, so we must know the capacity of each and how many rooms it has already booked. However, we do not assume a separate booking status for different room categories. Only the overall status of availability is kept and is compared to the total hotel capacity.

To meet this fully booked condition during application execution, we provide, in our sample list of hotels, some fully booked hotels. The pricing is computed from a base price for each hotel, to which we apply a multiplier according to room category.

**Car Reservation**

If you want to reserve a car, you can first look at the pictures of the types of car for rent in each category—compact, economy, intermediate, full-size, luxury, or sport. You can also select a favorite rental car company. We also keep track of the total number of cars in stock and the number of reservation made per car category. We do not have separate bookkeeping for every rental company. The total price is computed from the daily price of each category.
Introduction

**Package Confirmation**

When you are happy with all of your selections, you can request that the application checks for availability and price. Once the application does its checking, it displays a summary of your selections with corresponding status. If everything is available, it gives the individual and total prices. Here, you can still modify your selections on the appropriate menus, until you are satisfied with all aspects of the package. Then you are asked to confirm the package.

You are now requested to produce personal information. If you are already known to the application (you have already made some reservations through this Web site), all you have to do is supply your name and identification number. If you are a first-time customer, you must give your name, address, and telephone number. The application generates an identification number that you can use for your next visit to this Web site.

To pay for the package, you must also provide the name of a credit card company, and the credit card number and expiration date. You do not have to use the same credit card for all reservations.

Once the application has this information, the actual booking takes place. But what if someone else makes the same reservation between the time the application checks for availability and the time you confirm your selection? Then you receive the information about nonavailability as it was displayed during the first availability check. This may not be completely satisfactory.

To improve the situation, however, we would have to make a tentative reservation at the start. It is not clear how we would account for a tentative reservation when someone else wants the same hotel. Complex transaction systems such as SABRE and AMADEUS are designed to handle all these conditions (and many more). To keep the sample application simple enough for our purpose of understanding the specific design decisions inherent in this WWW environment, we thus accept the possibility of reselection.

**Personal Data Modification**

If you move from one city to another, the travel agent must still be able to send you some mail or call you. Therefore, our application allows for changing addresses and telephone numbers through the same Web browser interface. Providing your name and personal identification enables you to retrieve your personal data and modify it.
Package Inquiry

As our application has a friendly interface and offers good service, you can use it to prepare your next trip. Having already planned many other trips, you want to inspect all your packages. Providing your name and personal identification allows you to see a list of your packages and view any one of them in detail.

So far, we have discussed how you can use the Web Travel Agent. The travel agent itself also has some tasks to perform. In the next session, we review those tasks.

Travel Agent Interactions

Apart from a WTA customer there is another actor in our scenario: the travel agent who is also the administrator of the WTA application. The following sections describe the interactions with the WTA database from the administrator’s point of view.

Customer Listing

As the travel agents are paid commissions on packages they sell, they want to know all the new packages that have been created. They also need to know all the customers in the list, so that they can thank them for choosing their travel agency. They may want to see all of the packages a specific customer has booked, to check for some problems they have heard about. All of this is possible through the same Web interface, because the system hosting the application is not necessarily on the premises of the agent.

Administrative Tasks

To keep the application running smoothly, the travel agents, who have not hired a dedicated system administrator can also execute some basic system administration tasks with their Web browser.

Package Deletion

A travel package is no longer needed after a trip. We assume that a background program starts every morning and goes through all packages, deleting those whose return date is older than today’s date. This cleanup process is controlled by the travel agent. Every morning the travel agent selects an option from a menu, and the application not only deletes all the completed packages but also resets the corresponding number of booked rooms and cars.
Database Cleanup

For a complete cleanup of the database that holds all this information, you can delete all packages and reset the numbers of cars, flights, and hotel rooms to their installation default, through the same Web interface.

VMT Overview

Before starting to implement an object-oriented application, we first go through an analysis and design phase. The objective of an analysis is to understand the entities of the application and their interactions. This section briefly describes the VMT methodology that we use for our application. It is not our intent to give a formal explanation of this methodology, but rather to give the main ideas that allows you to understand the work we have done for this application. For more information on VMT refer to Visual Modeling Technique.

Object-Oriented Analysis

Object-oriented analysis covers the following activities in most methodologies, although the approaches, sequences, and techniques used may differ:

- Understanding the problem
- Defining the classes
- Establishing class relationships
- Defining class properties and behaviors
- Modeling object interactions
- Studying object state changes

The model depicts the relationship between the classes, the knowledge each class must have of other classes, the services each class must provide, and the description of how external events activate object interactions. To provide a proper description of what the application does, object analysis must address both the static class structure and the dynamic behavior of objects.

Modeling is an implementation of the analysis effort, using tools to produce deliverables. The terminology we use here is taken from current object-oriented technologies:

- Use cases, originated by I. Jacobson’s Object-Oriented Software Engineering Methodology (OOSE)
- Class Collaborators and Responsibilities cards (CRC), originated by K. Beck and W. Cunningham.
Object Model notation, from J. Rumbaugh’s Object-Modeling Technique (OMT).

The VMT modeling process, as shown in Figure 36, consists of the following steps:

1. Define an initial set of actors and use cases based on the problem statement and input from users and experts.
2. Build a basic functional interface prototype to understand the user interaction and the complete use case model.
3. From both the problem statement and the use case model, build an overview of the classes that describe the problem domain of the application, and the primary relationships and attributes of those classes.
4. Create CRC cards to define class responsibilities and collaborators.
5. From the class responsibilities, determine the services and attributes in the object model and refine the classes and their relationship.
6. From the use case and the required services for each class, build an event trace diagram or a state diagram to show the dynamic behavior of the system.
7. Iterate through this process to achieve an acceptable level of completeness.

Figure 36. VMT Modeling Process
VMT Overview

We now offer a short description of the main components of the VMT modeling process before applying it to our application.

Use Cases Description

The use case model is fundamental, as it not only collects all the requirements but also serves as input for all subsequent models developed. This requirements modeling is done using concepts and terms that relate to the business domain, not to the computer implementation. The system overall function is reflected in the use case model. It determines actors, their roles, and their transactions with the systems.

A use case contains a list of actions and specifies the interaction between the actor and the rest of the system. Identifying actors helps us determine what is inside and outside the system; that is, to identify the system boundaries. The process also enables us to determine the services that the system must provide.

Use cases can be described in different levels of detail; the correct level lies between two contrasting extremes: gathering enough information (which entails a highly detailed description) and not committing too early to a specific solution (which calls for a more general description).

CRC Cards Description

A responsibility-driven analysis is based on two concepts:

- A class has certain responsibilities, such as maintaining knowledge and providing services to other classes.
- An object provides requested services by collaborating with some other objects.

Class responsibilities and collaborators can be derived from use cases, with the help of CRC cards. To that end, we define system responsibilities implied from the use case and then we assign responsibilities. The gathered information is used to:

- Define object services.
- Refine attributes.
- Verify and refine classes and their relationship.
- Discover some new classes.
Object Model Description

The object model is the primary source of information about the results of the analysis work and the starting point of the design and implementation. Two different aspects must be addressed: the static object model and the dynamic object model.

Static Object Model

We call this the object model for short. It shows the hierarchy and the coherence of objects. Coherence is the static relationship among the objects, also called the association. We adopt Rumbaugh’s OMT notation to draw the object model and the associations (see Figure 37 for a notation subset). The main relationships between the object classes are these:

- **Generalization** is the relationship between a superclass and its subclasses. A subclass inherits all attributes, operations, and associations from its superclass. It builds the inheritance hierarchy.

- **Association** represents a link between two classes. Each end is a role showing how its class is viewed by the other class. Association is not always bidirectional. It can be qualified by a special attribute, such as Employee works for Company. It can be further specified by:
  
  ➢ Multiplicity, which specifies how many instances of one class may relate to a single instance of an associated class. This association can be one-to-one or one-to-many.
  
  ➢ Modeling an association as a class. For instance, we can make the position and salary of an employee an employment class to relate an employee and a company.

- **Aggregation** is a whole-part relationship, in which the components are part of the aggregated whole. For instance, the engine and wheels are part of a car.

Dynamic Object Model

VMT applies two different kinds of diagrams to form the dynamic model: the event trace diagram and the state transition diagram. The event trace diagram describes how the participating objects interact during the execution of a use case. In an event trace diagram, objects requesting or providing services are shown as vertical bars; events are shown as horizontal links between the objects, with an indication of direction. The sequence of events is shown proceeding from top to bottom. You can see a sample event trace diagram in Figure 38.
Figures 37 and 38 illustrate OMT Notation Subset and Sample Event Trace Diagram, respectively. The OMT Notation Subset shows the relationships between classes, objects, and attributes, including inheritance and association. The Sample Event Trace Diagram represents the flow of events and responses in a user interaction scenario.
The state transition diagram focuses on one object only. It shows every state that the object takes as the result of an executed function. Each object has an initial state, one or more intermediate states, and optionally a final state. Each state is implemented as a distinct value of an attribute of the object. The diagram represents the states as nodes and the event that causes the change of state as an arc between the original state and the resulting state (see Figure 39). Usually, we develop such diagrams for objects whose state changes are significant for the process flow, but we must check these state transition diagrams against each event trace diagram where the specific object is involved to ensure consistency.

**Figure 39. Sample State Transition Diagram**

**Summary of VMT Modeling Process**

We can now apply the VMT methodology for our Web Travel Agent. This application is kept simple enough that we do not distract you from our main goal, which is building a World Wide Web application. However, even a simple application enables us to show the main features of this methodology, which are summarized here.
Use Cases

From the initial requirements document, we define the actors and use cases, and build a GUI prototype. From the use cases, we then build a first-cut object model. We next write the CRC cards to define class responsibilities and collaborators. From the class responsibilities, we determine the services and attributes in the object model. From collaboration patterns, we refine the classes and their relationship. We build event trace diagrams and state diagrams showing the system dynamic behavior. This modeling process is iterated until the application is complete.

In the development cycle of an object-oriented application, the analysis phase is followed by the design phase. After modeling the problem domain, we proceed by designing and building a solution for it. During system design, we decompose the analysis model into subsystems. The decomposition process is comparable to the process of partitioning the entire application into subapplications. After a while, design and analysis become part of the same continuum where it is hard to separate them. The deliverables are similar, except that, in the design phase, they are taken to a finer granularity than for the analysis phase and are designed more from an internal (system) viewpoint rather than an external (user) viewpoint. We understand that the distinction between analysis and design is useful. The developer must always consider whether the decisions are related to the business domain or to an implementation issue.

For our book, we describe completely only one subsystem, that for reserving hotel rooms. The first view corresponds to our first steps into the analysis phase. The second view corresponds to a detailed design step.

At this step, we must consider the presentation characteristics of the target platform. Here, the application is displayed on a Web browser, using HTML format, and the information is carried over the HTTP protocol. We must consider the principles explained in Chapter 1, “Introduction and Overview,” on page 3.

Use Cases

Use cases are written using two different but common styles. For higher level (more general) use cases, a book-like paragraph style is used to describe function in a conversational tone. Lower-level use cases utilize a list style, as the actions required become more detailed. Ordered lists are used when the actions occur in a certain order. Unordered lists are used when the actions can occur in any sequence.

The use cases that make up the Web Travel Agent example are described in the following sections.
Hotel Room Selection

This section describes the hotel room reservation function of the Web Travel Agent. The first use case describes at a high level the complete path followed to reserve a hotel room. The following use cases then describe lower level functions that participate in the reservation of a hotel room.

Overview

The system prompts the customer for an initial travel agency service where the customer chooses to create a travel package. The system prompts for a destination, start date, and duration of the trip, and the customer provides the requested information. The system then prompts for a package component (hotel, flight, or car) selection or calculating the package price. The customer selects the hotel component.

The system displays the hotels located in the destination city and prompts for hotel preferences. The customer selects a hotel and specifies hotel room options. The system then prompts for a package component selection or price check. The customer selects the price check function. The system checks the selected hotel for availability of the requested room, and finds that rooms are available.

The system calculates the price of the hotel room. The system displays the selected city, dates, hotel name, room options, price, and provides a purchase option. The customer selects the purchase package option.

Initial Service Selection

The system displays a page that contains a welcome message, and allows the customer to choose an initial service. The customer selects one of four available initial services:
- Create a travel package
- Display existing travel packages
- Modify personal information
- View city attractions

Get Destination and Travel Time

The system displays a page that allows a customer to provide a destination, start date, and duration. The customer
- Selects a destination city from a list of cities.
- Selects a starting month from a month list.
- Selects a starting date from a date list.
- Selects a starting year from a year list.
- Enters the duration in number of days.
Use Cases

Package Component Selection

The system displays a page that allows a customer to choose a component of the travel package, or to check package’s availability and price. The customer selects one of the following:

- Flight component
- Car component
- Hotel component
- Package price

Get Flight Preferences

The system displays a page that allows a customer to specify flight preferences. The customer

- Selects the flight departure city from a list of cities.
- Enters number of seats needed.
- Selects seat class:
  - Economy
  - Business
  - First

Get Hotel Preferences

The system displays a page that allows a customer to specify hotel preferences. The customer

- Selects a hotel from a list.
- Enters the number of rooms.
- Selects smoking or nonsmoking.
- Selects a room category:
  - Standard
  - Deluxe
  - Suite

Get Car Preferences

The system displays a page that allows a customer to specify car rental preferences. The customer

- Selects a car rental company from a list of companies.
- Selects a car class:
  - Class A – compact
  - Class B – economy
Use Cases

➢ Class C – utility
➢ Class D – full-size
➢ Class E – luxury
➢ Class F – sport

Check Hotel Availability

The system determines whether the requested rooms in the selected hotel are available for fulfilling the travel reservation.

1. System queries selected hotel for total room capacity.
2. System queries selected hotel for number of rooms reserved.
3. System determines number of rooms available by subtracting number of rooms reserved from total room capacity.
4. System queries hotel package component for number of requested rooms.
5. System determines hotel component availability by comparing number of rooms available to requested number of rooms.

Price Package

The system calculates the price of each requested package component. The package can include all of the three package components (flight, hotel, and car) or can include any combination of the three, as long as at least one is specified. The cost of each requested component is added to the total package price.

Price Hotel Component

The system calculates the rooms price by multiplying the number of requested rooms with the standard room rate and an additional markup.

1. Package pricer routine queries selected hotel for standard room price.
2. Package pricer queries hotel package component for customer options.
3. Package pricer marks up standard room price 125% for a deluxe room.
4. Package pricer marks up standard room price 150% for a suite.
5. Package pricer multiplies room price by number of rooms requested.
CRC Cards

The CRC cards that are associated with the development of the Web Travel Agency application are the product of many iterations and changes. See Appendix A, “CRC Cards,” on page 429 for a complete listing of all CRC cards.

CRC Cards On-line

All CRC cards are provided in HTML format for viewing with a browser (C:\WTA\HTML\CRC\CRC.HTM). These Web CRC cards have hypertext links from the collaborator field to the corresponding class for easy viewing. This also may facilitate analysis between remote analysis teams.

User Interface Analysis

At this point, a simple prototyping can be done to better explain the user interface and the customer interaction with the application. Since we have described the application’s function in terms of use cases, we know which functions each actor can invoke, and we can illustrate what the screens might look like. Because we are developing a WWW application, where the user interacts through a Web browser, we implement this prototype using HTML. One Web page looks like the one displayed in Figure 40. You can run the prototype application through loading the HTML files provided with this book.

Figure 40. Prototype Page for Flight Reservation
From the use cases, we also know how the application windows are hierarchically displayed to answer the user’s requests. Figure 41 allows you to follow the paths that a user can take when navigating through the application. You always start at the top screen (Welcome View) and you eventually end at the Package Confirmation View. Refer to Table 8 for a short description of all views.

**Figure 41. Web Travel Agent Part Hierarchy**
### Table 8. View Descriptions

<table>
<thead>
<tr>
<th>View</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>WelcomeView</td>
<td>Home page of the application. It is used by the customer to select among the four main features available to access (package creation, packages listing, personal information, attraction viewing).</td>
</tr>
<tr>
<td>PackageCreationView</td>
<td>Enables the customer to enter destination, departure date, and length of stay.</td>
</tr>
<tr>
<td>PackageListView</td>
<td>Displays a short form of customer packages.</td>
</tr>
<tr>
<td>PersonalIdView</td>
<td>Displays a logon view to access user’s personal data.</td>
</tr>
<tr>
<td>AttractionView</td>
<td>Enables the user to choose among a list of cities in order to see a video or the WWW page for the selected city.</td>
</tr>
<tr>
<td>ComponentSelectionView</td>
<td>Displays a list of the three possible reservation types to choose from (flight, hotel, car).</td>
</tr>
<tr>
<td>FlightSelectionView</td>
<td>Allows for a choice of departing cities and selection of seats.</td>
</tr>
<tr>
<td>HotelSelectionView</td>
<td>Displays a list of hotels from which to choose city that matches the user’s destination. Allows for selection of category and number of rooms.</td>
</tr>
<tr>
<td>CarSelectionView</td>
<td>Allows for a choice of car rental company and car category.</td>
</tr>
<tr>
<td>PackagePurchaseView</td>
<td>When all reservations are successful, this window summarizes the selections, giving individual and total prices. Allows for effective booking or reselection.</td>
</tr>
<tr>
<td>PackageFailedView</td>
<td>When at least one reservation fails, this window enables the customer to reselect.</td>
</tr>
<tr>
<td>CustomerIdView</td>
<td>Collects name and address and generates ID for a new customer, or collects name and ID from a known customer.</td>
</tr>
<tr>
<td>CreditCardInfoView</td>
<td>Enables the user to give credit card information for the specific package.</td>
</tr>
<tr>
<td>PackageConfirmationView</td>
<td>Summarizes the booked package content.</td>
</tr>
<tr>
<td>PackageDetailsView</td>
<td>For a selected package, provides detailed information.</td>
</tr>
<tr>
<td>PersonalInfoView</td>
<td>Enables the user to change personal data (address and phone number).</td>
</tr>
<tr>
<td>MovieView</td>
<td>Enables the user to see a movie of a selected city.</td>
</tr>
</tbody>
</table>
Dynamic Object Model

The dynamic object model consists of two parts: the event trace diagrams and the state-transitions diagrams. Examining the steps for each of them would be too tedious; instead, we show some specific examples.

Event Trace Diagrams

This is the first step into our dynamic model. It must provide insight into the application’s dynamic behavior. During our analysis phase, the event trace diagram of the hotel room booking subsystem does not show internal details of objects, just their global role in the application. In Figure 42, you can see the flow of events for this subsystem. It reflects the use case described in “Use Cases” on page 116.

In the detailed design phase, we add all the internal behavior of objects needed to perform the task (see Figure 43). Here, we describe only the events associated with the internal objects. The others are fairly straightforward and follow the description of the use cases.
The location manager is added to hold the list of cities. It is created when the user chooses package creation to show a list of possible cities. The *RequestedPackage* must also be initialized, and an *endDate* computed from the *startDate* and *duration* to be used later on (for return flight, for instance). To show the list of hotel in the *HotelSelectionView*, the *Hotel* manager is created to hold the list of available hotels in the requested city. The *RequestedHotelRooms* object needs to be created to hold the information about the user’s choice. The number of nights needs also to be computed from the duration of the stay (duration minus one).

To ensure that the request for these rooms can be satisfied, the *RequestAvailabilityChecker* is started to verify that the conditions chosen can be met. The *Hotel* must be refreshed to account for the latest availability sta-
The booked, capacity, and rooms attributes can then be checked. If available, the PackagePricer is started; it computes the price from the Hotel and RequestedHotelRooms and updates the RequestedPackage (Figure 44).

Once the procedure for all the selections is completed, the user can proceed with the purchase of the package.

![Figure 44. Event Trace Diagram for Hotel Room Reservation – Design Phase](image)

**State Transition Diagrams**

We are going to choose one of the objects that has been used in the event trace diagram, so that we can check for consistency. One good candidate is the Hotel object.

You can see a diagram of the possible states that this object can take in Figure 45. The hotel can be selected from among the various hotels in the city where the customer wants to go. When the check for availability occurs, it can return available or not. If it is not available the user must propose another choice. If it is available, the price may not please the customer, who may deselect it. When the customer has filled in all the personal information, availability is again checked. Depending on this result, the booking can be confirmed, or the customer may be led back to the original selection.
Static Object Model

This object model is built during the analysis phase in parallel with the writing of the CRC cards. It is later refined in the design phase through multiple iterations to come to the final diagram.

Analysis Phase

In Figure 46, we show a first draft of such a diagram done during the analysis phase. It is taken from the use cases. In this first phase, we try to identify potential object classes from nouns such as customer, package, hotels, together with their attributes such as name, address, and start date. The links between objects represent association relationships. For
instance, the word *has* expresses an association between two objects. We try to assign meaningful names to relationships, so that looking at this diagram tells us, for instance, that the link between the customer and the requested hotel room is made by the customer’s preferences, therefore we name that link *prefers*. The customer can have many preferences, one set for each selectable components (flight, hotel, car). The package has the same number of components. The package involves a customer. A flight offers seats, and so on.

**Figure 46. Analysis Object Model**

**Hotel Room Reservation Design Phase**

In the design phase, the object model becomes too complex to be kept in one piece. We therefore divide it into subsystems, keeping related objects together. For instance, the reservation process can be separated from the booking process. Each type of component (flight, hotel, car) is a good candidate for simplification and the process of reserving a hotel involves relationships between classes similar to those for reserving a flight. We again choose to focus more completely on the hotel room reservation.
The number of classes that we come to in the design phase is larger than those discovered in the early analysis phase, but it may still be smaller than the number of classes in the actual implementation (for example, because of a service class to access the database). We do not cover implementation classes here, because they are often related to the implementation platform such as VisualAge for Smalltalk or VisualAge for C++. However, we show the visual classes (views), which are mapped to visual parts, and the business and services classes, which are mapped to nonvisual parts.

Combining the CRC cards, the event trace diagrams, and the view hierarchy, we can refine the object model for the hotel reservation path. As a first example, let's see what the PackageCreationView visual part has as attributes and what its relationships are with other classes. The view must be able to show a list of the cities the customer can choose to go to. It must also provide start date selection and duration of stay. The list of cities (locName attribute) is displayed in the view inside a container object, which retrieves information from the database controller.

We use this database controller terminology to speak about the nonvisual class that relates to the database. You can get more details on database usage for this application in “Database Design” on page 129.

When the customer has filled in all the necessary fields and clicked on a Continue push button, the PackageCreationView transfers control to the PackageCreationProcessor class. The PackageCreationProcessor class receives the customer data and updates the RequestedPackage class with the information provided, and thus stores the first attributes (locName, startDate, days). The end date needs to be computed for the return flight, for example, and this computation is a responsibility of this processor class. The other attributes created for this class derive from needs expressed in the rest of the application and qualify the package more completely. They are explained when actually needed.

After the PackageCreationProcessor finishes storing and processing data, it transfers control to the ComponentSelectionView class. Clicking, for example, on the Hotel button causes the HotelSelectionView page to be shown, for specifying hotel room options. The hotel name selection brings an additional step. The list of hotels in a city must be shown in the view. Therefore a clause is built to retrieve the list from the database and display them inside a list box. When the user has selected a hotel, an instance is created of RequestedHotelRooms and of Hotel.

The PackagePricer has a central role. When the user clicks on the Done push button indicating that the travel package is completed, control is transferred to this nonvisual class. The class uses the Hotel and RequestedHotelRooms instances to compare the booked and capacity values with the numbers of rooms requested. This computation allows the update of requestStatus according to availability. This field is later used for deciding whether to show the PackagePurchaseView or the PackagedFailedView.
If the hotel has sufficient available rooms, a price is calculated, using attributes such as the `roomMarkup` and `seatMarkup` as you can see in the Figure 47.

The flight selection, as well as the car selection are very similar and somewhat less complex, since no availability check is done for the flight, and you need not provide a list of specific items that changes with the selected city, as you do for hotel rooms.

**Database Design**

We have already defined the user interface, and how the user interacts with the business model. We have also designed the model layer, consisting of the application objects that implement real-world objects of the
application. Now we must study the data layer, providing data-access functions because, while objects are created and later deleted in this application, some information needs to remain. This is commonly called object persistence.

**Object Persistence**

When objects are created, space is allocated in memory and attributes are initialized as required. They are later deleted and memory is reclaimed, at least when the application ends. However, the life of these objects must be longer than the life of the application. For example, we must keep the name and personal information of the customer, as well as the content of the package that was built, for later inquiries. Therefore, we need a mechanism to save and retrieve these objects, thus ensuring object persistence.

In a multiuser environment, we also require:

- Data uniqueness, that is, no two packages or customers can have the same identification.
- Security. Access to personal information must be protected.
- Integrity. When a customer books a component, nobody else must be booking the same component at the same time. In addition, when the administrator deletes a package, the relevant components must also be deleted.

Only a database management system (DBMS) can provide all these functionalities. The widespread use of relational databases leads us to choose this technology to store object attribute values in the cells of relational tables, and later to retrieve the data to recreate the object. Generally speaking, we can say that:

- A relational database presents data as a collection of tables.
- A table consists of data logically arranged in columns and rows.
- A view is an alternative representation of data from one or more tables. It can include all or some of the columns contained in the tables on which it is defined.
- An index key is a column or an ordered collection of columns on which an index is defined. Using a unique index ensures that each value in the indexed column or columns is unique.

The objects are going to be stored inside these cells according to specific rules. This process is frequently called *flattening* or *streaming out*.
Mapping Object Classes to Tables

In the Web Travel Agent, the user interface logic and data access logic are the primary application features. We can call it a data-centric application. Most of the nonvisual parts we use are mapped from database tables. VisualAge for Smalltalk and VisualAge for C++ are meant to ease the development of this type of application by providing facilities such as the Composition Editor and database support parts. For instance, wrapper parts are created to take care of the real access to the database. For details on the different implementations, refer to Chapter 8, “Implementation with VisualAge for Smalltalk,” on page 159 and Chapter 9, “Implementation with VisualAge for C++,” on page 251.

We must choose the right data structures to support object attributes and relationships. This mapping allows translating the object to internal tables and views. An attribute of a table maps to a column of an SQL table definition. Each object class may map to one or more tables just as a table may correspond to more than one class. Usually, however, an object class maps to one table, because an object has knowledge about its attributes. Associations with other objects are implemented by defining additional columns within the tables, with another column representing the end of the other side of the association. No additional attributes of the associated object should be contained within the table representing the referencing object. The following are some examples from our application (see the diagram of these tables in Figure 48).

- The list of customers is represented as a table called CUSTOMER, whose attributes are directly mapped. One column equals one attribute, with corresponding attribute types (integer, character). This is really the simplest mapping.
- The list of available hotels is represented as a table called HOTEL, with each row being a different hotel. The information that must remain includes its name, its capacity, the number of rooms booked, and the daily price for a standard room. The hotel is located in a city, so that a column, LOC_NAME links to the city, but no other information on the city is provided here.
- The list of booked rooms maps to a table called BROOMS where the columns map directly to the customer requests—that is, number of rooms, room category, smoking or nonsmoking, and number of nights. None of this information is found in any other table (number of nights is computed from duration of stay). A column must link the booked rooms to the hotel and to the package that contains the booked rooms.
- The list of packages maps to a table called PACKAGE. The attributes unique to this package (start date, duration, end date, credit card information) are mapped to columns. A column links to the customer who has bought this package.
The other tables can be mapped in the same way and the result of this process is shown in Figure 48. But some features still need to be explained to enable the complete definition of the database.

### Defining Keys

Some details that are not mentioned in object modeling, such as primary and foreign keys, must be supplied. Each set of attributes that ensures the uniqueness of a row can be a candidate key. There may be multiple candidate keys in a table, but one of them must be designated as the primary key. Each class-derived table has a primary key as the unique identifier of the objects stored in the table, in conformity with the object-oriented notion that objects have an identity apart from their properties. IDs can be artificial; they do not necessarily have a counterpart in the real world. They are used for convenience and performance reason within the DBMS. When these IDs are artificial constructs, they are often implemented as...
unique integers. In an object-oriented environment, any instantiation of a
class has an object ID that is unique within all the objects. Two hotels that
have the same name in different cities are really different. In our case, it is
then convenient to use an artificial ID for one hotel and to put all refer-
ences to this hotel through this HOTELID primary key.

You can also see in the description of the database other IDs such as CUS-
TID or PACKID. The case of CUSTID is slightly different. If it were
designed only to ensure uniqueness for a customer based on the complete
customer information set (name, address, phone number), it would not
need to appear in the object model, only in the database. But is also used
in the business logic to act as an authentication mechanism when the user
wants to access personal data. PACKID also identifies uniquely the pack-
age.

Association between two classes is somewhat complex. Association with
multiplicity of 0 or 1 between Class A and Class B is represented by a spe-
cific column in each table, called the foreign key. According to Rumbaugh’s
Object-Oriented Modeling and Design, each one-to-many association should
map to a distinct table with a foreign key in the table for the many classes.
For example, in our application:

- Because a customer can have many packages, the PACKAGE table has
  a CUSTID foreign key.
- Because a city has many hotels, the HOTEL table has a LOC_NAME
  foreign key.

Each one-to-one association maps to a distinct table with a foreign key in
the table of either class:

- A package has a room reservation.
- A package has a flight reservation.

We choose to have the foreign key inside the BROOMS, BSEATS, BCARS
to relate to the PACKID, so that the PACKAGE does not need to have three
foreign keys. Furthermore, if we want to add a new component for reser-
vation, such as a train, the PACKAGE table definition is still valid, we
merely have to add two new tables (TRAIN, BTRAIN).

Our application is simple enough that we need not go into the mapping
of inheritance into relational tables.

Referential Integrity

Foreign keys are also used in maintaining referential integrity. From the
perspective of data value control, referential integrity constraints allow
you to control the uniqueness of the values of a key (one or more col-
umns) and existence of a row with a specified foreign key when the pri-
mary key does not exist. Referential integrity constraints, which are used
to enforce your rules on the data, may span one or more tables. If the rules apply for all applications that use the data, then using these constraints centralizes the rules in the database, making it generally applicable and easier to maintain.

Referential integrity is not directly provided within the object-oriented environment, but it is almost always a part of the DBMSs in the market. Therefore, an application can use this functionality and imitate it for the objects of the model. In our application, deletion of objects is done only in the administrative part of the application, through the DB2 WWW Connection gateway. As an example, when a city is no longer in the list, there cannot be any hotel located there. Also, when a package is deleted, the corresponding rows in BROOMS, BSEATS, and BCARS must be deleted. Usage of referential integrity can be seen in detail in Chapter 11, “Implementation of Administrative Functions,” on page 365.

The deletion of dependent table rows, as described above, is defined as a trigger, that is, executed in response to (triggered by) an update, insert, or delete operation on a specified base table. The use of such triggers promotes faster application development because triggers are stored in the relational database, and the actions performed by triggers do not have to be coded in each application. This functionality can be achieved through the so-called cascade delete mechanism of the database. Cascade delete enforces deletion of all rows in the dependent table that have the same foreign key value as the primary key of the deleted row in the parent table. This constraint is very powerful; it enforces consistency within the database. It must therefore be used by an application-driven database access only, otherwise data may be lost because of some mistake in formulating an SQL delete statement.

Appendix B, “Database Definition,” on page 451 lists the data definition language (DDL) commands that are used to build the WTA database and to create sample records.

For implementing our application, we chose to access a LAN-based DB2/2 server through TCP/IP and NetBIOS. With the facilities provided by the database access parts, however, the underlying database can be changed to any database supporting Open Database Connection (ODBC).
Application in Action

The application can be run from any Web browser that supports HTML 3.2, as every browser on the market should do by the time you read this book. On the server side, make sure the application, its database, and the Internet Connection server are installed and running. Refer to Chapter 12, “Installation and Setup,” on page 377 for detailed instructions.

Using the Web Travel Agent

The following sections describe the use of the Web Travel Agency application by going through some scenarios in which a potential customer wants to book a travel package using our application.

Startup

For easy startup of your Web Travel Agent, you might want to add an icon of the packaged application to your desktop. Drag a Program icon from the Templates folder to the desktop and change the settings as
shown in Figure 49. Notice that the file name of the executable might be
different. The one in Figure 49 refers to version 3.56 of the application.
Change the title of the icon to Web Travel Agent and close the settings view.

![Figure 49. Web Travel Agent Settings](image)

As soon as you start the Web Travel Agency application, VisualAge needs
to connect to the underlying database. In order to do that, the OS/2 User
Profile Management prompts you to log on to the system locally. If you
are running on Windows, you need to log on to DB2 through calling
DB2LOGON (see Figure 50). To avoid interfering with your network
conventions, we used the default user ID and password, which are USERID
and PASSWORD.

The Web Travel Agency Database (WTADB) is owned by the USERID
user profile, so that you now have complete authority over the database.
In our test environment, WTADB resided on a server separate from the
VisualAge and Internet Connection Server. This is not necessary; you can
place the database on either a remote or a local system. As long as the
database is cataloged correctly, the application can connect to it.

![Figure 50. Local Logon](image)
Using the Web Travel Agent

Chapter 7. Application in Action

137

Booking a Travel Package

If you have successfully installed all components of the application—VisualAge for Smalltalk or VisualAge for C++, the Web Connection feature, the Web Travel Agent (WTA) application, the DB2 database, and the IBM Internet Connection Server—you are ready to book your first WTA travel package. Start your Web browser, such as IBM WebExplorer or Netscape, and enter the URL of the Web server; for example, in our setup it is http://bavaria.almaden.ibm.com/cgi-bin/abtcgil/WtaWelcomeView. The application greets you with its Welcome View (Figure 51). From here you can

- Create a new travel package.
- Display existing travel packages.
- Modify your personal information.
- View attractions (videos).

To do so, click on the appropriate one of the four push buttons. The Web page footer lets you link to the IBM Redbooks home page, several product pages, various IBM Web pages and a WTA About page.

Figure 51. Welcome View
Our first step is to create a new travel package. Click on the Create button and the next Web page lets you select a destination city, a travel date, and the number of days for your trip (see Figure 52). We choose to go to San Francisco on November 12, 1997 (planning ahead), for a total of five days. Click on the Continue button to start the creation process.

![IBM Web Explorer: Create a Travel Package](image)

**Figure 52. Create a Travel Package**

The WTA Component Selection View (see Figure 53) allows you now to specify a hotel, a flight, and a rental car for your trip. After you have finished selecting a component, the program returns to this screen for further processing. To start putting our package together, click on the Hotel button; we first want to make sure that we have a place to stay.
Using the Web Travel Agent

Figure 53. Select Package Components

The application returns with the next Web page showing all San Francisco hotels (see Figure 54). The occasion merited a deluxe room at the Hyatt, and we select a nonsmoking room there. Click on Continue to get back to the Component Selection View (Figure 53), and then click on the Flight button to further define our itinerary.
The WTA application now asks us to specify our air travel plans (Figure 55). Our sample home town is the Windy City, thus we select Chicago as the departure city. For this 4-hour flight, we choose two seats in first class. Again, click on **Continue** to return to the Component Selection View (Figure 53), and then on the **Car** button to complete our travel package.
Using the Web Travel Agent

Chapter 7. Application in Action

Figure 55. Specify Flight Preferences

The Car Selection View (Figure 56) allows us to specify one of four car rental companies and a car class of our choice. We first want a closer look at the cars that represent classes A through F. Therefore, we click on the Classes button.
Using the Web Travel Agent

Depending on your taste, budget, and travel plans (off-road, autobahn) select the appropriate automobile and click on the corresponding button on the Class Selection View (Figure 57) to return to the previous screen.

Figure 56. Specify Rental Car Preferences
Depending on your choice on the Class Selection View, the appropriate radio button of the car class is automatically selected. Now pick your favorite rental company and click on **Continue** to finish the package.

You are now back where you started, on the Package Selection View (Figure 58). To let you review your package contents, the application put your selections next to the push buttons. If you need to change anything, you can click the corresponding button again and revise the travel component. If you decide that a trip to Boston would be more fun, click on **Cancel** to start over the whole package creation. If everything is OK, click on **Continue** for a package summary and the price. (You don’t think the trip is free, do you?).
The WTA application returns with a complete itinerary of your trip, the selected destination, travel date, hotel, flight, car and the price for all options (see Figure 59). You still have time to think it over and select, for example, a less expensive hotel, a coach class seat, or an even faster car. To revisit any component, click on the Change button. If you agree with the itinerary (and the total package price), click on the Purchase button.

**Figure 58. Select Package Components (Package Complete)**
Using the Web Travel Agent

The application now needs to know whether this is your first visit to the Web Travel Agent (click on the New button) or if you are a loyal cyber customer and came back for another trip purchase (click on the Existing button). See Figure 60 for details.
Since this is the first travel package we booked through the Web Travel Agent, we choose the **New** button. The next page requests us to enter our personal information, that is, name, address, and telephone number (Figure 62). After you filled all entry fields, click on the **Done** button.
Using the Web Travel Agent

You are almost there. Using your name and address, the application verifies the credit card information that you enter on the purchase page (see Figure 62).

**Figure 61. New Customer Personal Information**

You are almost there. Using your name and address, the application verifies the credit card information that you enter on the purchase page (see Figure 62).
That's it. The last screen of the application once again confirms the itinerary with all selected travel package components (Figure 63). The Web Travel Agent thanks you for your business, and asks you, please, to write down your customer ID, because you will need it to book your next trip. By clicking on the home page icon you can return to the Welcome View of the application.

Figure 62. Credit Card Information
List Existing Packages

After you have booked all your trips, you can review the packages by clicking on the **Display** button on the Welcome View (Figure 51). You are asked to enter your name and your customer ID, as shown in Figure 64.
Rikki Don’t Lose That Number!

If you forgot your customer ID, you would be out of luck, since the Web Travel Agent does not employ customer service representatives nor has it an 800 number. But because this is not real life (and we do not care about security), you can query the appropriate table from the DB2 database. Do the following from an OS/2 prompt or a Windows DB2 Command Window to view the customer table:

```
DB2 CONNECT TO WTADB
DB2 SELECT * FROM CUSTOMER
```

Alternatively, overwrite the part name in the current URL in your browser with WtaDatabaseView. This part is not used within the application, but was helpful during application development to quickly verify updates to the database.

Figure 64. Customer Authentication

If you entered a correct customer name and ID, the application returns with a list of all the trip packages booked under that customer ID. You can request more information about any shown package by selecting a line from the listbox and then clicking on the Details button (Figure 65).
Using the Web Travel Agent

Chapter 7. Application in Action

The application uses your selection (the number at the end of the line in the listbox is the generated package number) to retrieve the complete package data from the DB2 table and shows the details on the browser (Figure 66). If you want to review more than one package at this time, click on the Return button to go back to the package selection (Figure 65).
As a Web Travel Agent customer you travel to exciting destinations and someday you will also move to a new home, in another city or at least on another street. That is why the application provides you with a way to update your personal information.

The customer authentication is the same as for viewing the package list (Figure 64). Refer to the box on page 150 for instructions on how to retrieve your customer ID in case you cannot remember it. After entering your name and customer ID (see Figure 67), press the Continue button.
On the Personal Information page, you can modify your current address (you want to make sure that the Web Travel Agent mails the tickets and vouchers to the right address) and telephone number. To update the database with the new information, press **Change**. Clicking on the **Done** button or on the icon on the bottom of the page returns you again to the Welcome View of the application.

![Customer Authentication](image-url)
View Attractions

Nobody expects Web Travel Agent customers to book a travel package without seeking some information about the places they are going to. The WTA application allows for viewing a video about the selected location or directly connecting to a Web site about the travel destination (Figure 69).

You can retrieve all necessary information for your travel plans from the city’s home page or Chamber of Commerce page. After all, as a purely online service, the Web Travel Agent does not do mass mailings of catalogs.
Using the Web Travel Agent

Figure 69. View Attractions

Select a city from the listbox (Figure 69) and click on either the Video button or the WWW button. The application accordingly returns a video (AVI) file (see example in Figure 70) or links to the appropriate Internet site (see example in Figure 71).

To let you watch the video, your Web browser may launch an external viewer; in order to be able to see the external Web pages on the Internet, your browser must have a configured proxy or socks server or must have an active connection to your Internet service provider (ISP).
Using the Web Travel Agent

Figure 70. Tahiti Video

Figure 71. Chicago Home Page
Exception

There is no exception to the rule that an exception is always the rule. Apart from handling errors that occur when you forget to fill mandatory fields on a Web page or when you enter invalid data (these exceptions are usually handled by the affected part itself) one error results from actually doing nothing. Suppose you start putting together your travel package but the phone rings or your in-laws stop by for a surprise visit. After an hour, you return to your browser and try to continue with your trip booking. What you see is an error (Figure 72) because the Web Travel Agent assumed that you changed your mind and decided not to book your trip after all. The application invalidates your unfinished package to reduce the amount of required memory. The Web Travel Agent invalidates packages after the Web Connection default idle time of 900 seconds, that is, if you do not connect to the application within 15 minutes, you have to start the package over.

Figure 72. Exception
Implementation with VisualAge for Smalltalk

In this chapter, we discuss the Smalltalk implementation of the Web Travel Agency application. We will explain the usage of the Web Connection parts and how they fit into the visual web pages by showing the parts’ Composition Editor, the Smalltalk scripts and the resulting HTML.

Application Architecture

The overall concept of the different products and other application-related areas is shown in Figure 73. The center of the graphic is dominated by VisualAge for Smalltalk and its components, which build the core of our sample application.
VisualAge surrounds the database classes, our written application (WTA), and the Web Connection feature. On the top of the VisualAge block the VisualAge image interfaces with the DB2/2 database, although the actual database may reside on another system on the network. On the bottom of the VisualAge block, the CGI Link Server interfaces with the Internet Connection Server, to which it sends the generated HTML. The Internet Connection Server itself communicates with Web browsers on the Internet or on an intranet through HTTP.

The four blocks on the left represent application-related additions, such as regular HTML files, images, video clips, or Java applets. All those files are stored in their appropriate directories outside the VisualAge tree. They are referenced from within those parts of the application that contain a path to the external resources.

The right-most block stands for the DB2 WWW Connection gateway, that is used for the administrator application described in Chapter 11, “Implementation of Administrative Functions,” on page 365. The gateway function accesses the WTA database and presents its data through the CGI.

Figure 73. WTA Application Architecture
VisualAge Parts

In this section, we describe all parts that constitute the Web Travel Agency application. Instead of following an alphabetical order, we arrange them in the order in which you would pass through the application.

Several different part types are used, from nonvisual parts that represent the business objects (such as customer, hotel, or car), to various flavors of visual Web parts. The View parts contain mostly HTML pages and little business logic. The Processor parts are called from an HTML form of a View part to execute the selected business logic, depending on the pressed form button. The Controller parts do preprocessing before a View part generates its HTML. For more information on part types, refer to “Part Types” on page 349.

Now, let us discuss the internal workings of the VisualAge for Smalltalk implementation of the Web Travel Agency application. The purpose is to point out some of the issues and considerations when programming with the Web Connection. Our review is not exhaustive, nor is every possibility examined, but it does provide a working example and some hints and tips. We trust that when examining our code, you will find ways to improve on it in the applications you write.

Figure 74 shows a schematic diagram of the Web Travel Agency application.

WtaWelcomeView

The application begins with a request to the Web browser.

http://<wta server>/cgi-bin/abtcgil.exe/WtaWelcomeView

Using the IBM WebExplorer, what the user sees is shown in Figure 75.
VisualAge Parts

WtaWelcomeView
WtaWelcomeProcessor
*create* WtaPackageCreationController
WtaPackageCreationView
WtaPackageCreationProcessor
!error ! WtaPackageCreationView
*continue* WtaComponentSelectionView
WtaComponentSelectionProcessor
*hotel...* WtaHotelSelectionView
WtaHotelSelectionProcessor
*flight...* WtaFlightSelectionView
WtaFlightSelectionProcessor
*car.....* WtaCarSelectionView
WtaCarSelectionProcessor
*cancel..* WtaPackageCreationController
*continue* WtaPackagePricer
!failed! WtaPackageFailedView
!ok....! WtaPackagePurchaseView
WtaPackagePurchaseProcessor
*purchase* WtaCustomerIdView
WtaCustomerIdProcessor
*existing* WtaCustomerAuthenticationView
WtaCustomerAuthenticationProcessor
!error ! WtaCustomerAuthenticationView
*continue* WtaCreditCardInfoView
WtaCommiter
WtaPackageConfirmationView
*new.....* WtaNewCustomerIdView
WtaNewCustomerIdProcessor
*reset...* WtaCustomerIdProcessor
*done....* WtaCreditCardInfoView
WtaReminder
WtaPackagePurchaseView
WtaPersonalInfoView
WtaPersonalInfoProcessor
*display* WtaCustomerAuthenticationView
WtaCustomerAuthenticationProcessor
*continue* WtaPackageListView
WtaPackageListProcessor
*modify.* WtaCustomerAuthenticationView
WtaCustomerAuthenticationProcessor
*continue* WtaPersonalInfoView
WtaPersonalInfoProcessor
*view...* WtaAttractionView
WtaAttractionProcessor

Legend
-------
! ... ! error condition
* ... * push button

Figure 74. Schematic Diagram of the WTA Application
Chapter 8. Implementation with VisualAge for Smalltalk

Since this is the Smalltalk plumbing chapter, let us look at the HTML provided to the Web browser to render the Welcome View.

Figure 75. Welcome to the Web Travel Agency

What? No graphics in the Web browser?

Your Composition Editor shows graphics but not the Web browser? The graphic shown for a part is defined in its settings view. Graphics are displayed during run-time and edit time. The absolute reference for the graphics subdirectory is c:\wta\wtapix\. In the settings view, the relative reference and file name brings up the graphic, using /wtapix/wta.gif. Verify that the graphics are exactly where the settings view for the part expects them.

Since this is the Smalltalk plumbing chapter, let us look at the HTML provided to the Web browser to render the Welcome View.
Welcome to the Web Travel Agency!
Providing one-stop shopping for your business and leisure travel needs.

Click the button for the service you desire:

- Create a new travel package.
- Display your existing travel packages.
- Modify your personal information.
- See attractions in cities around the world.

See attractions in cities around the world.

About the Web Travel Agent
The form is the interesting part of the HTML because it provides the interface between the Web and VisualAge for Smalltalk. The interface is four push buttons, each of the type submit. Now open the Composition Editor for \textit{WtaWelcomeView}. The Composition Editor for this class looks like the window shown in Figure 76.

![Figure 76. WtaWelcomeView](image)

**What? No graphics in the Composition Editor?**

Your Web browser shows graphics but not the Composition Editor? The graphic shown for a part is defined in its settings view. Graphics are displayed during run-time and edit time. When developing the application all the graphics for the run-time copy were placed in \texttt{c:\wta\wtapix\}. Unfortunately, in the release used to develop this application the file specification is absolute, not relative. Verify that the graphics are exactly where the settings view for the part expects them.

Simply stated, the Composition Editor contains two parts: an \textit{HTML Page} part and a \textit{CGI Link Session Data} part. Requested Package Session Data is our renaming of the \textit{CGI Link Session Data} part (see Chapter 6, “Analysis and Design,” on page 105 for a discussion of a requested package.) The \textit{Session Data} part holds an instance of the business domain class, \textit{WtaRequestedPackage}. Tying this class to the session data allows us to store information for the duration of the end user’s interaction with the application.
(The session will end after a specified period of inactivity, the default is set to 900 seconds.) We make frequent use of the Requested Package Session Data part. All the interesting state information concerning the end user’s interaction with the application is kept here. When a customer makes a purchase, the state information is made persistent by updating the database.

The HTML Form part is the central portion of the HTML Page part. It appears to the VisualAge for Smalltalk programmer as a group box, yet it is invisible on the Web browser. The Web Connection provides a visual hint (a thin rectangle) to show where the <FORM ...> tag starts and where it ends (with a </FORM> tag). A nonvisual part is represented as a visual part! This is a difference from traditional VisualAge for Smalltalk. It occurs because the Composition Editor is used to specify HTML pages, not a GUI window.

---

**Part List**

A container icon tree list of parts used in the Composition Editor is available by moving the mouse pointer to any open space on the free-form surface and pressing mouse button 2. From the pop-up menu, click on View Parts List. You will see a window like this:

Look at the settings view for the HTML Form part. (You can double-click anywhere within the box on the HTML Page part or double-click on the icon in the parts list.) The text entry field labeled Part name contains the class name for the part that will process the form data. When the end user clicks on a button, the form data is returned to the Web Connection and WtaWelcomeProcessor is invoked. This class is defined on the action attribute of the <FORM> tag.

The short lines on both sides of the four push buttons are actually small graphics that we use to get some space between the icons, the buttons, and the text. We had rather put all icons, buttons, and texts into a table, but the HTML Table part does not support inclusion of graphics in this version of Web Connection.
The box shown on the bottom of the HTML Page part is the upper left portion of the reusable composite footer. We describe the footer in “Reusable Footer” on page 246.

WtaWelcomeProcessor

The Composition Editor for WtaWelcomeProcessor (Figure 77) consists entirely of nonvisual parts to process the user interaction captured in the Web browser and transmitted to our application as form data. Look at the settings view for the HTML Form Data part, named Welcome View Form Data. It refers back to WtaWelcomeView, the part were we just clicked on any of the four buttons. The tear-off attributes for the four push buttons are exposed as though they are standalone parts. The parts are referenced in the script, storeServiceTypeAndRoute. This script is fired from the CGI Link Request part’s event-to-script connection. The requestReceived event provides a means for triggering program logic.

![Figure 77. WtaWelcomeProcessor](image)

The script tests each of the push button’s clicked events to determine which has a value. When a clicked button is identified, an instance variable, requestedService, in the session data composite class, WtaRequestedPackage is assigned a value to remember which button was clicked, control is then passed to the appropriate class.
In the application, only View Attractions goes directly to a class suggested by the button title. Display Package and Modify Personal Info both reference an intermediate class for authentication. Create Package goes to WtaPackageCreationController.

The WtaWelcomeProcessor Composition Editor (Figure 77) further shows three page wrappers, but we have four push buttons. A page wrapper can be any class; it can be an HTML Page part (the Web Connection’s equivalent of a visual part) or it can consist entirely of nonvisual parts. All three HTML Page Wrapper settings points to a class name that is called for further processing. The Requested Package Session Data part is also available.

The path for the Create a new travel package branch of the application is where most users will spend their time. So let’s follow that code.

**WtaPackageCreationController**

The WtaPackageCreationController part (Figure 78) sets all the values of the session data composite part (WtaRequestedPackage) to nil, except one. The requestedService attribute holds the state of the push button clicked in WtaWelcomeView. Once any possible residue from an abandoned package creation is cleared, control is transferred to WtaPackageCreationView.
All logic is called through the CGI Link Request part’s connection to the initializeRequestedPackage method. The Session Data part’s torn-off value attribute and the Page Wrapper part are referenced in this script.

```smalltalk
initializeRequestedPackage

"A new package has been requested, so initialize the session data, which may have values left over from a previous create package request in this session. Note that requestedService is not reset since we are still in the package creation function."

| reqPkg |
reqPkg := (self subpartNamed: 'value of Requested Package Session Data') value.
reqPkg.
customerId: nil;
locationName: nil;
fromCity: nil;
packageId: nil;
packagePrice: nil;
startDate: nil;
endDate: nil;
numberOfDays: nil;
cardCompany: nil;
cardNumber: nil;
cardExpirationDate: nil;
hotel: nil;
flight: nil;
car: nil;
requestedHotelRooms: nil;
requestedFlightSeats: nil;
requestedCar: nil;
errorMessage: nil;
video: nil;
url: nil.

^(self subpartNamed: 'Package Creation Page Wrapper') transferRequest.
```
WtaPackageCreationView

To book a travel package the location, date of departure, and duration of stay need to be obtained. A screen is dedicated for collection this information (Figure 79). If there are problems with the entered date, for example, specifying a departure date in the past, this screen is modified to display an error message and prompt for the correct data.

WtaPackageCreationView also uses nonvisual parts and scripts. The list of travel destinations is read from the LOCATION table.

The formatLocations script uses the LOC_NAME column from the resultTable of the database query to add all retrieved city names to the list box. The city names in the list box are formatted as a collection of strings. The current date is obtained as a string and included on the page as a cue to the end user for avoiding dates in the past.
Create a Travel Package

Select a fabulous destination city:

- Atlanta
- Boston
- Chicago
- Dallas
- Frankfurt
- San Francisco
- San Jose

Select a start date. Today's date is Thursday, October 17, 1996.

Select the number of days:

Enter the number of days:

Return to the Web Travel Agency Home Page
Smalltalk

```
formatLocations
"Fetch candidate locations from the query result table."
| locations city |
locations := OrderedCollection new.
((self subpartNamed: 'resultTable of Multiple Row Query1')
 valueOfAttributeNamed: #rows selector: #IS_rows')
do: [:each | each isNil ifFalse: [
   city := (each at: 'LOC_NAME') trimBlanks.
   locations add: city.]].
^locations
```

WtaPackageCreationProcessor

Figure 80 shows the Composition Editor of the Package Creation Processor, which is called when the user clicks on **Continue** button on the WtaPackageCreationView.
The first part to examine is the HTML Form Data part. It specifies WtaPackageCreationView. Examining the data captured in the end user’s interaction with this view, we can:

- Validate data.
- Create an error message.
- Update persistent state information.
- Determine the next view to present.

These tasks are accomplished by tearing off attributes from the form data and session parts and a few lines of Smalltalk. The scripts begin by checking that the date is valid and that it has not passed. As soon as an error is detected, an error message is set in WtaRequestedPackage and control is passed to WtaPackageCreationView. The event-to-script connection from the CGI Link Request part starts the first method, checkDaysInMonth, and sequences through:

- checkIfDatesPassed
- processDuration
- processDateString
- processLocationName

The first three methods can raise data validation errors. The last three methods store information in the WtaRequestedPackage. The customer is presented with the WtaPackageCreationView part until everything is in order or until the decision is made to return to the Web Travel Agency application home page.

```
checkDaysInMonth

"Make sure that the customer did not select a day
that does not exist for the month/year."
| month monthSymbol day dayNumber year yearNumber maxDayNumber |

month := (self subpartNamed: 'Month List of Package Creation Form Data') value.
monthSymbol := month asSymbol.
day := (self subpartNamed: 'Day Number List of Package Creation Form Data') value.
dayNumber := day asNumber.
year := (self subpartNamed: 'Year List of Package Creation Form Data') value.
yearNumber := year asNumber.
maxDayNumber := Date daysInMonth:
monthSymbol forYear: yearNumber.

dayNumber > maxDayNumber ifTrue: [
  (self subpartNamed: 'errorMessage of value of Requested Package Session Data')
  value: 'Oops, you entered day ', day, ' but there are only ',
  maxDayNumber printString, ' days in ', month, ', ', year, '. Try again!'.

  ^(self subpartNamed: 'Package Creation View Page Wrapper') transferRequest]

ifFalse: [
  (self subpartNamed: 'errorMessage of value of Requested Package Session Data')
  value: nil.

  ^self checkIfDatesPassed]
```

Chapter 8. Implementation with VisualAge for Smalltalk
VisualAge Parts

Smalltalk

checkIfDatesPassed
"Check if customer entered a date before today's date."
| month monthSymbol day dayNumber year yearNumber dateEntered |

month := (self subpartNamed: 'Month List of Package Creation Form Data') value.
monthSymbol := month asSymbol.
day := (self subpartNamed: 'Day Number List of Package Creation Form Data') value.
dayNumber := day asNumber.
year := (self subpartNamed: 'Year List of Package Creation Form Data') value.
yearNumber := year asNumber.
dateEntered := Date newDay: dayNumber month: monthSymbol year: yearNumber.
dateEntered < Date today
ifTrue: [ 
  (self subpartNamed: 'errorMessage of value of Requested Package Session Data')
  value: 'Oops, you entered a date preceeding today’s date. Try again!'.
  ^(self subpartNamed: 'Package Creation View Page Wrapper') transferRequest]
ifFalse: [
  (self subpartNamed: 'errorMessage of value of Requested Package Session Data')
  value: nil.
  ^(self processDuration).
]

Smalltalk

processDuration
"Make sure trip duration is at least 1."
| duration durationNumber reqPkg |

reqPkg := (self subpartNamed: 'value of Requested Package Session Data') value.
duration := (self subpartNamed: 'Duration Entry Field of Package Creation Form Data')
  value asNumber.
duration isInteger ifTrue: [ 
  duration strictlyPositive ifTrue: [ 
    (self subpartNamed: 'errorMessage of value of Requested Package Session Data')
    value: nil.
    reqPkg numberOfDays: duration printString.
    ^(self processDateStrings)].
  ^(self subpartNamed: 'errorMessage of value of Requested Package Session Data')
  value: 'Oops, you entered ,
  (self subpartNamed: 'Duration Entry Field of Package Creation Form Data')
  value printString, ‘ for the number of days. Try again!’.]
  ^(self subpartNamed: 'Package Creation View Page Wrapper') transferRequest.

Smalltalk

processDateStrings
"Create the start date and end date from the Customer selected list strings and store." 
| startDate monthSymbol day dayNumber year yearNumber reqPkg |

reqPkg := (self subpartNamed: 'value of Requested Package Session Data') value.
month := (self subpartNamed: 'Month List of Package Creation Form Data') value.
monthSymbol := month asSymbol.
day := (self subpartNamed: 'Day Number List of Package Creation Form Data') value.
dayNumber := day asNumber.
year := (self subpartNamed: 'Year List of Package Creation Form Data') value.
yearNumber := year asNumber.
startDate := Date newDay: dayNumber month: monthSymbol year: yearNumber.
reqPkg startDate: startDate.
reqPkg endDate: (startDate addDays: (reqPkg numberOfDays asNumber - 1)).
^(self processLocationName).
WtaComponentSelectionView

For our purposes a travel package consists of a hotel, a round-trip flight, and a car. The Component Selection View (Figure 81) allows selection of one or more of these components. Status information about previous choices is provided by the attribute-to-attribute connections between text fields and WtaRequestedPackage. This method is used when the session data value is a string that requires no further formatting. Scripts are used to imbed a session data text in a sentence or format a date. Push buttons are available to begin pricing the travel package, discard all choices, and start fresh with component selection, or go back to WtaWelcomeView.

```smalltalk
processLocationName
"Save location in requested package."

(self subpartNamed: 'locationName of value of Requested Package Session Data') value:
(self subpartNamed: 'Location List of Package Creation Form Data') value.

^(self subpartNamed: 'Component Selection View Page Wrapper') transferRequest
```
Figure 81. WtaComponentSelectionView
VisualAge Parts

The HTML Form part shows WtaComponentSelectionProcessor as interpreter of the button clicks. This view is a prelude to other views where detailed processing occurs. The status information in the Component Selection View helps the end user remember what was chosen and what is available for selection.

The sayStartDate and sayEndDate methods retrieve the startDate and endDate attributes from the session data and edit it so that the dates look like Tuesday, December 24, 1996. Through attribute-to-attribute connections, the resulting strings are fed into the HTML page. The hotelPreferences, flightPreferences, and carPreferences methods retrieve and display a text for the corresponding travel package component, if it is selected. See Figure 58 for a Component Selection View with all components selected.
VisualAge Parts

Smalltalk

sayStartDate
"Answer the receiver with the package start date in English."
| startDate |

startDate := (self subpartNamed: 'startDate of value of Requested Package Session Data') value.

"startDate dayName asString, ', '
startDate monthName asString, ', '
startDate dayOfMonth printString, ', '
startDate year printString.

Smalltalk

sayEndDate
"Answer the receiver with the package end date in English."
| endDate |

endDate := (self subpartNamed: 'endDate of value of Requested Package Session Data') value.

"endDate dayName asString, ', '
endDate monthName asString, ', '
endDate dayOfMonth printString, ', '
endDate year printString.

Smalltalk

hotelPreferences
"answer receiver with any customer entered car preferences."
| reqPkg reqHotelRooms |

reqPkg := (self subpartNamed: 'value of Requested Package Session Data') value.
reqPkg requestedHotelRooms isNil
ifTrue: [^nil]
ifFalse: [
    reqHotelRooms := reqPkg requestedHotelRooms.
    ^'You have requested ', reqHotelRooms numberOfRooms, ' ', reqHotelRooms smoking, ' ', reqHotelRooms roomCategory, ' room(s) at the ', reqHotelRooms hotelName, 'Hotel.'].

Smalltalk

flightPreferences
"answer receiver with any customer entered flight preferences."
| reqPkg reqFlightSeats |

reqPkg := (self subpartNamed: 'value of Requested Package Session Data') value.
reqPkg requestedFlightSeats isNil
ifTrue: [^nil]
ifFalse: [
    reqFlightSeats := reqPkg requestedFlightSeats.
    ^'You have requested ', reqFlightSeats numberOfSeats, ' ', reqFlightSeats seatClass, ' airline seats departing from ', reqPkg fromCity, '.'].
WtaComponentSelectionProcessor

The Component Selection Processor examines which of the five buttons on the Component Selection View has been clicked and then routes to the appropriate next HTML page. The Component Selection Processor was originally implemented through pure visual programming, the WtaComponentSelectionProcessor part contained no scripts. The logic was contained in event-to-action connections from the HTML Form Data part and the appropriate page wrapper (see Figure 82).

Figure 82. WtaComponentSelectionProcessor (First Implementation)
After we decided to implement some rudimentary error handling (for example, when session data times out), we had to change the part so that before transferring the request to any view wrapper, session data is checked whether it is still alive. Therefore, we now have an event-to-script connection from the clicked event of every button to the appropriate method. The Composition Editor now looks like Figure 83.

**Figure 83. WtaComponentSelectionProcessor**

The `processHotel`, `processFlight`, `processCar`, `processContinue`, and `processCancel` scripts are shown below. They are basically identical, with the only exception that they route to different parts which are represented through HTML View Wrappers on the Composition Editor.

```smalltalk
processHotel
   " verify that session data is still alive, then process hotel button "
   ((self subpartNamed: 'value of Requested Package Session Data')
    valueOfAttributeNamed: #locationName selector: #'IS_locationName') isNil
    ifTrue: [
        ^(self subpartNamed: 'Exception View Page Wrapper') transferRequest
    ].

    ^(self subpartNamed: 'Hotel Selection Page Wrapper') transferRequest
```

World Wide Web Programming with VisualAge for C++ and Smalltalk
VisualAge Parts

Smalltalk

processFlight
" verify that session data is still alive,
then process flight button "
((self subpartNamed: 'value of Requested Package Session Data')
valueOfAttributeNamed: #locationName selector: #IS_locationName) isNil
ifTrue: [
  ^(self subpartNamed: 'Exception View Page Wrapper') transferRequest ].
^((self subpartNamed: 'Flight Selection Page Wrapper') transferRequest)

processCar
" verify that session data is still alive,
then process car button "
((self subpartNamed: 'value of Requested Package Session Data')
valueOfAttributeNamed: #locationName selector: #IS_locationName) isNil
ifTrue: [
  ^(self subpartNamed: 'Exception View Page Wrapper') transferRequest ].
^((self subpartNamed: 'Car Selection Page Wrapper') transferRequest)

processContinue
" verify that session data is still alive,
then process Continue button "
((self subpartNamed: 'value of Requested Package Session Data')
valueOfAttributeNamed: #locationName selector: #IS_locationName) isNil
ifTrue: [
  ^(self subpartNamed: 'Exception View Page Wrapper') transferRequest ].
^((self subpartNamed: 'Package Pricer Wrapper') transferRequest)

processCancel
" verify that session data is still alive,
then process Cancel button "
((self subpartNamed: 'value of Requested Package Session Data')
valueOfAttributeNamed: #locationName selector: #IS_locationName) isNil
ifTrue: [
  ^(self subpartNamed: 'Exception View Page Wrapper') transferRequest ].
^((self subpartNamed: 'Package Creation Controller Wrapper') transferRequest)
WtaHotelSelectionView

The database holds all the hotel accommodations available to this application; the table could theoretically contain several hundred rows; however, we present to the customer only those hotels in the destination city. The Hotel Selection View (Figure 84) allow the customer to select the desired hotel in the destination city, the preferred room category (standard, deluxe, or suite), a smoking or nonsmoking room, and the total number of rooms. (In a real-life application, a customer would be able to select, for example, one nonsmoking suite and two deluxe smoking rooms. In our sample application, the room category and the smoking selection applies to all rooms.) Default values are indicated for the room category radio buttons and the smoking check box. The HTML Entry Field part, number of rooms, is the only customer input validated. An integer converter is specified in the settings view and an integer greater than zero is tested for in the script. No check is made for reasonableness. (To amaze your non-Smalltalk friends enter a long integer value for the number of rooms and then click on Continue to check the price in Select Package Components.) WtaRequestedPackage is updated after the validation checks are satisfied. Hotel availability is checked in a downstream class when the customer is committing to the travel plans.

Figure 84. WtaHotelSelectionView
The `queryForHotelsByLocation` method is used to load the listbox on the HTML page with a list of hotel names for the selected city. The HOTEL table is queried through the following SQL statement:

```sql
SELECT HOTEL.HOTEL_NAME FROM HOTEL
WHERE HOTEL.LOC_NAME = location
```

The term `location` represents the `locationName` attribute from the HTML Session Data part. The result table is fed into a `SortedCollection` part, which is returned at the end of the method.
The Hotel Selection Processor (Figure 85) investigates the customer selections on the preceding Hotel Selection View. The Composition Editor shows the HTML Form Data part with its torn-off attributes for the hotel listbox, number of rooms entry field, room category radio buttons, and the smoking check box. These attributes are used in the processHotelSelectionForm method, that is triggered by the CGI Link Request part. 

```
queryForHotelsByLocation
"get hotel names for the location requested."
| activeConnection querySpec result aSC reqPkg |
reqPkg :=(self subpartNamed: 'value of Requested Package Session Data') value.
aSC := SortedCollection sortBlock: [:a :b | a < b].
activeConnection := (AbtDbmSystem activeDatabaseConnection).
querySpec := (AbtQuerySpec new) statement:
   'select HOTEL.HOTEL_NAME from HOTEL where HOTEL.LOC_NAME = ',
reqPkg locationName printString.
result := activeConnection resultTableFromQuerySpec: querySpec.
result do: [:each | each isNil ifFalse: [aSC add: (each at: 'HOTEL_NAME') trimBlanks.]].
^aSC
```

**WtaHotelSelectionProcessor**

The Hotel Selection Processor (Figure 85) investigates the customer selections on the preceding Hotel Selection View. The Composition Editor shows the HTML Form Data part with its torn-off attributes for the hotel listbox, number of rooms entry field, room category radio buttons, and the smoking check box. These attributes are used in the processHotelSelectionForm method, that is triggered by the CGI Link Request part.

**Figure 85. WtaHotelSelectionProcessor**
The `processHotelSelectionForm` method first verifies the availability of the session data before further processing can continue. If session data is invalid, the part transfers the request to the Exception View, where an error message is shown. If session data is valid, the method continues and calls the `processNumberOfRooms` method to verify that a positive integer is entered in the form’s entry field. If a mistake is encountered, an error message is returned, and the Hotel Selection View is shown again.

Then, the radio buttons and the check box are examined. If the customer did not select a room category or did not choose smoking or nonsmoking, default values are set and stored in the requested package session data. The number of nights for the hotel stay is calculated by length of trip minus one. The result is also copied into session data. Finally, the method ends by calling the `processHotel` method.

```
Smalltalk

processHotelSelectionForm
"Process the hotel preferences form setting the Requested Hotel and Hotel objects."
| reqPkg |

|((self subpartNamed: 'value of Requested Package Session Data')
valueOfAttributeNamed: #locationName selector: #'IS_locationName') isNil
ifTrue: [ ^(self subpartNamed: 'Exception View Page Wrapper') transferRequest ].

reqPkg := (self subpartNamed: 'value of Requested Package Session Data') value.
reqPkg requestedHotelRooms isNil
ifTrue: [reqPkg requestedHotelRooms: WtaRequestedHotelRooms new].

self processNumberOfRooms.
reqPkg errorMessage isNil
ifTrue: [
  reqPkg requestedHotelRooms hotelName:
  (self subpartNamed: 'Hotel Name List of Hotel Selection Form Data') value.
  (self subpartNamed: 'Room Category Radio Buttons of Hotel Selection Form Data')
  value isEmpty
  ifTrue: [ reqPkg requestedHotelRooms roomCategory: 'Standard'
  ifFalse: [ reqPkg requestedHotelRooms roomCategory: (self subpartNamed:
  'Room Category Radio Buttons of Hotel Selection Form Data') value].
  (self subpartNamed: 'Smoking Check Box of Hotel Selection Form Data')
  value
  ifTrue: [reqPkg requestedHotelRooms smoking: 'Smoking'
  ifFalse: [reqPkg requestedHotelRooms smoking: 'Nonsmoking'].
  reqPkg requestedHotelRooms numberOfNights:
  reqPkg numberOfDays asNumber asNumber - 1.
  ^self processHotel
  ifFalse: [ ^(self subpartNamed: 'Hotel Selection Page Wrapper') transferRequest

```
The `processHotel` method loads the row from the HOTEL table for the selected hotel. The following SQL query is used:

```
SELECT * FROM HOTEL
WHERE LOC_NAME = location AND HOTEL_NAME = name
```

This database query returns only one row, because we require a hotel name in a particular city to be unique. All column values are retrieved from the result row and stored in the corresponding attribute of the Hotel class. After the Hotel part is added to the requested package session data, the application returns to the Component Selection View.

```
Smalltalk

processNumberOfRooms
"Make sure number of rooms is at least 1."
| number reqPkg |

reqPkg := (self subpartNamed: 'value of Requested Package Session Data') value.
number := (self subpartNamed: 'Number of Rooms Entry Field of Hotel Selection Form Data') value asNumber.
number isInteger ifTrue: [
  number strictlyPositive ifTrue: [
    reqPkg requestedHotelRooms numberOfRooms: number printString.
    ^reqPkg errorMessage: nil]
].

^reqPkg errorMessage:
"Oops, you entered ", (self subpartNamed: 'Number of Rooms Entry Field of Hotel Selection Form Data') value printString,
' for the number of rooms. Try again!'.

Smalltalk

processHotel
"Create the hotel object using the hotel requested by the customer."
| activeConnection querySpec result reqPkg |

reqPkg := (self subpartNamed: 'value of Requested Package Session Data') value.
reqPkg hotel isNil ifTrue: [reqPkg hotel: WtaHotel new].
activeConnection := (AbtDbmSystem activeDatabaseConnection).
querySpec := (AbtQuerySpec new) statement: 
'select from HOTEL where HOTEL.LOC_NAME = ', reqPkg locationName printString,
' and HOTEL.HOTEL_NAME = ', reqPkg requestedHotelRooms hotelName printString.
result := activeConnection resultTableFromQuerySpec: querySpec.
result do: [:each | each isNil ifFalse: [
  reqPkg hotel hotelName: (each at: 'HOTEL_NAME') trimBlanks;
  hotelId: (each at: 'HOTELID') printString;
  locationName: (each at: 'LOC_NAME') trimBlanks;
  capacity: (each at: 'CAPACITY') printString;
  roomsBooked: (each at: 'BOOKED') printString;
  dailyPrice: (each at: 'DAILY_PRICE') printString.
  reqPkg requestedHotelRooms hotelId: (each at: 'HOTELID') printString]]].

^(self subpartNamed: 'Component Selection Page Wrapper') transferRequest.
```
WtaFlightSelectionView

The Flight Selection View (Figure 86) allows the customer to select a departure city for the itinerary, a seat class (Economy, Business, or First), and the number of required seats. The `aboutToGenerateHtml` event of the `HTML Page` part is connected to the `queryForLocations` script.

To make life simpler we decided a flight is available to the destination city from all the potential departure cities. In SQL, that means a SELECT is done on the LOCATION table for all location names that differ from the destination city, because you would otherwise be able, for example, to fly from Dallas to Dallas. We use a `SortedCollection` part to ensure that departure city location names are sorted alphabetically. An ORDER BY clause in the SELECT statement could be used for the same effect.

As shown in Figure 86, this page requires the customer to specify the seat class. No default is provided. If a button is not clicked, the customer is requested to make a choice. Validation of the number of seats is identical to that for number of rooms when selecting a hotel. The `WtaRequestedPackage` part is updated when all validation checks are satisfied. We assume every flight has infinite capacity and no subsequent check is made for availability.

![WtaFlightSelectionView - Composition Editor](image)

Figure 86. `WtaFlightSelectionView`
Specify Flight Preferences

Select a departure city:
- Atlanta
- Boston
- Chicago
- Dallas
- Frankfurt
- San Jose

Select a seat class:
- Economy
- Business
- First Class

Enter the number of seats:

Submit

Smalltalk

\[\text{queryForLocations} \]

\[
\text{\{get departure location names, but leave out the name of the destination city.\}}
\]

\[
| \text{activeConnection} \text{ requiSpec result} aSC \text{ reqPkg} |
\]

\[
\text{reqPkg := (self partNamed: \text\{value of Requested Package Session Data\}} value.}
\]

\[
\text{aSC := SortedCollection sortBlock: [:a :b | a < b].}
\]

\[
\text{activeConnection := (AbtDbmSystem activeDatabaseConnection).}
\]

\[
\text{querySpec := (AbtQuerySpec new) statement:}
\]

\[
\text{\{select LOCATION.LOC_NAME from LOCATION where not LOCATION.LOC_NAME = \'.\}}
\]

\[
\text{reqPkg locationName printString.}
\]

\[
\text{result := activeConnection resultTableFromQuerySpec: querySpec.}
\]

\[
\text{result do: [:each | each isNil ifFalse: [aSC add: (each at: \text\{LOC_NAME\}} trimBlanks.]].}
\]

\[
\text{\{(self partNamed: \text\{Departure City List\}} items: aSC.}
\]
WtaFlightSelectionProcessor

The Flight Selection Processor (Figure 87) investigates the customer selections on the preceding Flight Selection View. The Composition Editor shows the HTML Form Data part with its torn-off attributes for the departure city listbox, the number of seats entry field, and the seat class radio buttons. These attributes are used in the `processFlightSelectionForm` method, that is triggered by the CGI Link Request part.

![Composition Editor Diagram]

**Figure 87.** WtaFlightSelectionProcessor

The `processFlightSelectionForm` method retrieves the session data instance and stores it in a local variable. Then it calls the `processNumberOfSeats` method to verify that a positive integer is entered in the form’s entry field. If a mistake is encountered, an error message is returned, and the Flight Selection View is shown again.

Then, the seat class radio buttons are examined. If the customer did not select a seat class, an error message is generated and the application returns to the Flight Selection View. The number of nights for the hotel stay is calculated by length of trip minus one. The result is also copied into session data.
Finally, the departure city is added to the session data object and the *smoking* attribute of the *requestedFlightSeats* part is set to zero (we assume that all our flights are nonsmoking). The script ends by calling the *processFlight* method.

```smalltalk
| reqPkg |
reqPkg := (self subpartNamed: 'value of Requested Package Session Data') value.
reqPkg requestedFlightSeats isNil ifTrue: [reqPkg requestedFlightSeats: WtaRequestedFlightSeats new].

self processNumberOfSeats. reqPkg errorMessage notNil ifTrue: [
  ^(self subpartNamed: 'Flight Selection Page Wrapper') transferRequest].

(reqPkg requestedFlightSeats seatClass: (self subpartNamed: 'Seat Class Radio Buttons of Flight Selection Form Data') value.
reqPkg fromCity: (self subpartNamed: 'Departure City List of Flight Selection Form Data') value.
reqPkg requestedFlightSeats smoking: 0. *self processFlight.

processNumberOfSeats
"Make sure number of seats is at least 1."
| duration number reqPkg |
reqPkg := (self subpartNamed: 'value of Requested Package Session Data') value.
number := (self subpartNamed: 'Number of Seats Entry Field of Flight Selection Form Data') value asNumber.

number isInteger ifTrue: [
  number strictlyPositive ifTrue: [
    reqPkg requestedFlightSeats numberOfSeats: number printString.
    *reqPkg errorMessage: nil]].

*reqPkg errorMessage:
'Oops, you entered ',
(self subpartNamed: 'Number of Seats Entry Field of Flight Selection Form Data') value,
' for the number of seats. Try again!'.
```
The `processFlight` method first verifies the availability of the session data. If session data has timed out, the part transfers the request to the Exception View, where an error message is shown, otherwise the script continues and adds a new `WtaFlight` instance to session data. Then, the FLIGHT table from the database is queried to find out the base price for the connection between the two selected cities. The following SQL statement is used:

```
SQL
SELECT * FROM FLIGHT
WHERE FROM_CITY = departure AND TO_CITY = location
```

The `departure` attribute is the city name entered on the Flight Selection View, the `location` attribute is the destination entered on the Package Creation View. The columns from the result row are copied to the `flight` attribute of session data, the flight number is also copied to the `requested-FlightSeats` attribute of session data. Finally, the method returns to the Component Selection View.

```
Smalltalk

processFlight
"Get the flight from the specified departure city
to the specified destination and create the flight object."
| activeConnection aSC querySpec result reqPkg |

((self subpartNamed: 'value of Requested Package Session Data')
  valueOfAttributeNamed: #locationName selector: #'IS_locationName') isNil
  ifTrue: [ ^(self subpartNamed: 'Exception View Page Wrapper') transferRequest ].

reqPkg :=(self subpartNamed: 'value of Requested Package Session Data') value.
reqPkg flight isNil ifTrue: [reqPkg flight: WtaFlight new].

activeConnection := (AbtDbmSystem activeDatabaseConnection).
querySpec := (AbtQuerySpec new) statement:
  'select * from FLIGHT where FLIGHT.FROM_CITY = ', reqPkg fromCity printString,
   ' and FLIGHT.TO_CITY = ', reqPkg locationName printString.
result := activeConnection resultTableFromQuerySpec: querySpec.
result do: [ :each | each isNil ifFalse: [ reqPkg flight flightNumber: (each at: 'FLIGHTNUM') trimBlanks;
   fromCity: (each at: 'FROM_CITY') trimBlanks;
   toCity: (each at: 'TO_CITY') trimBlanks;
   basePrice: ((each at: 'BASE_PRICE') printString).
   reqPkg requestedFlightSeats flightNumber: (each at: 'FLIGHTNUM') trimBlanks]].

^(self subpartNamed: 'Component Selection Page Wrapper') transferRequest.
```
**WtaCarSelectionView**

This is a very simple and straightforward part. No data is fetched from a data base. Only two sets of radio buttons are presented and no defaults are provided (see Figure 88). The **Classes** button allows to view images of cars representing the various classes.

All validation takes place in *WtaCarSelection Processor*. There, the method *processCarSelectionForm* checks that the form data for the radio button sets is not empty.

![WtaCarSelectionView](image)

*Figure 88. WtaCarSelectionView*
Notice the *Car Class Form Data* part on the Composition Editor in Figure 88. At this point, this form has not been filled out, thus its attributes have no values. The form will be filled out, when the customer clicks on the *Classes* button and then returns to the current Car Selection View. When the customer is presented this part the second time, the *aboutToGenerateHtml* event fires again and executes the *processCarClass* script. This time, a button on the Car Class View is selected and, depending on which button the customer clicked on, the *selectionIndex* attribute is set for the corresponding radio button.
WtaCarClassView

The Car Class View is a pure visual Web part, with no scripts defined. The central component on the HTML page is an HTML Table part, that is assembled by using literal HTML Text parts, buttons, and images. The appearance of the part does not at all represent the actual Web page, as you can see by comparing Figure 89 with Figure 57.

The text literals such as <td>, </td>, and <tr> are placed in between the buttons, images, horizontal rules, and regular text, so that the Web browser shows a correctly formatted HTML table. The form action points back to the Car Selection View, where the form data (the clicked button) is investigated.

To be able to edit the whole HTML page in the Composition Editor without extensive scrolling, we did not want the original car images to be shown in the Composition Editor. Instead, we chose a small car icon and changed all image settings to show different image files during run-time and edit-time. See Figure 90 for the settings of the Class B image.
Figure 89. *WtaCarClassView*
<!-- generated by VisualAge Web Connection on 10-17-96 at 8:47:54 PM -->
<html><head><title>Select Car Class</title></head>
<body background="/wtapix/backgrnd.jpg">
<img src="/wtapix/wta.gif">
<p>Select the car class of your choice.</p>
<form method="POST" action="WtaCarSelectionView">
<table cellpadding=5 border=1>
<tr><td><b>Compact</b></td>
<td><img src="/wtapix/metro.gif"></td>
<td><b>Economy</b></td>
<td><img src="/wtapix/bmw.gif"></td></tr>
<tr><td><b>Utility</b></td>
<td><img src="/wtapix/blazer.gif"></td>
<td><b>Full-size</b></td>
<td><img src="/wtapix/cadillac.gif"></td></tr>
<tr><td><b>Luxury</b></td>
<td><img src="/wtapix/mercedes.gif"></td>
<td><b>Sport</b></td>
<td><img src="/wtapix/ferrari.gif"></td></tr>
</table>
<p>
<input type="submit" name="Class A Button" value="Class A">
<input type="submit" name="Class B Button" value="Class B">
<input type="submit" name="Class C Button" value="Class C">
<input type="submit" name="Class D Button" value="Class D">
<input type="submit" name="Class E Button" value="Class E">
<input type="submit" name="Class F Button" value="Class F">
</p>
<input type="submit" name="Cancel Button" value="Cancel">
<input type="hidden" name="_ABT_FROM_PART" value="WtaCarClassView">
<input type="hidden" name="_ABT_SESSION_KEY" value="524F84AD1...68A1B7BA">
</form>
<img src="/wtapix/rainban.gif">
<p><a href="WtaWelcomeView"> Return to Web Travel Agency Home Page</a></p>
</body>
</html>
WtaCarSelectionProcessor

The Car Selection Processor examines the form on the Car Selection View and accordingly routes the next request. The part’s Composition Editor (Figure 91) show the CGI Link Request that connects to the processCarSelectionForm script.

The shown subparts, session data, page wrappers, and form data are used in the script for further processing.

The processCarSelectionForm method first validates session data, then creates a new requestedCar instance and adds it to the requested package.

Next, the Classes button on the form is examined, and, if clicked, the Car Class View is loaded. If the Classes button was not clicked, the customer must have selected the Continue button, thus the radio buttons on the form are inspected. If the customer did not select a car class or a rental company, an error message is generated and the Car Selection View is shown again.
processCarSelectionForm
"Store customer car preferences in requested car object."
| reqPkg |
((self subpartNamed: 'value of Requested Package Session Data')
  valueOfAttributeNamed: #locationName selector: #IS_locationName) isNil
  ifTrue: [ ^(self subpartNamed: 'Exception View Page Wrapper') transferRequest ].
reqPkg := (self subpartNamed: 'value of Requested Package Session Data') value.
reqPkg requestedCar isNil
  ifTrue: [reqPkg requestedCar := WtaRequestedCar new].
(reqPkg errorMessage: nil.
reqPkg requestedCar carCompany:
  (self subpartNamed: 'Car Company Radio Buttons of Car Selection Form Data') value.
reqPkg requestedCar carClass:
  (self subpartNamed: 'Car Class Radio Buttons of Car Selection Form Data') value.
  ^(self subpartNamed: 'Component Selection View Page Wrapper') transferRequest.
When all is well, the customer eventually returns to the Package Creation View. A summary of customer selections is compiled from each of the trip components and displayed. Now a decision needs to be made about the last two push buttons:

- **Continue**, check the price and purchase the package
- **Cancel**, go back to the Package Creation Controller (see page 168) and start again.

Let’s cover new ground and click on **Continue**.

**WtaPackagePricer**

The Package Pricer is called when the customer selected all desired package components. The pricing for the package is carried out in three steps:

1. Price car
2. Price flight
3. Price hotel

Each step is handled by its own method. The *CGI Link Request* part on the Composition Editor (see Figure 92), however, triggers the *pricePackage* script, which acts as the controlling method. The Package Pricer can have two results:

- All selected package components are available for purchase.
- At least one package component is unavailable.

In the first case, the application is routed to the Package Purchase View, in the second case, it is routed to the Package Failed View. The Composition Editor contains wrappers for both parts.

![Figure 92. WtaPackagePricer](image)
The *pricePackage* method first initializes the *packagePrice* variable with zero. To this variable we eventually add the separately calculated component prices. We check whether any of the components (car, flight, or hotel) is selected by the customer. If so, we call the appropriate methods, *priceCar*, *priceFlight*, and *priceHotel*.

The *priceHotel* method is the only one that can return an error. If we receive the error message *No Vacancy*, the application is routed to the Package Failed View. If all components are available (or no hotel was selected), the total price is copied to the Requested Package Session Data part, and the Package Purchase View is requested.

```smalltalk
pricePackage

"Price the package by pricing each requested component."

| reqPkg |

reqPkg := (self subpartNamed: 'value of Requested Package Session Data') value.
reqPkg packagePrice: 0.
reqPkg requestedCar notNil
  ifTrue: [
    self priceCar.
    reqPkg packagePrice: (reqPkg packagePrice + (reqPkg requestedCar price)).
  ].
reqPkg requestedFlightSeats notNil
  ifTrue: [
    self priceFlight.
    reqPkg packagePrice: (reqPkg packagePrice + (reqPkg requestedFlightSeats price)).
  ].
reqPkg requestedHotelRooms notNil
  ifTrue: [
    self priceHotel.
    reqPkg errorMessage = 'No Vacancy'
    ifTrue: [
      reqPkg requestedHotelRooms: nil.
      "Note - keep hotel object for now to output hotel name in PackageFailedView"
      ^(self subpartNamed: 'Package Failed Page Wrapper') transferRequest
    ].
    ifFalse: [
      reqPkg packagePrice: (reqPkg packagePrice + (reqPkg requestedHotelRooms price)).
    ].
  ].
reqPkg packagePrice = 0
  ifTrue: [^(self subpartNamed: 'Package Failed Page Wrapper') transferRequest]
  ifFalse: [^(self subpartNamed: 'Package Purchase Page Wrapper') transferRequest].
```

The *priceCar* method first retrieves the selected car class from session data. The result is a string, such as 'Class D'. We are interested only in the seventh character of that string, thus we extract it from the *carClass* attribute.

To make our application simpler, instead of going back to the CAR table in the database to find out the price for the selected class, we hard-coded the class price in the method. The price is multiplied by the *numberOfDay* attribute, that we also get from session data. The total price for the car is put back into session data, and the method ends.
The `priceFlight` script gets the seat class and the base price for the requested flight from session data. The `seatClass` attribute is retrieved from the `requestedFlightSeats` subpart, and the `basePrice` attribute is retrieved from the `flight` subpart.

We implemented a fairly simple model for pricing the various classes of service. The base price represents economy class, if you booked business class, an uplift of 25% is added, in first class you have to pay a 50% markup.

The total price for all flight seats is calculated by multiplying the seat price, the possible uplift, and the number of seats. The result is stored in the `price` attribute of the `requestedFlightSeats` subpart in session data.
VisualAge Parts

In our application, the hotel entity is the only one that can be exhausted and make a reservation unavailable. We assume, however, that you can always successfully reserve a rental car and seats on any flight (obviously, this is not a real-life application).

Atlanta Olympics

To be able to demonstrate the No Vacancy message in our WTA application, we had to set up the HOTEL table in a way that a customer can select a hotel that is completely booked. The WTA application was developed in the summer of 1996, the year of the Atlanta Olympic Games. Therefore, to reflect a real-life situation, try finding a hotel room in Atlanta.

The priceHotel method first compares the roomsBooked and capacity attributes from the hotel subpart of session data. If they are equal, no rooms are available and an error message is generated.

If rooms are available, the method then also retrieves the selected room category and the daily price from session data. As in the priceFlight method, we use a simple calculation model for the room category uplift to determine the total price. A standard room costs the base price, a deluxe room costs 25% more, and a suite costs 50% more.

Smalltalk

priceHotel

"Price the hotel."

| reqPkg aRoomCategory aDailyPrice aRoomPrice |
reqPkg := (self subpartNamed: 'value of Requested Package Session Data') value.
reqPkg hotel roomsBooked = reqPkg hotel capacity
ifTrue: ['reqPkg errorMessage: 'No Vacancy'].
aRoomCategory := reqPkg requestedHotelRooms roomCategory.
aDailyPrice := reqPkg hotel dailyPrice asNumber.
aRoomCategory = 'Standard' ifTrue: [aRoomPrice := aDailyPrice].
aRoomCategory = 'Deluxe' ifTrue: [aRoomPrice := aDailyPrice * 1.25].
aRoomCategory = 'Suite' ifTrue: [aRoomPrice := aDailyPrice * 1.50].
*reqPkg requestedHotelRooms price:
(aRoomPrice *
  (reqPkg requestedHotelRooms numberOfRooms asNumber))
* reqPkg numberOfDays asNumber).

WtaPackagePurchaseView

Until now, the pattern in the application flow has been to who an HTML page followed by a processor page. The WtaPricer class uses the information collected in the WtaRequestedPackage part and the database to determine the financial effect of the customer’s choices. This is a processing step concerned only with nonvisual information. After WtaPackagePricer, the remainder of the application focuses on displaying the price, getting the customer’s commitment, and updating the database.
It is here that the availability of a package component, such as a hotel, is determined. When a component is not available, an error message is stored in *WtaRequestedPackage* and displayed in the Package Failed View. The customer clicks on **Continue** and is returned to the Component Selection View.

When all the package components are available, prices for each component and the grand total are displayed by the Package Purchase View (see Figure 93). Prices for the hotel and the flight are calculated from base values stored in the Web Travel Agency database. The price for the car is hardcoded in the Package Pricer.

Hotel and flight prices are read from the database when the package component is selected. During the database read, the price information is stored in *WtaRequestedPackage*, anticipating its use in pricing. By using the persistence provided by the CGI Link Session Data part, and a little memory, we avoid a database access in *WtaPackagePricer*.

In a CGI Link program, everything received from the browser is a string. In our code we store the strings as we receive them. Conversion to a number is postponed until computation or writing to the database requires conversion.

Figure 93. *WtaPackagePurchaseView*
The Composition Editor of the Package Purchase View shows a number of place holder strings, for example, \textit{cityName}, \textit{daysDuration}, or \textit{Hotel preferences go here}. All those strings are replaced at run-time either through an attribute-to-attribute connection from a session data subpart or through the result from an attribute-to-script connection. Refer to the following HTML and to Figure 59 for an example of a completed Package Purchase View.

\begin{verbatim}
<html><head><title>Package Summary and Purchase Option</title></head>
<body background="/wtapix/backgrnd.jpg">
<img src="/wtapix/wta.gif">
<p>You have selected a trip to <font size="4"><b>San Francisco</b></font> for <font size="4"><b>4</b></font> days, from <font size="4"><b>Tuesday, December 31, 1996</b></font> to <font size="4"><b>Friday, January 3, 1997</b></font>.
<hr>
<h3>Hotel Preferences</h3>
You have requested 1 Nonsmoking Deluxe room(s) at the Hyatt Embarcadero Hotel.
<br>
The price for the hotel is $1,400.00.
<br>
<h3>Flight Preferences</h3>
You have requested 2 Economy airline seats departing from Chicago.
<br>
The price for the flight is $850.00.
<br>
<h3>Car Preferences</h3>
You have requested a Class C - Utility car from Hertz.
<br>
The price for the car is $168.00.
<br>
<h3>Total Price</h3>
The total package price is $2,418.00.
<br>
<input type="submit" name="Do Purchase Push Button" value="Purchase" >
Preferences and price are satisfactory, buy the package.
<br>
<input type="submit" name="Change Preferences Push Button" value=" Change " >
Return to change preferences.
<br>
<input type="hidden" name="_ABT_FROM_PART" value="WtaPackagePurchaseView">
<input type="hidden" name="_ABT_SESSION_KEY" value="524F54AD1...6AA1B7BA">
</form>
</body>
</html>
\end{verbatim}

The following scripts are used in the attribute-to-script connections from the HTML Text parts on the HTML Page. All methods retrieve the appropriate subpart from session data and use the attributes to assemble a string that is shown on the Web page.

After reviewing the package components and prices, the customer can go back to the Component Selection View to change a selection or, if everything is satisfactory, click on the \textbf{Purchase} button.
hotelPreferences  
"answer receiver with any customer entered car preferences."

| reqPkg |

reqPkg := (self subpartNamed: 'value of Requested Package Session Data') value.
reqPkg requestedHotelRooms isNil
ifTrue: [ ^'A hotel room was not requested.']
ifFalse: [ ^'You have requested ',
  reqPkg requestedHotelRooms numberOfRooms, ' ',
  reqPkg requestedHotelRooms smoking, ' ',
  reqPkg requestedHotelRooms roomCategory,
  ' room(s) at the ',
  reqPkg hotel hotelName,
  ' Hotel.'].

Smalltalk

hotelPrice  
"answer receiver with the hotel price."

| reqPkg |

reqPkg := (self subpartNamed: 'value of Requested Package Session Data') value.
reqPkg requestedHotelRooms isNil
ifTrue: [ ^nil ]
ifFalse: [ ^'The price for the hotel is ',
  (self formatMoney: reqPkg requestedHotelRooms price), '.'].

Smalltalk

flightPreferences  
"answer receiver with any customer entered flight preferences."

| reqPkg |

reqPkg := (self subpartNamed: 'value of Requested Package Session Data') value.
reqPkg requestedFlightSeats isNil
ifTrue: [ ^'A flight was not requested.']
ifFalse: [ ^'You have requested ',
  reqPkg requestedFlightSeats numberOfSeats, ' ',
  reqPkg requestedFlightSeats seatClass,
  ' airline seats departing from ',
  reqPkg flight fromCity,
  '.'].
VisualAge Parts

Smalltalk

flightPrice
"answer receiver with the requested flight price."
| reqPkg |

reqPkg := (self subpartNamed: 'value of Requested Package Session Data') value.
reqPkg requestedFlightSeats isNil
ifTrue: ['nil']
ifFalse: ["The price for the flight is ", (self formatMoney: reqPkg requestedFlightSeats), "."].

Smalltalk

carPreferences
"answer receiver with any customer entered car preferences."
| reqPkg reqCar |

reqPkg := (self subpartNamed: 'value of Requested Package Session Data') value.
reqPkg requestedCar isNil
ifTrue: ['A car was not requested.']
ifFalse: [reqCar := reqPkg requestedCar.
"You have requested a ', reqCar carClass, ' car from ', reqCar carCompany, "."].

Smalltalk

carPrice
"answer receiver with the requested car price."
| reqPkg |

reqPkg := (self subpartNamed: 'value of Requested Package Session Data') value.
reqPkg requestedCar isNil
ifTrue: ['nil']
ifFalse: ["The price for the car is ", (self formatMoney: reqPkg requestedCar price), "."].

Smalltalk

formatMoney: aMoney
"Format a monetary amount, money, using Locale current specifications."
"Use the following expression to change the system wide format specifications.
alcMoney fracDigits: 2.
alcMoney monDecimalPoint: "."
| alcMoney aStream |

alcMoney := Locale current lcMonetary copy.
aStream := WriteStream on: String new.
alcMoney printNumber: aMoney on: aStream.
"aStream contents"
WtaPackagePurchaseProcessor

The Package Purchase Processor is a straightforward router part. There are no scripts. The processing logic is contained in two event-to-action connections between the HTML Form Data part and the two HTML Page Wrapper parts.

If the customer clicks on the Change button, the request is transferred to the Component Selection View. If the customer clicks on Purchase, the request is transferred to the Customer ID View.

Figure 94. WtaPackagePurchaseProcessor

We’ve seen what selecting a package component is like. Now, let’s proceed with purchasing the package.
WtaPackageFailedView

The Package Failed View is shown if any of the selected components (in our case, only the hotel) is not available. The part’s Composition Editor (Figure 95) contains basically the same information as it is shown on the Package Purchase View (see Figure 93), except that

- The title on the Web page says *Package Unavailable*.
- Instead of showing the hotel price, the page shows an error message.
- The *Purchase* button is not shown.

All scripts in the Package Failed View are the same as in the Package Purchase View, except that the *hotelPrice* method shows the message *Sorry, the requested hotel has not vacancies at this time. Try a different hotel.*
The customer has agreed to purchase the travel package. In return, the WTA application asks whether the customer is new or has already been entered into the system. Using the Customer ID View (Figure 96), the customer tells the application if he or she has an entry in the Web Travel Agency database or if one needs to be created. The Web page offers a simple selection of two alternatives, an Existing button and a New button.

Interestingly, there is no CGI Link Session Data part on the Composition Editor of WtaCustomerIdProcessor. So, how do we keep the session data information for use in the following parts?

When the HTTP server sends a request to the CGI Link Server, a Smalltalk process (thread) is started. The form data provided by the HTTP server is examined for a session key, which is matched to the CGI Link Session Data part. The session data composite object, WinRequestedPackage, is part of that same Smalltalk process.

```
hotelPrice "answer receiver with the hotel price."
| reqPkg |

reqPkg := (self subpartNamed: 'value of Requested Package Session Data') value.
reqPkg errorMessage = 'No Vacancy'
  ifTrue: [reqPkg errorMessage: nil.
    ^'Sorry, the requested hotel has no vacancies at this time. Try a different hotel.'].

reqPkg requestedHotelRooms isNil
  ifTrue: ['nil']
  ifFalse: [['The price for the hotel is ',
    (self formatMoney: reqPkg requestedHotelRooms price), ''].
```

**WtaCustomerIdView**

The customer has agreed to purchase the travel package. In return, the WTA application asks whether the customer is new or has already been entered into the system. Using the Customer ID View (Figure 96), the customer tells the application if he or she has an entry in the Web Travel Agency database or if one needs to be created. The Web page offers a simple selection of two alternatives, an Existing button and a New button.

Interestingly, there is no CGI Link Session Data part on the Composition Editor of WtaCustomerIdProcessor. So, how do we keep the session data information for use in the following parts?

When the HTTP server sends a request to the CGI Link Server, a Smalltalk process (thread) is started. The form data provided by the HTTP server is examined for a session key, which is matched to the CGI Link Session Data part. The session data composite object, WinRequestedPackage, is part of that same Smalltalk process.
Figure 96. WtaCustomerIdView

```
<html><head><title>Customer Identification</title></head>
<body background="/wtapix/backgrnd.jpg">
<img src="/wtapix/wta.gif">
<p>
<form method="POST" action="WtaCustomerIdProcessor">
You have asked to purchase a travel package. Please indicate whether you are a new or existing Web Travel Agency customer.
<br>
<br>
<input type="submit" name="Existing Customer Push Button" value="Existing">
Click here if you are an existing customer.
<br>
<br>
<input type="submit" name="New Customer Push Button" value=" New ">
Click here if you are a new customer.<br>
<input type="hidden" name="_ABT_FROM_PART" value="WtaCustomerIdView" />
<input type="hidden" name="_ABT_SESSION_KEY" value="524F84AD1...6A3DB7BA" />
</form>
</p>
<img src="/wtapix/rainban.gif">
<p><a href="WtaWelcomeView"><img src="/wtapix/skyhome.gif"></a>
<a href="WtaWelcomeView"> Return to the Web Travel Agency home page.</a>
</p></body>
</html>
```
WtaCustomerIdProcessor

In the Customer ID Processor, we check for which of the two buttons is clicked on. If the customer clicked on **Existing**, we route the application to the Customer Authentication View; if the customer clicked on **New**, we route to the New Customer ID View. There is no need for a Cgi Link Session Data part because the WtaRequestedPackage is kept alive in the IBM Smalltalk process.

Let’s take the longer of the two routes and click on **Existing**.

![WtaCustomerIdProcessor](image)

**Figure 97. WtaCustomerIdProcessor**

WtaCustomerAuthenticationView

We need to verify that the person claiming to be an existing customer is who he says he is. The name, exactly as entered in the database, and customer number are keyed in (see Figure 98). This authentication scheme, while not sophisticated, illustrates how information can be collected and processed.

For security reasons, the **Customer ID entry field** part could have been set up with a password protection attribute. When the customer keys in the field, the password character is displayed for each keystroke. To avoid transmission in the clear over the Internet, a secured Web browser and HTTP server must be used.

The Customer Authentication View is also a gateway part when modifying personal information and displaying packages. These application functions are started from the initial page, **WtaWelcomeView**.
VisualAge Parts

**Figure 98.** WtaCustomerAuthenticationView

```html
<!-- generated by VisualAge Web Connection on 10-17-96 at 8:50:31 PM -->
<html><head><title>Customer Authentication</title></head>
<body background="/wtapix/backgrnd.jpg">
<img src="/wtapix/wta.gif">
<p>
<form method="POST" action="WtaCustomerAuthenticationProcessor">
Please enter your name: <br>
<input type="text" name="Customer Name" size="20" maxlength="20" value="" />
<br>Please enter your customer ID: <br>
<input type="text" name="Customer ID" size="10" maxlength="10" value="" />
<br>
<input type="submit" name="Html Push Button1" value="Continue" />
<br><font size="3"><b></b></font>
<input type="hidden" name="_ABT_FROM_PART" value="WtaCustomerAuthenticationView">
<input type="hidden" name="_ABT_SESSION_KEY" value="524F84AD16A3D015D497DA2988A1B7BA">
</form>
<img src="/wtapix/rainban.gif">
<p><a href="WtaWelcomeView"><img src="/wtapix/skyhome.gif"></a> <a href="WtaWelcomeView">Return to the Web Travel Agency home page.</a></p>
</body>
</html>
```
WtaCustomerAuthenticationProcessor

The Customer Authentication Processor uses the contents of the entry fields on the Customer Authentication View to query the CUSTOMER table of the WTA database. The CGI Link Request part on the Composition Editor (Figure 99) triggers the execution of the checkInput method.

The checkInput script verifies that the customer filled out both the name field and the ID field on the Customer Authentication View. If a field is left blank, an error message is generated and the application returns to the view. If something is entered in both fields, the queryDb script is called.

```smalltalk
checkInput
"Private - Make sure a name and customer ID were entered. "
| reqPkg |

reqPkg := (self subpartNamed: 'value of Requested Package Session Data') value.

(self subpartNamed: 'Customer Name Entry Field of Customer Authentication Form Data')
  value isEmpty ifTrue: [ reqPkg errorMessage: 'Please enter a customer name.'.
  ^(self subpartNamed: 'Customer Authentication View Page Wrapper') transferRequest].

(self subpartNamed: 'Customer ID Entry Field of Customer Authentication Form Data')
  value isEmpty ifTrue: [ reqPkg errorMessage: 'Please enter a customer ID.'.
  ^(self subpartNamed: 'Customer Authentication View Page Wrapper') transferRequest].

reqPkg errorMessage: nil.
^self queryDb.
```
In order to verify the combination of customer name and customer ID, the `queryDb` method uses the entered values to access the CUSTOMER table. The following SQL statement is carried out:

```
SELECT * FROM CUSTOMER
WHERE CUST_NAME = name AND CUSTID = id
```

The result can only be a single row or no row at all, in which case the combination of name and ID is not valid. The handling of the query result is done in the next called method, `processCustomer`.

```
queryDb

"Private - query database looking for customer and if found build customer object."
where name id activeConnection querySpec result reqPkg |

name := (self subpartNamed:
  'Customer Name Entry Field of Customer Authentication Form Data') value.

id := (self subpartNamed:
  'Customer ID Entry Field of Customer Authentication Form Data') value.

reqPkg := (self subpartNamed:
  'value of Requested Package Session Data') value.

reqPkg customer isNil ifTrue: 
  [reqPkg customer: WtaCustomer new].

activeConnection := (AbtDbmSystem activeDatabaseConnection).

querySpec := (AbtQuerySpec new) statement:
  'select * from CUSTOMER where CUSTOMER.CUST_NAME = ', name printString,
  ' and CUSTOMER.CUSTID = ', id.

result := activeConnection resultTableFromQuerySpec: querySpec.

result do: [:each | each isNil ifFalse: 
  
  reqPkg customer customerId: (each at: 'CUSTID') printString;
  customerName: (each at: 'CUST_NAME') trimBlanks;
  customerAddress: (each at: 'ADDRESS') trimBlanks;
  customerPhone: (each at: 'PHONE') trimBlanks].

^self processCustomer.
```

The `processCustomer` method investigates the customer ID that is stored in the `Requested Package Session Data` part. If the ID is `nil` (wrong customer number), an error is generated indicating that the customer name does not match the number according to the database. In case there is a valid customer ID in session data (customer verified), the method calls `routeToService` to transfer the request to the next logically correct part.
Chapter 8. Implementation with VisualAge for Smalltalk

VisualAge Parts

As stated before, the customer authentication routine is used in various parts of the application. The `requestedService` attribute stores the information in which branch of the application we are navigating. The attribute is loaded at the beginning of the application (Welcome View), according to the button (Create, Display, Modify, or View) the customer clicked.

The `routeToService` method examines the `requestedService` attribute and issues a `transferRequest` message to one of the three page wrappers:

- Package List View
- Personal Info View
- Credit Card Info View

```
processCustomer
"Private - Create Customer object if needed. Verify customer exists in database.
Set error messageg and retry if not existing or proceed."
| reqPkg |
reqPkg := (self subpartNamed: 'value of Requested Package Session Data') value.

reqPkg customer customerId isNil
ifTrue: [
  reqPkg errorMessage: 'Sorry, ',
  (self subpartNamed:
    'Customer Name Entry Field of Customer Authentication Form Data') value, ' and ',
  (self subpartNamed:
    'Customer ID Entry Field of Customer Authentication Form Data') value,
  ' is not a valid combination. Try again!'.
  ^(self subpartNamed: 'Customer Authentication View Page Wrapper') transferRequest]

ifFalse: [
  reqPkg customerId: reqPkg customer customerId.
  reqPkg errorMessage: nil.
  ^(self routeToService).
]
```

```
routeToService
"Customer is verified, proceed based on customer’s service request."
| reqPkg |
reqPkg := (self subpartNamed: 'value of Requested Package Session Data') value.

reqPkg requestedService = 'Display Packages'
ifTrue: [^(self subpartNamed: 'Package List View Page Wrapper') transferRequest].

reqPkg requestedService = 'Modify Personal Info'
ifTrue: [^(self subpartNamed: 'Personal Info Page Wrapper') transferRequest].

reqPkg requestedService = 'Create Package' "Customer is now in the purchase phase."
ifTrue: [^(self subpartNamed: 'Credit Card Info Page Wrapper') transferRequest].
```
When a customer creates the first travel package in the WTA application, we request some personal information:

- Name
- Address
- Phone number

All fields on the page (Figure 100) are later stored in the database. The entered name is to be used for identification when returning to the application.

When all fields are filled out, the customer clicks on the **Done** button and the application is routed to the New Customer ID Processor.

![Figure 100. WtaNewCustomerIdView](image-url)
VisualAge Parts

WtaNewCustomerIdProcessor

The New Customer ID Processor carries out a series of methods. The execution is triggered by the CGI Link Request part on the Composition Editor (Figure 101), which calls the checkInput method.
The `checkInput` script verifies that all entry fields on the New Customer ID View are filled out. If one is left blank, the application returns to the view along with an error message. If all fields carry values, the `queryDb` method is called.

```smalltalk
checkInput:
  "Private - Make sure a name, address and phone number were entered."
  | reqPkg |
  ((self subpartNamed: 'value of Requested Package Session Data')
    valueOfAttributeNamed: #locationName selector: #'IS_locationName') isNil
  ifTrue: [ ^ (self subpartNamed: 'Exception View Page Wrapper') transferRequest ].
  reqPkg := (self subpartNamed: 'value of Requested Package Session Data') value.
  (self subpartNamed: 'Customer Name Entry Field of New Customer Id Form Data') value isEmpty ifTrue: [
    reqPkg errorMessage: 'Please enter your customer name.'.
    ^(self subpartNamed: 'New Customer Id Page Wrapper') transferRequest].
  (self subpartNamed: 'Customer Address Entry Field of New Customer Id Form Data') value isEmpty ifTrue: [
    reqPkg errorMessage: 'Please enter your address.'.
    ^(self subpartNamed: 'New Customer Id Page Wrapper') transferRequest].
  (self subpartNamed: 'Customer Phone Entry Field of New Customer Id Form Data') value isEmpty ifTrue: [
    reqPkg errorMessage: 'Please enter your phone number.'.
    ^(self subpartNamed: 'New Customer Id Page Wrapper') transferRequest].
  reqPkg errorMessage: nil.
  ^self queryDb.
```

Figure 101. `WtaNewCustomerIdProcessor`
Because we need a new ID for our new customer, the `queryDb` method accesses the CUSTOMER table in the database to determine the currently highest customer ID. The following SQL statement is executed:

```
SELECT MAX(CUSTID) FROM CUSTOMER
```

This query returns an integer value that represents the highest ID. We increase the value by one and that defines our new customer ID. The method now calls `processCustomer`.

```
queryDb

*Private - query database for the highest customer number and create new customer number.*
 |
activeConnection querySpec result reqPkg maxCustomerNumber |
reqPkg := (self subpartNamed: 'value of Requested Package Session Data') value.
activeConnection := (AbtDbmSystem activeDatabaseConnection).
querySpec := (AbtQuerySpec new) statement:
   'select max(CUSTOMER.CUSTID) from CUSTOMER'.
result := activeConnection resultTableFromQuerySpec: querySpec.
result do: [ :each | each isNil ifFalse: [ maxCustomerNumber := (each at: '1')]].
reqPkg customerId: (maxCustomerNumber + 1) printString.
^self processCustomer.
```

The `processCustomer` method gathers all information about the new customer (name, address, phone, and ID), and stores all data in the `customer` attribute of the `Requested Package Session Data` part. Now we need to store the new customer in the database.

```
processCustomer

*Private - Create Customer object.*
 |
reqPkg |
reqPkg := (self subpartNamed: 'value of Requested Package Session Data') value.
reqPkg customer: WtaCustomer new.
reqPkg customer:customerId: reqPkg customerId.
reqPkg customer:customerName: (self subpartNamed:
   'Customer Name Entry Field of New Customer Id Form Data') value.
reqPkg customer:customerAddress: (self subpartNamed:
   'Customer Address Entry Field of New Customer Id Form Data') value.
reqPkg customer:customerPhone: (self subpartNamed:
   'Customer Phone Entry Field of New Customer Id Form Data') value.
^self updateDb.
```
The `updateDb` method opens the CUSTOMER table and adds one row for the new customer. All attribute values from the session data’s `customer` subpart are copied to their counterparts in the table, and the transaction is committed. Finally, the application is routed to the Credit Card Info View.

```smalltalk
updateDb
 "Private - store new customer information to the database and finish purchasing package."
| reqPkg database newRow table |

reqPkg := (self subpartNamed: 'value of Requested Package Session Data') value.
database := AbtDbmSystem activeDatabaseMgr activeConnection.
table := (database openTableNamed: 'CUSTOMER').
newRow := table emptyRow.
newRow at: #CUSTID put: reqPkg customer customerId asNumber;
at: #CUST_NAME put: reqPkg customer customerName;
at: #ADDRESS put: reqPkg customer customerAddress;
at: #PHONE put: reqPkg customer customerPhone.
table addRow: newRow.
database commitUnitOfWork.
^(self subpartNamed: 'Credit Card Info Page Wrapper') transferRequest.
```

WtaCreditCardInfoView

One last piece of information needs to be collected; how to pay for the trip. The Credit Card Info View (Figure 102) contains a radio button set for five different credit cards. Provision is made to enter the credit card number and expiration date. With this information, authorization from a financial institution could be obtained.

The Web Travel Agency application does not use any of the data presented on this Web page. In the interest of brevity, we leave the implementation of the `WtaCreditCardInfoProcessor` part to the reader.

When the customer clicks on **Continue**, the application is routed to the `WtaCommitter` part.
Figure 102. WtaCreditCardInfoView
VisualAge Parts

The Commiter uses the data collected in WtaRequestedPackage to write it to the Web Travel Agency database. Methods are provided for inserts into seven of the nine tables in the Web Travel Agency database. In the interest of brevity, there is no provision for reporting database errors. We assume all goes well.
The commitPackageController method carries out a sequence of scripts to finally store all data in the database. First, a new package ID is required, then, one after the other, the following parts are committed (if selected):

- Package
- Requested hotel rooms
- Hotel
- Requested flight seats
- Flight
- Requested car
- Car

The methods below are shown to demonstrate how the objects are flattened and stored in the appropriate database tables. The commitPackage, commitRequestedHotelRooms, commitRequestedFlightSeats, and commitRequestedCar methods all follow the same structure:

1. Obtaining requested package
2. Opening database table
3. Creating empty row
4. Copying attributes into columns
5. Adding row to table

The methods commitHotel, commitFlight, and commitCar are not implemented.
commitPackageController
"Commit the package by committing each component then the requested package."
| reqPkg database |

(reqPkg := (self subpartNamed: 'value of Requested Package Session Data') value.
reqPkg requestedHotelRooms notNil
ifTrue: [self commitRequestedHotelRooms; commitHotel].
reqPkg requestedFlightSeats notNil
ifTrue: [self commitRequestedFlightSeats; commitFlight].
reqPkg requestedCar notNil
ifTrue: [self commitRequestedCar; commitCar].

Self generatePackageld.
Self commitPackage.

database commitUnitOfWork.

^reqPkg packageId: (maxPackageldNumber + 1) printString.

generatePackageld
"Create a new package ID by querying the database
for the highest existing package id and adding 1 to it."
| activeConnection querySpec result resultCollection reqPkg maxPackageldNumber |

(reqPkg := (self subpartNamed: 'value of Requested Package Session Data') value.
resultCollection := OrderedCollection new.
activeConnection := (AbtDbmSystem activeDatabaseConnection).
querySpec := (AbtQuerySpec new) statement:
'select TPACKAGE.PACKID from TPACKAGE'.
result := activeConnection resultTableFromQuerySpec: querySpec.
result do: [ :eachRow | resultCollection add: (eachRow asString)].
(resultCollection size = 0)
ifFalse: [
  querySpec := (AbtQuerySpec new) statement:
  'select max(TPACKAGE.PACKID) from TPACKAGE'.
  result := activeConnection resultTableFromQuerySpec: querySpec.
  result do: [ :each | each isNil
  ifFalse: [ maxPackageldNumber := (each at: '1')]]
  ifTrue: [ maxPackageldNumber := 0].
^reqPkg packageId: (maxPackageldNumber + 1) printString.
commitPackage
"Commit the package by committing each component then the requested package."

reqPkg := (self subpartNamed: 'value of Requested Package Session Data') value.
database := AbtDbmSystem activeDatabaseMgr activeConnection.
table := (database openTableNamed: 'TPACKAGE').
newRow := table emptyRow.

reqPkg cardCompany: 'VISA'. " no logic implemented "
reqPkg cardNumber: '123 456 789'. " for credit card purchase "
reqPkg cardExpirationDate: Date today.

newRow
at: #CUSTID put: reqPkg customerId;
at: #PACKID put: reqPkg packageId;
at: #LOC_NAME put: reqPkg locationName;
at: #FROM_CITY put: reqPkg fromCity;
at: #PRICE put: reqPkg packagePrice;
at: #START_DATE put: reqPkg startDate;
at: #END_DATE put: reqPkg endDate;
at: #DAYS put: reqPkg numberOfDays;
at: #CARD_COMPANY put: reqPkg cardCompany;
at: #CARD_NB put: reqPkg cardNumber;
at: #EXP_DATE put: reqPkg cardExpirationDate.
^table addRow: newRow.

commitRequestedHotelRooms
"Private - update database with hotel information."

reqPkg := (self subpartNamed: 'value of Requested Package Session Data') value.
database := AbtDbmSystem activeDatabaseMgr activeConnection.
table := (database openTableNamed: 'BROOMS').
newRow := table emptyRow.

newRow
at: #PACKID put: reqPkg packageId;
at: #HOTELID put: reqPkg requestedHotelRooms hotelId;
at: #ROOM_CAT put: reqPkg requestedHotelRooms roomCategory;
at: #NIGHTS put: reqPkg requestedHotelRooms numberOfNights;
at: #ROOMS put: reqPkg requestedHotelRooms numberOfRooms;
at: #SMOKING put: reqPkg requestedHotelRooms smoking;
at: #PRICE put: reqPkg requestedHotelRooms price.
^table addRow: newRow.

commitHotel
"Would be used to update the number of booked rooms in this hotel."

^self.
### VisualAge Parts

#### Smalltalk

**commitRequestedFlightSeats**

"Private - update database with customer requested flight information."

```smalltalk
| reqPkg database newRow table |

reqPkg := (self subpartNamed: 'value of Requested Package Session Data') value.
database := AbtDbmSystem activeDatabaseMgr activeConnection.
table := (database openTableNamed: 'BSEATS').
newRow := table emptyRow.
newRow at: #PACKID put: reqPkg packageId;
at: #FLIGHTNUM put: reqPkg requestedFlightSeats flightNumber;
at: #SEAT_CLASS put: reqPkg requestedFlightSeats seatClass;
at: #SEATS put: reqPkg requestedFlightSeats numberOfSeats;
at: #SMOKING put: reqPkg requestedFlightSeats smoking;
at: #PRICE put: reqPkg requestedFlightSeats price.
^table addRow: newRow.
```

#### Smalltalk

**commitFlight**

"Would be used to update the number of booked seats on this flight."

*

**commitRequestedCar**

"Private - update database with car information."

```smalltalk
| reqPkg database newRow table |

reqPkg := (self subpartNamed: 'value of Requested Package Session Data') value.
database := AbtDbmSystem activeDatabaseMgr activeConnection.
table := (database openTableNamed: 'BCAR').
newRow := table emptyRow.
newRow at: #PACKID put: reqPkg packageId;
at: #CAR_CLASS put: reqPkg requestedCar carClass;
at: #CAR_COMPANY put: reqPkg requestedCar carCompany;
at: #PRICE put: reqPkg requestedCar price.
^table addRow: newRow.
```

#### Smalltalk

**commitCar**

"Would be used to update the number of booked cars for the car rental company."

*

### WtaPackageConfirmationView

The **WtaPackageConfirmationView** part echoes back the customer choices and the **WtaRequestedPackage** part provides the information displayed. This prevents any delay in performance by not accessing the database.
The Package Confirmation View is the last view part of the travel booking sequence. The part’s Composition Editor (Figure 104) and its method protocol are basically the same as for the Package Purchase View. The difference is the text on top of the confirmation page, where we thank the customer and display the customer ID and package ID. Also, because there cannot be any more changes to the package, which has already been purchased at this point, the buttons that are shown on Figure 93 are missing from the view in Figure 104.

Figure 104. WtaPackageConfirmationView

WtaPackageListView

Figure 105 shows the Composition Editor of the Package List View. This part is eventually requested when the customer clicks on Display on the Welcome View. Before showing the stored travel packages, the customer first has to be identified. Only after successful verification of customer name and ID, the packages are displayed.
Figure 105. WtaPackageListView

HTML

<!-- generated by VisualAge Web Connection on 10-17-96 at 8:53:01 PM -->
<html><head><title>Select an Existing Package</title></head>
<body background="/wtapix/backgrnd.jpg">
<img src="/wtapix/wta.gif">
<p>
<form method="POST" action="WtaPackageListProcessor">
Select a travel package:
<br>
<select size="5" name="Requested Package List">
<option>From Oct 1, 1996 to Oct 3, 1996 destination of Atlanta 6
<option>From Oct 12, 1996 to Oct 14, 1996 destination of San Francisco 11
<option>From Nov 1, 1996 to Nov 9, 1996 destination of Boston 12
</select>
<p>
<input type="submit" name="continueButton" value="Details">
Display details for the selected travel package.
</form>
</p>
</body>
</html>
The `queryForPackages` method uses the customer ID entered on the Customer Authentication View to query the TPACKAGE table for all travel packages matching the ID. The following SQL statement is carried out:

```
SELECT START_DATE, END_DATE, LOC_NAME, PACKID
FROM TPACKAGE
WHERE CUSTID = customerID
```

Smalltalk

```smalltalk
queryForPackages
"get all existing packages for a customer from the database."
| activeConnection querySpec result rowOC aSC anOC reqPkg |

reqPkg := (self subpartNamed: 'value of Requested Package Session Data') value.
aSC := SortedCollection sortBlock: [ :a :b | (a at: 1) < (b at: 1)].
activeConnection := (AbtDbmSystem activeDatabaseConnection).
querySpec := (AbtQuerySpec new) statement:
' select TPACKAGE.START_DATE, TPACKAGE.END_DATE, TPACKAGE.LOC_NAME, TPACKAGE.PACKID
from TPACKAGE where TPACKAGE.CUSTID = ', reqPkg customerId.
result := activeConnection resultTableFromQuerySpec: querySpec.
result do: [:each | each isNil ifFalse: [ |
anOC := OrderedCollection new.
aSC add:
('anOC addLast: (each at: 'START_DATE');
addLast: (each at: 'END_DATE');
addLast: (each at: 'LOC_NAME');
addLast: (each at: 'PACKID') ;
yourself).
reqPkg packageId: (each at: 'PACKID').
]]].
anOC := OrderedCollection new.
1 to: (aSC size) do: [ n |
rowOC := aSC at: n.
anOC addLast:
('From ', (self formatDate: (rowOC at: 1)),
'to ', (self formatDate:
(rowOC at: 2)),
'destination of ', ((rowOC at: 3) trimBlanks),
', ', ((rowOC at: 4) printString)). ].

^(self subpartNamed: 'Requested Package List') items: anOC
```
WtaPackageListProcessor

The Package List Processor (Figure 106) identifies the selected package from the Package List View and requests the corresponding data from the database. When all package data is restored, the application routes to the Package Details View.

The requested package is regenerated by retrieving all parts from the various database tables. The following sequence of methods is carried out:

- processPackageListForm
- reinitializeRequestedPackage
- pumpUpRequestedPackage
- pumpUpRequestedHotelRooms
- pumpUpHotel
- pumpUpRequestedFlightSeats
- pumpUpFlight
- pumpUpRequestedCar

All `pumpUp` methods are performing the opposite of the methods in the Committer; instead of copying attributes to table columns, here the methods use the columns to fill the attributes.
VisualAge Parts

Smalltalk

processPackageListForm
"Get the selected package from the list and rebuild (pump up) the package objects."
| reqPkg aSelectedItem |

reqPkg := (self subpartNamed: 'value of Requested Package Session Data') value.
self reinitializeRequestedPackage.
aSelectedItem :=
(self subpartNamed: 'Requested Package List of Package List Form Data') value.
reqPkg packageId: aSelectedItem subStrings last asNumber.
self pumpUpRequestedPackage.

self pumpUpRequestedHotelRooms.
reqPkg requestedHotelRooms notNil ifTrue: [self pumpUpHotel].
self pumpUpRequestedFlightSeats.

reqPkg requestedFlightSeats notNil ifTrue: [self pumpUpFlight].
self pumpUpRequestedCar.

^(self subpartNamed: 'Package Details Page Wrapper') transferRequest

Smalltalk

reinitializeRequestedPackage
"Reset requested package fields for another package details showing."
| reqPkg |

reqPkg := (self subpartNamed: 'value of Requested Package Session Data') value.

*reqPkg locationName: nil;
fromCity: nil;
packageId: nil;
packagePrice: nil;
startDate: nil;
endDate: nil;
numberOfDays: nil;
cardCompany: nil;
requestedFlightSeats: nil;
requestedHotelRooms: nil;
requestedCar: nil;
flight: nil;
hotel: nil;
car: nil.

Smalltalk

pumpUpRequestedPackage
"Retrieve requested package from database and complete requested package object."
| activeConnection querySpec result reqPkg |

reqPkg := (self subpartNamed: 'value of Requested Package Session Data') value.
activeConnection := (AbtDbmSystem activeDatabaseConnection).
querySpec := (AbtQuerySpec new) statement:
'select * from TPACKAGE where TPACKAGE.PACKID = ',
reqPkg packageId asInteger printString.
result := activeConnection resultTableFromQuerySpec: querySpec.
result do: [ :each | each isNil ifFalse: [
reqPkg
locationName: (each at: 'LOC_NAME');
fromCity: (each at: 'FROM_CITY');
packagePrice: (each at: 'PRICE');
startDate: (each at: 'START_DATE');
endDate: (each at: 'END_DATE');
numberOfDays: (each at: 'DAYS') printString;
cardCompany: (each at: 'CARD_COMPANY') trimBlanks]].
VisualAge Parts

**Smalltalk**

```smalltalk
pumpUpRequestedHotelRooms
"Retrieve requested hotel room database information
and create requested hotel rooms object."
| activeConnection querySpec result reqPkg |

reqPkg := (self subpartNamed: 'value of Requested Package Session Data') value.
activeConnection := (AbtDbmSystem activeDatabaseConnection).
querySpec := (AbtQuerySpec new) statement:
   'select * from BROOMS where BROOMS.PACKID = ', reqPkg packageId printString.
result := activeConnection resultTableFromQuerySpec: querySpec.
result do: [:each | each isNil ifFalse: [
   reqPkg requestedHotelRooms: WtaRequestedHotelRooms new.
   reqPkg requestedHotelRooms
      packageId: (each at: 'PACKID') printString;
      hotelId: (each at: 'HOTELID') printString;
      roomCategory: (each at: 'ROOM_CAT') trimBlanks;
      numberOfNights: (each at: 'NIGHTS') printString;
      numberOfRooms: (each at: 'ROOMS') printString;
      price: (each at: 'PRICE')
]].

^self
```

**Smalltalk**

```smalltalk
pumpUpRequestedFlightSeats
"Retrieve requested flight seats database information
and create requested flight seats object."
| activeConnection querySpec result reqPkg |

reqPkg := (self subpartNamed: 'value of Requested Package Session Data') value.
activeConnection := (AbtDbmSystem activeDatabaseConnection).
querySpec := (AbtQuerySpec new) statement:
   'select * from BSEATS where BSEATS.PACKID = ', reqPkg packageId printString.
result := activeConnection resultTableFromQuerySpec: querySpec.
result do: [:each | each isNil ifFalse: [
   (reqPkg requestedFlightSeats)
      packageId: (each at: 'PACKID');
      flightNumber: (each at: 'FLIGHTNUM');
      seatClass: (each at: 'SEAT_CLASS') trimBlanks;
      numberOfSeats: (each at: 'SEATS') printString;
      smoking: (each at: 'SMOKING') printString;
      price: (each at: 'PRICE')
]].

^self
```
pumpUpRequestedCar

"Retrieve requested car database information and create the requested car object."

| activeConnection querySpec result reqPkg |

reqPkg := (self subpartNamed:'value of Requested Package Session Data') value.
activeConnection := (AbtDbmSystem activeDatabaseConnection).
querySpec := (AbtQuerySpec new) statement:
'select * from BCAR where BCAR.PACKID = ', reqPkg packageld printString.

result := activeConnection resultTableFromQuerySpec: querySpec.
result do: [ :each | each isNil ifFalse: [
reqPkg requestedCar: WtaRequestedCar new.
(reqPkg requestedCar)
packageId: (each at: 'PACKID');
carClass: (each at: 'CAR_CLASS') trimBlanks;
carCompany: (each at: 'CAR_COMPANY') trimBlanks;
price: (each at: 'PRICE')].]

^self

pumpUpHotel

"Retrieve hotel database information and create the hotel object."

| activeConnection querySpec result reqPkg |

reqPkg := (self subpartNamed:'value of Requested Package Session Data') value.
activeConnection := (AbtDbmSystem activeDatabaseConnection).
querySpec := (AbtQuerySpec new) statement:
'select * from HOTEL where HOTEL.HOTELID = ', reqPkg hotelRooms hotelld.

result := activeConnection resultTableFromQuerySpec: querySpec.
result do: [ :each | each isNil ifFalse: [
reqPkg hotel: WtaHotel new.
(reqPkg hotel)
hotelName: (each at: 'HOTEL_NAME');
hotelId: (each at: 'HOTELID');
locationName: (each at: 'LOC_NAME') trimBlanks].]

^self

pumpUpFlight

"Retrieve flight database information and create the flight object."

| activeConnection querySpec result reqPkg |

reqPkg := (self subpartNamed:'value of Requested Package Session Data') value.
activeConnection := (AbtDbmSystem activeDatabaseConnection).
querySpec := (AbtQuerySpec new) statement:
'select * from FLIGHT where FLIGHT.FLIGHTNUM = ', reqPkg requestedFlightSeats flightNumber printString.

result := activeConnection resultTableFromQuerySpec: querySpec.
result do: [ :each | each isNil ifFalse: [
reqPkg flight: WtaFlight new.
(reqPkg flight)
fromCity: (each at: 'FROM_CITY');
toCity: (each at: 'TO_CITY')].]

^self
WtaPackageDetailsView

The Package Details View (Figure 107) shows the restored package, when the customer clicks on the Details button on the Package List View. The appearance of the part is basically identical to the Package Purchase View and the Package Confirmation View. Refer to these parts for a description of the methods.

Figure 107. WtaPackageDetailsView
WtaPersonalInfoView

The Personal Info View lets you display and modify your customer record from the database. This view part is loaded when you click on the Modify button on the Welcome View. On the Composition Editor (Figure 108) the HTML page contains a text field (customer name) and two entry fields (address, telephone) that will be initially filled through running the showName, showAddress, and showPhoneNumber scripts that follow Figure 108. The form points to the WtaPersonalInfoProcessor part where the routing of the application is decided depending on which button on the form was pressed.
The invisible text field below the **Done** button is an empty string that eventually contains an error message from the torn-off attribute of the session data part. This session data attribute is set in the Personal Info Processor in case the customer leaves one of the mandatory entry fields blank.

---

**Smalltalk**

```smalltalk
showName
  "answer the receiver with the customer's name."
| reqPkg |
reqPkg :=(self subpartNamed: 'value of Requested Package Session Data') value.
*reqPkg customer customerName

showAddress
  "Print customer's address." | reqPkg | reqPkg :=(self subpartNamed: 'value of Requested Package Session Data') value.
*reqPkg customer customerAddress.
```
VisualAge Parts

WtaPersonalInfoProcessor

The Personal Info Processor (Figure 109) examines the buttons on the preceding Personal Info View. If the customer clicks on Done, the application routes to the Welcome View. If the customer clicks on Change, the entry fields are validated and the modified customer information is stored in the database.

Smalltalk

showPhoneNumber
"Print customer's phone number."
| reqPkg |

reqPkg :=(self subpartNamed: 'value of Requested Package Session Data') value.
*reqPkg customer customerPhone.

Figure 109. WtaPersonalInfoProcessor

Smalltalk

checkWhichButtonPressed
"Private - was change or done pressed?"

(self subpartNamed: 'Done Push Button of Personal Info Form Data') value
ifTrue: [(self subpartNamed: 'Welcome Page Wrapper') transferRequest]
ifFalse: ['self checkInput'].

Chapter 8. Implementation with VisualAge for Smalltalk 237
processCustomer
"Private - Update address and phone number in the customer object."
| reqPkg |

reqPkg := (self subpartNamed: 'value of Requested Package Session Data') value.
reqPkg customer customerAddress:
(self subpartNamed: 'Customer Address Entry Field of Personal Info Form Data') value.
reqPkg customer customerPhone:
(self subpartNamed: 'Phone Number Entry Field of Personal Info Form Data') value.
^self updateDb.

updateDb
"Private - update database with new address and phone number for customer."
| activeConnection querySpec reqPkg database oldRow newRow table |

((self subpartNamed: 'value of Requested Package Session Data') valueOfAttributeNamed: #customer selector: #'IS_customer') isNil
ifTrue: [ ^(self subpartNamed: 'Exception View Page Wrapper') transferRequest ].

reqPkg := (self subpartNamed: 'value of Requested Package Session Data') value.
database := AbtDbmSystem activeDatabaseMgr activeConnection.
querySpec := (AbtQuerySpec new)
statement: 'SELECT CUSTOMER.ADDRESS, CUSTOMER.PHONE from CUSTOMER
where CUSTOMER.CUSTID = ', reqPkg customer customerId;
hostVarsShape: (nil).
oldRow := (database resultTableFromQuerySpec: querySpec) first.
newRow := (oldRow deepCopy)
at: 'ADDRESS' put: reqPkg customer customerAddress asString;
at: 'PHONE' put: reqPkg customer customerPhone asString;
yourself.
table := (database openTableNamed: 'CUSTOMER')
atRow: oldRow putRow: newRow;
yourself. database commitUnitOfWork.
^\(self subpartNamed: 'Personal Info Page Wrapper') transferRequest.
WtaAttractionView

The Attraction View (Figure 110) is the anchor for viewing videos and linking to external Internet sites. It is loaded when you click on the View button on the Welcome View.

The Location Query part is connected to the aboutToGenerateHtml action of the window, making the query run just before the page appears on the browser. The part executes the following query:

```sql
SELECT LOCATION.LOC_NAME FROM LOCATION
```

The torn-off resultTable subpart therefore contains all locations from the database. The subpart is used in the formatLocations script to fill the list-box of the HTML form on the page.

```smalltalk
formatLocations
"Fetch candidate locations from the query result table."
locations city |
locations := OrderedCollection new.
((self subpartNamed: 'resultTable of Location Query')
 valueOfAttributeNamed: #rows selector: #'IS_rows')
do: [:each | each isNil ifFalse: [ 
     city := (each at: 'LOC_NAME') trimBlanks. locations add: city. 
]].
```

Figure 110. WtaAttractionView
The HTML form action links to the WtaAttractionProcessor part. The HTML list now contains all locations, but has been limited to view only three lines at a time. The two push buttons are configured as submit buttons, both are named in order to use them easily as subparts in the Attraction Processor.

WtaAttractionProcessor

The Attraction Processor (Figure 111) is called from the form on the Attraction View. The form data contains the location list box and the two buttons for Video and WWW. Those three subparts are torn off from the form data in order to use them in the processLocationName script. The script is called through the CGI Link Request part. The remaining parts on the Composition Editor show the session data with its tear-off attributes for the location, the video file, and the URL.
The `processLocationName` method receives the selected location through the form data subpart and stores it in session data. Then it uses the location as a variable for the SQL query on the location table to retrieve the video file name and the URL for the location. From the result, we use the URL and `VIDEO_FILE` columns to fill the appropriate session data attributes. Then, we use the form data again to find out which button has been clicked on the Attraction View, and we use the `self transaction response location` construct to return the session data values for either the path to the video file or the URL.

![Diagram of WtaAttractionProcessor](image)

**Figure 111. WtaAttractionProcessor**
VisualAge Parts

WtaAboutView

The About View is basically only a short description of the project and the people that are behind the development of the Web Travel Agent application. It is called through an HTML link from the Welcome View.

The Composition Editor (Figure 112) contains only regular text with embedded links to other project-related information, such as ITSO, the first prototype, the used CRC cards, and the Developer View.

Smalltalk

```smalltalk
processLocationName
"Save location in requested package, reload location and link to video or URL"

| activeConnection querySpec result reqPkg |

(self subpartNamed: 'locationName of value of Requested Package Session Data')
value: (self subpartNamed: 'Location List of Attraction View Form Data') value.

reqPkg := (self subpartNamed: 'value of Requested Package Session Data') value.

activeConnection := (AbtDbmSystem activeDatabaseConnection).
querySpec := (AbtQuerySpec new) statement:
'SELECT * FROM LOCATION WHERE LOCATION.LOC_NAME = ','
reqPkg locationName printString.

result := activeConnection resultTableFromQuerySpec: querySpec.
result do: [:each | each isNil ifFalse: [ reqPkg url: (each at: 'URL') trimBlanks;
video: '/wta/media/', (each at: 'VIDEO_FILE') trimBlanks. ]].

((self subpartNamed: 'Video Push Button of Attraction View Form Data') value)
ifTrue: [ self transaction response location:
(sefl subpartNamed: 'video of value of Requested Package Session Data') value
].

((self subpartNamed: 'WWW Push Button of Attraction View Form Data') value)
ifTrue: [ self transaction response location:
(self subpartNamed: 'url of value of Requested Package Session Data') value
].

^nil
```
VisualAge Parts

Chapter 8. Implementation with VisualAge for Smalltalk

VisualAge Parts

Chapter 8. Implementation with VisualAge for Smalltalk

Figure 112. WtaAboutView

WtaDeveloperView

The Developer View allows for viewing pictures of the project team that designed and implemented the WTA application. To demonstrate the ability of VisualAge for Smalltalk to embed Java applets in the Composition Editor, we implemented a Java applet that lets you flip through the guys by clicking on a Next or Previous button. Depending on the button clicked, the appropriate picture (JPEG) is loaded along with the person’s name, country of origin, and project responsibilities. For the Java applet’s source code refer to Appendix C, “Java Applet,” on page 455.

The Composition Editor’s central part is the black rectangle which represents the applet (Figure 113). The actual applet size is bigger, however, because the black box does not really show anything, we reduced the size so that it fits into the window in Figure 113.
The generated HTML shows where the applet is imbedded in the HTML page. The two important parameters, *code* and *codebase*, define which Java class responds to the request and where to find it on the Web server. How to set those two parameters is shown in Figure 114.

The *Guys.class* file (case sensitive!) has to be located in your server’s C:\WTA\HTML\JAVA directory. The HTTPD.CNF Web server configuration file contains a *Pass* entry for /wta. Through the additional specification of the Java subdirectory the applet is found.

---

**Figure 113. WtaDeveloperView**

The generated HTML shows where the applet is imbedded in the HTML page. The two important parameters, *code* and *codebase*, define which Java class responds to the request and where to find it on the Web server. How to set those two parameters is shown in Figure 114.

The *Guys.class* file (case sensitive!) has to be located in your server’s C:\WTA\HTML\JAVA directory. The HTTPD.CNF Web server configuration file contains a *Pass* entry for /wta. Through the additional specification of the Java subdirectory the applet is found.

---

**HTML**

```html
<!-- generated by VisualAge Web Connection on 10-21-96 at 2:50:57 PM -->
<html><head><title>Development Team</title></head>
<body background="/wtapix/backgrnd.jpg">
<img src="/wtapix/wta.gif" align="TOP" alt="">
<p><applet code="Guys.class" width="600" height="400"
codebase="/wta/java" name="Java Applet" align="TOP">
A Java applet is supposed to be shown here, ... if you had a Java-enabled browser. Oops!
</applet>
</p>
<p><a href="WtaWelcomePage"><img src="/wtapix/skyhome.gif"></a>
<a href="WtaWelcomeView"> Return to Web Travel Agency Home Page</a>
</body>
</html>
```
The WTA application transfers the page request to the Exception View (Figure 115) when the class for the session data times out. This can happen from almost every Web Connection part in the WTA application. Because there is no way to recover the lost session data, the customer has to start over booking the travel package. Therefore, the Exception View’s only link points to the Web Travel Agent home page.
Database Parts

Reusable Footer

This part is used to demonstrate how to incorporate one HTML-generating part in another. The Reusable Footer part (Figure 116) contains an HTML composite with various literal text subparts. We used it in the application as a footer on the Welcome View.

To employ the footer as a reusable part, use the Add Part menu item from a Web Connection part’s Options pull-down menu and specify the reusable part’s name.

For more information on reuse refer to “Reuse of Web Parts” on page 351.

Figure 116. Reusable Footer

Database Parts

All WTADB database tables have a corresponding class in the VisualAge development image. The classes are generated in the Public Interface Editor with one attribute for every table column. The generated default scripts for all attributes allow us to get and set the values of each attribute. Table 9 shows some examples of matching columns and attributes, and Figures 117 through 120 show the Public Interface Editor’s Attribute page of the corresponding Customer, Location, Flight, and Hotel classes.
## Table 9. Matching Table Columns and Attributes

<table>
<thead>
<tr>
<th>Table</th>
<th>Column</th>
<th>Attribute</th>
</tr>
</thead>
<tbody>
<tr>
<td>Customer</td>
<td>CUSTID</td>
<td>customerId</td>
</tr>
<tr>
<td></td>
<td>CUST_NAME</td>
<td>customerName</td>
</tr>
<tr>
<td></td>
<td>ADDRESS</td>
<td>customerAddress</td>
</tr>
<tr>
<td></td>
<td>PHONE</td>
<td>customerPhone</td>
</tr>
<tr>
<td>Location</td>
<td>LOC_NAME</td>
<td>locationName</td>
</tr>
<tr>
<td></td>
<td>VIDEO_FILE</td>
<td>videoFile</td>
</tr>
<tr>
<td></td>
<td>URL</td>
<td>webPageUrl</td>
</tr>
<tr>
<td>Flight</td>
<td>FLIGHTNUM</td>
<td>flightNumber</td>
</tr>
<tr>
<td></td>
<td>FROM_CITY</td>
<td>fromCity</td>
</tr>
<tr>
<td></td>
<td>TO_CITY</td>
<td>toCity</td>
</tr>
<tr>
<td></td>
<td>BASE_PRICE</td>
<td>basePrice</td>
</tr>
<tr>
<td>Hotel</td>
<td>HOTEL_NAME</td>
<td>hotelName</td>
</tr>
<tr>
<td></td>
<td>HOTELID</td>
<td>hotelId</td>
</tr>
<tr>
<td></td>
<td>LOC_NAME</td>
<td>locationName</td>
</tr>
<tr>
<td></td>
<td>CAPACITY</td>
<td>capacity</td>
</tr>
<tr>
<td></td>
<td>BOOKED</td>
<td>roomsBooked</td>
</tr>
<tr>
<td></td>
<td>DAILY_PRICE</td>
<td>dailyPrice</td>
</tr>
</tbody>
</table>

![Customer (Public Interface)](image)

**Figure 117.** Customer (Public Interface)
Database Parts

Figure 118. Location (Public Interface)

Figure 119. Flight (Public Interface)
Database Access Set

The database access set of the WTA application is listed here only for informational purposes. It is not a part that you change through, for example, the Script Editor. Rather, every time you create or update a database query, the corresponding AbtQuerySpec is changed accordingly.

```
Smalltalk

AllCars
  "Private - ** Warning ** This method is generated by VisualAge and should not
  be modified or deleted. This method is responsible for returning a featureSpec
  that describes the implementation of a particular feature of the receiver"

^AbtQuerySpec new name: 'AllCars';
  statement: 'SELECT * FROM CAR ORDER BY CAR.CLASS ASC';
  hostVarsShape: (nil)

CarPriceWhereClass

^AbtQuerySpec new name: 'CarPriceWhereClass';
  statement: 'SELECT * FROM CAR :whereClause';
  hostVarsShape: (nil)

flightQuery

^AbtQuerySpec new name: 'flightQuery';
  statement: 'SELECT * FROM FLIGHT';
  hostVarsShape: (nil)

getAllCustomers

^AbtQuerySpec new name: 'getAllCustomers';
  statement: 'SELECT * FROM CUSTOMER';
  hostVarsShape: (nil)
```

Figure 120. Hotel (Public Interface)
**Database Parts**

```
Smalltalk

getCustomerNameAndId

^AbtQuerySpec new name: 'getCustomerNameAndId';
  statement: 'SELECT CUSTOMER.CUST_NAME, CUSTOMER.CUSTID
  FROM CUSTOMER :whereClause';
  hostVarsShape: (nil)

GetHotelAllColumnsAllRows

^AbtQuerySpec new name: 'GetHotelAllColumnsAllRows';
  statement: 'SELECT * FROM HOTEL';
  hostVarsShape: (nil)

GetHotelsForALocation

^AbtQuerySpec new name: 'GetHotelsForALocation';
  statement: 'SELECT HOTEL.HOTEL_NAME FROM HOTEL :whereClause';
  hostVarsShape: (nil)

getLocationAllColumnsAllRows

^AbtQuerySpec new name: 'getLocationAllColumnsAllRows';
  statement: 'SELECT * FROM LOCATION ORDER BY LOCATION.LOC_NAME ASC';
  hostVarsShape: (nil)

getLocationsFromDb

^AbtQuerySpec new name: 'getLocationsFromDb';
  statement: 'SELECT LOCATION.LOC_NAME FROM LOCATION';
  hostVarsShape: (nil)

getPackage

^AbtQuerySpec new name: 'getPackage';
  statement: 'SELECT TPACKAGE.CUSTID, TPACKAGE.PACKID,
  TPACKAGE.FROM_CITY, TPACKAGE.LOC_NAME, TPACKAGE.PRICE,
  TPACKAGE.START_DATE, TPACKAGE.END_DATE, TPACKAGE.DAYS
  FROM TPACKAGE';
  hostVarsShape: (nil)

DbConnectionAlias

"Private - ** Warning ** This method is generated by VisualAge and should not
be modified or deleted. This method is responsible for returning a featureSpec
that describes the implementation of a particular feature of the receiver"

^AbtDatabaseConnectionSpec new
  dbmClass: AbtIbmDatabaseManager;
  dsn: 'WTADB';
  promptEnabled: false;
  yourself
```
In this chapter we detail how to implement the Web Travel Agent application using the VisualAge for C++ Web parts. As mentioned in Chapter 4, “VisualAge for C++ Web Parts,” on page 61, you use the Waikiki beta code to develop the sample application. Therefore, some implementation details explained in this chapter may be irrelevant when the official Web parts code is available. Nevertheless, the application architecture and the techniques explained here should remain up to date.

To mirror Chapter 8, “Implementation with VisualAge for Smalltalk,” on page 159, this chapter is divided into three main sections:

- **Application architecture** introduces the architecture adopted for implementing the sample application in C++. 


VisualAge parts details the visual and nonvisual parts that you have to build to implement the application. This parts are the direct mapping of the CRC cards that you flesh out during the application design.

Database parts introduces the parts needed to make the application accessing the WTA database. These parts are mapped from the database tables by using the Data Access Builder tool provided with VisualAge for C++.

Application Architecture

In terms of the visual and nonvisual parts developed, both Smalltalk and C++ are similar, and you can refer to Chapter 4, “VisualAge for C++ Web Parts,” on page 61 for detailed information.

Although the number and type of parts are the same in both implementations, the C++ application architecture differs significantly from the Smalltalk application architecture. The difference results from the differences between the languages. Below we explain how storage of session data is implemented in VisualAge for Smalltalk and how you can implement it, using a similar mechanism, in VisualAge for C++. The solution that we propose is obviously temporary and should be different when the VisualAge for C++ Web parts are generally available.

Web Connection Session Data

The Smalltalk language does not support writing or reading standard output (usually the display) and standard input (usually the keyboard) of the operating system. The Transcript window is available to programmers as a general-purpose blackboard from which they can control input and output for their code. For this reason, a “software pump,” implemented as the abtcgil executable, is necessary to provide an interface between the Smalltalk image that contains each part and the standard input and output of the operating system. When a Smalltalk part is loaded in memory, it redirects its input and output through the CGI link, which in turn redirects the input and output to the operating system. The session data is stored in memory and allocated in the heap. It is accessible through a global object, which acts as a shopping cart to collect the data. This “shopping cart” is accessible by every part.

Implementing Session Data for C++

Programming with the VisualAge for C++ parts enables you to build executable or dynamic link library files that generate HTML flow on at runtime. Unlike Smalltalk, C++ can be used directly as a CGI language
because it supports writing to standard output and reading from standard input. To illustrate how an executable uses the standard input and standard output, let us suppose that you build a simple HTML form that contains an entry field to enter someone’s name and a push button to submit the form. When users connect to the form, they enter their names and push the submit button to send the information to the Web server. Let us suppose that the form action is an executable (greeting.exe) that is posted and displays a greeting message in return to the user. The greeting message is generated as an HTML flow and uses the name information that the user entered (Figure 121).

Writing and reading from the standard output and input is the basis of CGI programming. The executable reads the parameter transmitted from the Web server to the operating system standard input. Similarly, the executable writes a text flow formatted with HTML to the operating system standard output. The Web server collects the flow to redirect it to the Web browser of the user connected to the application.

Once the user enters the data, and it is sent to the CGI program, the data is lost unless it is stored in some place. Furthermore other CGI programs can access the data only if it is stored in a shareable place and if a mechanism is provided to let them access the data. To share data between different executable files that execute as different processes, two main options can be envisioned: (1) using interprocess communication mechanisms (IPCs) to let different processes access the same data, or (2) storing the data persistently in a place accessible by separate processes.1

1 We do not cover Web server programming, which is also an alternative when sharing data between different processes.
Application Architecture

**Interprocess Communication**

Several options are available for implementing IPC between the CGI programs to make up the Web application. Unfortunately, they lead to drastic modification of the code generated by the Visual Builder:

- **Shareable memory:** Different programs can share the same memory location. The memory location must be associated with an identifier that each program can use to access the location. Each user connected to the application would be allotted a memory location for his or her session data. No mutual exclusion mechanisms are thus necessary to ensure that the data is not overridden by several programs accessing the data at the same time. This option implies a modification of the Visual Builder code to define the memory location and implement the mechanism for each program to access the memory location.

- **Pipes:** Named and nonnamed pipes can be used to make programs communicate with an agent program that would manage the session data. The agent program could send data upon request by the CGI programs using a communication protocol that has to be decided. Again, this option would require modifying the code generated by the Visual Builder and result in code that is hardly portable.

**Persistent Storage**

Session data can be stored persistently with different approaches:

- **Flat-file storage:** Flat files can be used to store the session data. Unfortunately flat file parts are not provided with the Visual Builder, so you have to implement such parts. In addition, you should be able to retrieve session data easily, using a key access mechanism, if you want to implement the session data management mechanism in a way that is similar to the Smalltalk approach. A file format should be adopted in this regard.

- **Database storage:** Databases can be used to store session data, but, obviously, they must be installed. Unfortunately this can lead to a costly configuration especially if the CGI application does not require that the session data be stored once the session ends.

- **Profile-file storage:** Keys in a unique profile file associated with the Web application can be used to store session data. Each user can be associated with a unique key, and the session information is stored under this key by using subkeys. This approach has several advantages over the flat-file and database storage approaches:
➢ *It is portable:* The IProfile part can be used to implement profile-file storage. The IProfile part is mapped to a profile file in OS/2 and to a key in HKEY_LOCAL_MACHINE\SOFTWARE in the registry in Windows 95 or Windows NT.

➢ *It is an inexpensive solution:* Because profile-file storage relies on the Windows registry or the OS/2 profile files, it does not require a database.

➢ *It is compatible with the code generated by the Visual Builder:* Because the IProfile part can be used directly with the Visual Builder, the code generated does not have to be patched or updated.

Below we explain how to use the IProfile part to persistently store the session data. Notice that this solution is just an example which shows how you could implement session data. Obviously this solution is not satisfactory for a production application which must support a large number of hits. In addition, the application implementation with Waikiki could be improved by adding a clean up feature to delete timed-out session data after a certain amount of time.

### Implementation Details

A unique profile, DATA.INI, is used to hold each new Web session key generated each time a user connects to the application. This profile is mapped to an OS/2 INI file and in Windows 95 or Windows NT to a key in the HKEY_LOCAL_MACHINE\SOFTWARE entry of the Windows registry.

When the first CGI program is executed, a numerical key, also referred as to the *session key* is associated with the current Web connection. This numerical key consists of the current date and time of the server machine. It is added as a subkey of DATA.INI. All information related to the current connection is stored under this key using an identifier and a value. In Figure 122, for example, the Customer Name identifier is associated the value “Marc Carrel” under the session key 28739. The information for the same session key constitutes the session data.
Session Data Stored in the Windows Registry

The session key is transmitted during the session across every CGI program by adding an extra hidden entry field in every dynamic HTML page. The value of this hidden entry field is initialized with the session key.

Naming Conventions

Several visual and nonvisual parts are used to build the Web Travel Agent application. You should adopt a naming convention for your parts, you also must decide on the naming conventions for the C++ source files.

We adopted the following conventions:

- Each file name has a maximum length of eight characters.
- Each visual part file name terminates with the character V (like view).
- Each nonvisual part file name terminates with the character P (like processor). Two exceptions to this rule: the WtaPackagePricer and the WtaPackageCommitter.

Tables 1 and 2 respectively list the visual and nonvisual parts and their corresponding file names.
### Table 10. Visual Parts

<table>
<thead>
<tr>
<th>Part Name</th>
<th>File Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>WtaCarClassView</td>
<td>WTCARCLV</td>
</tr>
<tr>
<td>WtaCarSelectionView</td>
<td>WTCARSLV</td>
</tr>
<tr>
<td>WtaComponentSelectionView</td>
<td>WTCMPNNV</td>
</tr>
<tr>
<td>WtaCreditCardInfoView</td>
<td>WTCINFV</td>
</tr>
<tr>
<td>WtaCustomerAuthenticationView</td>
<td>WTCSTMAV</td>
</tr>
<tr>
<td>WtaCustomerIdView</td>
<td>WTCSTMIV</td>
</tr>
<tr>
<td>WtaFlightSelectionView</td>
<td>WTFLGHTV</td>
</tr>
<tr>
<td>WtaNewCustomerIdView</td>
<td>WTCSTMIV</td>
</tr>
<tr>
<td>WtaPackageConfirmationView</td>
<td>WTPCKGOV</td>
</tr>
<tr>
<td>WtaPackageCreationView</td>
<td>WTPCKGCV</td>
</tr>
<tr>
<td>WtaPackageDetailsView</td>
<td>WTPCKGDV</td>
</tr>
<tr>
<td>WtaPackageFailedView</td>
<td>WTPCKGFV</td>
</tr>
<tr>
<td>WtaPackageListView</td>
<td>WTPCKGLV</td>
</tr>
<tr>
<td>WtaPackagePurchaseView</td>
<td>WTPCKGPV</td>
</tr>
<tr>
<td>WtaPersonalInfoView</td>
<td>WTPINFOV</td>
</tr>
<tr>
<td>WtaWelcomeView</td>
<td>WTWLCMEV</td>
</tr>
</tbody>
</table>

### Table 11. Nonvisual Parts

<table>
<thead>
<tr>
<th>Part Name</th>
<th>File Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>WtaCarSelectionProcessor</td>
<td>WTCARSLP</td>
</tr>
<tr>
<td>WtaCommitter</td>
<td>WTCMMTRP</td>
</tr>
<tr>
<td>WtaComponentSelectionProcessor</td>
<td>WTCMPSLP</td>
</tr>
<tr>
<td>WtaCustomerAuthenticationProcessor</td>
<td>WTCSTMAP</td>
</tr>
<tr>
<td>WtaCustomerIdProcessor</td>
<td>WTCSTMIP</td>
</tr>
<tr>
<td>WtaFlightSelectionProcessor</td>
<td>WTLGSLP</td>
</tr>
<tr>
<td>WtaHotelSelectionProcessor</td>
<td>WHTHLPSLP</td>
</tr>
<tr>
<td>WtaNewCustomerIdProcessor</td>
<td>WTN'CSTLP</td>
</tr>
</tbody>
</table>
VisualAge Parts

For each part, the file name of its associated source code is set up in the Part Interface Editor.

Below we fully describe each part. The order we choose to describe the parts mirrors the order in which the Smalltalk parts are described. This order does not reflect the order you use to implement the C++ parts. Each part source code should be generated in the same directory (for example \WTA\WORKING) to facilitate the dependencies resolution when compiling and link-editing the CGI programs.

### Table 11. Nonvisual Parts

<table>
<thead>
<tr>
<th>Part Name</th>
<th>File Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>WtaPackageCreationProcessor</td>
<td>WTPCKGCP</td>
</tr>
<tr>
<td>WtaPackageListProcessor</td>
<td>WTPCKGLP</td>
</tr>
<tr>
<td>WtaPackagePricer</td>
<td>WTPCKGPR</td>
</tr>
<tr>
<td>WtaPackagePurchaseProcessor</td>
<td>WTPCKGPP</td>
</tr>
<tr>
<td>WtaPersonalInfoProcessor</td>
<td>WTPINFOF</td>
</tr>
<tr>
<td>WtaRouteServiceProcessor</td>
<td>WTRSRVRP</td>
</tr>
<tr>
<td>WtaWelcomeProcessor</td>
<td>WTWLCMEP</td>
</tr>
</tbody>
</table>

VisualAge Parts

In this section we detail each visual and nonvisual VisualAge part.

**WtaWelcomeView**

The *WtaWelcomeView* is the first page displayed when a Web surfer connects to the Web Travel Agent application. Because this page is static, its HTML code is directly generated by the Visual Builder. You build the page as a visual part that inherits from *IFrameWindow* in (Figure 123).
The title of the IFrameWindow is set to the home page title. A background is associated with the page by filling in the User HTML Attributes entry field in the IFrameWindow settings notebook (Figure 124). The User HTML Attributes entry field is useful for setting up specific page parameters that you cannot specify with the Visual Builder.
You replace the standard canvas provided by default as the frame window client area with an \texttt{IMultiCellCanvas}. The \texttt{IMultiCellCanvas} part is mapped to an HTML table and is useful for laying out your controls.

An \texttt{IImgControl} is added to the first cell of the first row of the \texttt{IMultiCellCanvas} to display a banner image. You specify the proper image file by filling in the image URL attribute in the settings notebook of the \texttt{IImgControl} (see Figure 125).

![IImgControl Settings Notebook](image)

\textbf{Figure 125. IImgControl Settings Notebook}

Three \texttt{IHeadControl} parts are laid out on the \texttt{IMultiCellCanvas} and their head level is set appropriately from the settings notebook (Figure 126). Another \texttt{IMultiCellCanvas} part is used to hold the various buttons and images for each option available. Make sure you name the push buttons by specifying the names in their settings notebook as shown in Table 12.

\begin{table}[h]
\centering
\begin{tabular}{|l|l|}
\hline
\textbf{Push Button Name} & \textbf{Text Attribute} \\
\hline
PBCreate & Create \\
PBDisplay & Display \\
PBMModify & Modify \\
PBView & View \\
\hline
\end{tabular}
\caption{Push Button Names and TextAttributes}
\end{table}

Because the page is static, you do not have to set the \texttt{HTML Name} for the push buttons. In effect, the \texttt{HTML Name} attribute is used to associate a name with your HTML FORM control when the CGI program dynamically generates the HTML flow. Setting the \texttt{HTML Name} attribute is, however, a good habit to get into. An \texttt{IImgControl} at the bottom of the page displays a colored ruler.
You must associate a form action and a form method with the outmost `IMultiCellCanvas` part by setting up the proper entry fields in its settings notebook (Figure 127). The name given to the form action corresponds to the `WtaWelcomeProcessor` executable file that you build in the next section (see also “Naming Conventions” on page 256).

Finally, you can generate the HTML code for the Web Travel Agent application by selecting **Save** and **Generate Part** from the **File** menu item of the Visual Builder menu bar. The code should look like this HTML code:
<html><head>
<title>Web Travel Agency</title>
<meta name="keywords" content="Web Travel Agency">
<meta name="abstract" content="Web Travel Agency">
<meta name="cc" content="int">
<meta name="language" content="en.us">
</head>
<body background="/wtapix/backgrnd.jpg">
<form action="/wta/cgi-bin/wtwlcmep.exe" method="post">
<a name="ImgControl1"><img src="/wtapix/wta.gif" alt="ImgControl1"></a>
<h2><i><a name="HeadControl1">Welcome to the Web Travel Agency</a></i></h2>
<h4><i><a name="HeadControl2">Providing one-stop shopping for your business and leisure travel needs.</a></i></h4>
<h3><a name="HeadControl3">Click the button for the service you desire:</a></h3>
<table>
<tr><td>&nbsp;</td><td><a name="ImgControl2"><img src="/wtapix/electro.gif" alt="ImgControl2"></a></td><td>&nbsp;</td><td><input type="submit" value="Create" name="PBCreate"></td><td>&nbsp;</td><td><a name="StaticText1">Create a new travel package</a></td></tr>
<tr><td>&nbsp;</td><td><a name="ImgControl3"><img src="/wtapix/see.gif" alt="ImgControl3"></a></td><td>&nbsp;</td><td><input type="submit" value="Display" name="PBDisplay"></td><td>&nbsp;</td><td><a name="StaticText2">Display your existing travel packages</a></td></tr>
<tr><td>&nbsp;</td><td><a name="ImgControl4"><img src="/wtapix/feedback.gif" alt="ImgControl4"></a></td><td>&nbsp;</td><td><input type="submit" value="Modify" name="PBModify"></td><td>&nbsp;</td><td><a name="StaticText3">Modify your personal information</a></td></tr>
<tr><td>&nbsp;</td><td><a name="ImgControl5"><img src="/wtapix/movie.gif" alt="ImgControl5"></a></td><td>&nbsp;</td><td><input type="submit" value="View" name="PBView"></td><td>&nbsp;</td><td><a name="StaticText4">See attractions in cities around the world</a></td></tr>
</table>
</form>
</body></html>
WtaWelcomeProcessor

The WtaWelcomeProcessor generates the session key for the current connection. This session key is unique. It consists of the current date and time of the server machine. The date and time are retrieved from the ITime and IDate parts (Figure 128). The generated session key is used to initialized the defaultApplicationName of an IProfile part whose name is set to DATA.INI. This session key is transmitted to the constructor of two factories to set a hidden entry that which holds the key. The factories enable the creation of either WtaPackageCreationView or WtaCustomerAuthenticationView according to the push button the customer clicks on in the WtaWelcomeView.

Figure 128. WtaWelcomeProcessor

An ICGL part is used to parse the input sent by the Web server. Custom logic is attached to the ICGL part to select the HTML page to be generated (Figure 129). The custom logic also sets the requestedService information to the proper service before creating the corresponding HTML page. This session data is used in “WtaCustomerAuthenticationProcessor” on page 305 to create the correct HTML page from the customer authentication form. The requestedService key and its value are added to the session data through the IProfile part, SessionData, dropped inside the free-form surface.

If the customer clicks the Create push button, the WtaPackageCreationView is displayed. Upon clicking either the Display or Modify push buttons, the WtaCustomerAuthenticationView is displayed to the user to prompt him for his name and his identifier.
if (target->get("PBCreate") == "Create")
{
    source->iSessionData->addOrReplaceElementWithKey("requestedService",
        "Create Package");
    source->iCreatePackage->create();
}

if ((target->get("PBDisplay") == "Display") ||
    (target->get("PBReturn") == "Return"))
{
    source->iSessionData->addOrReplaceElementWithKey("requestedService",
        "Display Package");
    source->iCustomerAuthentication->create();
}

if (target->get("PBModify") == "Modify")
{
    source->iSessionData->addOrReplaceElementWithKey("requestedService",
        "Modify Personal Info");
    source->iCustomerAuthentication->create();
}

Figure 129. Custom Logic Source Code for WtaWelcomeProcessor

An IDatstore part creates, at runtime, a connection to the WTADB database before the HTML page is generated.

Remark: To create the WtaWelcomeProcessor, you first have to build WtaPackageCreationView and WtaCustomerAuthenticationView to set the type of each factory part.

WtaPackageCreationView

When the Web surfer chooses to create a new package, he is prompted to select a destination, a departure date, and duration (Figure 130). WtaPackageCreationView is built for this purpose and ensures that all information is properly entered by posting the CGI program, WTPCKGCPEXE. (You build this program in the next section.)

Two parts are added to the free-form surface: a CityLocationManager part, which retrieves all possible destinations available from the WTADB database, and an IDate part, which retrieves the system date.

An event-to-action connection from the ready event of the free-form surface to the refresh action of the LocationManager fetches all cities available from the LOCATION table. The CityLocationManager part populates the ICollectionViewListBox with the cities, using an attribute-to-attribute connection from the items attribute of the LocationManager to the items of the ICollectionViewListBox.
Chapter 9. Implementation with VisualAge for C++

VisualAge Parts

Figure 130. WtaPackageCreationView

The session key is transmitted to WtaPackageCreationView from the WtaWelcomeProcessor part. A hidden entry field is used to hold the session key. Its text attribute is promoted to allow transmission between the two parts (Figure 131).

An IStaticText part, underneath the Continue push button (it is not displayed in Figure 130 because its text attribute is set to NULL) is used to display a potential error message if the Web surfer input is not valid. The Web surfer input is checked by WtaPackageCreationProcessor. The text attribute of the static text is also promoted to make it accessible from WtaPackageCreationProcessor.
The list of all possible destinations is populated by the CityLocationManager part that you build with the Data Access Builder tool (refer to “Database Parts” on page 340). The cities are displayed inside an ICollectionViewListBox that manages objects of type CityLocation* (Figure 132).

Figure 132. ICollectionViewListBox Settings Notebook
To display the city names in the ICollectionViewListBox, select the default string generator provided with CityLocationManager. In the Data Access Builder settings notebook of the City part, only the LOCATION_NAME attribute is chosen to be displayed (Figure 133).

![CityLocation Settings Notebook](image)

**Figure 133. CityLocation Settings Notebook**

**WtaPackageCreationProcessor**

The *WtaPackageCreationProcessor* part checks the input provided by the Web surfer in *WtaPackageCreationView*. If the input is valid, it displays the *WtaComponentSelectionView* HTML page. If the input is invalid, it redisplays *WtaPackageCreationView* with the corresponding error message.

The structure of the nonvisual part is typical of all of the processors that are created in the application. In this section we devote some time to explaining how to build a nonvisual part.

The *WtaPackageCreationProcessor* part is a nonvisual part which inherits from *IStandardNotifier*. It is built half visually and half by coding the specific actions needed to check the user input (Figure 134).

The part consists of an *ICGI* part that transmits the session key from the hidden entry field of the HTML form that called this CGI program. The session key is used to initialize the *defaultApplicationName* of an *IProfile* part that represents the session data.
Two factories are used to create the next HTML page (in this case, \texttt{WtaComponentSelectionView}) or \texttt{WtaPackageCreationView} if the user input is invalid.

The \emph{defaultApplicationName} attribute of the \texttt{IProfile} part is transmitted to each factory to initialize the text attribute of a hidden entry field that holds the session key for this purpose.

An \texttt{IDatastore}, \texttt{DBConnection}, is added to the free-form surface because the HTML pages that are created from the factories parts access the database.

An \texttt{IVBStringPart}, \texttt{ErrorMsg}, stores the error message returned upon checking the user input. This error message is transmitted to the \texttt{WtaPackageCreationView} factory to display the error message to the user.

The Part Interface Editor allows you to define eight actions, six of them map the Smalltalk implementation of the same part (Figure 135). \texttt{CheckEntry} is the general action that calls all other actions. It returns a potential error message as a string. If the string is null, the input is checked as valid, and the package creation can proceed. Otherwise \texttt{WtaPackageCreationView} is displayed again with the error message.
Figures 136 through 143 show the source code of each action.

```c++
IString WtaPackageCreationProcessor::processDuration()
{
    IString errorMsg = "";
    IString duration = iCGI->get("duration");
    if (duration.isDigits())
    {
        int durationInt = duration.asInt();
        if (durationInt <= 0)
            errorMsg = "Oops, you entered " + duration + " for
                  the number of days. Try again.";
        else
            iSessionData->addOrReplaceElementWithKey("numberOf
                  Days", duration.strip());
    }
    else
        errorMsg = "The number of days is not a number.
                  Try again!";
    return errorMsg;
}
```

Figure 136. ProcessDuration Source Code
IString WtaPackageCreationProcessor::checkIfDateIsPassed()
{
    IString errorMsg = "";
    IString monthSt = iCGI->get("month");
    IString daySt   = iCGI->get("day");
    IString yearSt  = iCGI->get("year");
    if (convertDate(yearSt, monthSt, daySt) < IDate::today())
        errorMsg = "Oops, you entered a date preceeding today's date. Try again!";
    return errorMsg;
}

Figure 137. CheckIfDateIsPassed Source Code

void WtaPackageCreationProcessor::processDateStrings()
{
    IString monthSt = iCGI->get("month");
    IString daySt   = iCGI->get("day");
    IString yearSt  = iCGI->get("year");
    int duration = iCGI->get("duration").asInt();
    IDate startDate = convertDate(yearSt, monthSt, daySt);
    IDate endDate   = startDate + duration - 1;
    iSessionData->addOrReplaceElementWithKey("startDate", startDate.asString().strip());
    iSessionData->addOrReplaceElementWithKey("endDate", endDate.asString().strip());
}

Figure 138. ProcessDateStrings Source Code

IString WtaPackageCreationProcessor::processLocationName()
{
    IString errorMsg = "";
    IString location = iCGI->get("location");
    if (location != "location")
        iSessionData->addOrReplaceElementWithKey("location-Name", location.strip());
    else
        errorMsg = "Oops, you forgot to choose a location. Try again!";
    return errorMsg;
}

Figure 139. ProcessLocationName Source Code
IString WtaPackageCreationProcessor::checkEntry()
{
    IString errorMsg = "";
    errorMsg = checkDaysInMonth();
    if (errorMsg == "")
        errorMsg = checkIfDateIsPassed();
    if (errorMsg == "")
        errorMsg = processDuration();
    if (errorMsg == "")
        errorMsg = processLocationName();
    if (errorMsg == "")
        processDateStrings();
    return errorMsg;
}

Boolean WtaPackageCreationProcessor::checkIfDateIsValid(IString yearSt, IString monthSt, IString daySt)
{
    int day = daySt.asInt();
    int year = yearSt.asInt();
    Boolean dateValid = false;
    if (monthSt == "January")
        dateValid = IDate::isValid(IDate::January, day, year);
    if (monthSt == "February")
        dateValid = IDate::isValid(IDate::February, day, year);
    if (monthSt == "March")
        dateValid = IDate::isValid(IDate::March, day, year);
    if (monthSt == "April")
        dateValid = IDate::isValid(IDate::April, day, year);
    if (monthSt == "May")
        dateValid = IDate::isValid(IDate::May, day, year);
    if (monthSt == "June")
        dateValid = IDate::isValid(IDate::June, day, year);
    if (monthSt == "July")
        dateValid = IDate::isValid(IDate::July, day, year);
    if (monthSt == "August")
        dateValid = IDate::isValid(IDate::August, day, year);
    if (monthSt == "September")
        dateValid = IDate::isValid(IDate::September, day, year);
    if (monthSt == "October")
        dateValid = IDate::isValid(IDate::October, day, year);
    if (monthSt == "November")
        dateValid = IDate::isValid(IDate::November, day, year);
    if (monthSt == "December")
        dateValid = IDate::isValid(IDate::December, day, year);
    return dateValid;
}
IString WtaPackageCreationProcessor::checkDaysInMonth()
{
  IString errorMsg = "";
  IString monthSt = iCGI->get("month");
  IString daySt = iCGI->get("day");
  IString yearSt = iCGI->get("year");
  if (!checkIfDateIsValid(yearSt, monthSt, daySt))
    errorMsg = "Oops, you entered an invalid date. Try again!";
  return errorMsg;
}

Figure 142. CheckDaysInMonth Source Code

IDate WtaPackageCreationProcessor::convertDate(IString year, IString month, IString day)
{
  int day = daySt.asInt();
  int year = yearSt.asInt();
  IDate aDate;
  if (monthSt == "January")
    aDate = IDate(IDate::January, day, year);
  if (monthSt == "February")
    aDate = IDate(IDate::February, day, year);
  if (monthSt == "March")
    aDate = IDate(IDate::March, day, year);
  if (monthSt == "April")
    aDate = IDate(IDate::April, day, year);
  if (monthSt == "May")
    aDate = IDate(IDate::May, day, year);
  if (monthSt == "June")
    aDate = IDate(IDate::June, day, year);
  if (monthSt == "July")
    aDate = IDate(IDate::July, day, year);
  if (monthSt == "August")
    aDate = IDate(IDate::August, day, year);
  if (monthSt == "September")
    aDate = IDate(IDate::September, day, year);
  if (monthSt == "October")
    aDate = IDate(IDate::October, day, year);
  if (monthSt == "November")
    aDate = IDate(IDate::November, day, year);
  if (monthSt == "December")
    aDate = IDate(IDate::December, day, year);
  return aDate;
}

Figure 143. ConvertDate Source Code
A few connections are drawn on the free-form surface:

- A connection to the WTADB database is established by connecting the `ready` event from the free-form surface to the `connect` action of the `IDatastore` part.

- The session key is retrieved from the hidden entry field of the previous HTML page by connecting the `ready` event of the free-form surface to the `get` method of the `ICGI` part. The connection parameter is the entry field name, in our case, `sessionId`.

- The `checkEntry` action is fired by connecting the `ready` event of the free-form surface to the `checkEntry` of the free-form surface.

- The `actionResult` attribute of the preceding connection is connected to the `text` attribute of the `errorMsg` part.

- The `actionResult` attribute is also connected to a `customLogic` attribute of the free-form surface. The `customLogic` attribute holds a code snippet that tests the error message string returned by `checkEntry` and creates the proper HTML page by calling the `create` method of one of the factories.

- The `text` attribute of the `errorMsg` part is connected to a promoted `text` attribute of the static text that holds the error message in `WtaPackageCreationView`.

- The `sessionKey` is transmitted to each factory by connecting the `defaultApplicationName` attribute of the `IProfile` part to the promoted `text` attribute of the hidden entry field of each factory.

The order of the connections from the `ready` event is important (Figure 144) because it implies the order the connections are executed at run time.

**Remark:** To create the `WtaPackageCreationProcessor` part, you first have to build `WtaPackageCreationView` and `WtaComponentSelectionView` to set the type of each factory part.
**WtaComponentSelectionView**

The **WtaComponentSelectionView** part (Figure 145) allows the Web surfer to choose a package component for which he or she wants to make a reservation. This view is the “home base” view of the Web Travel Agent application, because each time the user selects a component, the view is displayed to summarize the choice he or she just made. On completing the package, the Web surfer can click on the **Continue** push button to validate the package and book it.

![Figure 145. WtaComponentSelectionView](image)

The client view is built from an **IMultiCellCanvas** on which are laid out **IStaticText**, **IImgControl**, and **IPushButton** parts as shown in Figure 145. An entry field whose **text** attribute is promoted, is used as usual to carry the session key from one view to another (Figure 146).

An **IProfile** part is used to collect the session data from the session key received by the entry field. The second row of **IMultiCellCanvas** displays the trip destination, its duration, and start and end dates. That data is collected from the **IProfile** part by using the **elementWithKey** action.

Three actions are defined for displaying the choice for each already selected component: **hotel**, **flight**, and **car** (Figure 147). Those actions are coded in the **hpv** and **cpv** files associated with **WtaComponentSelectionView**.
Figure 146. *Promote Features from SessionKey*

Notice that we present here two techniques for accessing and displaying session data: one using an `IProfile` part and visual programming to access the session data, the other using straight C++ coding by developing source code action.

Figure 147. *WiaComponentSelectionView Part Interface Editor*
Figures 148 through 150 show the source code of the `hotelPreferences`, `flightPreferences`, and `carPreferences` actions.

```cpp
IString WtaComponentSelectionView::hotelPreferences()
{
    IString hotelPref= "";
    if (iSessionData->containsKeyName("hotelName"))
    {
        IString numberOfRooms = iSessionData->elementWithKey("numberOfRooms");
        IString roomCategory = iSessionData->elementWithKey("roomCategory");
        IString roomSmoking = iSessionData->elementWithKey("roomSmoking");
        IString hotelName = iSessionData->elementWithKey("hotelName");
        hotelPref = "You have requested "
                    + numberOfRooms + " "
                    + roomSmoking + " "
                    + roomCategory
                    + " room(s) at the "
                    + hotelName
                    + ".";
    }
    return hotelPref;
}
```

**Figure 148. HotelPreferences Action Source Code**

```cpp
IString WtaComponentSelectionView::flightPreferences()
{
    IString flightPref= "";
    if (iSessionData->containsKeyName("flightNum"))
    {
        IString numberOfSeats = iSessionData->elementWithKey("numberOfSeats");
        IString seatClass = iSessionData->elementWithKey("seatClass");
        IString fromCity = iSessionData->elementWithKey("fromCity");
        flightPref = "You have requested "
                     + numberOfSeats
                     + " "
                     + seatClass
                     + " airline seat(s) departing from "
                     + fromCity
                     + ".";
    }
    return flightPref;
}
```

**Figure 149. FlightPreferences Action Source Code**
WtaComponentSelectionProcessor

The WtaComponentSelectionProcessor part (Figure 151) is executed when the Web surfer clicks a button on WtaComponentSelectionView. Its structure is similar to WtaWelcomeProcessor. It inherits from IStandardNotifier and is built visually using the following parts:

- IProfile, named SessionKey, stores the session data
- ICGI, retrieves the sessionKey from the WtaComponentSelectionView form
- IDatastore, named DBConnection, establishes a database connection with WTADB
- IVBFactory parts create the corresponding view when the Web surfer selects the corresponding option (that is, select a hotel, a flight, and so on).

For each possible component selection, a factory part allows to display the corresponding view to choose the details related to the component. A custom logic connection (Figure 152) checks the push button that is clicked by the user in WtaComponentSelectionView and creates the proper object from its factory.

If the Web surfer clicks Cancel, the session data is deleted by deleting all the element recorder for the connection session key.

**IString WtaComponentSelectionView::carPreferences()**

```c++
IString carPref = "";
if (iSessionData->containsKeyName("carCompany"))
{
    IString carCompany = iSessionData->elementWithKey("carCompany");
    IString carClass = iSessionData->elementWithKey("carClass");
    carPref = "You have requested a "
    + carClass
    + " from "
    + carCompany
    + ".";
}
return carPref;
```

Figure 150. CarPreferences Action Source Code
An event-to-action connection gets the current session key from the ICGI part. This session key is located in the SessionKey hidden entry field of WtaComponentSelectionView that carries the key through the different views.

The session key initializes the defaultApplicationName attribute of the IProfile part sessionKey. This defaultApplicationName attribute of sessionKey is transmitted to each factory to initialize the content of its own SessionKey hidden entry field. For this purpose, the text attribute of this entry field must be promoted for each corresponding visual part.

```cpp
if (target->get("PBHotel") == "Hotel")
    source->iHotelSelection->create();
if (target->get("PBFlight") == "Flight")
    source->iFlightSelection->create();
if (target->get("PBCar") == "Car")
    source->iCarSelection->create();
if (target->get("PBContinue") == "Continue")
    source->iPackagePricer->create();
if (target->get("PBCancel") == "Cancel")
{
    IProfile myProfile("DATA.INI");
    myProfile.deleteElementWithApplication(source->isessionKey->text());
    source->iDBConnection->disconnect();
    source->iWelcomeSelection->create();
}
```

**Figure 151. WtaComponentSelectionProcessor**

**Figure 152. Custom Logic Source Code for Wta Component Selection Processor**
The execution order of each connection is important (Figure 153).

![Figure 153. Execution Order of the Connections](image)

**WtaHotelSelectionView**

The **WtaHotelSelectionView** allows the user to select his or her hotel preferences. This form is built similarly to **WtaPackageCreationView**.

As usual, a hidden entry field receives the session key from the processor that builds this view. The session key is transmitted to an **IProfile** part (see **SessionData** in Figure 154), which retrieves the trip location and displays it in the static text located at the top of the form.

![Figure 154. WtaHotelSelectionView](image)
An SQL clause is built from the package destination and stored in the `IVBStringPart` clause. The clause is transmitted to the `HotelManager` to perform an SQL select against the HOTEL table. The resulting hotels are displayed in an `ICollectionViewListBox`.

Similarly to the `ICollectionViewListBox` part we use in `WtaPackageCreationView`, a string generator class allows to display only the name of each hotel (Figure 155).

![ICollectionViewListBox String Generator](image)

**Figure 155.** `ICollectionViewListBox String Generator`

Underneath the `Continue` push button (see Figure 154), an `ISetCanvas` part holds an `IStaticText` part whose purpose is to display an error message, generated by the `WtaHotelSelectionProcessor`, if the user selection is not valid. The `text` attribute of this static text is promoted (Figure 156), to let `WtaHotelSelectionProcessor` send back the error message to `WtaComponentSelectionView`.

**Remark:** To create the `WtaHotelSelectionView` part, you first have to build `WtaHotelSelectionView` and `WtaComponentSelectionView` to set the type of each factory part.
The `WtaHotelSelectionProcessor` part processes the Web surfer input from `WtaHotelSelectionView`. If the Web surfer input is invalid, an error message is returned by the `processHotel` member function. The message is displayed in `WtaHotelSelectionView` when an instance of this part is created dynamically by the `HotelSelection` factory object (Figure 157). If the user input is valid and the `processHotel` member function returns a NULL string as the error message, the `ComponentSelection` factory object creates a `WtaComponentSelectionView` object. When the `WtaComponentSelectionView` object is created, the `hotelPreferences` action is fired from its `ready` event, and user selection is summarized in the corresponding static text.

The `processHotel` action is defined in the Part Interface Editor of the `WtaHotelSelectionProcessor` (Figure 158).
A custom logic connection is used to create either a `WtaHotelSelectionView` or `WtaComponentSelectionView` of the proper object according to whether or not the error message returned by the `processHotel` action is NULL (Figure 159).
Figure 159. Custom Logic Source Code for WtaSelectionProcessor

Notice in the `processHotel` (Figure 160 and Figure 161) source code the creation of session data that is used by the `hotelPreferences` action of `WtaComponentSelectionView`.

```c++
if (target->iErrorMsg->text() == "")
    target->iComponentSelection->create();
else
    target->iHotelSelection->create();

Figure 160. ProcessHotel Action Source Code

// Feature source code generation begins here...
IString WtaHotelSelectionProcessor::processHotel()
{
    // Initialize errorMsg
    IString errorMsg = "";
    // Point the session data in the registry
    IProfile mySessionData("DATA.INI");
    mySessionData.setDefaultApplicationName(iSessionKey-Text());
    // Check if a hotel is selected
    IString aHotelName = ICGI->get("hotelName");
    if (aHotelName == "")
        errorMsg = "Oops, you forgot to select a hotel.  
                     Try again!";
    else
    { // Get room category
        IString aCategory = ICGI->get("roomCategory");
        // Check number of rooms
        IString aNumberOfRooms = ICGI->get("numberOfRooms");
        if (!aNumberOfRooms.isDigits())
            errorMsg = "Oops, the number of rooms you 
                        entered is not a number. Try again!";
        else
        { int aNumberOfRoomsInt = aNumberOfRooms.asInt();
            if (aNumberOfRoomsInt <= 0)
                errorMsg = "Oops, the number of seats you 
                            entered is negative or null. Try again!.";
            else
            { // Get the number of nights spent
                int numberOfDays = mySessionData.elementWith-
                                  Key("numberOfDays") asInt();
                int numberOfNights = numberOfDays  ? numberOf-
                                    OfDays - 1 : numberOfDays;
                // Get smoking or nonsmoking
                IString smoking = ICGI->get("smok-
                                ing").strip();
                if (smoking != "nonsmoking")
                    smoking = "smoking";
```
// Get information regarding the hotel
IString area = mySessionData.elementWithKey("locationName");
    HotelManager aHotelMng;
    IString aClause = "HOTEL_NAME='"+aHotelName+"' AND LOC_NAME='"+area+"'";
    aHotelMng.select(aClause);
    Hotel* theHotel = aHotelMng.items()->firstElement();

    // Update the session data
    mySessionData.addOrReplaceElementWithKey("hotelId", (theHotel->hotelidAsString()).strip());
    mySessionData.addOrReplaceElementWithKey("hotelCapacity", (theHotel->capacityAsString()).strip());
    mySessionData.addOrReplaceElementWithKey("hotelBooked", (theHotel->bookedAsString()).strip());
    mySessionData.addOrReplaceElementWithKey("hotelPrice", (theHotel->daily_priceAsString()).strip());
    mySessionData.addOrReplaceElementWithKey("hotelName", aHotelName.strip());
    mySessionData.addOrReplaceElementWithKey("roomSmoking", smoking);
    mySessionData.addOrReplaceElementWithKey("roomCategory", aCategory.strip());
    mySessionData.addOrReplaceElementWithKey("numberOfRooms", aNumberOfRooms.strip());
    mySessionData.addOrReplaceElementWithKey("numberOfNights", IString(numberOfNights));
}
}
return errorMsg;

// Feature source code generation ends here.

Figure 161. ProcessHotel Source Code (Continued)
WtaFlightSelectionView

WtaFlightSelectionView (Figure 162) is similar to WtaHotelSelectionView. It allows the Web surfer to select his or her flight preferences.

![WtaFlightSelectionView diagram]

Figure 162. WtaFlightSelectionView

An IVBStringPart, Clause, holds an SQL clause which is passed as a parameter to the select action of CityLocationManager. The SQL select retrieves all available cities but the destination city.

WtaFlightSelectionProcessor

The WtaFlightSelectionProcessor part (Figure 163) is similar to WtaHotelSelectionProcessor. The processFlight action processes the user input retrieved from WtaFlightSelectionView. If the input is valid, processFlight returns an empty string as a potential error message, and the ComponentSelection factory creates a WtaComponentSelectionView object. When the WtaComponentSelectionView is ready to be displayed as an HTML page, the flightPreferences action is fired. This action displays the flight preferences from the session data updated in the processFlight action. If the customer’s input is invalid, processFlight returns a string message that contains the error message. The error message is sent as a constructor parameter to the FlightSelection factory, which creates a WtaFlightSelectionView object and prompts the Web surfer to correct the input.
A custom logic connection fires a `create` action on one of the factory objects regarding the string returned by the `processFlight` action (Figure 164).

```cpp
if (target->iErrorMsg->text() == "")
    target->iComponentSelection->create();
else
    target->iFlightSelection->create();
```

**Figure 164. Custom Logic Source Code for WtaFlightSelectionProcessor**

Figures 165 and 166 show the source code for the `processFlight` action.
// Feature source code generation begins here...
IString WtaFlightSelectionProcessor::processFlight()
{
    // Initialize errorMsg
    IString errorMsg = "";
    // Point the session data in the registry
    IProfile mySessionData("DATA.INI");
    mySessionData.setDefaultApplicationName(iSessionKey-
    >text());
    // Check if a departure city is selected
    IString aDepartureCity = ICGI->get("locationName");
    if (aDepartureCity == "")
        errorMsg = "Oops, you forgot to select a departure
city. Try again!";
    else
    {
        // Get seat category
        IString aSeatClass = ICGI->get("seatClass");
        // Check number of seats
        IString aNumberOfSeats = ICGI->get("numberOfSeats");
        if (!aNumberOfSeats.isDigits())
            errorMsg = "Oops, you forgot to enter the number of
seats, or the number of seats you entered is not a number.
Try again!";
        else
        {
            int aNumberOfSeatsInt = aNumberOfSeats.asInt();
            if (aNumberOfSeatsInt <= 0)
                errorMsg = "Oops, the number of seats you entered
is negative or null. Try again!.";
            else
            {
                // Get information regarding the flight
                IString area = mySessionData.elementWith-
Key("locationName");
                FlightManager aFlightMng;
                IString aClause = "FROM_CITY='"+aDeparture-
City+' AND TO_CITY='"+area+'";
                aFlightMng.select(aClause);
                Flight* theFlight = aFlightMng.items()-
->firstElement();

Figure 165. ProcessFlight Action Source Code (Part 1/2)
WtaCarSelectionView

The WtaCarSelectionView part (Figure 167) allows the Web surfer to select a car preference. A “helper” view enables the Web surfer visualize the car category from which the application allows to select. Clicking on the Classes push button displays the WtaCarClassView part that you build in “WtaCarClassView” on page 290.

When the WtaCarClassView is displayed, the Web surfer can choose a car category by clicking on the proper push button and the WtaCarSelectionView part is displayed again. An ICGI part retrieves the push button that was clicked and selects the corresponding radio button through a custom logic connection (Figure 168).

Listing 166. ProcessFlight Action Part (2/2)

// Update the session data
mySessionData.addOrReplaceElementWithKey("flightNum", (theFlight->flightnumAsString()).strip());
mySessionData.addOrReplaceElementWithKey("seatClass", aSeatClass.strip());
mySessionData.addOrReplaceElementWithKey("fromCity", (theFlight->from_cityAsString()).strip());
mySessionData.addOrReplaceElementWithKey("toCity", (theFlight->to_cityAsString()).strip());
mySessionData.addOrReplaceElementWithKey("basePrice", (theFlight->base_priceAsString()).strip());
mySessionData.addOrReplaceElementWithKey("numberOfSeats", aNumberOfSeats.strip());
}
)
return errorMsg;
} // Feature source code generation ends here.
if (target->get("PBClassA") == "Class A")
    source->iRBClassA->select();
if (target->get("PBClassB") == "Class B")
    source->iRBClassB->select();
if (target->get("PBClassC") == "Class C")
    source->iRBClassC->select();
if (target->get("PBClassD") == "Class D")
    source->iRBClassD->select();
if (target->get("PBClassE") == "Class E")
    source->iRBClassE->select();
if (target->get("PBClassF") == "Class F")
    source->iRBClassF->select();

Figure 167. WtaCarSelectionView

Figure 168. Custom Logic Source Code for WtaCarSelectionView
The *WtaCarClassView* part (Figure 169) displays a selection of categories from which the Web surfer can choose. We used the *IMultiCellCanvas* and *ISetCanvas* parts to align the various car bitmaps. Notice also the use of `<HR>` to create an HTML rule.

Table borders are shown by checking the *Has Border* check box in the Web page of the table settings notebook (Figure 170).
The `WtaCarSelectionProcessor` part (Figure 171) is slightly different from the other processor parts. In effect, because the user can access an extra HTML page, `WtaCarClassView`, the processor must check the Web surfer input only when the **Continue** push button is clicked. When a button of `WtaCarClassView` is clicked, the processor should create an object of `WtaCarSelectionView` to redisplay the car selection HTML page. When the **Classes** button of `WtaCarSelectionView` is clicked, the processor should create an object of `WtaCarClassView` to display the different categories available.

**Figure 171. WtaCarSelectionProcessor**
A custom logic connection handles this routing by retrieving the push button that is clicked from an ICGI part (Figure 172). Once the custom logic is executed, the processCar action (Figure 173) is executed to validate the user input retrieved from WtaCarSelectionView.

if (target->get("PBClasses") == "Classes")
    source->iCarClassView->create();
if (target->get("PBCancel") == "Cancel")
    source->iCarSelection->create();
if ( target->get("PBClassA") == "Class A" ||
    target->get("PBClassB") == "Class B" ||
    target->get("PBClassC") == "Class C" ||
    target->get("PBClassD") == "Class D" ||
    target->get("PBClassE") == "Class E" ||
    target->get("PBClassF") == "Class F")
    source->iCarSelection->create();

Figure 172. Custom Logic Source Code for WtaCarSelectionProcessor

// Feature source code generation begins here...
IString WtaCarSelectionProcessor::processCar()
{ // Initialize errorMsg
    IString errorMsg = "";
    // Point the session data in the registry
    IProfile mySessionData("DATA.INI");
    mySessionData.setDefaultApplicationName(iSessionKey->text());
    // Check if a car company is selected
    IString aCarCompany = ICGI->get("carCompany");
    IFUNCTRACE_DEVELOP();
    ITRACE_DEVELOP("----> carCompany = " + aCarCompany);
    if (aCarCompany=="carCompany")
        errorMsg = "Oops, you forgot to select a rental company. Try again!";
    else  // Get the car class
        IString aCarClass = ICGI->get("carClass");
        ITRACE_DEVELOP("----> carClass = " + aCarClass);
        if (aCarClass == "carClass")
            errorMsg = "Oops, you forgot to select a car class. Try again!";
    else
        { // Update the session data
            mySessionData.addOrReplaceElementWithKey("carClass",aCarClass.strip());
            mySessionData.addOrReplaceElementWithKey("carCompany",aCarCompany.strip());
        }
    return errorMsg;
} // Feature source code generation ends here.

Figure 173. ProcessCar Action Source Code
Like the previous processors, this action returns a string that contains a potential error message. If the string is NULL, the ComponentSelection factory creates a WtaComponentSelectionView, which displays a summary of the user preferences by executing the carPreferences action on the ready event. If the string is not NULL, the error message is transmitted to the CarSelection factory, which redisplay WtaCarSelectionView and prompts the user to correct the input. A custom logic connection controls the return value of the processCar action (Figure 174).

```cpp
if (target->iErrorMsg->text() == "")
    target->iComponentSelection->create();
else
    target->iCarSelection->create();
```

**Figure 174. Custom Logic Source Code for WtaCarSelectionProcessor**

The order of the connections from the ready event of this part is crucial to ensure that the processCar action is fired after the custom logic connection (Figure 175).

**Figure 175. Connection Order**

**WtaPackagePricer**

Once the different package components are selected, the Web surfer can proceed by clicking the Continue push button of WtaComponentSelectionView. The package is then priced by WtaPackagePricer, a nonvisual part (Figure 176).
The structure of the `WtaPackagePricer` part is similar to the other processor parts. The part's `ready` event fires the `PricePackage` action, which chains the other actions that you define in the Part Interface Editor (Figure 177). The `PricePackage` action returns a string that contains a potential error message. If the string is NULL, the `PackagePurchase` factory creates a `WtaPackagePurchaseView` object. If the string is not NULL, the error message is transmitted to the `PackageFailed` factory for display in the `WtaPackageFailedView` that the factory creates.
A custom logic connection checks the return code from the 
PricePackage action and calls the create action of a factory accordingly (Figure 178).

```c++
if (target->iErrorMsg->text() == "")
    target->iPackagePurchase->create();
else
    target->iPackageFailed->create();
```

**Figure 178. Custom Logic Source Code**

Figures 180 through 182 list the source code of the 
PriceFlight, PriceHotel, PriceCar, and PricePackage actions. Notice in the source code of each action that the session data is updated by adding the price of each component and the total price for the package.

```c++
IString WtaPackagePricer::priceFlight()
{
    // Initialize errorMsg
    IString flightPrice = "";
    float seatPrice, packageFlightPrice;
    // Point the session data in the registry
    IProfile mySessionData("DATA.INI");
    mySessionData.setDefaultApplicationName(iSessionKey->text());
    // Check if a flight reservation has to be priced
    if (mySessionData.containsKeyName("flightNum"))
    {
        IString seatClass = mySessionData.elementWithKey("seatClass");
        float basePrice = mySessionData.elementWithKey("basePrice").asDouble();
        int numberOfSeats = mySessionData.elementWithKey("numberOfSeats").asInt();
        if (seatClass == "Economy")
            seatPrice = basePrice;
        if (seatClass == "Business")
            seatPrice = basePrice * 1.25;
        if (seatClass == "First Class")
            seatPrice = basePrice * 1.50;
        packageFlightPrice = (int)(seatPrice * numberOfSeats * 100 + 0.5) / 100;
        flightPrice = IString(packageFlightPrice);
        mySessionData.addOrReplaceElementWithKey("packageFlightPrice", flightPrice.strip());
    }
    return flightPrice;
}
```

**Figure 179. PriceFlight Action Source Code**
IString WtaPackagePricer::priceHotel()
{
    IFUNCTRACE_DEVEL();
    // Initialize hotelPrice
    IString hotelPrice = "";
    // Point the session data in the registry
    IProfile mySessionData("DATA.INI");
    mySessionData.setDefaultApplicationName(iSessionKey->text());
    ITRACE_DEVEL(" ----> key: "+ iSessionKey->text());
    // Check if a hotel reservation has to be priced
    if (mySessionData.containsKeyName("hotelName"))
        {
            int roomsBooked = mySessionData.elementWithKey("hotelBooked").asInt();
            int numberOfRooms = mySessionData.elementWithKey("numberOfRooms").asInt();
            int hotelCapacity = mySessionData.elementWithKey("hotelCapacity").asInt();
            float roomPrice, packageHotelPrice;
            if (roomsBooked == hotelCapacity)
                hotelPrice = "All the rooms have been booked up.
Reselect a hotel.";
            else
                {
                    IString roomCategory = mySessionData.elementWithKey("roomCategory");
                    float dailyPrice = mySessionData.elementWithKey("hotelPrice").asDouble();
                    int numberOfNights = mySessionData.elementWithKey("numberOfNights").asInt();
                    if (roomCategory == "Standard")
                        roomPrice = dailyPrice;
                    if (roomCategory == "Deluxe")
                        roomPrice = dailyPrice * 1.25;
                    if (roomCategory == "Suite")
                        roomPrice = dailyPrice * 1.50;
                    packageHotelPrice = (int)(roomPrice * numberOfRooms
                        * numberOfNights) * 100 + 0.5) / 100;
                    hotelPrice = IString(packageHotelPrice);
                    mySessionData.addOrReplaceElementWithKey("package-
HotelPrice", hotelPrice.strip());
                }
        }
    return hotelPrice;
}

Figure 180. PriceHotel Action Source Code
IString WtaPackagePricer::priceCar()
{
    // Initialize errorMsg
    IString carPrice = "";
    float packageCarPrice = 0.0;
    // Point the session data in the registry
    IProfile mySessionData("DATA.INI");
    mySessionData.setDefaultApplicationName(iSessionKey->text());
    // Check if a car reservation has to be priced
    if (mySessionData.containsKeyName("carCompany"))
    {
        IString carClass = mySessionData.elementWithKey("carClass");
        int numberOfDays = mySessionData.elementWithKey("numberOfDays").asInt();
        if (carClass == "Class A")
            packageCarPrice = 30. * numberOfDays;
        if (carClass == "Class B")
            packageCarPrice = 36. * numberOfDays;
        if (carClass == "Class C")
            packageCarPrice = 42. * numberOfDays;
        if (carClass == "Class D")
            packageCarPrice = 48. * numberOfDays;
        if (carClass == "Class E")
            packageCarPrice = 54. * numberOfDays;
        if (carClass == "Class F")
            packageCarPrice = 72. * numberOfDays;
        carPrice = IString(packageCarPrice);
        mySessionData.addOrReplaceElementWithKey("packageCarPrice", carPrice.strip());
    }
    return carPrice;
}

Figure 181. PriceCar Action Source Code
Once each component of the package is checked available and priced, the `WtaPackagePurchaseView` summarizes the different components selected by the user.

This visual part is very similar to `WtaComponentSelectionView` where an `IProfile` part is used to collect the session data from the session key held by the hidden entry field (Figure 183).

Seven actions are defined in the Part Interface Editor for displaying the user preferences and the price for each selected component (Figure 184).

The code of the `carPrice`, `hotelPrice`, `flightPrice` and `totalPrice` actions produces a string which contains the price information for each component. The price is formatted using the locale currency of the server machine. The currency is retrieved from the operating systems settings using the `formatMoney` action.

The code for the first three actions is the same as the code in Figure 148, Figure 149, and Figure 150.

```cpp
IString WtaPackagePricer::pricePackage()
{
    IString errorMsg = "", aPrice;
    float packagePrice = 0.;
    // Test if a package has to be priced
    if ((priceHotel()+priceCar()+priceFlight()) != "")
    {
        // Point the session data in the registry
        IProfile mySessionData("DATA.INI");
        mySessionData.setDefaultApplicationName(iSessionKey-
>text());
        aPrice = priceHotel();
        if (aPrice.asDouble() != 0.)
            packagePrice += aPrice.asDouble();
        else
            errorMsg = aPrice;
        packagePrice += (priceCar()).asDouble();
        packagePrice += (priceFlight()).asDouble();
        mySessionData.addOrReplaceElementWithKey("packageTo-
>talPrice", IString(packagePrice).strip());
    }
    return errorMsg;
}
```

**Figure 182. PricePackage Action Source Code**
Figure 183. WtaPackagePurchaseView

Figure 184. WtaPackagePurchaseView Part Interface Editor
The price for each component is retrieved from the session data, which is updated in the `WtaPackagePricer` part (see Figures 180, Figure 181, and 179).

By clicking on either of the two push buttons, the `WtaPackagePurchaseProcessor` nonvisual part is executed.

Figures 185 through 189 list the source code of the `carPrice`, `hotelPrice`, `flightPrice`, `totalPrice`, and `formatMoney` actions.

```cpp
IString WtaPackagePurchaseView::carPrice()
{
  IFUNCTRACE_DEVELOP();
  IString carPrice= "";
  if (iSessionData->containsKeyName("carCompany"))
  {
    IString aPrice = iSessionData->elementWithKey("packageCarPrice");
    ITRACE_DEVELOP("----> function packageCarPrice "
      + aPrice);
    carPrice = "The price for the car is "
      + formatMoney(aPrice) + ".";
  }
  return carPrice;
}

Figure 185. CarPrice Action Source Code
```

```cpp
IString WtaPackagePurchaseView::hotelPrice()
{
  IString hotelPrice= "";
  if (iSessionData->containsKeyName("hotelName"))
  {
    IString aPrice = iSessionData->elementWithKey("packageHotelPrice");
    hotelPrice = "The price for the hotel is " + formatMoney(aPrice) + ".";
  }
  return hotelPrice;
}

Figure 186. HotelPrice Action Source Code
```
IString WtaPackagePurchaseView::flightPrice()
{
    IString flightPrice = "";
    if (iSessionData->containsKeyName("flightNum"))
    {
        IString aPrice = iSessionData->elementWithKey("packageFlightPrice");
        flightPrice = "The price for the flight is " + formatMoney(aPrice) + ";"
    }
    return flightPrice;
}

IString WtaPackagePurchaseView::totalPrice()
{
    IString totalPrice = "No component were selected to price.";
    if (iSessionData->containsKeyName("packageTotalPrice"))
    {
        IString aPrice = iSessionData->elementWithKey("packageTotalPrice");
        totalPrice = formatMoney(aPrice);
    }
    return totalPrice;
}

IString WtaPackagePurchaseView::formatMoney(IString aPrice)
{
    struct lconv* mylocale;
    IString defaultCurrency = "";
    IString formatedPrice = "";
    char string[80];
    float money = aPrice.asDouble();
    mylocale = localeconv();
    defaultCurrency = IString(mylocale->currency_symbol);
    if (defaultCurrency == "")
    {
        formatedPrice = "$ " + aPrice;
    }
    else
    {
        snprintf(string, 100, "%n", money);
        formatedPrice = string;
    }
    return formatedPrice;
}

Figure 187. FlightPrice Action Source Code

Figure 188. TotalPrice Action Source Code

Figure 189. FormatMoney Action Source Code
WtaPackagePurchaseProcessor

The WtaPackagePurchaseProcessor part is a router that displays the proper HTML page according to the push button clicked by the user in WtaPackagePurchaseView (Figure 190). If the user clicks the Change push button, the ComponentSelection factory creates a WtaPackagePurchaseView object to let the user change a package component. If the user clicks the Purchase push button, the CustomerId factory creates a WtaCustomerIdView to prompt the user to indicate whether he or she is a new or existing customer for the Web Travel Agent application.

![Diagram of WtaPackagePurchaseProcessor](image)

Figure 190. WtaPackagePurchaseProcessor

A custom logic connection controls the routing to WtaCustomerIdView or to WtaComponentSelectionView (Figure 191).

```cpp
if (target->get("PBPurchase") == "Purchase")
    source->iCustomerId->create();
if (target->get("PBChange") == "Change")
    source->iComponentSelection->create();
```

Figure 191. Custom Logic Source Code for WtaPackagePurchaseProcessor
WtaPackageFailedView

When a package component is not available, WtaPackageFailedView displays the reason and prompts the user to change the preferences by clicking the Change push button (Figure 72). The WtaPackagePurchaseProcessor nonvisual part is called when the user clicks the Change push button. And, as shown in Figure 191, the WtaComponentSelectionView is displayed to let the user select another component.

![Figure 192. WtaPackageFailedView](image)

The WtaPackageFailedView part is the same as WtaPackagePurchaseView, except for the error message for the hotel component. Of course, the last three components could be included in a separate multicell canvas to achieve better reuse. You might implement this improvement as an exercise.

WtaCustomerIdView

When the Web surfer is ready to purchase the package, the WtaCustomerIdView asks for his or her identification as an existing or new customer (Figure 193).
WtaCustomerIdProcessor

The WtaCustomerIdProcessor part (Figure 74) is a router that displays the proper HTML page according to the push button clicked by the user in WtaCustomerIdView (Figure 193). If the Web surfer clicks the Existing push button, the CustomerAuthentication factory creates a WtaCustomerAuthenticationView object to let the Web surfer enter his or her name and identifier. If the Web surfer clicks the New push button, the NewCustomer factory creates a WtaNewCustomerIdView to prompt the Web surfer to enter his or her name, address, and phone number. A custom logic connection controls the routing to either WtaCustomerAuthenticationView or WtaNewCustomerIdView (Figure 195).
if ((target->get("PBExisting")) == "Existing")
    source->iCustomerAuthentication->create();
else
    source->iNewCustomer->create();

Figure 195. Custom Logic Source Code for WtaCustomerIdProcessor

WtaCustomerAuthenticationView

The WtaCustomerAuthenticationView part collects the Web surfer name and identifier and transmit the information to WtaCustomerAuthenticationProcessor for authentication. Notice the three event-to-action connections from the free-form surface which set the HTML name attribute of each entry field (Figure 196).

Figure 196. WtaCustomerAuthenticationView

WtaCustomerAuthenticationProcessor

The WtaCustomerAuthenticationProcessor part (Figure 197) is a router that checks the customer authentication. If the customer authentication is valid, the RouteService factory creates a WtaCreditCardInfoView object to let the Web surfer enter his or her credit card information. If the customer authentication is invalid, the CustomerAuthentication factory creates a WtaCustomerAuthenticationView to prompt the Web surfer to enter a correct authentication. Four actions are defined in the part interface editor to verify the customer authentication (Figure 198).
VisualAge Parts

Figure 197. WtaCustomerAuthenticationProcessor

Figure 198. WtaCustomerAuthenticationProcessor Part Interface Editor
A custom logic connection controls the routing to WtaCreditCardInfoView or WtaCustomerAuthenticationView. Figures 200 through 204 list the source code for the checkInput, QueryDb, processCustomer, and checkCustomer actions.

```cpp
if (target->iErrorMsg->text() == "")
    target->iRouteService->create();
else
    target->iCustomerAuthentication->create();
```

**Figure 199. Custom Logic Source Code**

```cpp
IString WtaCustomerAuthenticationProcessor::checkInput()
{
    IString errorMsg = "";
    IString customerName = iCGI->get("customerName");
    IString customerID   = iCGI->get("customerID");
    if (customerName == "")
        errorMsg = "Please enter a customer name.";
    if (customerID == "")
        if (errorMsg == "")
            errorMsg = "Please enter a customer ID.";
        else
            errorMsg = "Please enter a customer name and a customer ID.";
    return errorMsg;
}
```

**Figure 200. CheckInput Action Source Code**

```cpp
void WtaCustomerAuthenticationProcessor::queryDb()
{
    Customer aCustomer;
    IProfile mySessionData("DATA.INI");
    IString customerName = iCGI->get("customerName").strip();
    IString customerID   = iCGI->get("customerID").strip();
    IFUNCTRACE_DEVELOP();
    mySessionData.setDefaultApplicationName(iSessionKey->text());
    aCustomer.setCustid(customerID.strip());
    try
    {
        aCustomer.retrieve();
    }
    catch (IDAAccessError &exc)
    {
        ITRACE_DEVELOP("Get An exception ");
        mySessionData.addOrReplaceElementWithKey("exception", exc.errorAsString());
    }
```

**Figure 201. QueryDb Action Source Code (Part 1/2)**
if ((exc.errorCode()) == 100)
{
    mySessionData.addOrReplaceElementWithKey("customerId","");
    mySessionData.addOrReplaceElementWithKey("customerName","");
    mySessionData.addOrReplaceElementWithKey("customerAddress","");
    mySessionData.addOrReplaceElementWithKey("customerPhone","");
}

ITRACE_DEVELOP("--> Found the Customer Name = " + aCustomer.cust_name());
IString tempo = IString(aCustomer.cust_name()).strip();
if (tempo == customerName)
{
    ITRACE_DEVELOP("--> Found the Customer!!!!!!");
    IString customerPhone = aCustomer.phone();
    IString customerAddress = aCustomer.address();
    mySessionData.addOrReplaceElementWithKey("customerId", customerID.strip());
    mySessionData.addOrReplaceElementWithKey("customerName", customerName.strip());
    mySessionData.addOrReplaceElementWithKey("customerAddress", customerAddress.strip());
    mySessionData.addOrReplaceElementWithKey("customerPhone", customerPhone.strip());
}
else
{
    mySessionData.addOrReplaceElementWithKey("customerId","");
    mySessionData.addOrReplaceElementWithKey("customerName","");
    mySessionData.addOrReplaceElementWithKey("customerAddress","");
    mySessionData.addOrReplaceElementWithKey("customerPhone","");
}

Figure 202. QueryDb Source Code (Part 2/2)
WtaNewCustomerIdView

When the Web surfer clicks the **New** push button of the *WtaCustomerIdView*, the *WtaCustomerIdProcessor* creates a *WtaNewCustomerIdView* object to prompt the Web surfer for his or her name, address, and phone number. After the user fills out the form and clicks the **Done** push button, the information is stored in the CUSTOMER table (Figure 205).
The Web surfer can also reset the information entered by clicking the **Reset** push button. A push button behaves like a reset push button if you select the *Enable Reset* check box in its settings notebook (Figure 206).

Notice the event-to-action connections from the *ready* event of the freeform surface to the *htmlName* attribute of each entry field. These connections ensure that the name of each entry field is properly generated at runtime for this HTML page.
An IStaticText part is added underneath the two push buttons to receive a potential error message if the Web surfer input is invalid (this IStaticText part is not shown on Figure 205 because its text attribute is set to NULL). The input is checked by WtaNewCustomerIdProcessor (see below). The text attribute of this IStaticText part is promoted to enable WtaNewCustomerIdView to send the error message when redisplaying this view.

WtaNewCustomerIdProcessor

The WtaNewCustomerIdProcessor part (Figure 207) retrieves the user input from WtaNewCustomerIdView, checks the validity of the input and stores the information in the CUSTOMER table.

The UpdateDB action is called at the ready event and returns a string as a potential error message. If the string contains an error message, the NewCustomerId factory creates a WtaNewCustomerIdView object which displays the error message and prompts the user to reenter the information. If the string is NULL, the CreditCardInfo factory creates a WtaCreditCardInfoView object, which prompts the Web surfer to enter his or her credit card information.

```
if (target->iErrorMsg->text() =="")
    target->iCreditCardInfo->create();
else
    target->iNewCustomerId->create();
```

Figure 207. WtaNewCustomerIdProcessor

A custom logic connection checks the string returned by the UpdateDB action and calls the create method on the correct factory (Figure 208).
Four actions are defined in the Part Interface Editor to verify the Web surfer input and store the information in the database. The *UpdateDB* action is the first action called and chains the other actions (Figure 209).

![Figure 209. WtaNewCustomerIdProcessor Part Interface Editor](image)

Figures 210 through 213 list the source code for the *checkInput*, *QueryDb*, *processCustomer*, and *updateDB* actions. Note in the *processCustomer* action (Figure 212) how the session data is updated by calling the *addOrReplaceElementWithKey* function on the DATA.INI profile file.

```cpp
IString WtaNewCustomerIdProcessor::checkInput()
{
    IString errorMsg = "";
    IString customerName = iCGI->get("customer-Name").strip();
    IString customerAddress = iCGI->get("customerAddress").strip();
    IString customerPhone = iCGI->get("customer-Phone").strip();
    if (customerName == "" ||
        customerAddress == "" ||
        customerPhone == ""
    )
        errorMsg = "Please enter your customer name, address and phone number."
    return errorMsg;
}
```

![Figure 210. CheckInput Action Source Code](image)
IString WtaNewCustomerIdProcessor::queryDb()
{
    CustomerManager aCustomerMng;
    Customer aCustomer;
    IString clause = "CUST_NAME LIKE '%' ORDER BY CUSTID DESC";
    IString newId;
    try
    {
        aCustomerMng.select(clause);
        if (!(aCustomerMng.items()->isEmpty()))
        {
            newId = IString(aCustomerMng.items()->firstElement()->custid() + 1);
        }
        else
        {
            newId = "1";
        }
        catch (IDAAccessError &exc)
        {
            ITRACE_DEVELOP("Get An exception " + exc.errorAsString());
        }
    return newId;
    }

Figure 211. QueryDb Action Source Code

void WtaNewCustomerIdProcessor::processCustomer()
{
    IString customerId = queryDb();
    IString customerName = iCGI->get("customerName").strip();
    IString customerAddress = iCGI->get("customerAddress").strip();
    IString customerPhone = iCGI->get("customerPhone").strip();
    IProfile mySessionData("DATA.INI");
    mySessionData.setDefaultApplicationName(iSessionKey->text());
    mySessionData.addOrReplaceElementWithKey("customerId", customerId);
    mySessionData.addOrReplaceElementWithKey("customerName", customerName);
    mySessionData.addOrReplaceElementWithKey("customerAddress", customerAddress);
    mySessionData.addOrReplaceElementWithKey("customerPhone", customerPhone);
}

Figure 212. ProcessCustomer Action Source Code
WtaCreditCardInfoView

The WtaCreditCardInfoView part (Figure 214) enables a user to enter the customer’s credit card information. However, this information is not really required because our sample is not intended to be a commercial application.

Once the user enters the information and clicks the Continue push button, the transaction is committed in the database by the WtaCommitter part (see below).

IString WtaNewCustomerIdProcessor::updateDB()
{
    IString errorMsg = "";
    Customer aCustomer;
    IProfile mySessionData("DATA.INI");
    mySessionData.setDefaultApplicationName(iSessionKey-
>text());
    errorMsg = checkInput();
    if (errorMsg =="")
    {
        processCustomer();
        IString customerId = mySessionData.elementWith-
Key("customerId");
        IString customerName = mySessionData.elementWith-
Key("customerName");
        IString customerAddress = mySessionData.elementWith-
Key("customerAddress");
        IString customerPhone = mySessionData.elementWith-
Key("customerPhone");
        aCustomer.setCustid(customerId);
        aCustomer.setCust_name(customerName);
        aCustomer.setAddress(customerAddress);
        aCustomer.setPhone(customerPhone);
        aCustomer.add();
        iDBConnection->commit();
    }
    return errorMsg;
}

Figure 213. CheckInput Action Source Code
The **WtaCommitter**, a nonvisual part, commits the whole transaction and the session data in the database (Figure 215).

The **WtaCommitter** part retrieves the session key from the ICGI part and saves it in the IVBStringPart SessionKey. Then it calls the CommitPackage-Controller action, which chains all of the other actions defined in the Part Interface Editor (Figure 216). Each action commits a subset of the session data, one for each component and at last for the package itself. When the transaction is committed, an event-to-action connection from the ready event of the free-form surface to the create action of the PackageConfirmation factory creates a WtaPackageConfirmationView, which summarizes what the user has purchased.
Nine actions are defined in the Part Interface Editor of WtaCommitter:

- The `commitPackageController` method commits the whole package and calls the other methods.
- The `generatePackageId` method generates a package identifier for the new package to be stored in the database.
- The `commitPackage` method commits the package information.
- The `commitRequestedHotelRooms` method commits the user preferences regarding the hotel booking.
- The `commitRequestedFlightSeats` method commits the user preferences regarding the flight booking.
- The `commitRequestedCar` method commits the user preferences regarding the car booking.
- The `commitHotel` method updates the HOTEL table for the number of available rooms.
- The `commitFlight` method updates the FLIGHT table for the number of available seats.
- The `commitCar` method updates the CAR table for the number of available cars.

Figures 217 through 224 list the source code for the actions in the WtaCommitter Part Interface Editor.
void WtaCommitter::commitPackageController()
{
    IDatastore aDBConnection;
    IProfile mySessionData("DATA.INI");
    mySessionData.setDefaultApplicationName(iSessionKey-
   >text());
    aDBConnection.setAuthentication("userid");
    aDBConnection.setDatastoreName("WTADB");
    aDBConnection.setUserName("PASSWORD");
    aDBConnection.connect();
    generatePackageId();
    commitPackage();
    if (mySessionData.containsKeyName("hotelName"))
        commitRequestedHotelRooms();
    if (mySessionData.containsKeyName("flightNum"))
        commitRequestedFlightSeats();
    if (mySessionData.containsKeyName("carCompany"))
        commitRequestedCar();
    aDBConnection.commit();
}

Figure 217. CommitPackageController Action Source Code
void WtaCommitter::generatePackageId()
{
    TpackageManager aTpackageMng;
    Tpackage aTpackage;
    IString clause = "LOC_NAME LIKE '%' ORDER BY PACKID DESC";
    IString newId;
    IProfile mySessionData("DATA.INI");
    mySessionData.setDefaultApplicationName(iSessionKey->text());
    try
    {
        aTpackageMng.select(clause);
        if (!(aTpackageMng.items()->isEmpty()))
            {
                newId = IString(aTpackageMng.items()->firstElement()->packid() + 1);
                mySessionData.addOrReplaceElementWithKey("packageId", newId);
            }
        else
            mySessionData.addOrReplaceElementWithKey("packageId", "1");
    } catch (IDAAccessError &exc)
    {
        ITRACE_DEVELOP("Get An exception " + exc.errorAsString());
        mySessionData.addOrReplaceElementWithKey("exception", exc.errorAsString());
        mySessionData.addOrReplaceElementWithKey("packageId", "1");
    }
}

Figure 218. GeneratePackageId Action Source Code
void WtaCommitter::commitPackage()
{
    Tpackage aTpackage;
    IProfile mySessionData("DATA.INI");
    mySessionData.setDefaultApplicationName(iSessionKey->text());
    IString packageId = mySessionData.elementWithKey("packageId");
    IString customerId = mySessionData.elementWithKey("customerId");
    IString locationName = mySessionData.elementWithKey("locationName");
    IString packageTotalPrice = mySessionData.elementWithKey("packageTotalPrice");
    IString startDate = mySessionData.elementWithKey("startDate");
    IString endDate = mySessionData.elementWithKey("endDate");
    IString numberOfDays = mySessionData.elementWithKey("numberOfDays");
    IString fromCity, month, day, year, dbDate;
    if (mySessionData.containsKeyName("fromCity"))
        fromCity = mySessionData.elementWithKey("fromCity");
    else
        fromCity = locationName;
    IString cardCompany = iCGI->get("cardCompany").strip();
    IString cardNumber = iCGI->get("cardNumber").strip();
    IString cardExpirationDate = iCGI->get("cardExpirationDate").strip();
    mySessionData.addOrReplaceElementWithKey("cardCompany", cardCompany);
    mySessionData.addOrReplaceElementWithKey("cardNumber", cardNumber);
}

Figure 219. CommitPackage Action Source Code (Part 1/2)
mySessionData.addOrReplaceElementWithKey("cardExpirationDate", cardExpirationDate);
ATpackage.setPackid(packageId);
ATpackage.setCustid(customerId);
ATpackage.setLoc_name(locationName);
ATpackage.setFrom_city(fromCity);
ATpackage.setPrice(packageTotalPrice);
startDate >> month >> "/" >> day >> "/" >> year;
dbDate = month + "/" + day + "/19" + year;
ATpackage.setStart_date(dbDate);
endDate >> month >> "/" >> day >> "/" >> year;
dbDate = month + "/" + day + "/19" + year;
ATpackage.setEnd_date(dbDate);
ATpackage.setDays(numberOfDays);
ATpackage.setCard_company(cardCompany);
ATpackage.setCard_nb(cardNumber);
ATpackage.setExp_date("10/22/1994");
IFUNCTRACE_DEVELOP();
ITRACE_DEVELOP(" ----> Add!!!!!!!");
ATpackage.add();
}

**Figure 220. CommitPackage Action Source Code (Part 2/2)**
void WtaCommitter::commitRequestedHotelRooms()
{
    aBrooms;
    IProfile mySessionData("DATA.INI");
    mySessionData.setDefaultApplicationName(iSessionKey->text());
    IString packageId = mySessionData.elementWithKey("packageId");
    IString hotelId = mySessionData.elementWithKey("hotelId");
    IString roomCategory = mySessionData.elementWithKey("roomCategory");
    IString numberOfNights = mySessionData.elementWithKey("numberOfNights");
    IString numberOfRooms = mySessionData.elementWithKey("numberOfRooms");
    IString roomSmoking = mySessionData.elementWithKey("roomSmoking");
    IString packageHotelPrice = mySessionData.elementWithKey("packageHotelPrice");
    aBrooms.setPackid(packageId);
    aBrooms.setHotelid(hotelId);
    aBrooms.setRoom_cat(roomCategory);
    aBrooms.setNights(numberOfNights);
    aBrooms.setRooms(numberOfRooms);
    aBrooms.setSmoking(roomSmoking);
    aBrooms.setPrice(packageHotelPrice);
    aBrooms.add();
}

Figure 221. CommitRequestedHotelRooms Action Source Code
void WtaCommitter::commitRequestedFlightSeats()
{
    Bseats aBseats;
    IProfile mySessionData("DATA.INI");
    mySessionData.setDefaultApplicationName(iSessionKey->text());
    IString packageId = mySessionData.elementWithKey("packageId");
    IString flightNum = mySessionData.elementWithKey("flightNum");
    IString seatClass = mySessionData.elementWithKey("seatClass");
    IString numberOfSeats = mySessionData.elementWithKey("numberOfSeats");
    IString packageFlightPrice = mySessionData.elementWithKey("packageFlightPrice");
    aBseats.setPackid(packageId);
    aBseats.setFlightnum(flightNum);
    aBseats.setSeat_class(seatClass);
    aBseats.setSeats(numberOfSeats);
    aBseats.setSmoking("NO");
    aBseats.setPrice(packageFlightPrice);
    aBseats.add();
}

Figure 222. CommitRequestedFlightSeats Action Source Code

void WtaCommitter::commitRequestedCar()
{
    Bcar aBcar;
    IProfile mySessionData("DATA.INI");
    mySessionData.setDefaultApplicationName(iSessionKey->text());
    IString packageId = mySessionData.elementWithKey("packageId");
    IString carClass = mySessionData.elementWithKey("carClass");
    IString carCompany = mySessionData.elementWithKey("carCompany");
    IString packageCarPrice = mySessionData.elementWithKey("packageCarPrice");
    aBcar.setPackid(packageId);
    aBcar.setCar_class(carClass);
    aBcar.setCar_company(carCompany);
    aBcar.setPrice(packageCarPrice);
    aBcar.add();
}

Figure 223. CommitRequestedCar Action Source Code
void WtaCommitter::commitHotel()
{
    // Would be used to update the number of rooms in this hotel
}
void WtaCommitter::commitFlight()
{
    // Would be used to update the number of seats in this flight
}
void WtaCommitter::commitCar()
{
    // Would be used to update the number of car for the rental company
}

Figure 224. CommitHotel, CommitFlight, and CommitCar Actions Source Code

The execution order of the connections is shown in Figure 225.

Figure 225. Connection Order

WtaPackageConfirmationView

The WtaPackageConfirmationView part (Figure 226) is similar to WtaPackagePurchaseView except that a FORM is not defined in the corresponding page. The Web surfer must click the Home anchor to get back to the Web Travel Agent home page.

The actions defined for WtaPackagePurchaseView are also defined for this view and are not reproduced here.
The **WtaPackageListView** part lists all packages of a user of the Web Travel Agent application who has been authenticated (Figure 227). The SQL clause is built up from the customer identifier retrieved from the session data. A custom logic connection (Figure 228) build up the clause that is transmitted to the `select` action of the `TPackageManager` part.

**Figure 226. WtaPackageConfirmationView**

**WtaPackageListView**

The **WtaPackageListView** part lists all packages of a user of the Web Travel Agent application who has been authenticated (Figure 227). The SQL clause is built up from the customer identifier retrieved from the session data. A custom logic connection (Figure 228) build up the clause that is transmitted to the `select` action of the `TPackageManager` part.

**Figure 227. WtaPackageListView**
target->setText("CUSTID = " +
    source->iSessionData->
    elementWithKey("customerId"));

Figure 228. Custom Logic Source Code of WtaPackageListView

WtaPackageListProcessor

Once the Web surfer selects a package in the ICollectionViewListBox of WtaPackageListView, he or she can click the Details push button to obtain from WtaPackageListProcessor detailed information about the different package components (Figure 229).

This nonvisual part is similar to WtaCommitter. The package identifier from the selected package is retrieved from WtaPackageListView by using the ICGI part. Then the processPackageListForm action is fired to retrieve from the session data all the information for each package component.

Figure 229. WtaPackageListProcessor

Seven actions are defined in the WtaPackageListProcessor Part Interface Editor to retrieve the information from the session data (Figure 230). The processPackageListForm chains all of these actions.
Figure 230. WtaPackageListProcessor Part Interface Editor

The connections drawn from the ready event of the free-form surface must be executed in a proper order as shown in Figure 231.

Figure 231. Connection Order for WtaPackageListProcessor

In the following figures we detail the source code of each action.
void WtaPackageListProcessor::processPackageListForm() {
  IProfile mySessionData("DATA.INI");
  IString packid, locationName, startDate, endDate;
  IString selectedItem = iCGI->get("PackageList");
  reinitializeRequestedPackage();
  mySessionData.setDefaultApplicationName(iSessionKey->
>text());
  ITRACE_DEVELOP();
  ITRACE_DEVELOP("----> Item = " + selectedItem);
  selectedItem >> packid >> locationName >> startDate >>
>endDate;
  ITRACE_DEVELOP("----> Packid    = " + packid);
  ITRACE_DEVELOP("----> Location  = " + locationName);
  ITRACE_DEVELOP("----> startDate = " + startDate);
  ITRACE_DEVELOP("----> endDate   = " + endDate);
  mySessionData.addOrReplaceElementWithKey("pack-
>ageId", packid);
  pumpUpRequestedPackage();
  pumpUpRequestedHotelRooms();
  if (mySessionData.containsKeyName("hotelId"))
    pumpUpHotel();
  pumpUpRequestedFlightSeats();
  if (mySessionData.containsKeyName("flightNum"))
    pumpUpFlight();
  pumpUpRequestedCar();
}

Figure 232. ProcessPackageListForm Action Source Code

void WtaPackageListProcessor::reinitializeRequestedPack-
>age() {
  IProfile mySessionData("DATA.INI");
  mySessionData.deleteElementWithApplication(iSessionKey->
>text());
  mySessionData.setDefaultApplicationName(iSessionKey->
>text());
  mySessionData.addOrReplaceElementWithKey("requested Ser-
>vice", "Display Package");
}

Figure 233. ProcessPackageListForm Action Source Code
void WtaPackageListProcessor::pumpUpRequestedPackage()
{
    TpackageManager aTpackageMng;
    Tpackage* aTpackage;
    IString clause,
        packid,
        locationName,
        fromCity,
        packageTotalPrice,
        startDate,
        endDate,
        numberOfDays,
        cardCompany,
        cardNum,
        cardExpDate;
    IProfile mySessionData("DATA.INI");
    IFUNCTRACE_DEVELOP();
    mySessionData.setDefaultApplicationName(iSessionKey-
>text());
    packid = mySessionData.elementWithKey("packageId");
    clause = "PACKID = " + packid;
    try
    {
        aTpackageMng.select(clause);
        if (!(aTpackageMng.items()->isEmpty()))
        {
            aTpackage = aTpackageMng.items()->firstElement();
            locationName = aTpackage->loc_name();
            ITRACE_DEVELOP("----> location = "+locationName);  
            fromCity = aTpackage->from_city();
            packageTotalPrice = aTpackage->priceAsString();
            startDate = aTpackage->start_dateAsString();
            endDate = aTpackage->end_dateAsString();
            numberOfDays = aTpackage->daysAsString();
            cardCompany = aTpackage->card_company();
            cardNum = aTpackage->card_nbAsString();
            cardExpDate = aTpackage->exp_dateAsString();
            mySessionData.addOrReplaceElementWithKey("locationName", locationName);
                mySessionData.addOrReplaceElementWithKey("fromCity", fromCity);
        mySessionData.addOrReplaceElementWithKey("packageTotalPrice", packageTotalPrice);
            mySessionData.addOrReplaceElementWithKey("startDate", startDate);
            mySessionData.addOrReplaceElementWithKey("endDate", endDate);
        }
    }

Figure 234. ProcessPackageListForm Action Source Code (1/2)
mySessionData.addOrReplaceElementWithKey("numberOfDays", numberOfDays);
   mySessionData.addOrReplaceElementWithKey("cardCompany", cardCompany);
   mySessionData.addOrReplaceElementWithKey("cardNum", cardNum);
   mySessionData.addOrReplaceElementWithKey("cardExpDate", cardExpDate);
}  
else
   ITRACE_DEVELOP("----> Empty list!");
}
catch (IDAAccessError &exc)
{
   ITRACE_DEVELOP("Get An exception " + exc.errorAsString());
   mySessionData.addOrReplaceElementWithKey("exception", exc.errorAsString());
}
}

Figure 235. ProcessPackageListForm Action Source Code (2/2)

void WtaPackageListProcessor::pumpUpRequestedHotelRooms()
{
   BroomsManager aBroomsMng;
   Brooms*       aBrooms;
   IString clause,
      packid,
      hotelId,
      roomCategory,
      numberOfNights,
      numberOfRooms,
      smoking,
      packageHotelPrice;
   IProfile mySessionData("DATA.INI");
   IFUNCTRACE_DEVELOP();
   mySessionData.setDefaultApplicationName(iSessionKey->text());
   packid = mySessionData.elementWithKey("packageId");
   clause = "PACKID = " + packid;
   try
   {
      aBroomsMng.select(clause);
      if (!(aBroomsMng.items()->isEmpty()))
      {

Figure 236. PumpUpRequestedHotelRooms Action Source Code (1/2)
aBrooms = aBroomsMng.items()->firstElement();
hotelId = aBrooms->hotelidAsString().strip();
roomCategory = aBrooms->room_catAsString().strip();
numberOfNights = aBrooms->nightAsString().strip();
numberOfRooms = aBrooms->roomsAsString().strip();
smoking = (aBrooms->smoking() == 0) ? "smoking" : "nonsmoking";
packageHotelPrice = aBrooms->priceAsString().strip();
mySessionData.addOrReplaceElementWithKey("hotelId", hotelId);
mySessionData.addOrReplaceElementWithKey("roomCategory", roomCategory);
mySessionData.addOrReplaceElementWithKey("numberOfNights", numberOfNights);
mySessionData.addOrReplaceElementWithKey("numberOfRooms", numberOfRooms);
mySessionData.addOrReplaceElementWithKey("roomSmoking", smoking);
mySessionData.addOrReplaceElementWithKey("packageHotelPrice", packageHotelPrice);
} else
  ITRACE_DEVELOP("----> Empty list!");
} catch (IDAAccessError &exc) {
  ITRACE_DEVELOP("Get An exception "+ exc.errorAsString());
  mySessionData.addOrReplaceElementWithKey("exception", exc.errorAsString());
}
void WtaPackageListProcessor::pumpUpRequestedFlightSeats()
{
    BseatsManager aBseatsMng;
    Bseats*       aBseats;
    IString clause,
    packid,
    flightNum,
    seatClass,
    numberOfSeats,
    packageFlightPrice;
    IProfile mySessionData("DATA.INI");
    IFUNCTRACE_DEVELOP();
    mySessionData.setDefaultApplicationName(iSessionKey-
>text());
    packid = mySessionData.elementWithKey("packageId");
    clause = "PACKID = " + packid;
    ITRACE_DEVELOP("Clause = "+clause);
    try {
        aBseatsMng.select(clause);
        if (!(aBseatsMng.items()->isEmpty()))
        {
            aBseats = aBseatsMng.items()->firstElement();
            flightNum = aBseats->flightnumAsString().strip();
            seatClass = aBseats->seat_classAsString().strip();
            numberOfSeats = aBseats->seatsAsString().strip();
            packageFlightPrice = aBseats->priceAsString().strip();
            mySessionData.addOrReplaceElementWithKey("flight-
Num", flightNum);
            mySessionData.addOrReplaceElementWithKey("seat-
Class", seatClass);
            mySessionData.addOrReplaceElementWithKey("number-
OfSeats", numberOfSeats);
            mySessionData.addOrReplaceElementWithKey("pack-
ageFlightPrice", packageFlightPrice);
        }
        else
            ITRACE_DEVELOP("----> Empty list!");
    }
    catch (IDAAccessError &exc)
    {
        ITRACE_DEVELOP("Get An exception "+ exc.erro-
rAsString());
        mySessionData.addOrReplaceElementWithKey("excep-
tion", exc.errorAsString());
    }
}

Figure 238. PumpUpRequestedFlightSeats Action Source Code
void WtaPackageListProcessor::pumpUpRequestedCar()
{
    BcarManager aBcarMng;
    Bcar* aBcar;
    IString clause,
    packid,
    carCompany,
    carClass,
    packageCarPrice;
    IProfile mySessionData("DATA.INI");
    IFUNCTRACE_DEVELOP();
    mySessionData.setDefaultApplicationName(iSessionKey->text());
    packid = mySessionData.elementWithKey("packageId");
    clause = "PACKID = " + packid;
    ITRACE_DEVELOP("Clause = " + clause);
    try
    {
        aBcarMng.select(clause);
        if (!(aBcarMng.items()->isEmpty()))
        {
            aBcar = aBcarMng.items()->firstElement();
            carCompany = aBcar->car_companyAsString().strip();
            carClass = aBcar->car_classAsString().strip();
            packageCarPrice = aBcar->priceAsString().strip();
            mySessionData.addOrReplaceElementWithKey("carCompany", carCompany);
            mySessionData.addOrReplaceElementWithKey("carClass", carClass);
            mySessionData.addOrReplaceElementWithKey("packageCarPrice", packageCarPrice);
        }
        else
        {
            ITRACE_DEVELOP("----> Empty list!");
        }
    }
    catch (IDAAccessError &exc)
    {
        ITRACE_DEVELOP("Get An exception " + exc.errorAsString());
        mySessionData.addOrReplaceElementWithKey("exception", exc.errorAsString());
    }
}

Figure 239. PumpUpRequestedCar Action Source Code
void WtaPackageListProcessor::pumpUpHotel()
{
    HotelManager  aHotelMng;
    Hotel*        aHotel;
    IString clause,
                  hotelName,
                  hotelId,
                  packageHotelPrice;
    IProfile mySessionData("DATA.INI");
    IFUNCTRACE_DEVELOP();
    mySessionData.setDefaultApplicationName(iSessionKey->text());
    hotelId = mySessionData.elementWithKey("hotelId");
    clause = "HOTELID = " + hotelId;
    try
    {
        aHotelMng.select(clause);
        if (!(aHotelMng.items()->isEmpty()))
        {
            aHotel = aHotelMng.items()->firstElement();
            hotelName = aHotel->hotel_nameAsString().strip();
            mySessionData.addOrReplaceElementWithKey("hotel-
Name", hotelName);
        }
        else
            ITRACE_DEVELOP("----> Empty list!");
    }
    catch (IDAAccessError &exc)
    {
        ITRACE_DEVELOP("Get An exception " + exc.errorAsString());
        mySessionData.addOrReplaceElementWithKey("excep-
tion", exc.errorAsString());
    }
}

Figure 240. PumpUpHotel Action Source Code
void WtaPackageListProcessor::pumpUpFlight()
{
    FlightManager  aFlightMng;
    Flight*        aFlight;
    IString clause,
    flightNum,
    fromCity,
    toCity,
    basePrice;
    IProfile mySessionData("DATA.INI");
    IFUNCTRACE_DEVELOP();
    mySessionData.setDefaultApplicationName(iSessionKey->text());
    flightNum = mySessionData.elementWithKey("flightNum");
    clause = "FLIGHTNUM = "+ flightNum + ";";
    try
    {
        aFlightMng.select(clause);
        if (!(aFlightMng.items()->isEmpty()))
        {
            aFlight = aFlightMng.items()->firstElement();
            toCity = aFlight->to_cityAsString().strip();
            fromCity = aFlight->from_cityAsString().strip();
            toCity = aFlight->base_priceAsString().strip();
            mySessionData.addOrReplaceElementWithKey("toCity",
            toCity);
            mySessionData.addOrReplaceElementWithKey("from-
            City", fromCity);
            mySessionData.addOrReplaceElementWithKey("base-
            Price", basePrice);
        }
        else
            ITRACE_DEVELOP("----> Empty list!");
    }
    catch (IDAAccessError &exc)
    {
        ITRACE_DEVELOP("Get An exception "+ exc.errorAsString());
        mySessionData.addOrReplaceElementWithKey("excep-
        tion", exc.errorAsString());
    }
}

Figure 241. PumpUpFlight Action Source Code
WtaPackageDetailsView

The WtaPackageDetailsView part lists each component of the package, similarly to WtaPackageConfirmationView. A Return push button allows the Web surfer to reaccess WtaPackageListView (Figure 242) to choose another package.

![WtaPackageDetailsView](image)

Figure 242. WtaPackageDetailsView

WtaPersonalInfoView

The WtaPersonalInfoView (Figure 243) allows the Web surfer to change his or her personal information. The Change push button commits the changes in the database, the Done push button displays WtaWelcomeView, and the Reset push button resets the personal information to the default value.
WtaPersonalInfoProcessor

The WtaPersonalInfoProcessor part (Figure 244) is similar to the other processor parts you have built. Depending on the push button clicked in WtaPersonalInfoView, the CheckInput action is called. It chains processCustomer and updateDb to store the personal information changes in the database.
You define the `checkInput`, `processCustomer` and `updateDb` actions in the `WtaPersonalInfoProcessor` Part Interface Editor (Figure 244) and update their code in the hpv and cpv files you provide for this purpose. Figures 246 through 248 list the source code for each action.

**Figure 245. Part Interface Editor**

```c++
IString WtaPersonalInfoProcessor::checkInput()
{
    ICGI aCGI;
    IString aAddress, aPhone;
    IString errorMsg = "";
    IFUNCTRACE_DEVELOP();
    ITRACE_DEVELOP("In checkinput");
    aAddress = aCGI.get("customerAddress").strip();
    aPhone = aCGI.get("customerPhone").strip();
    if (aAddress == "")
        errorMsg = "A blank address was entered."
    if (aPhone == "")
        errorMsg = "A blank phone number was entered."
    if (errorMsg == "")
        processCustomer();
    return errorMsg;
}

**Figure 246. CheckInput Source Code**
VisualAge Parts

```cpp
void WtaPersonalInfoProcessor::processCustomer()
{
    IProfile mySessionData("DATA.INI");
    I/cgi a CGI;
    IString customerAddress, customerPhone;
    IFUNCTRACE_DEVELOP();
    ITRACE_DEVELOP("In processCustomer");
    mySessionData.setDefaultApplicationName(iSessionKey-
&text());
    customerAddress = a CGI.get("customerAddress").strip();
    customerPhone = a CGI.get("customerPhone").strip();
    mySessionData.addOrReplaceElementWithKey("customerAd-
dress", customerAddress);
    mySessionData.addOrReplaceElementWithKey("customerPhone",
customerPhone);
    updateDb();
}

Figure 247. ProcessCustomer Source Code
In the last section of this chapter we detail how to create the database parts which you need to access the various relational tables.

```cpp
void WtaPersonalInfoProcessor::updateDb()
{
    Customer aCustomer;
    IString customerId, customerName, customerAddress, customerPhone;
    IDatastore dbConnection;
    IProfile mySessionData("DATA.INI");
    mySessionData.setDefaultApplicationName(iSessionKey->text());
    dbConnection.setDatastoreName("WTADB");
    dbConnection.connect();
    IFUNCTRACE_DEVELOP();
    ITRACE_DEVELOP("In updateDb");
    customerId      = mySessionData.elementWithKey("customerId");
    customerName    = mySessionData.elementWithKey("customerName");
    customerAddress = mySessionData.elementWithKey("customerAddress");
    customerPhone   = mySessionData.elementWithKey("customerPhone");
    aCustomer.setCustid(customerId);
    aCustomer.setCust_name(customerName);
    aCustomer.setAddress(customerAddress);
    aCustomer.setPhone(customerPhone);
    ITRACE_DEVELOP("About to Update!!!!");
    try
    {
        aCustomer.update();
    }
    catch (IDAAccessError &exc)
    {
        ITRACE_DEVELOP("Get An exception " + exc.errorAsString());
        mySessionData.addOrReplaceElementWithKey("exception",exc.errorAsString());
        if ( exc.errorProvided())
        {
            mySessionData.addOrReplaceElementWithKey("exception",exc.errorAsString());
        }
    }
    dbConnection.commit();
}
```

**Figure 248. UpdateDb Source Code**
Database Parts

You use Data Access Builder which comes with VisualAge for C++ for Windows, to build your database parts. Data Access Builder maps the relational tables to parts that you reuse in the Composition Editor and connect to other parts. We detail in the following sections how to use Data Access Builder the generate these parts.

Mapping the Relational Tables to Parts

Start Data Access Builder by double clicking its icon in the VisualAge for C++ folder. Data Access Builder displays the startup window. To create a mapping from scratch, click on the **Create Classes...** push button. The Create Classes window prompts you to select the database type which defines the data access method. Select DB2 Version 2.1 as the database type. Data Access Builder accesses your database directory and presents a list of all database names cataloged on your machine.

Select the WTADB database, the database owner (usually the database is created by the local administrator USERID), the **TABLE, VIEW** type, and click on **Connect**.

![Create Classes Window](image)

Figure 249. Create Classes Window

Select the WTADB database, the database owner (usually the database is created by the local administrator USERID), the **TABLE, VIEW** type, and click on **Connect**.
When you click on **Connect**, Data Access Builder tries to connect to the selected database. If DB2 Database Manager is not already running, it is started. If you are not logged on, you are asked to give the user ID and password of the database creator (in our case, the user ID is `userid`, and the password is `password`).

Once the database connection is established, Data Access Builder lists the tables and views, prefixed with `USERID`. Click on each table or view you want to map to a part. In our case, select only the tables and views related to the Property subsystem:

- `USERID.BCAR`
- `USERID.BROOMS`
- `USERID.BSEATS`
- `USERID.CAR`
- `USERID.CUSTOMER`
- `USERID.FLIGHT`
- `USERID.HOTEL`
- `USERID.LOCATION`
- `USERID.TPACKAGE`

Click on **Create classes**. Data Access Builder prompts you for the class options. By default, Data Access Builder generates C++ Visual Builder parts and accesses the database with the embedded SQL method. Click on **OK** to confirm these options and to get to the primary window, which shows the mapping of each table on the free-form surface (Figure 250).
To access the settings window of each mapped class, click on it with the right mouse button and select **Open settings** from the pop-up menu or, as a shortcut, double-click with the left mouse button (Figure 251).

Before you generate the code for each part, you can check the associated file names in the Names page. The files generated for each part depend on your class options and whether you chose to generate a make file. In our case the following files are generated:

- *.hpp, header for the parts
- *.sqx, part code with embedded SQL
- *.vbe, import file for Visual Builder

Since the table names are 8 or less character long, you can use the default file stem proposed by Data Access Builder. The following parts will be generated for the respective table:

- BCar for BCAR
- BRooms for BROOMS
- BSeats for BSEATS
- Car for CAR
- Customer for CUSTOMER
- Flight for FLIGHT
- Hotel for HOTEL
- Location for LOCATION
- TPackage for TPACKAGE
Make sure you change the name of the Location part to a different name. In effect, this name conflicts with a Location enum defined already in the Open Class Library for specifying the locations for a frame window extension. We decide to change the name to CityLocation.

Notice that the Generate MAKEFILE check box is checked and a make file will be generated when you generate the parts. This make file is used to build each dynamic link library (one per table) to access the database.

Notice also that the Automate link to application option is enabled for each part. The Data Access Builder puts a #pragma library statement with the given library name in the generated .hpp file. Then, the linker automatically looks for this library and, if it is not found, produces a link error.

![Image of Customer Part Settings]

**Figure 251. Customer Part Settings**

When you use Data Access Builder to generate one dynamic link library for each table mapping, make sure that the file names are unique for each DLL generated. In our example, such a problem does not exist but it is always a good idea to double check. In addition, the maximum length for a database package is eight characters in DB2 2.1. Thus, you should avoid file names with more than eight characters.

The Data identifier (Figure 252) is used to identify a row. Before the update, delete, and retrieve operations, the unique values of the data identifiers must be set to locate the row in the relational table. When the Data identifier check box of an attribute is selected, that attribute is used to identify a row in the table. By default, each primary key is an identifier. If the table does not have a primary key, the first attribute is selected as the default data identifier. You must ensure that the attribute contains unique
values. If the values in a data identifier identify more than one row, errors occur during the retrieve operation, and multiple rows are affected during the update and delete operations.

Figure 252. HOTEL Part Settings

Table 13 lists the identifiers for each relational table.

<table>
<thead>
<tr>
<th>Table</th>
<th>Identifier</th>
</tr>
</thead>
<tbody>
<tr>
<td>BCAR</td>
<td>PACKID</td>
</tr>
<tr>
<td>BROOMS</td>
<td>PACKID</td>
</tr>
<tr>
<td>BSEATS</td>
<td>PACKID</td>
</tr>
<tr>
<td>CAR</td>
<td>MAKE</td>
</tr>
<tr>
<td>CUSTOMER</td>
<td>CUSTID</td>
</tr>
<tr>
<td>FLIGHT</td>
<td>FLIGHTNUM</td>
</tr>
<tr>
<td>HOTEL</td>
<td>HOTELID</td>
</tr>
<tr>
<td>LOCATION</td>
<td>LOC_NAME</td>
</tr>
<tr>
<td>TPACKAGE</td>
<td>PACKID</td>
</tr>
</tbody>
</table>
The *Is Displayed* check box (Figure 252) allows you to specify whether the selected attribute is included as part of the returned value from the *forDisplay* method. *forDisplay* returns an *IString* containing all the class attributes that have their *Is Displayed* check box marked, in the format specified by the *asString* method.

Because the Web Travel Agent application displays the hotels by their names only, you must uncheck the *Is Displayed* check box for all the attributes of the *Hotel* part except the HOTEL_NAME attribute (Figure 252). Similarly, you must uncheck the *Is Displayed* check box for all attributes of the *CityLocation* part except the LOC_NAME attribute.

Access the pop-up menu of each part (right-click with your mouse on each part) and select the option *Generate* to generate the source code and the corresponding make file.

When each part has been generated, open a DOS session and change to the directory where you generated the source code. Compile and link-edit the code to produce one dynamic library for each relational table by issuing, for each generated make file, the command: `nmake makefile`. Make sure to start database manager when you run this command since the SQL preprocessor requires it.

### Parts Produced

When you map a table, for example, *T*, to a part, several parts are generated. By default, their names start with the capitalized name of the table:

- *T*, a persistent object class that represents a row in the table
- *TManager*, a persistent object manager class that represents a collection of persistent objects *T*
- *TDataId*, a class that contains one or multiple columns that uniquely identify a persistent object
- *TDatastore*, a class that represents the datastore that *T* belongs to
- *TManagerTemplate*, a persistent object manager class that you use to represent a collection of persistent objects derived from *T*
- *TManagerBase*, an abstract base class for the persistent object manager classes

You use only the *T* and *TManager* in the application. The others are needed for more sophisticated tasks. The part *T* is derived from the *IPersistentObject* class and represents a row of the *T* table. Using the *T* part, you can access the information of the table, because each column is mapped to a corresponding part attribute (Data Access Builder handles type conversion between the data types defined in the database and C++). Data Access Builder generates methods to get and set the value of each attribute as well as check or set the attributes to NULL. For each
attribute, there is also an `AttributeAsString` method that returns an `IString` representation. In addition, the `T` part supports the actions you usually apply on a table row: `add`, `delete`, `update`, and `retrieve`.

Before using these methods, you must indicate the attribute you will use to retrieve the entire row (see Figure 251 and Table 13) and then check the corresponding `Data identifier` check box. You can select several attributes as data identifiers, to identify the row by a combination of these attributes.

The `TManager` part is derived from `IPOManager` and accesses multiple rows of data. Using this part, you can retrieve several rows of the table. Use the `Refresh` method to retrieve all rows of the table, and use the `Select` method to retrieve a selected set of rows according to an SQL clause.

The rows are maintained through an attribute of type `IVSequence<T*>`, called `iItems`. As you will see in the section below, the `iItems` attribute is used through attribute-to-attribute connections with other visual parts, such as a container or a list box, to display the contents of a set of rows.

In the current release of Data Access Builder, you must use only the Select method of the manager part to limit the number of rows that are read from a table and added to the `IVSequence` of the `Manager` part.

All database access is executed through the exception handler framework. In this way, exceptions are thrown by the parts whenever an error occurs, and your application can catch the exceptions to react accordingly. You can refer to the action `updateDb` (Figure 248) as an example of catching a database update exception.
Implementation Considerations

This chapter describes various issues to consider during the development of a VisualAge for Smalltalk Web Connection application. It ranges from naming conventions to reuse opportunities.

Part Naming

When you create Web Connection user interface parts on an HTML page, such as buttons, check boxes, entry fields and so on, it is a good idea to name the parts according to their meaning in the application (see Figure 253). Instead of the default names generated by VisualAge, such as

- HTML Push Button1
- HTML Entry Field1
- HTML Check Box1
you should use such application-related names as

- Purchase Button
- Address Entry Field
- Nonsmoking Check Box.

The effort you invest in providing more descriptive names pays off when you start referencing those components on the HTML form of another part. You will instantly know which form element you are referring to, when, for example, you are tearing off attributes from a **HTML Form Data** part, or connecting form data events to **HTML Page Wrapper** parts. Figure 254 shows the settings of an **HTML Form Data** part that refers to another HTML page, where we named the subparts as shown.

If you now need to connect to an event that happened on the form of the previous page, the pop-up menu of the form data part shows event names that you can easily identify (Figure 255).
The process of developing the Web Travel Agent application taught us that there are different types of Web Connection parts to be considered. As in an ordinary application that runs on the OS/2 Presentation Manager or on Windows, you will certainly have visual and nonvisual parts. You can relate visual parts to Web Connection parts that generate some HTML through the use of an HTML Page part and its graphical subparts. Nonvisual parts in a Web application are certainly usable for your business objects (in the case of the Web Travel Agent application, for example, Hotel, Flight, or Package).

User interactions with a Web application, however, are handled somewhat differently than the ordinary application. Usually, if you press a button on a window, control is transferred to the application immediately and the processing continues. When you use Web parts to build your application, once you generate the HTML and send it out through the CGI Link server, the application is completely detached from the user interface, that is, the Web browser. For example, clicking on a button on an HTML form, cannot trigger direct events in the application. Even the HTML-generating Web part does not stay around to monitor any user interactions. As soon as the HTML is generated, the part’s instance goes away.

The structure we recommend to implement the various types of Web Connection parts is shown in Figure 256. There are four distinct types of parts used in the Web Travel Agency application:

- Controller parts
- View parts
- Router parts
- Processor parts
Controller parts do processing that is required before you actually consider generating HTML. Typically, you would put business logic and database access here.

View parts contain HTML pages and represent the user interface of the application. They might also have some GUI-related scripts that run before the HTML is actually generated.

Router parts interpret form data (buttons) and transfer the request to other view parts. They can entirely be programmed visually by using Form Data and Page Wrapper parts (see Figure 257).

Processor parts also eventually route to other parts, but do some post-processing on the input the part gets from the form data. For example, they may store entry-field data, check box status, or radio button selections in an HTML Session Data part for future requests. Processor parts can also do data verification on user input and, in the case of an error, return to the view part.

Figure 256. Part Types and their Relationships

Controller parts do processing that is required before you actually consider generating HTML. Typically, you would put business logic and database access here.

View parts contain HTML pages and represent the user interface of the application. They might also have some GUI-related scripts that run before the HTML is actually generated.

Router parts interpret form data (buttons) and transfer the request to other view parts. They can entirely be programmed visually by using Form Data and Page Wrapper parts (see Figure 257).

Processor parts also eventually route to other parts, but do some post-processing on the input the part gets from the form data. For example, they may store entry-field data, check box status, or radio button selections in an HTML Session Data part for future requests. Processor parts can also do data verification on user input and, in the case of an error, return to the view part.
Reuse of Web Parts

In this section, we explain how to create reusable Web parts from visual Web parts. There is basically no difference between reusing parts that are developed for a GUI and those developed for HTML-generating Web parts. However, because reuse is an important feature of object-oriented programming, we cover it in detail.

Build Composite

The easiest way to create a reusable part is by developing a Web part based on a composite. The Web Connection category on the Composition Editor contains an HTML Composite icon that we can use to assemble the visual (HTML) parts we later want to reuse. We start with creating a new part (choose type Web Connection). When the Composition Editor appears, we delete the default HTML page from the free-form surface and replace it with an HTML Composite (Figure 258).

Now we can start to build the reusable part. For our example, we use the TPACKAGE table from the Web Travel Agent database. We want to execute an SQL query that generates an HTML table through the use of the Quick HTML option. The table is to be used from another part. We promote attributes from the query’s result table to make the reusable part more flexible by allowing the table to be customized. Let’s start from the beginning.
Reuse of Web Parts

From the Composition Editor’s Database category, drop a *Multiple Row Query* part on the free-form surface and call it *Package Query*. Double-click on the icon to open the settings. We prefabricated a query that counts the number of packages per travel location, so creating a hit list of destinations. Figure 259 shows the query settings.

![Figure 258. HTML Composite](image1)

![Figure 259. Settings: Query Specifications](image2)
We want the result table to contain the city and the number of visits as columns (Figure 260), with the location most often travelled to at the top. That is why we select the LOC_NAME column from the table and the computed column COUNT(*)

For a correct calculation of the COUNT clause, we need to group the output by location. We add the clause GROUP BY TPACKAGE.LOC_NAME to the SQL statement. Finally, we define the sort order we want, in our case it is by descending values in the computed COUNT(*) column.

Because the VisualAge SQL generator allows entering column names only in the ORDER BY clause, we edited the SQL statement manually (Figure 261). The term COUNT(*) is not valid in an ORDER BY clause, so we use the sort column’s number in the result table: ORDER BY 2 DESC. Once you change the SQL statement manually, you cannot edit it again in the VisualAge SQL generator. The Edit button on the settings (Figure 259) will consequently be disabled.
Now click on the **Quick HTML** option from the *Package Query* part’s pop-up menu. Select the *resultTable* attribute from the list shown. A torn-off subpart appears on the free-form surface and the cursor is loaded with a *Table* part. Drop the *Table* part on the *Composite* part and adjust the columns.

The table size and the column widths of the table on the composite have no influence on how it will look on the Web browser. For cosmetic reasons, we change the headings of the two columns to *City* and *Visits*, and rename the part as *LocationHitListTable*, otherwise we leave the table unchanged.

The reusable part does not allow encapsulating the execution of the query. That is why we have to promote the *executeQuery* action of the *Package Query* part. Later, in a visual Web Connection part we can then connect to the encapsulated query.

From the query’s pop-up menu, select Promote Part Feature. From the feature list, select the *executeQuery* action. Instead of using the system-generated promote feature name of *packageQueryExecuteQuery*, we renamed the promote feature name to *packageQueryExecute* to better reflect the action.

---

**No Quick HTML?**

If your pop-up menu on the query does not show the Quick HTML option you have not executed the file-in as written in the README.TXT file in the `\VISUALAG` directory of VisualAge version 3.0a.
Because we want to enable another user of the reusable query part to customize, for example, the HTML border width and the cell padding of the table, we also promote these features from the LocationHitListTable part on the composite (Figure 264).

The reusable query is now ready; we save the part for use in another HTML view.
Reuse of Web Parts

Reuse Composite

In order to take advantage of our newly built reusable part, we create an ordinary Web Connection part that will serve as a parent to the *WtaReusableQuery* part. We use a graphic as a masthead on the page (we get the graphic from the Web Travel Agent application) and underneath we want to show some header text and the table with the location hit list.

After adding the graphics and the text, select **Add Part** from the Options pull-down menu of the Composition Editor. Enter the name of our reusable part, *WtaReusableQuery*, and give it a meaningful part name for the view (Figure 265).

![Add Part](image)

Figure 265. **Add Part**

The cursor is loaded and allows you to drop the composite part anywhere on the HTML page. Arrange the composite so that it shows underneath your graphics and text. Only the composite bearing the result table of the reusable query is shown; the query itself, its torn-off result table, and all connections among the parts are hidden (that is, encapsulated). Also, you cannot make any changes to the table, such as sizing or moving columns. The table does not even reflect the way you saved it in the reusable query part. Again, it does not matter how the part looks on the Composition Editor. Because the final rendering of the part is the responsibility of the browser, the generated HTML will always be unchanged.

The promote features of the table, however, give us a way to influence the appearance of the HTML table. Double-click on the embedded part to open the settings. There you have the promoted attributes for changing the border and the cell padding (Figure 266).

![Figure 266](image)

Figure 266. **Figure 266**

When is the query going to run? We cannot encapsulate the execution of the query. From our primary (parent) part, we have to notify the query to be processed and thus the table to be generated.
Reuse of Web Parts

Chapter 10. Implementation Considerations

Connect the *aboutToGenerateHtml* event of the HTML page to the promoted *packageQueryExecute* action of the embedded composite. This will run the query in the subpart, generate the HTML for the table, and embed it in our current HTML page. The final part in the Composition Editor looks as it is shown in Figure 267.

To verify that our part with its reused child composite really works, we try it out in our Web browser (Figure 268). Obviously, the part works because all locations show up, nicely ordered by number of visits, and the table has its customized border and cell padding, as we can verify in the generated HTML that follows Figure 268.
Figure 268. Location Hit List

HTML

<!-- generated by VisualAge Web Connection on 10-23-96 at 6:34:48 PM -->
<html><head><title>Location Hit List</title></head>
<body>
<img src="/wtapix/wta.gif"
<br/>
<h1>Location Hit List</h1>
Our most travelled destinations from the WTADB database:<br/>
<table border="2" cellpadding="5">
<tr><th>City</th><th>Visits</th></tr>
<tr><td>Atlanta</td><td>9</td></tr>
<tr><td>San Francisco</td><td>8</td></tr>
<tr><td>San Jose</td><td>6</td></tr>
<tr><td>Dallas</td><td>5</td></tr>
<tr><td>Frankfurt</td><td>5</td></tr>
<tr><td>Boston</td><td>4</td></tr>
<tr><td>Chicago</td><td>4</td></tr>
</table>
</body>
</html>
Single Row Query

For our example we will be using the Web Travel Agent database again. This time we want to see the complete database details of one particular travel package. We can imagine having one part with an HTML list containing abbreviated package data, where a user selects a package and the application returns our detailed view on the next page. For our purposes, we just create the SQL query so that it selects only one record from the TPACKAGE table to show the Quick HTML function for a single-row query.

This function is useful for data entry applications, where you might want to display or change all or some of the fields in a database row.

For single-row queries, Quick HTML builds a table that displays a label and an entry field for each column in the row. This makes it possible to build a user interface in which end users can edit the contents of the fields in the row, and then submit the changes for processing by another page. Because Quick HTML uses entry fields for this, you must place the generated table for a single-row query inside a form.

To display the results of a single-row query, create a new part (Figure 269). We call it LastPackageView because we will set up the SQL query so that it always returns the newest travel package (that is, the one with the highest package ID).

![New Part for Single Row Query](image)

From the Composition Editor’s Database category, select a Single Row Query part and drop it on the free-form surface. Double-click on the icon to open the settings, and select the Create option from the Query pull-down menu. Enter a name and a description for the query, select the TPACKAGE table, and click on the SELECT * check box to include all columns (Figure 270). This query now returns the whole table, that is, all rows and all columns. To restrict the result table to one row, we must add a WHERE clause to the SQL statement.
Pull down the **Clause** menu from the action bar and select the **WHERE()** item. To create the required clause, you need to define the operands. The left operand is the package ID (PACKID) column and the right operand is a nested SELECT (Figure 271) that will be defined on the next screen.

From a list of queries that were defined earlier, select a subquery that returns the comparison operand for the **WHERE** clause, in this case, the highest package ID (see Figure 272).
Chapter 10. Implementation Considerations

Figure 272. Subquery for WHERE Clause

After you clicked on OK, the subquery is inserted into the WHERE clause (Figure 273) completing the SQL statement. When you return to the SELECT Details screen (Figure 270) the Show SQL button allows you to view the full SQL statement that will be executed within the Single Row Query part created in the beginning (Figure 274).

Figure 273. WHERE Details (Complete)

Figure 274. SQL Statement
Now that you have set up your database query using the *Single Row Query* part, select *Quick HTML* from the pop-up menu of the *Single Row Query* part. When prompted, select the *resultRow* attribute from the list. VisualAge automatically tears off the *resultRow* attribute and loads the mouse pointer with the generated table. Place the generated table within a form on the HTML page.

At edit time, the table generated for a single-row query appears as HTML tagging rather than as an *HTML Table* part. To see how the table looks at run time, test it with your Web browser. Connect the *aboutToGenerateHtml* event of the *HTML Page* part to the *executeQuery* action of the *Single Row Query* part. Your Composition Editor should look as it is shown in Figure 275.

![LastPackageView - Composition Editor](image)

**Figure 275. LastPackageView - Composition Editor**

Connect to the newly created *LastPackageView* part from the Web browser to see the final rendering of the HTML table with entry fields filled through the execution of the SQL query (Figure 276).
Figure 276. LastPackageView - Web Browser
Implementation of Administrative Functions

The Web Travel Agency application depends on five main administrative functions:

- Completing packages whose end date is older than today’s date
- Listing selected customers, or all customers, in the database, and providing a detailed view of one selected customer, along with the list of packages booked by that customer
- Listing all the booked travel packages and providing a detailed view of one selected package
- Resetting the whole database, deleting all the booked packages, hotel rooms, cars, and flights
- Displaying the database statistics
By choice, we use the DB2 World Wide Web Connection gateway to implement the administrative functions of our application. We use the naming convention adminxx.d2w to name our DB2 WWW macro files, where xx is 0, 1, 2, and sometimes 2a, 2b, and so on.

The macro files are placed in the DB2WWW\MACRO subdirectory. If you want to change to a different path, you can modify the MACRO_PATH directive of the file DB2WWW.INI, which is located in the HTML document subdirectory of your Internet server (default is C:\WWW\HTML).

Administrator Authentication

When you try to access the administrative functions for the first time, the server prompts you to enter your administrator ID and password for authentication purposes. In our Web Travel Agency application, we let the IBM Internet Connection Server to do the checking to protect the DB2 WWW Connection CGI program DB2WWW.EXE.

Note

The Internet server can restrict access to the DB2WWW.EXE executable or its directory, but cannot protect the macro files. This happens because the macro files usually reside on a directory (the default is C:\DB2WWW\MACRO) that is not under the hierarchy of the Web server. In fact, the Web server does not know the path to the macros, as the MACRO_PATH variable is defined in the DB2WWW.INI file, and the information in this file is retrieved by DB2WWW.EXE. Thus, the only file IBM Internet Connection Server can protect is the executable DB2WWW.EXE.

When you run this program for the first time, you need to enter your administrator ID and password to access the home page of administrative functions. For ongoing pages, however, you need not enter the password again.

Here are the steps needed to set up DB2 WWW protection:

1. Create a new directory, say WTAADMIN, under your Web server home directory (default is C:\WWW)
2. Move your DB2 WWW Connection CGI program DB2WWW.EXE from the CGI-BIN subdirectory to the WTAADMIN subdirectory.
3. Modify the configuration file for the IBM Internet Connection Server, HTTPD.CNF, on your machine as follows:
Administrator Authentication

a. Search for the Protect directive. You should find a line that reads Protect /admin-bin/* PROT-ADMIN, which specifies the protection details for accessing the Configuration and Administration Forms when configuring your IBM Internet Connection Server. Add the following block to HTTPD.CNF just after that directive:

Protection WTA-ADMIN {
    Serverid WTA_Administrator
    AuthType Basic
    GetMask All@(*)
    PutMask All@(*)
    PostMask All@(*)
    Mask All@(*)
    PasswdFile C:/TCPIP/ETC/WTAADMIN.PWD
}

Protect /wta-bin/* WTA-ADMIN

This block of directives shows how to set up protection of the DB2WWW.EXE file in the /wta-bin subdirectory.

b. As we have created the directory WTAADMIN before, we need to define a mapping in the server, such that when the client requests access to /wta-bin, the server knows that it actually maps to the WTAADMIN directory in our local system.

What we need to do is to search for the line Exec \cgi-bin\*
C:\WWW\CGI-BIN\* and add the following line just after it:

Exec \wta-bin\* c:\www\wtaadmin\*

4. As we have specified the password file

    PasswdFile C:/TCPIP/ETC/WTAADMIN.PWD

in the Protection directive, we need to define that file. Create a one-line file called WTAADMIN.PWD that contains the following line:

    wtaadmin:wtaibm:WTA Administrator

and save it in the ETC directory of your machine. (Check it by typing SET ETC in an OS/2 prompt.)

The file has three fields separated by colons. The first field specifies the administrator ID, and the second field is the corresponding password. The third field is just a comment or description of the line. You can change the administrator ID and password by modifying the contents of this file.

5. Restart your server.

Note

If you are not using IBM Internet Connection Server as your server, consult your Web server’s user’s guide for modifying the configuration file.
Home Page for Administrative Functions

The Administrative Functions main menu (see Figure 277) shows five links to the five administrative functions. You can click on any one link to go to the function specified.

![Administrative Functions Home Page](image)

**Figure 277. Administrative Functions Home Page**

---

**Note**

The DB2 WWW macro contains only the HTML_INPUT section. That means we can use an HTML file to construct this page instead. Since we want all the pages of administrative functions to be generated by DB2 WWW macros, and we need to protect all the files with one single set of Administrator ID and password, we therefore encapsulate this first page in the DB2 WWW macro for our implementation.

---

**Complete Old Packages**

When you click on the first item of the home page, you request the application to complete old packages. The application complies by deleting those packages whose end dates are older than today’s date.
Year 2000

Our application will run very well in the next millennium, because our database design includes 4-digit year fields. All date calculations will therefore execute properly beyond December, 31st, 1999.

The application first displays all the packages that are going to be deleted and prompts the administrator to confirm the action (see Figure 278).

---

**Figure 278. Complete Old Packages**

Here are the packages that are going to be deleted:

<table>
<thead>
<tr>
<th>From</th>
<th>Destination</th>
<th>Start Date</th>
<th>End Date</th>
<th>Customer</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>New York</td>
<td>Atlanta</td>
<td>02/19/1996</td>
<td>02/17/1996</td>
<td>Bernard Tsang</td>
<td>Hong Kong</td>
</tr>
<tr>
<td>San Francisco</td>
<td>Chicago</td>
<td>04/08/1996</td>
<td>04/24/1996</td>
<td>Bernard Tsang</td>
<td>Hong Kong</td>
</tr>
</tbody>
</table>

*Click here to confirm the deletion*

---

**SQL section**

```
%SQL(confirm){
SELECT from_city, loc_name, start_date, end_date, cust_name, address
FROM tpackage, customer
WHERE end_date < CURRENT DATE
   AND tpackage.custid = customer.custid
ORDER BY loc_name
%}
```

For the window shown in Figure 278, we use an HTML table to format the selected packages. Here is what we write for the SQL Report section:
In this section, we have variable names such as $(V1)$, $(V2)$. They represent the values for each field of a row returned by the previous SQL query. So, $(V1)$ actually means from_city in the query, $(V2)$ means loc_name, and so on.

After confirmation, the application deletes all the old packages by the following SQL statement:

```
%SQL(delete){
DELETE FROM tpackage
WHERE end_date < CURRENT DATE
%
}
```

In our database design, we specified in our database definition

```
REFERENCES USERID.TPACKAGE ON DELETE CASCADE
```

on all the three tables BCAR, BSEATS, and BROOMS. Because of that, we do not need to issue separate SQL commands to delete record on these tables. When we delete a package record, all of its booked rooms, booked seats and booked car records are deleted automatically.

**Search for Customer Information**

The second item of the Administrative Functions Home Page allows you to search for a particular customer by specifying the customer name (or part of the name to let the application do a partial match). Figure 279 shows what the first screen looks like.
In this screen, we use an HTML form to let you type in the customer name that you want to find. Here is the syntax of the HTML form:

```
<form method=POST action="/wta-bin/db2www/admin2.d2w/report">

Please enter the customer's name (case sensitive), or leave it blank for listing all the customers in the database:<br>
<input type=TEXT name=custname size=30><p>
<input type=SUBMIT value="Search">
</form><p>
```

If you type nothing in the entry field and press the **Search** button, the application returns all the customers in the database. Here is the SQL query for listing the customers:

```
%SQL(listcust){
SELECT custid, cust_name FROM customer
WHERE cust_name LIKE '$(custname)%%'
ORDER BY cust_name
}%
```

![Figure 279. Search for Customers](image-url)
Search for Customer Information

The above query is issued by the HTML Report section. The value of the variable $(custname)$ is what you have just typed in the HTML input form, as we use the same variable name=custname for both sections. This example shows how DB2 WWW Connection allows the sharing of variables between HTML forms and SQL queries.

In the next screen, shown in Figure 280, the application lists the names of all the matched customers and puts them into an HTML list box (which is in fact a <select> tag with line number greater than one).

Figure 280. Search for Customers (List Found)

Here is the SQL Report section for the page:

```sql
%SQL_REPORT{
  <form method=POST action="/wta-bin/db2www/admin2a.d2w/report">
    <select name=cid size=3>
      %ROW{
        <option value="$(V1)">$(V2)
      %}
    </select>
    <input type=SUBMIT value="Detail information">
  </form>
}%
```
In this page, we also add a simple SQL COUNT(*) query to let the user know how many customers are found and are placed in the list box:

```
SQL Report section

%SQL(countcust){
SELECT COUNT(*) FROM customer
  WHERE cust_name LIKE '$_{custname}$'
%
}
```

After you click on the Detail Information button, the next screen appears as shown in Figure 281.

![Detail Information of Customer](image)

**Figure 281. Detail Information of Customer**

In this screen, we have two HTML tables: one shows the customer information, and the other lists all the travel packages for that customer.
Here is the macro for getting the customer information:

```
SQL section

%SQL(findcust){
SELECT cust_name, address, phone
  FROM customer WHERE custid=${cid}
%

%SQL_REPORT{
  <center><table border=2>
  %ROW{
    <tr><th>Name:<td>${V1}<br>
    <tr><th>Customer ID:<td>${cid}<br>
    <tr><th>Address:<td>${V2}<br>
    <tr><th>Telephone:<td>${V3}<br>
  %}
  </table></center>
%
}
%
%

And here is the macro for getting the travel package information:

```

```
SQL section

%SQL(findpack){
SELECT loc_name, start_date, end_date, packid, from_city
  FROM tpackage WHERE custid=${cid}
  ORDER BY loc_name
%

%SQL_REPORT{
  <p>Here are the packages of this customer:<p>
  <center><table border=2>
  <tr><th>From<br><th>Destination<br>
    <th>Start Date<br><th>End Date<br>
  %ROW{
    <tr>
    <td>${V5}<br>
    <th><a href="/wta-bin/db2www/admin3a.d2w/report?pid=${V4}">${V1}</a><br>
    <td>${V2}<br>
    <td>${V3}<br>
  %}
  </table></center>
%
}
%
%

In this page, we add HTML links to the names of each package, so that you can click on the name and get the package details. Since they are not HTML forms but HTML anchor links only, we cannot pass the package ID to the following page using the POST method. Instead, we use the GET method. The syntax of the link is like this:

```
<a href="/wta-bin/db2www/admin3a.d2w/report?pid=${V4}"${V1}</a>
```
Search for Package Information

Our application provides two flexible ways for getting detailed information about a travel package. One way is to search for a customer first, and then choose a package of that customer to view, as described in “Search for Customer Information” on page 370. Another way is to list all the packages in the database and let the user click on one to see detailed information. The third administrative function allows you to list travel packages as shown in Figure 282.

The HTML table in this page is similar to that in the Complete Old Packages page, except that a user can click on a location in the Destination field to look at the details for that package.

After you click on a package, either from this screen or from the screen in Figure 281, you see the package details as shown in Figure 283.
Reset the Whole Database

This function deletes all the packages in the database. All the records in the TPACKAGE, BROOMS, BCAR, and BSEATS tables are deleted. The administrator is prompted to confirm this action.

Display Database Statistics

This function display all the database statistics. It shows the number of records in the HOTEL, CAR, FLIGHT, CUSTOMER, PACKAGE, and BCAR tables by the SQL statement `SELECT COUNT(*)`. For the BROOMS and BSEATS tables, it shows the total number of booked rooms and booked seats, in which we use the SQL statement `SELECT SUM(rooms)` and `SELECT SUM(seats)`, respectively.
In this chapter, we describe the steps necessary to install and set up the prerequisites for the Web Travel Agency application:

- Web Connection
- Waikiki
- Internet Connection Server
- DB2 WWW Connection
- TCP/IP Configuration
- Web Travel Agent

The book is provided with two CDROMs. The first CD holds the installation program for each product. It also contains the following products:

- IBM VisualAge for C++ for Windows Version 3.5 (try and buy)
- Waikiki (beta)
- Database 2 for Windows Version 2.1.2 (try and buy)
- Web Travel Agent Application
The second CD holds the try-and-buy version of IBM VisualAge for Smalltalk for Windows. In the following sections, we detail the installation of each product.

Web Connection

In addition to installing the Web Connection feature in your VisualAge image, you must also set up your HTTP server to relay client requests to VisualAge for Smalltalk.

The Web Connection feature uses the CGI protocol to receive requests from Web browsers. CGI is a standard interface that defines how HTTP servers can run programs in response to client requests, and how the results from those programs are returned to the clients. In order for your Web Connection application to be accessible, you must set up your HTTP server to handle CGI requests. You must also set up the CGI Link program, which is shipped with the Web Connection feature, on your HTTP server machine. For more information on setting up your server to handle CGI requests, refer to the documentation for your HTTP server software.

Setting Up the CGI Link

When you install the Web Connection feature, several files are added to your VisualAge for Smalltalk development directory:

- The CGI Link executable program files and source file
- A sample CGI Link configuration file

Before VisualAge can receive requests from your Web server, you must install these files in the correct locations and set up your configuration file.

Installing the CGI Link Files

To install the CGI Link files, follow these steps:

1. Select the correct executable program file for the operating system your Web server runs on. This need not be the same operating system VisualAge runs on, if the two are on separate machines.

   Compiled versions of CGI Link are provided for the following operating systems:

   - **OS/2**
     - abtcgil.os2
   - **Windows 95**
     - abtcgil.win
   - **Windows NT**
     - abtcgil.win
   - **AIX**
     - abtcgil.aix
Chapter 12. Installation and Setup

Web Connection

If your HTTP server runs on an operating system other than these, you will need to compile the source code (abtcgil.c) to create an executable for your operating system. You can use the make file (abtcgil.mak) as a starting point for creating a make file for your development environment.

2. Copy the executable file to the directory your Web server uses for CGI programs. (Typically, this directory is called cgi-bin.) To determine the correct location, check the configuration of your HTTP server, or consult your HTTP server administrator.

3. If your HTTP server runs Windows or OS/2, rename the CGI Link executable program abtcgil.exe, or another name with an EXE extension. On AIX, you can optionally rename the file abtcgil or any other valid file name.

4. Create a CGI Link configuration file called abtcgil.cnf and place it in the directory on your HTTP server where CGI Link will run. The easiest way to do this is to copy the sample abtcgil.cnf file shipped with the Web Connection feature, which you can then customize for your environment.

The correct location for abtcgil.cnf can vary depending upon how your HTTP server runs CGI processes; some HTTP servers change to the CGI directory before starting a CGI process, while others do not. See “Troubleshooting” on page 383 for more information about finding the correct location for abtcgil.cnf.

5. Edit the CGI Link configuration file to specify the correct values for your server and your VisualAge application. (See “Editing the CGI Link Configuration File” on page 379 for information on how to set up the CGI Link configuration file.)

The HTTP server is now set up to relay incoming requests to the VisualAge application. If the VisualAge application is on the same machine as the HTTP server, no further setup is required. If it is on a separate machine, you must also be running TCP/IP on that machine.

Editing the CGI Link Configuration File

Edit the CGI Link configuration file using an ASCII text editor such as Windows Notepad, OS/2 EPM, or Unix vi.

The CGI Link configuration file must contain at least one line beginning with the keyword link. These entries define how the CGI Link communicates with VisualAge.

The format of a link entry is as follows:

    link <path_info> <protocol> <protocol_info>

where:
<path_info>

Specifies a value for the CGI PATH_INFO environment variable. This entry will apply only when PATH_INFO is equal to the value you specify.

PATH_INFO is a CGI variable submitted as part of the request. By specifying PATH_INFO, you can set up multiple link entries on the same server, each with a different PATH_INFO value. This makes it possible to route different CGI requests to different locations depending upon the extra path information specified by incoming requests.

The PATH_INFO environment variable contains the information that comes after the name of the CGI program (typically abtcgil). For a Web Connection part, this is the same as the name of the part being accessed. With PATH_INFO, you can specify that different part names should be accessed in different locations.

If you want to handle all requests in the same way, specify an asterisk (*) in the <path_info> field.

<protocol>

Identifies the protocol to use to communicate with VisualAge. Currently, only tcp (specifying the TCP/IP protocol) is supported.

<protocol info>

Provides additional location information for communicating with VisualAge. For the TCP/IP protocol, this information consists of the host name or numeric IP address followed by the port number, separated by a space: hostname.ibm.com 8081

You can also specify a dash (-) in place of the host name to indicate that VisualAge is on the same host as CGI Link.

**Note:** Be sure you use a port that is not being used by any other process on your VisualAge machine. For example, if your VisualAge machine is also running an HTTP server on port 80, do not use port 80 for the CGI Link.

In addition to the link entries, the CGI Link configuration file can contain a terseErrors line:

```
terseErrors <on|off>
```
This line controls whether error messages generated by CGI Link, which appear in client browsers, will be terse or verbose. Terse error messages do not reveal any information about your server setup, which can be important if you are concerned about security. Verbose error messages include detailed information about your server setup, including:

- Version information about your CGI Link executable
- Version information about your HTTP server software
- The TCP/IP host name of your server
- Path and file name information about your CGI Link files

This information can be helpful during debugging, but is generally not of interest to users at runtime.

If your CGI Link configuration file does not contain a `terseErrors` line, the default value is `terseErrors off`.

In addition to the `link` entries, the CGI Link configuration file can contain comments, identified by a `#` character in column 1.

Following is an example of a CGI Link configuration file that routes all requests to the same location:

```plaintext
#==============================================================
# Sample cgi-link configuration file with one location
# # VisualAge is running on host webconnection.raleigh.ibm.com
# link * tcp webconnection.raleigh.ibm.com 8081
# # hide configuration information from users
# terseErrors on
# #==============================================================
```

This example routes any CGI Link requests via TCP/IP to port 8081 on the host webconnection.raleigh.ibm.com. The VisualAge application must be running on that host, and its CGI Link server process must be configured to listen to port 8081.

Following is an example of a more complex CGI Link configuration file that routes different requests to different locations:
Testing the Installation

After you install the Web Connection feature and set up CGI Link on your HTTP server, you can test your installation by building a small Web page. This will confirm that your installation is functioning properly, and that Web browsers can access applications you build with VisualAge.

To test your installation, follow these steps:

1. In the VisualAge Organizer, create a new part. For part type, select Web Connection part.
2. Open the Composition Editor on the new part. You should see a single subpart, an HTML Page.
3. Go to the Web Connection category and select an HTML Text part. Place the HTML Text part inside the HTML Page.
4. Open the settings of the HTML Text part. On the Text page, set the Text to be displayed to read Hello, World!
5. Save the part.
6. In the VisualAge Organizer, select Start a CGI Link Server from the Options menu. (You can also do this from the Smalltalk Tools menu of the System Transcript window.)
7. In the Start a new CGI Link server window, specify the port number through which CGI Link will connect to VisualAge. This should be the same port number you specified in the CGI Link configuration file on the HTTP server. (See “Editing the CGI Link Configuration File” on page 379 for more information about the CGI Link configuration file.)

8. Start your Web browser and open the URL specifying the part you just built. For example:

   http://hostname.ibm.com/cgi-bin/abtcgil.exe/WebHelloWorld

   Be sure to specify the correct host name for your HTTP server and the correct path and file name for the CGI Link program.

9. After the page loads, you should see the text *Hello, World!* appear in your Web browser.

**Troubleshooting**

If the *Hello, World!* application does not load correctly, there are several things you can check:

- If your Web browser cannot connect with the HTTP server, then something is wrong with your TCP/IP or HTTP server setup. Make sure you can ping the server, and check the configuration of your HTTP server software.

- If your Web server cannot find the CGI Link program, make sure you have installed the executable file in your CGI directory. Also confirm that you are specifying the correct file name in the URL (usually `abtcgil.exe` for OS/2 and Windows, `abtcgil` for AIX).

- If you see a CGI Link error reading *Unable to connect to host*, then the CGI Link program was unable to contact the host where VisualAge is running. Make sure all of the following are true:
  - The `abtcgil.cnf` file specifies the correct TCP/IP host name.
  - TCP/IP is running on the specified host. Try pinging the host from the HTTP server machine.
  - The CGI Link server is running in the VisualAge image. You should see the AbtCgiLinkServer window.
  - The CGI Link server is monitoring the correct port. Compare the port specified in the `abtcgil.cnf` file to the port number you specified when you started the CGI Link server.

- If you see a CGI Link error reading *No class responded to this request*, then the class name you specified in the URL does not exist. Make sure you are typing the class name exactly as it appears in VisualAge, including the correct capitalization.
If you see a CGI Link error reading *An exception occurred during the processing of this request*, then something caused the VisualAge application to fail. Check for errors in the application.

If you see a CGI Link error reading *Configuration file abtcgil.cnf not found or unreadable*, then the CGI Link program was unable to find the CGI Link configuration file, or the file was not in the correct format. Make sure the following are true:

- You have placed the ABTCGIL.CNF file in the directory where CGI Link runs. To find out the correct directory, check the current directory entry on the CGI Link Error page that appears in your browser. (This information appears only in verbose mode, so make sure your CGI Link configuration file includes a `terseErrors off` statement.)

- The `abtcgil.cnf` file is a readable ASCII file and conforms to the correct format. (See “Editing the CGI Link Configuration File” on page 379 for more information on the `abtcgil.cnf` file format.)

## Using the CGI Link Server Interface

When the CGI Link server process is running, it displays a window that shows statistics on incoming requests and the Web Connection parts that respond to them. For each Web Connection part you have defined, the CGI Link server window shows the following information:

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class</td>
<td>The Smalltalk class name of the Web Connection part</td>
</tr>
<tr>
<td>Count</td>
<td>The number of instances of the part that have been created in response to incoming requests</td>
</tr>
<tr>
<td>Avg</td>
<td>The average time (in milliseconds) the part has taken in responding to incoming requests</td>
</tr>
<tr>
<td>Min</td>
<td>The shortest time (in milliseconds) the part has taken to respond to an incoming request</td>
</tr>
<tr>
<td>Max</td>
<td>The longest time (in milliseconds) the part has taken to respond to an incoming request</td>
</tr>
</tbody>
</table>

The first two lines in the CGI Link server window show total counts and times for all incoming requests and outgoing responses. These lines are labeled *request receives* and *response sends*. In addition, the CGI Link server window shows the following information:

**Peak transactions**

The largest number of transactions that have been pending at any one time.
Active session keys

The total number of session keys that have been generated and have not timed out (this indicates roughly how many user sessions are currently in progress). See “Preserving Session Data” on page 47 for more information about sessions and session keys.

Waikiki

You must install VisualAge for C++ first before installing Waikiki on your machine. A trial version of VisualAge for C++ for Windows is included on the CDROMs that accompany this book. If VisualAge for C++ is already installed on your machine you do not need to install it again and you can jump directly to “Installing Waikiki” on page 386.

Installing VisualAge for C++ for Windows

Before you install VisualAge for C++ on Windows 95, you must increase the system environment size. The VisualAge for C++ for Windows product updates a number of system environment variables during installation and the default system environment size is too small. The required changes are described below.

CONFIG.SYS Change

Use your favorite editor to add the following line to your CONFIG.SYS file. Create the file with this line if the file does not exist.

```
SHELL=C:\WINDOWS\COMMAND.COM /p /e:20000
```

This line sets the system environment size to 20,000 bytes, which is sufficient for the VisualAge for C++ product. If you have other products that use a significant amount of system environment space, you may need to increase this value. You may also want to use a smaller value on systems that are constrained by memory.

This change will set the system environment size for programs run directly from folders.

NOTE: The above assumes that Windows 95 was installed in the C:\WINDOWS directory. If you installed in another directory, substitute the directory name.
**MS-DOS Prompt Change**

Start an MS-DOS Prompt. If you are using the Start button, select Programs and then MS-DOS Prompt. Open the MS-DOS Prompt Properties sheet by:

- Opening the system menu by clicking on the MS-DOS icon at the top-left corner of the MS-DOS Prompt, and then selecting the Properties item, or,
- Selecting the Properties icon on the toolbar if the toolbar is active. Switch to memory page by selecting the Memory tab at the top. Within the Conventional memory area is a drop-down entry field. Click on the arrow, scroll down until the 4096 entry is visible and select it. Now click on the OK button at the bottom of the Properties sheet.

This change sets the system environment size for each MS-DOS prompt started.

**Installing**

To install VisualAge for C++ for Windows, put the first CDROM in the CD drive and wait for the installation program to come up. Select VisualAge for C++ to install the product. Follow the various installation panels. You can refer to the README file located in the VACPP directory of your CDROM for the latest information regarding VisualAge for C++ setup.

**Installing Waikiki**

Waikiki requires 41 megabytes of disk space. An additional 10 megabytes is needed to hold the installation files. Please carry out these steps to install the Waikiki alpha driver:

1. Put the first CDROM in the CD drive and wait for the installation program to come up. Select Waikiki to install the product. A welcome panel is displayed (Figure 284). Click the Next push button to continue.
Chapter 12. Installation and Setup

2. Answer prompts about target directory or take the defaults (Figure 285). Choose a target drive with at least 41 megabytes of free space.

3. Select the folder name and location of Waikiki or take the defaults (Figure 286).

Figure 284. Welcome Panel of Waikiki Installation Program

Figure 285. Choose Destination Location Panel

Figure 286.
Once Waikiki is installed, the Waikiki folder contains four icons as shown in Figure 287.

From now on, each time you need to use the Visual Builder to design a static or dynamic HTML page, you must use the Waikiki Visual Builder from the new program folder you chose in Step 3 (Figure 288). In effect, the Waikiki Visual Builder is a special version of Visual Builder that has additional attributes for Web parts.
Also, make sure you select the DOS session provided in the Waikiki folder to compile and link-edit the code you generate from the Visual Builder. This DOS session is set up with the proper environment variables to access the libraries and header files required to produce your CGI programs.

Testing the Waikiki Installation

You can test the Waikiki installation by running the CInput/COutput sample provided in the \waikikiw\working directory. The sample is composed of the CInput.html, vbbase.gif, and COuput.exe files. CInput.html requests a CGI POST action to the COutput.exe CGI program. Place the CInput.html and vbbase.gif files in your server’s HTML document directory. COutput.exe is the target of the form method parameter. COutput.exe is a run time DLL-based. Place it in the server’s CGI-BIN directory (or whatever executable directory is set up). If the server PATH environment variable does not point to \ibmcppw\bin and \waikikiw\bin, you must place all needed DLLs in the executable directory. You may alternatively configure your working directory to be a server executable directory. This is convenient for iterative development.

The DLLs needed can be found by using the CHKMOD32 utility provided in with Waikiki and located in \waikikiw\bin. Simply open a DOS session and change to the directory where COutput.exe is located. At the command line, enter (Figure 289):

\waikikiw\bin\chkmod32 coutput.exe
Bring up `CInput.html` in your browser. Type in a name in the displayed entry field and click on the Submit button (Figure 290).

A dynamic Web page is generated by `COutput.exe` where you should see that name in the output name field (Figure 291). The address and other fields come from the defaults supplied by the `IAddress` and `ICustomer` sample nonvisual parts.

Figure 289. Using Chkmod32 to List DLLs

Figure 290. `CInput.html` Form
The name comes from the entry field in CInput.html. It’s attribute name is name. This attribute is transmitted to COutput.exe as a parameter. Look at the COutput part in the composition editor to see how to use the ICGI part to extract this parameter from the input to the server. ICGI has a key field that names the parameter to be extracted. Do not forget to fill in this field on the magenta parameter connection, or you will be requesting a non-existent parameter.

If you experience problems running COutput.exe, you can rebuild the CGI program as follows:

1. Start the Waikiki Visual Builder from the new program folder (see Figure 287 or Figure 288).
2. Load the COutput.vbb file in the Visual Builder by selecting the File→Load... option in the Visual Builder menu bar.
3. Select the COutput visual part and from its pop-up menu generates its code and the make file by selecting the following options: Generate→Part source and Generate→main() for part.

After the code is generated, start Waikiki command line from the new program folder. Change to the working directory where the COutput part’s code has been generated. Edit the COutput.mak file and remove the /pmt:ype:pm flag on the link command line.
Compile and link-edit your code, by issuing the command:

    nmake COutput.mak

Run CHKMOD32 against the COutput.exe executable to verify that CPPHOB3.DLL, CPPHOU3.DLL, and CPPOHV3.DLL are loading.

Uninstalling Waikiki

An uninstallation icon is created in the Waikiki program folder. You can use it to remove Waikiki from your machine. The preferred method is to use Add/Remove Programs from the Control Panel. That way, the program folder gets removed also.

Note that any files you create directly or indirectly will not be removed. These include any files you create in the working directory of the installation tree. You must remove these files yourself, because the uninstallation process assumes that you want to keep them.

IBM Internet Connection Server

The IBM Internet Connection Server for OS/2 Warp is a WWW server that runs on OS/2 Warp. It provides a way for you to share graphical, interactive information on the Internet or within a local network.

This section offers you a quick reference on installing, using, and configuring your IBM Internet Connection Server. For more detailed information, refer to the IBM Internet Connection Server for OS/2 Warp: User's Guide, and the README file on the IBM Internet Connection Server installation diskette or CD-ROM.

Hardware and Software Requirements

Prior to installing the IBM Internet Connection Server, make sure you have the following hardware and software ready:

- Any personal computer or PS/2 computer with OS/2 Warp 3.0 or Windows 95 installed.
- Approximately 2 MB of free disk space. If you are using OS/2, you need to install your IBM Internet Connection Server into a partition formatted using the High Performance File System (HPFS), as some system files require long file names.
- OS/2 TCP/IP 3.0, which comes with OS/2 Warp Connect. If you have OS/2 TCP/IP 2.0 instead, you must apply APAR fix PN71501.
- Windows 95 and its Service Pack 1.
❑ Your machine’s host name, IP address, domain name, domain name server address, IP address of your router, and so on. Usually, your system administrator has already configured these for you during the installation of TCP/IP. If you are not sure of any of these, ask your system administrator.

❑ A LAN adapter that is supported by OS/2 Warp 3.0 or Windows 95. When you start your IBM Internet Connection Server, you must have your machine connected to the LAN.

**Warning**

If you want to use HTTP to access your HTML documents and CGI programs, you must have your machine connected to the LAN, even if the files actually reside in the same machine as your Web browser. The reason is that when you type in your Uniform Resource Locator (URL) http://hostname.domainname, your browser tries to access your domain name server to search for the IP address of the host, even when the host is in fact your own machine.

If you are not connected to the LAN, you can only use Open File of your Web browser to view your local HTML files, and you will find it impossible to run any CGI program from the browser.

**Installation**

The installation of the IBM Internet Connection Server is basically identical on all supported platforms. The version on the enclosed CDROM runs on Windows 95 and NT, the steps described here are shown for OS/2. The installation requires six steps:

1. At an OS/2 prompt, type:

   x:install

   and press Enter, where x: is the drive on which you put your IBM Internet Connection Server diskette or CD-ROM (see Figure 292).
2. From the *Instructions for Installation* window, click on **Continue**.
3. Click on **OK** from the **Install** window.
4. From the **Install directories** window, select the product components that you want to install (see Figure 293).

![Figure 292. IBM Internet Connection Server Installation](image)

![Figure 293. Install Directories –Internet Connection Server](image)
It is recommended that you choose **Select all**, as all the components take up only 2 MB of disk space.

The default home directory for installing IBM Internet Connection Server is `C:\WWW\`, but you are allowed to change it to any other directory name you like, for example `D:\HTTPD\`. However, make sure that the partition in which you are going to install IBM Internet Connection Server is HPFS formatted.

5. From the **IBM Internet Connection Server Install Configuration** window (Figure 294), you can optionally change the default values for the host name and the port, and you can select to start the IBM Internet Connection Server automatically when you start OS/2. You can also change the fields Administrator ID and Password, which are required when you use the Configuration and Administration Forms to configure your server. For more details, see “Using the Configuration and Administration Forms” on page 397.

![Figure 294. Internet Connection Server Install Configuration](image)

6. Click on **OK**, and the installation completes after all the files have been copied to your hard disk. An **IBM Internet Connection Server for OS/2** folder is created on the OS/2 desktop.

**Starting the Server**

You can start the server by selecting the IBM Internet Connection Server for OS/2 Warp icon, or by entering the following command at an OS/2 prompt:

```
HTTPD
```
This starts your server with the settings you specified during installation along with default values for other configuration settings.

Now that you have the server started, you can:

- Use the Front Page to configure the server remotely or view an HTML version of the User’s Guide
- Create your own welcome page

The IBM Internet Connection Server front page gives you access to an assortment of tools and information you will find valuable as you use the server. It is an HTML document stored in a file named Frntpage.html on your HTML documents directory (the default is C:\WWW\HTML). To use the front page, open the Frntpage.html file from IBM WebExplorer or any other Web browser. The front page URL is

http://hostname.domainname/Frntpage.html

where hostname.domainname is your server’s host name.

From the front page you can link to:

- Configuration and Administration Forms — a set of forms that you can use to easily configure your IBM Internet Connection Server to meet your particular needs
- Sample HTML files — a set of samples that show some of the ways you can use HTML to create your own documents
- Documentation — an HTML version of the IBM Internet Connection Server for OS/2 Warp User’s Guide
- Resource list — an HTML document containing links to many WWW sites you may find interesting and useful as you work with the IBM Internet Connection Server

**Configuring the Server**

The IBM Internet Connection Server comes with a default configuration that gives you a fully operational Web server. You can change any part of the default configuration to make the server meet your specific needs.

You can configure the server either by using the Web server Configuration and Administration Forms, or by editing the server’s configuration file directly.
Using the Configuration and Administration Forms

The best and easiest way to configure your IBM Internet Connection Server is to use the Configuration and Administration Forms from the IBM Internet Connection Server front page. Using these forms, you can configure your server from the host itself or from a remote client that has access to the forms.

The URL for the main form is

http://hostname.domainname/admin-bin/cfgin.exe/initial

which is a CGI program located in your server. It prompts you to enter the Administrator ID and password, which you provided when you installed your server. If you have not made any modification during installation, the default Administrator ID is webadmin, and the password is webibm. You can change the Administrator ID and password by modifying the file ADMIN.PWD. This file is in the path specified on the SET ETC statement in your CONFIG.SYS file.

From the main form, you can link to each of the other input forms. Each input form lets you enter information about how you want to configure part of your server. The forms are displayed with current values in their input fields, and they provide instructions and help information to assist you in deciding what changes to make.

For each form you decide to use, make your changes to the input fields and then select the Apply button at the bottom of the form. The server shows you a message indicating whether your input was accepted.

If the input is accepted, the server prompts you to click on the Restart Server button, so that you do not need to restart your IBM Internet Connection Server manually. The changes take effect as soon as the server is restarted.

Editing the Configuration File

Since the Configuration and Administration Forms modify the contents of the configuration file HTTPD.CNF when configuring your server, you can in fact edit the configuration file manually. This file is located in the path specified on the SET ETC statement in your CONFIG.SYS file. You can type SET ETC in an OS/2 Command Prompt to check for the path.

The configuration file is made up of statements called directives. For more details on these directives, please refer to the IBM Internet Connection Server for OS/2 Warp User's Guide. The file HTTPD.CNF also contains detail comments that help you to configure your server.

Here are the descriptions of some directives that you may find useful.
ServerRoot Specify the home directory for the server.
This directive specifies the directory where you keep the IBM Internet Connection Server executable program HTTPD.EXE, and some other related data and programs. The default is C:\WWW\BIN, or the name you provided during installation.
You need to change it when you move your IBM Internet Connection Server to a different directory or a different drive.

HostName Specify the fully qualified host name for the server.
Use this directive to specify the host name of the machine on which the server is running. The default value is the same as the host name defined in your CONFIG.SYS file by SET HOSTNAME.

Port Specify the port number you want the server to use.
Use this directive to specify the port number the server should use. The default port number for HTTP is 80. Other port numbers less than 1024 are reserved for other TCP/IP applications (such as telnet, ftp, gopher) and should not be used.
Port numbers 8000, 8080, 8001, and 8008 are commonly used for testing servers. When a port other than 80 is used, clients are required to include a specific port number on requests to the server, for example:

   http://bavaria.almaden.ibm.com:8000/

Welcome Specify default file displayed when URL has a directory only.
Use this directive to specify the name of a welcome file to be returned for directory requests. When the server receives a request containing a directory name, it searches the directory for files matching a name specified on a Welcome directive.
You can have more than one occurrence of this directive, for example:

   Welcome Welcome.html
   Welcome Frntpage.html
   Welcome index.html

The server might find more than one match between a file name on a Welcome directive and a file name in the requested directory. The file name from the first Welcome directive that is matched is the one that is returned.
AddType

Specify the data type of files with particular extensions.

Use this directive to bind files with a particular extension to a Multipurpose Internet Mail Extension (MIME) type/subtype. The format of the directive is:

AddType .extension type/subtype encoding [quality]

For example:

- `AddType .html text/html 8bit 1.0`
- `AddType .au audio/basic binary 1.0`
- `AddType .tif image/tiff binary 1.0`
- `AddType .mpeg video/mpeg binary 1.0`
- `AddType .mov video/quicktime binary 1.0`
- `AddType .avi video/x-msvideo binary 1.0`
- `AddType .gz multipart/x-gzip binary 1.0`
- `AddType .c text/plain 7bit 0.5`

Protecting the Server

Part of the attraction of the World Wide Web is its openness, even though the Web is open abuse as well as constructive use. In particular, using CGI scripts or programs on a Web server can create a security exposure. If you use CGI scripts or programs, you need to protect them by controlling who has access to them.

There are several ways to limit access to your server resources. You can require that a user requesting a document from your server supply an ID and password. The IBM Internet Connection Server uses the basic authentication scheme, which sends the password as plain text, encoded but not encrypted. You can also restrict access to document directories, reject access to specific URLs (such as local user pages), or restrict access to only authenticated users. Refer to the IBM Internet Connection Server for OS/2 Warp User’s Guide for more details.

In our Web Travel Agency application, we protect the CGI program DB2WWW.EXE, the program we use for administration functions, by requiring administrator authentication. You can also refer to Chapter 11, “Implementation of Administrative Functions,” on page 365 for how we restrict user access.

DB2 World Wide Web Connection

This section offers a quick reference on installing and configuring DB2 WWW Connection gateway version 1.0 on your machine. For more detail, refer to the DB2 WWW Connection Application Developer’s Guide
Hardware and Software Requirements

To install the DB2 WWW Connection, you need to have IBM Internet Connection Server for OS/2 already installed in your machine. You also need DB2/2 version 1.2 or later.

If you have installed a previous (beta) version of DB2 WWW Connection, uninstall it first by running Install and selecting the Delete option.

Installing DB2 WWW Connection requires a minimum of 600 KB of free disk space, but 6.5 MB is recommended.

Installation

There are five steps to install the DB2 WWW Connection gateway on your system:

1. If you are installing from a diskette, type at an OS/2 prompt:

   x:install

   and press Enter, where x: is the drive on which you put your DB2 WWW Connection diskette (see Figure 295).

2. If you are installing from a CD-ROM, type at an OS/2 prompt:

   x:\db2www\install

   Figure 295. DB2 WWW Connection Gateway Installation

   If you are installing from a CD-ROM, type at an OS/2 prompt:

   x:\db2www\install
and press Enter, where x: is the drive on which you put your DB2 WWW Connection CD-ROM.

2. From the Instructions for Installation window, click on Continue.

3. Click on OK from the Install window.

4. From the Install directories window, select the product components that you want to install (see Figure 296).

![Figure 296. Install Directories – DB2 WWW](image)

Make sure the CGI-BIN, Document and Images directories match the directories in your Internet Connection Server configuration.

5. Click on OK, and the installation completes after all the files have been copied to your hard disk. An IBM DB2 World Wide Web Connection folder is created on the OS/2 desktop.

After installation, the following directories will be created in the DB2 WWW Connection installation directory (default is C:\DB2WWW):

- **DOC\**  
  Documentation and the README file

- **MACRO\**  
  DB2 WWW macro files

- **DB2V1\**  
  Executable and bind file for DB2/2 Version 1.2

- **DB2V2\**  
  Executable and bind file for DB2/2 Version 2.1

- **TOOLS\**  
  Command files to switch support for different DB2/2 versions
DB2 DATA \ Sample database import files
ICONS \ DB2 WWW icons for the desktop
NLS \ National Language Support DLL and catalog files

In addition, the following files will be installed into the appropriate Web Server directories:

WWW\CGI-BIN\db2www.exe  Executable from db2v2 directory
WWW\CGI-BIN\db2sql.bnd  Bind file from db2v2 directory
WWW\HTML\db2www.ini  DB2 WWW initialization file
WWW\HTML\*.htm  DB2 WWW HTML files
WWW\ICONS\*.gif  Images used by the above HTML files
WWW\HTML\tmplobs Directory to store temporary LOBS from queries

Configuration

The best and easiest way to configure your DB2 WWW Connection is to use the Configuration Forms. The URL for the form is:

http://hostname.domainname/admin-bin/db2in.exe

which is a CGI program located in your server. It prompts you to enter the Administrator ID and password, which you provided when you installed your IBM Internet Connection Server. If you have not made any modification in that window, the default Administrator ID is webadmin, where the password is webibm. You can change the Administrator ID and password by modifying the file ADMIN.PWD. This file is in the path specified on the SET ETC statement in your CONFIG.SYS file.

From this form, you can update the path for DB2WWW macro files (default is C:\DB2WWW\MACRO) and the DB2WWW Bind file (default is C:\WWW\CGI-BIN\db2sql.bnd).

Alternatively, you can modify the configuration file directly. The configuration information is in the file DB2WWW.INI, which resides in the HTML document root directory (default is C:\WWW\HTML).
TCP/IP Configuration

Three pieces of software are needed, all connected by TCP/IP:

- A Web browser
- An HTTP server
- A VisualAge for Smalltalk Professional version 3.0a development environment with the Web Connection feature loaded.

Web Browser

Any browser that supports HTML 2.0 will do, even a text-based browser. However, graphical browsers such as Netscape, IBM WebExplorer, or Microsoft Internet Explorer are far more useful. The Web Connection’s ability to use a Java applet or a Netscape embed requires a browser that incorporates these features.

HTTP Server

The server needs to support HTTP 1.0 and CGI 1.1. We do our work with the IBM Internet Connection Server. This is available for OS/2, AIX, Windows 95, Windows NT, HP-UX, and Sun Solaris. To find out more, visit the Internet Connection Server web site at http://www.ics.raleigh.ibm.com

The VisualAge for Smalltalk Web Connection provides a CGI Link program to run on the server. Executables of this program are provided for OS/2, AIX, Windows NT, and Windows 95. C language source and make files are provided for creating an executable on other operating systems. Netscape, Spry, WebSite, NCSA, Apache, CERN, and Microsoft Internet Server have been tested.

The CGI Link program and its configuration file need to be on the HTTP server. As the developers incorporate graphics (GIF and JPEG) into their Web pages, these graphic files are placed on the server. A development-controlled HTTP server or cooperative webmaster is needed.

VisualAge

The Web Connection parts are available in VisualAge for Smalltalk Professional Version 3.0a or later. No other VisualAge for Smalltalk features are required for Web development. Web Connection provides the necessary Smalltalk TCP/IP classes.

One, two, or three separate systems can be used to provide these pieces. With one machine, TCP/IP loopback must be configured. For two or more machines a conventional TCP/IP connection is required.
Loopback

The development and demonstration of the Web Travel Agency application was greatly aided by using the Web browser, HTTP Server, and VisualAge for Smalltalk development environment on one machine. With a capable laptop computer, you can develop this entire application without wires. (A powercord is needed to recharge the battery from time to time.) With the TCP/IP loopback facility, there is no need to access a modem or LAN adapter card.

There is a special network interface driver for loopback. The basic architecture is as follows:

```
+-----------------------------+  +-----------------------------+
|                            |  |                            |
|          Socket Layer       |  |          kernel space      |
|                            |  |                            |
|      T C P                  |  |      T C P                 |
|                            |  |                            |
|      I P                   |  |      I P                   |
|                            |  |                            |
|   lo0 | en0 | tr0 | etc...                   |
|      |     |     |                            |
+-----------------------------+  +-----------------------------+
```

Packets sent to the loopback interface `lo0` do not reach any adapter device drivers. These packets are looped inside the loopback network interface driver.

From the point of view of the socket layer or TCP/IP, `lo0` is just another network interface. Hence, a packet destined for the loopback interface would follow the same path inside these layers as would any other data packet going to a Token-ring (`tr0`) or Ethernet (`en0`) interface.

The Ethernet and Token-ring network interface drivers (`if_en`, `if_ie5`) are loaded as kernel extensions, while the loopback network-interface driver is part of the kernel.

For loopback to work in OS/2 Warp Connect, the configuration program object in the TCP/IP folder needs to be modified. The number 127.0.0.1 is a magic number associated with loopback, as it is used for all the IP loopback addresses. Three notebook pages need to be changed:
TCP/IP Configuration

Network
Be sure that loopback is enabled in the Interface to Configure listbox (see Figure 297). If your machine has other adapters, they can remain enabled.

Routing
In the Configure Routing Information table, one or more destination addresses are specified. Each destination address must be unique. That is, the system does not support two destination addresses beginning with the same number. Figure 298 shows the loopback routing configuration.

Hostnames
Only the Nameserver address needs to be changed. Values used in the other fields to support other configurations can remain unchanged. Figure 299 shows the loopback nameserver configuration.

Loopback and Nonloopback
You can set up TCP/IP to connect through the loopback interface and the LAN adapter concurrently. You have to enable the LAN interface (Figure 300), configure the routing to contain both loopback and nonloopback information (Figure 301), and configure nameserver addresses for both interfaces (Figure 302).

The loopback and nonloopback TCP/IP configuration allows you to have two Web browsers up. Use one to test your development of VisualAge for Smalltalk Web Connection applications and the other to surf the net and to monitor the status of your production Web applications. Figure 303 shows the server configuration in IBM WebExplorer for loopback, while Figure 304 shows the server configuration in IBM WebExplorer for accessing the Internet.

After you changed your TCP/IP configuration, save the new setup by closing the configuration settings notebook. Restart TCP/IP for the new values to take effect. Close any running TCP/IP programs, such as portmap or inetd, then issue tcpstart from an OS/2 command line.

Web Connection Loopback Setup
If you decide to connect to your VisualAge Web application through the loopback interface, the VisualAge for Smalltalk Web Connection program on the HTTP Server, ABTCGIL.EXE, also needs to be configured for loopback. Make sure the following is the last link statement in the CGI Link configuration file, ABTCGIL.CNF:

```plaintext
link * tcp 127.0.0.1 8081
```
TCP/IP Configuration

To access the Web Travel Agency application when you are not connected to the Internet, use the loopback address for the HTTP server in the URL, for example,

http://127.0.0.1/cgi-bin/abtcgil.exe/WtaWelcomeView

To access the Web Travel Agency application when you are connected to the Internet, specify your IP hostname for the HTTP server in the URL, for example,

http://bavaria.almaden.ibm.com/cgi-bin/abtcgil/WtaWelcomeView

![Figure 297. TCP/IP Configuration for Loopback Interface](image)
Chapter 12. Installation and Setup

TCP/IP Configuration

Figure 298. TCP/IP Configuration for Loopback Router

Figure 299. TCP/IP Configuration for Loopback Nameserver
TCP/IP Configuration

Figure 300. TCP/IP Configuration for LAN Interface

Figure 301. TCP/IP Configuration for Loopback and LAN Router
Figure 302. TCP/IP Configuration for Loopback and LAN Nameserver

Figure 303. IBM WebExplorer Server Configuration for Loopback
Web Travel Agent

In this section, we describe the steps required to install the Web Travel Agent application on your system. As prerequisites, you need to have the following products installed:

- For the VisualAge for Smalltalk implementation:
  - VisualAge for Smalltalk version 3.0a
  - Database 2 for OS/2 or Database 2 for Windows
  - Internet Connection Server or any other Web server program

- For the VisualAge for C++ implementation:
  - VisualAge for C++ version 3.5 for Windows
  - Database 2 for Windows
  - Waikiki
  - Internet Connection Server or any other Web server program

Notice that the Smalltalk implementation of the Web Travel Agent application is available either on OS/2 or Windows although the C++ implementation is available only for Windows 95 or Windows NT.

Application

To install the Web Travel Agent application follow these instructions:

1. Put the first CDROM in the CD drive and wait for the installation program to come up. Select Web Travel Agent Sample to install the product. A welcome panel is displayed (Figure 305). Click the Next push button to continue.
2. A short information window is displayed and let you know about the three types of installation you can choose (Figure 306):

- **Typical installation** allows you to install both implementation of the Web Travel Agent, the CRC cards in HTML format and the screenplay of the application that you can view in your browser.

- **Compact installation** allows you to install only the CRC cards and the screenplay of the application.

- **Custom installation** allows you to install any selection of components: the two implementations, the CRC cards, or the screenplay.

3. Click on the Next button and select the destination directory for your installation (Figure 307), then click on the Next button.
4. In the next panel, select the type of the installation for the Web Travel Agent application. If you select the custom installation and click on the Next button you are prompted to select the components you want to install (Figure 308). Notice that the disk usage changes according to the components you select.

5. Once you selected the components to install, click on the Next button to choose program folder for your installation (Figure 309).
6. Click on the **Next** button to display the summary panel then proceed with the installation by clicking the **Next** button.

Once the Web Travel Agent application is installed, the Web Travel Agent folder should contain icons as it is shown in Figure 310.

The CRC cards and the WTA screenplay can be viewed using your Web browser. A **readme** file is provided to get you started with the Web Travel Agent installation. If you install all the components of the Web Travel Agent application, the directory structure shown in Figure 311 will be created.
Table 14 details the content of each directory.

**Table 14. WTA Directory Details**

<table>
<thead>
<tr>
<th>Directory</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>WTA</td>
<td>Deinstallation and readme files.</td>
</tr>
<tr>
<td>WTA\DB2</td>
<td>WTADB database creation files</td>
</tr>
<tr>
<td>WTA\DB2\EXPORT</td>
<td>WTADB database export files</td>
</tr>
<tr>
<td>DB2WWW</td>
<td>DB2WWW code for maintaining WTADB</td>
</tr>
<tr>
<td>WTA\DB2WWW\CGI-BIN</td>
<td>DB2WWW CGI code</td>
</tr>
<tr>
<td>WTA\DB2WWW\IMAGES</td>
<td>Bitmaps for the DB2WWW application</td>
</tr>
<tr>
<td>WTA\EXE\OS2</td>
<td>WTA runtime for VisualAge Smalltalk for OS/2</td>
</tr>
</tbody>
</table>
We recommend that you install the WTA directory tree on your C: drive, because the Composition Editor of all Web Connection parts expects the location of the application graphics to be C:\WTA\WTAPix. You may later move all other WTA directories to a different drive, however, you should keep the WTAPix directory on the C: disk.

Java installations require a file system that supports long file names, not the typical 8+3 characters for file name and extension (such as in the FAT file system). Java source files have a file type of `.java` and the compiled byte code has an extension of `.class`. If you want to run the Java applet that comes with the WTA application, verify that at least the Java directory is stored on a file system such as HPFS. If you decide to move the Java directory, make sure you have a `Pass` directive in HTTPD.CNF pointing to it.

<table>
<thead>
<tr>
<th>Directory</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>WTA\EXE\WIN</td>
<td>WTA runtime files for VisualAge Smalltalk for Windows</td>
</tr>
<tr>
<td>WTA\HTML</td>
<td>HTML file for WTA C++ implementation</td>
</tr>
<tr>
<td>WTA\HTML\CRC</td>
<td>CRC cards in HTM format</td>
</tr>
<tr>
<td>WTA\HTML\JAVA</td>
<td>Java application for WTA</td>
</tr>
<tr>
<td>WTA\HTML\MEDIA</td>
<td>Movies for the View option of WTA</td>
</tr>
<tr>
<td>WTA\HTML\PROTOTYPE</td>
<td>Various prototype pages for WTA</td>
</tr>
<tr>
<td>WTA\LIBRARY\OS2</td>
<td>Smalltalk library for OS/2</td>
</tr>
<tr>
<td>WTA\LIBRARY\WIN</td>
<td>Smalltalk library for Windows</td>
</tr>
<tr>
<td>WTA\BIND</td>
<td>Bind files for WTA C++ implementation</td>
</tr>
<tr>
<td>WTA\CGI-BIN</td>
<td>CGI programs for WTA C++ implementation</td>
</tr>
<tr>
<td>WTA\WORKING</td>
<td>WTA C++ source code</td>
</tr>
<tr>
<td>WTA\WTAPIX</td>
<td>Images for WTA</td>
</tr>
<tr>
<td>WTA\PRESENTATION</td>
<td>WTA screenplay HTML source</td>
</tr>
</tbody>
</table>

We recommend that you install the WTA directory tree on your C: drive, because the Composition Editor of all Web Connection parts expects the location of the application graphics to be C:\WTA\WTAPix. You may later move all other WTA directories to a different drive, however, you should keep the WTAPix directory on the C: disk.

Java installations require a file system that supports long file names, not the typical 8+3 characters for file name and extension (such as in the FAT file system). Java source files have a file type of `.java` and the compiled byte code has an extension of `.class`. If you want to run the Java applet that comes with the WTA application, verify that at least the Java directory is stored on a file system such as HPFS. If you decide to move the Java directory, make sure you have a `Pass` directive in HTTPD.CNF pointing to it.
VisualAge for Smalltalk

For the Smalltalk implementation of the WTA application, VisualAge for Smalltalk requires to have these features installed:

- Database feature
- Web Connection feature
- Fixes from the \VISUALAG\INSTALL\FILEINS directory

Use the VisualAge import function to load the WTA application into the library, the WTA\Library\OS2 and WTA\Library\WIN directories contain the application for both OS/2 and Windows 95/NT. Then, load the application into your development image.

Verify that the appropriate CGI Link Server program for your platform is copied from the \VISUALAG\CGILINK directory to the CGI-BIN directory of your Internet Connection Server. For detailed instructions, refer to “Web Connection” on page 378.

VisualAge for C++ and Waikiki

To use the C++ implementation of the WTA application, you need to install VisualAge for C++ first with the Data Access Builder features. Then you can install Waikiki. Both products are located in the first CDROM which come with this book.

Refer to “Installing VisualAge for C++ for Windows” on page 385, “Installing Waikiki” on page 386 and the WTA readme file for details information about setting the products and the WTA application.

Database 2

Before you install the Web Travel Agent database, make sure you have a local user ID on your system named USERID. The database installer and the application expects USERID to be the owner of the WTADB database. If you use a non-US version of DB2, you might not have a USERID/PASSWORD default administrator on your system, for example, a German installation typically has a default administrator ID and password of BENUTZER and KENNWORT. In this case, create a new local administrator ID called USERID before proceeding.

If you want to create the WTA database on OS/2, you can run the DBBUILD.EXE program from the \WTA\DB2 directory. The program allows you to select a drive letter for the installation of the database (Figure 312).
The database builder creates the WTADB database, all tables, and a few records in each table. The two command files, EXPORT.CMD and IMPORT.CMD, allow you to unload and load the entire database using IXF files.

If you install the WTADB database on Windows, you can refer to the readme file which is installed with WTA. This file details each step to create the database using the DDL files located in \WTA\DB2.

Before the WTA application can access the newly created database, you must bind it to the database. Use the following command for binding with the Smalltak implementation of WTA:

```
SQLBIND D:\VISUALAG\ABT\ABTDC30.BND WTADB
```

Use the batch file WTABIND.BAT in the \WTA\BIND directory for binding the C++ implementation against the database. The readme file of WTA details each step of the binding process.

You have to execute the binding commands only once for your database. However, if you later decide to re-create the WTADB database, you need to rebind the program against your database by issuing again the commands.
Internet Connection Server

To direct the Internet Connection Server to the location of the WTA files, such as graphics, HTML, Java applets, you need to add two lines to the HTTPD configuration file. The HTTPD.CNF file resides in the ETC directory on your system.

Search in the file for the Pass directives, and before the last line, that usually reads something like

```
Pass /* D:WWW\HTML\*
```

add the two following lines:

```
Pass /wta/* D:WTA\HTML\*
Pass /wtapix/* D:WTA\WTAPIX\*
```

For the C++ implementation you also need to add one line for accessing the CGI programs:

```
Exec /wta/cgi-bin/* D:WTA\CGI-BIN\*
```

Restart the Internet Connection Server to notify the HTTP daemon of the new directories.
13

Packaging a Web Application

In this chapter we show you how to package the Web application that you develop with VisualAge for Smalltalk or VisualAge for C++.

VisualAge for Smalltalk

After you have finished developing and testing your application, you can package it into a run-time executable for installation on your server. By packaging your application, you can avoid having to put the entire VisualAge development environment on your HTTP server after your application is in production. Packaging a Web Connection application into a run-time executable is basically the same as packaging any VisualAge application, but there are a few extra things you need to do.
Creating a Run-time Image

Packaging a Web Connection application is essentially the same as packaging any VisualAge application. To package your application, follow these steps:

1. Make sure you have versioned and released the classes in your application, and that you have listed any required prerequisites.

2. In the VisualAge Organizer, select the application you want to package. Then select **Make Executable** from the Applications menu, or from the pop-up menu in the Applications list.

3. Because a Web Connection application contains no ordinary visual parts, VisualAge asks if you want to continue with packaging (see Figure 313). Select **Yes**.

4. Confirm that Make Executable is selected (see Figure 314), and make any necessary corrections to the list of prerequisite applications. When everything is correct, select **OK** to initiate packaging.
5. If your application is rather large, it may take a while for VisualAge to find all used methods and classes. That is why you must confirm the packaging process (see Figure 315). You may have time to go for a coffee.

![Figure 315. Prompter: Packaging Time](image1)

6. VisualAge now asks you for a file name for your packaged application. We chose WTA36.EXE for Web Travel Agent Version 3.6 (see Figure 316), thus your image file will be called WTA36.IMG.

![Figure 316. Save Packaged Image](image2)

7. If you have packaged and run your application more than once, you are likely to receive two warnings (see Figure 317) that the files RGB.TXT and CURSORS.OBJ already exist on your output directory. Click on Yes to replace both files.

![Figure 317. Prompter: Existing Files](image3)
8. VisualAge may return a window with packaging problems (see Figure 318). In most cases, you can ignore those messages because they relate to some missing implementers of TCP classes. Close the window.

![Figure 318. Packaging Problems](image1)

9. The last window of the packaging routine (see Figure 319) has actually been updated all along during the process. The packaging log reveals information, for example, about the removal of unused classes and methods from the image, the size of the image, or the time used to package. We recommend that you save the log for future reference (Figure 320).

![Figure 319. Packaging Messages](image2)
Image file name is: C:\WTA\exe\wta36.img.
Adding Applications...
Packaging the following applications:
  WtaApp 3.6
Calculating minimum components
Finding unused methods...
Found 15476 unused methods.
Found 547 unused classes.
Time taken was 00:01:49
Removing unused application classes...
Finding unreferenced globals...
Finding pool dictionaries to reduce...
Minimum components calculated
Finding problems...
Finding packaging problems...
  Filtering method symbol arguments...
Done finding packaging problems.
Packaging continues...
Image file name is: C:\WTA\exe\wta36.img
AbtRunStrings reduced from 39 entries to 20 entries
No entries found for pool dictionary: NisCatCWa
AbtCgiLinkServerViewStrings reduced from 21 entries to 19 entries
NisCatCW reduced from 23 entries to 14 entries
No entries found for pool dictionary: NisCatNLSa
No entries found for pool dictionary: PlatformGlobals
No entries found for pool dictionary: NisCatEWc
NisCatDWcPM reduced from 10 entries to 9 entries
NisCatESd reduced from 23 entries to 11 entries
No entries found for pool dictionary: NisCatEa
No entries found for pool dictionary: NisCatCGa
PlatformConstants reduced from 5126 entries to 418 entries
NisCatESW reduced from 9 entries to 6 entries
NisCatGPsPM reduced from 6 entries to 5 entries
No entries found for pool dictionary: NisCatCWcsPMWin
AbtFeatureLoadPool reduced from 5 entries to 0 entries
No entries found for pool dictionary: NisCatCGsPMa
AbtMsgCoop reduced from 40 entries to 2 entries
No entries found for pool dictionary: AbtRunHtmlPageStrings
No entries found for pool dictionary: NisCatCWc
NisCatKRN reduced from 32 entries to 23 entries
AbtVaMsgBase reduced from 59 entries to 6 entries
NisCatCFS reduced from 11 entries to 10 entries
No entries found for pool dictionary: NisCatCG
No entries found for pool dictionary: NisCatEA
No entries found for pool dictionary: NisCatCgPSPM
AbtCommMRI reduced from 55 entries to 0 entries
No entries found for pool dictionary: NisCatEPb
No entries found for pool dictionary: NisCatEa
NisCatCP reduced from 11 entries to 5 entries
NisCatEM reduced from 16 entries to 12 entries
No entries found for pool dictionary: AbtCgiLinkStrings
AbtIbmDbmPlatformFunctions reduced from 37 entries to 16 entries
AbtPrimitiveNLSStrings reduced from 37 entries to 16 entries
NisCatESTa reduced from 55 entries to 9 entries
Size of resulting image: 2609780
Time taken was 00:03:34

Figure 320. Packaging Log
Setting up the Run-time System

Once you have packaged the run-time image of your application, you can install it (along with the other required VisualAge files) on your server. Refer to the VisualAge for Smalltalk User's Guide for more information about the files required on a run-time system. In addition to the files required for any VisualAge application, a Web Connection application also requires the following files at run-time:

- **Required .mpr files:**
  - abtwce30.mpr
  - abtwve30.mpr
  - abtwre30.mpr
  - abtcpe30.mpr

- **Minimum required .cat files:**
  - cfs.cat
  - esta.cat
  - esw.cat
  - cp.cat
  - krn.cat
  - esd.cat
  - cw_e.cat
  - em.cat
  - cpswin.cat

- **TCP/IP files:**
  - abttcp30.dll (OS/2 and Windows)
  - abttcp30.w (AIX)

Remember that you must also set up the CGI Link program and its configuration file on your HTTP server. See “Setting Up the CGI Link” on page 378 for information on setting up the CGI Link.

Start a Web Connection application just as you would start any packaged VisualAge application. As the application starts, VisualAge prompts you for the startup parameters for the CGI Link server, just as when you start the server in a development environment. After you specify the CGI Link server parameters, the application loads and is ready to respond to incoming requests. You will not see any windows on the local screen except for the CGI Link server window.

If you prefer not to be prompted at application startup, you can use optional command-line arguments to specify the CGI Link server start-up parameters. (These same arguments are also valid when starting a development image.) The complete syntax for the optional Web Connection command-line arguments is as follows:

```
abt -wc:noprompt  or
abt -wc:port:[host]:[on|off]
```
Replace *abt* with the name of the executable file you created during packaging. Following is an explanation of the valid command-line arguments:

**noprompt**

Specifies that you do not want to be prompted for the CGI Link server startup parameters, and you also do not want to specify the parameters on the command line. If you use the *noprompt* argument, you must start the CGI Link server explicitly in your application, typically in the startup code. See “Starting the CGI Link Within an Application” on page 426 for more information on how to do this.

**port**

Stands for the port number that the CGI Link uses to relay CGI queries to VisualAge. The CGI Link server process will listen to this port. If you use the command-line arguments, *port* is required; if you fail to specify the port number, VisualAge will prompt you for it during startup.

**hostname**

This is the host name or IP address you want the CGI Link server to bind to. Use this option only if your VisualAge system has multiple TCP/IP addresses, and you want to specify which one will be used for incoming Web Connection requests.

**on|off**

Specifies the auto-refresh value for the CGI Link server window.

Each of these command-line arguments is optional; however, if you skip one of them, you must still include the colon for the missing argument. Following are some examples of valid command lines:

```
abt -wc:8081
```

Starts the CGI Link server on port 8081.

```
abt -wc:8081:server.almaden.ibm.com:on
```

Starts the CGI Link server on port 8081, binding to hostname *server.almaden.ibm.com*, and turns auto-refresh on.

```
abt -wc:8081::off
```

Starts the CGI Link server on port 8081 and turns auto-refresh off; does not specify a preferred host to bind to.

**Note:** If you want to start multiple CGI Link servers (on different ports), you can do so by specifying multiple *-wc* strings (separated by spaces) on the command line.
**Starting the CGI Link Within an Application**

The class `AbtCgiLinkServerView` provides an interface you can use to start the CGI Link server in a Smalltalk method, typically in your application’s startup code. You can use this technique if you want your application to start the CGI Link server automatically, without having to specify the startup parameters on the command line.

To start the CGI Link server from Smalltalk, use the following Smalltalk method:

```smalltalk
AbtCgiLinkServerView startFromSpec: aString
```

`aString` is a string containing the startup parameters for the server, exactly as you would type them on the command line (not including the `-wc` flag). For example, to start the server on port 8081, binding to hostname `server.almaden.ibm.com`, and turning auto-refresh on, you would use the following code:

```smalltalk
AbtCgiLinkServerView startFromSpec: '8081:server.almaden.ibm.com:on'
```

---

**VisualAge for C++**

Packaging your VisualAge for C++ Web application is simple and does not differ from packaging a standard VisualAge for C++ application. In the following section we describe how to install the application files on your Web server to have your application ready to run.

**Setting up the Run-time System**

After you have finished developing and testing your application, you end up with four type of run-time files:

- **HTML files** are the static Web pages of your application which are retrieved by the Web server from the appropriate directory.
- **Image files** are the logos or graphic files (GIF, JPEG) you create to enhance your application.
- **Executable files** are your CGI programs which are called by your Web server and should be also located in an appropriate directory.
- **Dynamic link libraries** are complementary libraries that your CGI programs might need to perform specific functions. Often they are located in the same directory as the CGI programs.
Most Web servers in the market place can be set up using configuration files similarly to the httpd.cnf configuration file of the IBM Internet Connection Server (see Chapter 12, “Installation and Setup,” on page 385). To set up the run-time environment of your Web application, these configuration files define different directories where the application CGI programs and static Web pages should be located. For the Web Travel Agent application, you direct the Web server to locate your files by providing the following Pass and Exec directives:

```
Pass /wta/* D:\WTA\HTML\*
Pass /wtapix/* D:\WTA\WTAPIX\*
Exec /wta/cgi-bin/* D:\WTA\CGI-BIN\*
```

The first directive directs the Web server to retrieve the HTML files, referenced at /wta, in the directory D:\WTA\HTML. Place your static Web pages in this directory.

The second directive directs the Web server to retrieve your picture files, referenced at /wtapix, in the directory D:\WTA\WTAPIX. Place your GIF files in this directory.

The last directive directs the Web server to call the CGI programs, referenced at /wta/cgi-bin, in the directory D:\WTA\CGI-BIN. Place your executable files and their related dynamic link libraries in this directory.

**Running Environment**

When the Web server calls your CGI programs, they run in the environment in which your Web server runs. This is the reason why you must be sure that from the environment the Web server runs from, all libraries necessary to run your CGI programs can be accessed. For example, the CGI programs created with Waikiki need to access specific libraries located in predefined directories. A specific DOS session is defined for running your CGI programs and the Web server must be started from this session as follows:

1. From the Windows Start menu access the Waikiki menu item and start the related DOS session.
2. From the DOS session change to the directory where your Web server is installed and start your Web server.

Once your Web server is started in the appropriate running environment, your CGI programs can run properly.
This appendix shows the Class-Responsibility-Collaborator cards as they were used during the development of the Web Travel Agent application. The online version of the CRC cards is also available by viewing the CRC.HTML file on the enclosed CDROM. The advantage of viewing the HTML version is that the entries in the Collaborator column are links to the collaborating class.
WelcomeView

Table 15. WelcomeView

<table>
<thead>
<tr>
<th>Responsibilities</th>
<th>Collaborators</th>
</tr>
</thead>
<tbody>
<tr>
<td>shows welcome message</td>
<td></td>
</tr>
<tr>
<td>allows selection of one of</td>
<td></td>
</tr>
<tr>
<td>create a package</td>
<td>WelcomeProcessor</td>
</tr>
<tr>
<td>list existing packages</td>
<td></td>
</tr>
<tr>
<td>modify personal information</td>
<td></td>
</tr>
<tr>
<td>see attractions in various cities</td>
<td></td>
</tr>
<tr>
<td>transfers control</td>
<td>AboutView</td>
</tr>
</tbody>
</table>

WelcomeProcessor

Table 16. WelcomeProcessor

<table>
<thead>
<tr>
<th>Responsibilities</th>
<th>Collaborators</th>
</tr>
</thead>
<tbody>
<tr>
<td>receives customer selection</td>
<td>WelcomeView (Form Data)</td>
</tr>
<tr>
<td>stores service type requested by customer</td>
<td>RequestedPackage</td>
</tr>
<tr>
<td>transfers control depending on selected service</td>
<td></td>
</tr>
<tr>
<td>create a package</td>
<td>PackageCreationProcessor</td>
</tr>
<tr>
<td>list existing packages</td>
<td>CustomerAuthenticationView</td>
</tr>
<tr>
<td>modify personal information</td>
<td>CustomerAuthenticationView</td>
</tr>
<tr>
<td>see attractions in cities</td>
<td>AttractionView</td>
</tr>
</tbody>
</table>

PackageCreationController

Table 17. PackageCreationController

<table>
<thead>
<tr>
<th>Responsibilities</th>
<th>Collaborators</th>
</tr>
</thead>
<tbody>
<tr>
<td>initializes data</td>
<td>RequestedPackage</td>
</tr>
<tr>
<td>transfers control</td>
<td>PackageCreationView</td>
</tr>
</tbody>
</table>
PackageCreationView

Table 18. PackageCreationView

<table>
<thead>
<tr>
<th>Responsibilities</th>
<th>Collaborators</th>
</tr>
</thead>
<tbody>
<tr>
<td>displays locations</td>
<td>Database</td>
</tr>
<tr>
<td>allows location selection</td>
<td></td>
</tr>
<tr>
<td>allows input of</td>
<td></td>
</tr>
<tr>
<td>starting date</td>
<td></td>
</tr>
<tr>
<td>trip length in days</td>
<td></td>
</tr>
<tr>
<td>displays error message if present</td>
<td>RequestedPackage</td>
</tr>
<tr>
<td>submits customer input</td>
<td>PackageCreationProcessor</td>
</tr>
</tbody>
</table>

PackageCreationProcessor

Table 19. PackageCreationProcessor

<table>
<thead>
<tr>
<th>Responsibilities</th>
<th>Collaborators</th>
</tr>
</thead>
<tbody>
<tr>
<td>receives customer input</td>
<td>PackageCreationView (Form Data)</td>
</tr>
<tr>
<td>checks for valid date in month and year</td>
<td></td>
</tr>
<tr>
<td>checks that start date is today or in future</td>
<td></td>
</tr>
<tr>
<td>checks that duration is at least one day</td>
<td></td>
</tr>
<tr>
<td>creates start date</td>
<td>RequestedPackage</td>
</tr>
<tr>
<td>creates end date</td>
<td>RequestedPackage</td>
</tr>
<tr>
<td>creates error message if necessary</td>
<td>RequestedPackage</td>
</tr>
<tr>
<td>transfers control depending on</td>
<td></td>
</tr>
<tr>
<td>no input errors</td>
<td>ComponentSelectionView</td>
</tr>
<tr>
<td>input errors</td>
<td>PackageCreationView</td>
</tr>
</tbody>
</table>
### ComponentSelectionView

**Table 20.** ComponentSelectionView

<table>
<thead>
<tr>
<th>Responsibilities</th>
<th>Collaborators</th>
</tr>
</thead>
<tbody>
<tr>
<td>shows</td>
<td></td>
</tr>
<tr>
<td></td>
<td>destination</td>
</tr>
<tr>
<td></td>
<td>duration</td>
</tr>
<tr>
<td></td>
<td>start date</td>
</tr>
<tr>
<td></td>
<td>end date</td>
</tr>
<tr>
<td></td>
<td>RequestedPackage</td>
</tr>
</tbody>
</table>

allows selection of

|                  |                      |
| flight preferences input |
| hotel preferences input  |
| rental car preferences input |
| price check            |
| cancellation            |
| ComponentSelectionProcessor |

### ComponentSelectionProcessor

**Table 21.** ComponentSelectionProcessor

<table>
<thead>
<tr>
<th>Responsibilities</th>
<th>Collaborators</th>
</tr>
</thead>
<tbody>
<tr>
<td>receives customer input</td>
<td>ComponentSelectionView (Form Data)</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>transfers control depending on selection of</td>
<td></td>
</tr>
<tr>
<td>flight preferences input</td>
<td>FlightSelectionView</td>
</tr>
<tr>
<td>hotel preferences input</td>
<td>HotelSelectionView</td>
</tr>
<tr>
<td>rental car preferences input</td>
<td>CarSelectionView</td>
</tr>
<tr>
<td>price check</td>
<td>PackagePricer</td>
</tr>
<tr>
<td>cancellation</td>
<td>PackageCreationView</td>
</tr>
</tbody>
</table>
### FlightSelectionView

<table>
<thead>
<tr>
<th>Responsibilities</th>
<th>Collaborators</th>
</tr>
</thead>
<tbody>
<tr>
<td>displays departure cities</td>
<td>Database</td>
</tr>
<tr>
<td>allows selection of</td>
<td></td>
</tr>
<tr>
<td>departure city</td>
<td></td>
</tr>
<tr>
<td>seat class</td>
<td></td>
</tr>
<tr>
<td>allows input of number of seats</td>
<td></td>
</tr>
<tr>
<td>displays error message if present</td>
<td>RequestedPackage</td>
</tr>
<tr>
<td>submits customer input</td>
<td>FlightSelectionProcessor</td>
</tr>
</tbody>
</table>

### FlightSelectionProcessor

<table>
<thead>
<tr>
<th>Responsibilities</th>
<th>Collaborators</th>
</tr>
</thead>
<tbody>
<tr>
<td>receives customer input</td>
<td>FlightSelectionView</td>
</tr>
<tr>
<td>checks for valid number of seats</td>
<td>(Form Data)</td>
</tr>
<tr>
<td>stores</td>
<td></td>
</tr>
<tr>
<td>number of seats</td>
<td>RequestedFlightSeats</td>
</tr>
<tr>
<td>seat class</td>
<td></td>
</tr>
<tr>
<td>departure city</td>
<td></td>
</tr>
<tr>
<td>stores</td>
<td></td>
</tr>
<tr>
<td>flight number</td>
<td>Flight</td>
</tr>
<tr>
<td>departure city</td>
<td></td>
</tr>
<tr>
<td>arrival city</td>
<td></td>
</tr>
<tr>
<td>base price</td>
<td></td>
</tr>
<tr>
<td>creates error message if necessary</td>
<td>RequestedPackage</td>
</tr>
<tr>
<td>transfers control depending on</td>
<td></td>
</tr>
<tr>
<td>no input errors</td>
<td>ComponentSelectionView</td>
</tr>
<tr>
<td>input errors</td>
<td>FlightSelectionView</td>
</tr>
</tbody>
</table>
**Flight**

<table>
<thead>
<tr>
<th>Responsibilities</th>
<th>Collaborators</th>
</tr>
</thead>
<tbody>
<tr>
<td>knows departure city</td>
<td></td>
</tr>
<tr>
<td>knows arrival city</td>
<td></td>
</tr>
<tr>
<td>knows flight number</td>
<td></td>
</tr>
<tr>
<td>knows price of an economy seat</td>
<td></td>
</tr>
</tbody>
</table>

**RequestedFlightSeats**

<table>
<thead>
<tr>
<th>Responsibilities</th>
<th>Collaborators</th>
</tr>
</thead>
<tbody>
<tr>
<td>knows flight number</td>
<td></td>
</tr>
<tr>
<td>knows package ID</td>
<td></td>
</tr>
<tr>
<td>knows seat class</td>
<td></td>
</tr>
<tr>
<td>knows number of seats</td>
<td></td>
</tr>
<tr>
<td>knows price</td>
<td></td>
</tr>
</tbody>
</table>

**HotelSelectionView**

<table>
<thead>
<tr>
<th>Responsibilities</th>
<th>Collaborators</th>
</tr>
</thead>
<tbody>
<tr>
<td>displays hotels in selected city</td>
<td>Database</td>
</tr>
<tr>
<td>allows selection of room category</td>
<td></td>
</tr>
<tr>
<td>smoking room</td>
<td></td>
</tr>
<tr>
<td>allows input of number of rooms</td>
<td></td>
</tr>
<tr>
<td>displays error message if present</td>
<td>RequestedPackage</td>
</tr>
<tr>
<td>submits customer input</td>
<td>HotelSelectionProcessor</td>
</tr>
</tbody>
</table>
### HotelSelectionProcessor

<table>
<thead>
<tr>
<th>Responsibilities</th>
<th>Collaborators</th>
</tr>
</thead>
<tbody>
<tr>
<td>receives customer input</td>
<td>HotelSelectionView (Form Data)</td>
</tr>
<tr>
<td>checks for valid number of rooms</td>
<td></td>
</tr>
<tr>
<td>stores</td>
<td>number of rooms</td>
</tr>
<tr>
<td></td>
<td>room category</td>
</tr>
<tr>
<td></td>
<td>smoking or nonsmoking</td>
</tr>
<tr>
<td></td>
<td>hotel ID</td>
</tr>
<tr>
<td>stores</td>
<td>hotel name</td>
</tr>
<tr>
<td></td>
<td>hotel ID</td>
</tr>
<tr>
<td></td>
<td>location name</td>
</tr>
<tr>
<td></td>
<td>daily price</td>
</tr>
<tr>
<td></td>
<td>hotel capacity</td>
</tr>
<tr>
<td></td>
<td>number of rooms booked</td>
</tr>
<tr>
<td>creates error message if necessary</td>
<td>RequestedPackage</td>
</tr>
<tr>
<td>transfers control depending on</td>
<td>no input errors</td>
</tr>
<tr>
<td></td>
<td>input errors</td>
</tr>
</tbody>
</table>

### Hotel

<table>
<thead>
<tr>
<th>Responsibilities</th>
<th>Collaborators</th>
</tr>
</thead>
<tbody>
<tr>
<td>knows hotel name</td>
<td></td>
</tr>
<tr>
<td>knows hotel ID</td>
<td></td>
</tr>
<tr>
<td>knows hotel city</td>
<td></td>
</tr>
<tr>
<td>knows hotel capacity</td>
<td></td>
</tr>
<tr>
<td>knows number of rooms booked</td>
<td></td>
</tr>
<tr>
<td>knows daily price of a standard room</td>
<td></td>
</tr>
</tbody>
</table>
### RequestedHotelRooms

**Table 29.** RequestedHotelRooms

<table>
<thead>
<tr>
<th>Responsibilities</th>
<th>Collaborators</th>
</tr>
</thead>
<tbody>
<tr>
<td>knows package ID</td>
<td></td>
</tr>
<tr>
<td>knows number of nights</td>
<td></td>
</tr>
<tr>
<td>knows hotel name</td>
<td></td>
</tr>
<tr>
<td>knows hotel ID</td>
<td></td>
</tr>
<tr>
<td>knows room category</td>
<td></td>
</tr>
<tr>
<td>knows number of rooms</td>
<td></td>
</tr>
<tr>
<td>knows price</td>
<td></td>
</tr>
<tr>
<td>knows smoking or nonsmoking</td>
<td></td>
</tr>
<tr>
<td>knows availability status</td>
<td></td>
</tr>
</tbody>
</table>

### CarSelectionView

**Table 30.** CarSelectionView

<table>
<thead>
<tr>
<th>Responsibilities</th>
<th>Collaborators</th>
</tr>
</thead>
<tbody>
<tr>
<td>allows selection of car class</td>
<td>RequestedCar</td>
</tr>
<tr>
<td></td>
<td>car rental company</td>
</tr>
<tr>
<td>browse car classes</td>
<td>CarClassView</td>
</tr>
<tr>
<td>submits customer input</td>
<td>CarSelectionProcessor</td>
</tr>
</tbody>
</table>

### CarSelectionProcessor

**Table 31.** (Part 1 of 2) CarSelectionProcessor

<table>
<thead>
<tr>
<th>Responsibilities</th>
<th>Collaborators</th>
</tr>
</thead>
<tbody>
<tr>
<td>receives customer input</td>
<td>CarSelectionView (Form Data)</td>
</tr>
<tr>
<td>checks for valid input</td>
<td></td>
</tr>
</tbody>
</table>
### Table 31. (Part 2 of 2) CarSelectionProcessor

<table>
<thead>
<tr>
<th>Responsibilities</th>
<th>Collaborators</th>
</tr>
</thead>
<tbody>
<tr>
<td>stores car class</td>
<td>RequestedCar</td>
</tr>
<tr>
<td>rental car company</td>
<td></td>
</tr>
<tr>
<td>creates error message if necessary</td>
<td>RequestedPackage</td>
</tr>
<tr>
<td>transfers control depending on no input errors</td>
<td>ComponentSelectionView</td>
</tr>
<tr>
<td></td>
<td>CarSelectionView</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### CarClassView

<table>
<thead>
<tr>
<th>Responsibilities</th>
<th>Collaborators</th>
</tr>
</thead>
<tbody>
<tr>
<td>allows selection of car class</td>
<td>RequestedCar</td>
</tr>
<tr>
<td>submits customer input</td>
<td>CarSelectionView</td>
</tr>
</tbody>
</table>

### RequestedCar

<table>
<thead>
<tr>
<th>Responsibilities</th>
<th>Collaborators</th>
</tr>
</thead>
<tbody>
<tr>
<td>knows package ID</td>
<td></td>
</tr>
<tr>
<td>knows car class</td>
<td></td>
</tr>
<tr>
<td>knows car rental company</td>
<td></td>
</tr>
<tr>
<td>knows number of rental days</td>
<td></td>
</tr>
<tr>
<td>knows price</td>
<td></td>
</tr>
</tbody>
</table>
### PackagePricer

<table>
<thead>
<tr>
<th>Responsibilities</th>
<th>Collaborators</th>
</tr>
</thead>
<tbody>
<tr>
<td>knows base price of a standard hotel room</td>
<td>Hotel</td>
</tr>
<tr>
<td>knows markup for deluxe rooms and suites</td>
<td></td>
</tr>
<tr>
<td>knows daily price for all car classes</td>
<td></td>
</tr>
<tr>
<td>knows base price of an economy flight seat</td>
<td>Flight</td>
</tr>
<tr>
<td>knows markup for business and first-class seats</td>
<td></td>
</tr>
<tr>
<td>computes flight price</td>
<td>RequestedFlightSeats</td>
</tr>
<tr>
<td>computes hotel price</td>
<td>Hotel</td>
</tr>
<tr>
<td>computes car price</td>
<td>RequestedHotelRooms</td>
</tr>
<tr>
<td>computes total package price</td>
<td>RequestedCar</td>
</tr>
<tr>
<td>transfers control depending on package unavailable</td>
<td>PackageFailedView</td>
</tr>
<tr>
<td>depending on package available</td>
<td>PackagePurchaseView</td>
</tr>
</tbody>
</table>

### PackageFailedView

<table>
<thead>
<tr>
<th>Responsibilities</th>
<th>Collaborators</th>
</tr>
</thead>
<tbody>
<tr>
<td>lists information for an unavailable package</td>
<td>RequestedPackage</td>
</tr>
<tr>
<td></td>
<td>RequestedHotelRooms</td>
</tr>
<tr>
<td></td>
<td>RequestedFlightSeats</td>
</tr>
<tr>
<td></td>
<td>RequestedCar</td>
</tr>
<tr>
<td>allows selection to transfer control</td>
<td>ComponentSelectionView</td>
</tr>
</tbody>
</table>
## PackagePurchaseView

**Table 36.** PackagePurchaseView

<table>
<thead>
<tr>
<th>Responsibilities</th>
<th>Collaborators</th>
</tr>
</thead>
<tbody>
<tr>
<td>lists flight information</td>
<td>RequestedFlightSeats</td>
</tr>
<tr>
<td></td>
<td>Flight</td>
</tr>
<tr>
<td>lists hotel information</td>
<td>RequestedHotelRooms</td>
</tr>
<tr>
<td></td>
<td>Hotel</td>
</tr>
<tr>
<td>lists rental car information</td>
<td>RequestedCar</td>
</tr>
</tbody>
</table>

- shows
  - destination
  - duration
  - start date
  - end date
  - price
  - RequestedPackage

- allows selection of
  - package purchase
  - return to change components

- transfers control
  - PackagePurchaseProcessor

## PackagePurchaseProcessor

**Table 37.** PackagePurchaseProcessor

<table>
<thead>
<tr>
<th>Responsibilities</th>
<th>Collaborators</th>
</tr>
</thead>
<tbody>
<tr>
<td>receives customer input</td>
<td>PackagePurchaseView (Form Data)</td>
</tr>
<tr>
<td>transfers control depending on selection of</td>
<td>package purchase</td>
</tr>
<tr>
<td></td>
<td>CustomerIdView</td>
</tr>
<tr>
<td></td>
<td>return to change components</td>
</tr>
<tr>
<td></td>
<td>ComponentSelectionView</td>
</tr>
</tbody>
</table>
## RequestedPackage

### Table 38. RequestedPackage

<table>
<thead>
<tr>
<th>Responsibilities</th>
<th>Collaborators</th>
</tr>
</thead>
<tbody>
<tr>
<td>knows customer ID</td>
<td></td>
</tr>
<tr>
<td>knows location name</td>
<td></td>
</tr>
<tr>
<td>knows package ID</td>
<td></td>
</tr>
<tr>
<td>knows package price</td>
<td></td>
</tr>
<tr>
<td>knows start date</td>
<td></td>
</tr>
<tr>
<td>knows end date</td>
<td></td>
</tr>
<tr>
<td>knows number of days</td>
<td></td>
</tr>
<tr>
<td>knows package price</td>
<td></td>
</tr>
<tr>
<td>knows error messages</td>
<td></td>
</tr>
<tr>
<td>knows departure city</td>
<td></td>
</tr>
</tbody>
</table>

## CustomerIdView

### Table 39. CustomerIdView

<table>
<thead>
<tr>
<th>Responsibilities</th>
<th>Collaborators</th>
</tr>
</thead>
<tbody>
<tr>
<td>allows indication of whether customer is new or existing</td>
<td>CustomerIdProcessor</td>
</tr>
</tbody>
</table>

## CustomerIdProcessor

### Table 40. CustomerIdProcessor

<table>
<thead>
<tr>
<th>Responsibilities</th>
<th>Collaborators</th>
</tr>
</thead>
<tbody>
<tr>
<td>receives customer input</td>
<td>CustomerIdView (Form Data)</td>
</tr>
<tr>
<td>transfers control depending on</td>
<td>existing customer</td>
</tr>
<tr>
<td></td>
<td>new customer</td>
</tr>
<tr>
<td></td>
<td>CustomerAuthenticationView</td>
</tr>
<tr>
<td></td>
<td>NewCustomerIdView</td>
</tr>
</tbody>
</table>

## NewCustomerIdView

440
**Table 41. NewCustomerIdView**

<table>
<thead>
<tr>
<th>Responsibilities</th>
<th>Collaborators</th>
</tr>
</thead>
<tbody>
<tr>
<td>allows entry of</td>
<td>name</td>
</tr>
<tr>
<td></td>
<td>address</td>
</tr>
<tr>
<td></td>
<td>phone number</td>
</tr>
<tr>
<td>transfers control</td>
<td>NewCustomerIdProcessor</td>
</tr>
</tbody>
</table>

**NewCustomerIdProcessor**

**Table 42. NewCustomerIdProcessor**

<table>
<thead>
<tr>
<th>Responsibilities</th>
<th>Collaborators</th>
</tr>
</thead>
<tbody>
<tr>
<td>checks for input of</td>
<td>name</td>
</tr>
<tr>
<td></td>
<td>address</td>
</tr>
<tr>
<td></td>
<td>phone number</td>
</tr>
<tr>
<td>creates error message if necessary</td>
<td>RequestedPackage</td>
</tr>
<tr>
<td>retrieves highest customer number</td>
<td>Database</td>
</tr>
<tr>
<td>creates new customer number</td>
<td>Requested Package</td>
</tr>
<tr>
<td>stores customer information</td>
<td>Customer</td>
</tr>
<tr>
<td>transfers control depending on</td>
<td>no input errors</td>
</tr>
<tr>
<td></td>
<td>input errors</td>
</tr>
<tr>
<td></td>
<td>CreditCardInfoView</td>
</tr>
<tr>
<td></td>
<td>NewCustomerIdView</td>
</tr>
</tbody>
</table>

**Committer**

**Table 43. (Part 1 of 2) Committer**

<table>
<thead>
<tr>
<th>Responsibilities</th>
<th>Collaborators</th>
</tr>
</thead>
<tbody>
<tr>
<td>retrieves highest package number</td>
<td>Database</td>
</tr>
<tr>
<td>generates new package number</td>
<td></td>
</tr>
<tr>
<td>stores new package number</td>
<td>Requested Package</td>
</tr>
<tr>
<td>Responsibilities</td>
<td>Collaborators</td>
</tr>
<tr>
<td>------------------</td>
<td>---------------</td>
</tr>
<tr>
<td>stores</td>
<td></td>
</tr>
<tr>
<td>customer ID</td>
<td>RequestedPackageDatabase</td>
</tr>
<tr>
<td>package ID</td>
<td></td>
</tr>
<tr>
<td>departure city</td>
<td></td>
</tr>
<tr>
<td>arrival city</td>
<td></td>
</tr>
<tr>
<td>package price</td>
<td></td>
</tr>
<tr>
<td>start date</td>
<td></td>
</tr>
<tr>
<td>end date</td>
<td></td>
</tr>
<tr>
<td>duration</td>
<td></td>
</tr>
<tr>
<td>credit card information</td>
<td></td>
</tr>
<tr>
<td>stores</td>
<td></td>
</tr>
<tr>
<td>number of seats</td>
<td>RequestedFlightSeatsDatabase</td>
</tr>
<tr>
<td>seat class</td>
<td></td>
</tr>
<tr>
<td>flight number</td>
<td></td>
</tr>
<tr>
<td>flight price</td>
<td></td>
</tr>
<tr>
<td>package ID</td>
<td></td>
</tr>
<tr>
<td>stores</td>
<td></td>
</tr>
<tr>
<td>number of rooms</td>
<td>RequestedHotelRoomsDatabase</td>
</tr>
<tr>
<td>room category</td>
<td></td>
</tr>
<tr>
<td>smoking or nonsmoking</td>
<td></td>
</tr>
<tr>
<td>hotel ID</td>
<td></td>
</tr>
<tr>
<td>number of nights</td>
<td></td>
</tr>
<tr>
<td>hotel price</td>
<td></td>
</tr>
<tr>
<td>number of rooms booked</td>
<td></td>
</tr>
<tr>
<td>package ID</td>
<td></td>
</tr>
<tr>
<td>stores</td>
<td></td>
</tr>
<tr>
<td>car class</td>
<td>RequestedCarDatabase</td>
</tr>
<tr>
<td>rental car company</td>
<td></td>
</tr>
<tr>
<td>rental car price</td>
<td></td>
</tr>
<tr>
<td>package ID</td>
<td></td>
</tr>
</tbody>
</table>
### CreditCardInfoView

**Table 44. CreditCardInfoView**

<table>
<thead>
<tr>
<th>Responsibilities</th>
<th>Collaborators</th>
</tr>
</thead>
<tbody>
<tr>
<td>shows customer name</td>
<td>RequestedPackage</td>
</tr>
<tr>
<td></td>
<td>Customer</td>
</tr>
<tr>
<td>allows selection of credit card company</td>
<td>RequestedPackage</td>
</tr>
<tr>
<td>allows input of credit card number</td>
<td></td>
</tr>
<tr>
<td>allows input of expiration date</td>
<td></td>
</tr>
</tbody>
</table>

### PackageConfirmationView

**Table 45. PackageConfirmationView**

<table>
<thead>
<tr>
<th>Responsibilities</th>
<th>Collaborators</th>
</tr>
</thead>
<tbody>
<tr>
<td>shows thank you message</td>
<td>RequestedPackage</td>
</tr>
<tr>
<td></td>
<td>Customer</td>
</tr>
<tr>
<td>lists customer information</td>
<td>RequestedFlightSeats</td>
</tr>
<tr>
<td></td>
<td>Flight</td>
</tr>
<tr>
<td>lists flight information</td>
<td>RequestedHotelRooms</td>
</tr>
<tr>
<td></td>
<td>Hotel</td>
</tr>
<tr>
<td>lists hotel information</td>
<td></td>
</tr>
<tr>
<td>lists rental car information</td>
<td>RequestedCar</td>
</tr>
<tr>
<td>shows destination</td>
<td>RequestedPackage</td>
</tr>
<tr>
<td>duration</td>
<td></td>
</tr>
<tr>
<td>start date</td>
<td></td>
</tr>
<tr>
<td>end date</td>
<td></td>
</tr>
<tr>
<td>package ID</td>
<td></td>
</tr>
<tr>
<td>price</td>
<td></td>
</tr>
<tr>
<td>allows return to welcome screen</td>
<td>WelcomeView</td>
</tr>
</tbody>
</table>

Appendix A. CRC Cards 443
**CustomerAuthenticationView**

<table>
<thead>
<tr>
<th>Responsibilities</th>
<th>Collaborators</th>
</tr>
</thead>
<tbody>
<tr>
<td>requests entry of name</td>
<td></td>
</tr>
<tr>
<td>requests entry of customer ID</td>
<td></td>
</tr>
<tr>
<td>displays error message if present</td>
<td>RequestedPackage</td>
</tr>
<tr>
<td>transfers control</td>
<td>CustomerAuthenticationProc.</td>
</tr>
</tbody>
</table>

**CustomerAuthenticationProcessor**

<table>
<thead>
<tr>
<th>Responsibilities</th>
<th>Collaborators</th>
</tr>
</thead>
<tbody>
<tr>
<td>checks for input of</td>
<td></td>
</tr>
<tr>
<td>customer name</td>
<td>CustomerAuthenticationView</td>
</tr>
<tr>
<td>customer ID</td>
<td>(Form Data)</td>
</tr>
<tr>
<td>creates error message if necessary</td>
<td>RequestedPackage</td>
</tr>
<tr>
<td>retrieves customer information</td>
<td>Database</td>
</tr>
<tr>
<td>stores customer information</td>
<td>Customer</td>
</tr>
<tr>
<td>transfers control based on customer request</td>
<td>PackageListView</td>
</tr>
<tr>
<td></td>
<td>PersonalInfoView</td>
</tr>
<tr>
<td></td>
<td>CreditCardInfoView</td>
</tr>
</tbody>
</table>

**PersonalInfoView**

<table>
<thead>
<tr>
<th>Responsibilities</th>
<th>Collaborators</th>
</tr>
</thead>
<tbody>
<tr>
<td>allows entry of</td>
<td></td>
</tr>
<tr>
<td>address</td>
<td></td>
</tr>
<tr>
<td>phone number</td>
<td></td>
</tr>
<tr>
<td>displays error message</td>
<td>RequestedPackage</td>
</tr>
</tbody>
</table>
Table 48. (Part 2 of 2) PersonalInfoView

<table>
<thead>
<tr>
<th>Responsibilities</th>
<th>Collaborators</th>
</tr>
</thead>
<tbody>
<tr>
<td>allows selection of</td>
<td>update information</td>
</tr>
<tr>
<td></td>
<td>return home</td>
</tr>
<tr>
<td>transfers control</td>
<td>PersonalInfoProcessor</td>
</tr>
</tbody>
</table>

**PersonalInfoProcessor**

Table 49. PersonalInfoProcessor

<table>
<thead>
<tr>
<th>Responsibilities</th>
<th>Collaborators</th>
</tr>
</thead>
<tbody>
<tr>
<td>checks for input of</td>
<td>address</td>
</tr>
<tr>
<td></td>
<td>phone number</td>
</tr>
<tr>
<td>creates error message if necessary</td>
<td>RequestedPackage</td>
</tr>
<tr>
<td>stores updated customer information</td>
<td>Customer</td>
</tr>
<tr>
<td></td>
<td>Database</td>
</tr>
<tr>
<td>transfers control based on service selected</td>
<td>update</td>
</tr>
<tr>
<td></td>
<td>PersonalInfoView</td>
</tr>
<tr>
<td></td>
<td>return home</td>
</tr>
<tr>
<td></td>
<td>WelcomeView</td>
</tr>
</tbody>
</table>

**Customer**

Table 50. Customer

<table>
<thead>
<tr>
<th>Responsibilities</th>
<th>Collaborators</th>
</tr>
</thead>
<tbody>
<tr>
<td>knows ID</td>
<td></td>
</tr>
<tr>
<td>knows name</td>
<td></td>
</tr>
<tr>
<td>knows address</td>
<td></td>
</tr>
<tr>
<td>knows telephone number</td>
<td></td>
</tr>
</tbody>
</table>
### PackageListView

**Table 51.** PackageListView

<table>
<thead>
<tr>
<th>Responsibilities</th>
<th>Collaborators</th>
</tr>
</thead>
<tbody>
<tr>
<td>lists all existing packages for a customer</td>
<td>RequestedPackage</td>
</tr>
<tr>
<td>allows selection of one of the listed packages</td>
<td>PackageDetailsView</td>
</tr>
</tbody>
</table>

### PackageListProcessor

**Table 52.** (Part 1 of 2) PackageListProcessor

<table>
<thead>
<tr>
<th>Responsibilities</th>
<th>Collaborators</th>
</tr>
</thead>
<tbody>
<tr>
<td>initialize package data</td>
<td>RequestedPackage</td>
</tr>
<tr>
<td>queries for package data</td>
<td>Database</td>
</tr>
</tbody>
</table>

stores

<table>
<thead>
<tr>
<th>stores</th>
<th>RequestedPackage</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

stores

<table>
<thead>
<tr>
<th>stores</th>
<th>RequestedFlightSeats</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

stores

<table>
<thead>
<tr>
<th>stores</th>
<th>Flight</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

stores

<table>
<thead>
<tr>
<th></th>
<th>Flight</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Table 52. (Part 2 of 2) PackageListProcessor

<table>
<thead>
<tr>
<th>Responsibilities</th>
<th>Collaborators</th>
</tr>
</thead>
<tbody>
<tr>
<td>number of rooms</td>
<td>RequestedHotelRooms</td>
</tr>
<tr>
<td>room category</td>
<td></td>
</tr>
<tr>
<td>smoking or nonsmoking</td>
<td></td>
</tr>
<tr>
<td>hotel ID</td>
<td></td>
</tr>
<tr>
<td>number of nights</td>
<td></td>
</tr>
<tr>
<td>hotel price</td>
<td></td>
</tr>
<tr>
<td>number of rooms booked</td>
<td></td>
</tr>
<tr>
<td>package ID</td>
<td></td>
</tr>
</tbody>
</table>

### PackageDetailsView

<table>
<thead>
<tr>
<th>Responsibility</th>
<th>Collaborators</th>
</tr>
</thead>
<tbody>
<tr>
<td>lists customer information</td>
<td>RequestedPackage</td>
</tr>
<tr>
<td></td>
<td>Customer</td>
</tr>
<tr>
<td>lists flight information</td>
<td>RequestedFlightSeats</td>
</tr>
<tr>
<td></td>
<td>Flight</td>
</tr>
</tbody>
</table>
### Table 53. (Part 2 of 2) PackageDetailsView

<table>
<thead>
<tr>
<th>Responsibilities</th>
<th>Collaborators</th>
</tr>
</thead>
<tbody>
<tr>
<td>lists hotel information</td>
<td>RequestedHotelRooms</td>
</tr>
<tr>
<td></td>
<td>Hotel</td>
</tr>
<tr>
<td>lists rental car information</td>
<td>RequestedCar</td>
</tr>
<tr>
<td>shows</td>
<td>RequestedPackage</td>
</tr>
<tr>
<td>destination</td>
<td></td>
</tr>
<tr>
<td>duration</td>
<td></td>
</tr>
<tr>
<td>start date</td>
<td></td>
</tr>
<tr>
<td>end date</td>
<td></td>
</tr>
<tr>
<td>package ID</td>
<td></td>
</tr>
<tr>
<td>price</td>
<td></td>
</tr>
<tr>
<td>transfers control</td>
<td>PackageListView</td>
</tr>
</tbody>
</table>

### AttractionView

#### Table 54. AttractionView

<table>
<thead>
<tr>
<th>Responsibilities</th>
<th>Collaborators</th>
</tr>
</thead>
<tbody>
<tr>
<td>allows location selection</td>
<td>Location</td>
</tr>
<tr>
<td>allows to</td>
<td></td>
</tr>
<tr>
<td>watch a video of the location</td>
<td></td>
</tr>
<tr>
<td>visit Web site of the location</td>
<td></td>
</tr>
<tr>
<td>transfers control</td>
<td>AttractionProcessor</td>
</tr>
</tbody>
</table>

### AttractionProcessor

#### Table 55. AttractionProcessor

<table>
<thead>
<tr>
<th>Responsibilities</th>
<th>Collaborators</th>
</tr>
</thead>
<tbody>
<tr>
<td>retrieves location data</td>
<td>Location</td>
</tr>
<tr>
<td>stores location</td>
<td>RequestedPackage</td>
</tr>
<tr>
<td>transfers control to external URL</td>
<td></td>
</tr>
<tr>
<td>returns video file</td>
<td></td>
</tr>
</tbody>
</table>
## Location

**Table 56. Location**

<table>
<thead>
<tr>
<th>Responsibilities</th>
<th>Collaborators</th>
</tr>
</thead>
<tbody>
<tr>
<td>knows location name</td>
<td></td>
</tr>
<tr>
<td>knows related video file</td>
<td></td>
</tr>
<tr>
<td>knows URL of related Web page</td>
<td></td>
</tr>
</tbody>
</table>

## AboutView

**Table 57. AboutView**

<table>
<thead>
<tr>
<th>Responsibilities</th>
<th>Collaborators</th>
</tr>
</thead>
<tbody>
<tr>
<td>describes residency project</td>
<td></td>
</tr>
<tr>
<td>shows application prototype</td>
<td></td>
</tr>
<tr>
<td>shows CRC cards</td>
<td></td>
</tr>
<tr>
<td>shows project members</td>
<td>DeveloperView</td>
</tr>
<tr>
<td>links to ITSO Web pages</td>
<td></td>
</tr>
<tr>
<td>returns home</td>
<td>WelcomeView</td>
</tr>
</tbody>
</table>

## DeveloperView

**Table 58. DeveloperView**

<table>
<thead>
<tr>
<th>Responsibilities</th>
<th>Collaborators</th>
</tr>
</thead>
<tbody>
<tr>
<td>shows pictures of project team members</td>
<td></td>
</tr>
<tr>
<td>returns home</td>
<td>WelcomeView</td>
</tr>
</tbody>
</table>
Database Definition

In this appendix we list the DDL commands that are used to build the database and show some samples how to create database records.
CREATE TABLE USERID.CUSTOMER (CUSTID INTEGER NOT NULL,  
    CUST_NAME CHAR(40) NOT NULL,  
    ADDRESS CHAR(50) NOT NULL,  
    PHONE CHAR(20) NOT NULL,  
    PRIMARY KEY (CUSTID));

CREATE TABLE USERID.LOCATION (LOC_NAME CHAR(20) NOT NULL,  
    VIDEO_FILE CHAR(20) NOT NULL,  
    URL CHAR(254) NOT NULL,  
    PRIMARY KEY (LOC_NAME));

CREATE TABLE USERID.TPACKAGE (CUSTID INTEGER NOT NULL,  
    PACKID INTEGER NOT NULL,  
    LOC_NAME CHAR(20) NOT NULL,  
    PRICE INTEGER NOT NULL,  
    START_DATE DATE NOT NULL,  
    END_DATE DATE NOT NULL,  
    DAYS SMALLINT NOT NULL,  
    CARD_COMPANY CHAR(20) NOT NULL,  
    CARD_NB CHAR(20) NOT NULL,  
    EXP_DATE DATE NOT NULL,  
    PRIMARY KEY (PACKID),  
    FOREIGN KEY(LOC_NAME)  
    REFERENCES USERID.LOCATION,  
    FOREIGN KEY(CUSTID)  
    REFERENCES USERID.CUSTOMER  
    ON DELETE CASCADE);

CREATE TABLE USERID.CAR (MODEL CHAR(10),  
    MAKE CHAR(10) NOT NULL,  
    CAR_CLASS CHAR(1) NOT NULL,  
    STOCK SMALLINT NOT NULL,  
    BOOKED SMALLINT NOT NULL,  
    DAILY_PRICE SMALLINT NOT NULL,  
    PICTURE CHAR(20));
To install the WTADB database on OS/2, you can use the provided DBBUILD.EXE program from the C:\WTA\DB2 directory. Alternatively, you can use the command prompt to create the database and insert the sample records into the tables. Use the following commands from the C:\WTA\DB2 directory:

```
DB2 CREATE DATABASE WTADB
DB2 CONNECT TO WTADB
DB2 -t -f CREATE.DDL
DB2 -t -f UPDATE.DDL
```
Here are some sample SQL commands that you can use to fill the database tables.

```
SQL

INSERT INTO CUSTOMER VALUES(100, 'John Doe', 'Albuquerque', '123-555-6789');

INSERT INTO LOCATION VALUES('Chicago', 'CHICAGO.AVI', 'http://www.ci.chi.il.us');

INSERT INTO FLIGHT VALUES('AA1157', 'Chicago', 'San Jose', 285);

INSERT INTO HOTEL VALUES('Hyatt Embarcadero', 700301, 'San Francisco', 600, 200, 280);

INSERT INTO CAR VALUES('325i', 'BMW', 'D', 5, 0, 48, 'BMW.GIF');
```
The WtaDeveloperView part contains a little Java applet to demonstrate the ability of VisualAge for Smalltalk to include Java as an HTML sub-part. The Composition Editor does not show anything else but a black box in the size of the Java applet, because the real code lies outside in a Java-compiled class. The Java source to the Guys.class file is shown below.
import java.lang.;
import java.awt.;
import java.applet.Applet;

public class Guys extends Applet
{
  Image pic;
  Button next, prev;

  String guypix[] = { "andi.jpg", "joe.jpg", "chris.jpg",
                     "marc.jpg", "bruno.jpg", "bernard.jpg" };

  String guynames[] = { "Andi Bitterer", "Joe Bianco", "Chris Bosman-Clark",
                        "Marc Carrel-Billard", "Bruno Georges", "Bernard Tsang" };

  String country[] = { "Germany", "USA", "USA", "France", "France", "Hong Kong" };

  String job1[] = { "Project Leader Smalltalk and", "Parts Development and",
                   "Smalltalk Plumbing and", "Project Leader C++ and",
                   "Object Design C++ and", "Rapid Prototyping and" };

  String job2[] = { "Application Architecture", "User Interface Design",
                   "Database Access", "Database Architecture",
                   "Database Design", "DB2WWW Gateway" };

  String name, jpg, ctry, line1, line2;

  int guy = 99;
  boolean ini = true;

  public void init() {
    setLayout(null);

    next = new Button("Next");
    prev = new Button("Previous");

    add(next);
    add(prev);

    next.reshape(295,315,90,50);
    prev.reshape(395,315,90,50);

    resize(600, 400);
  } // end init()
public void paint(Graphics g) {
  Font large = new Font("Helvetica", Font.BOLD, 24);
  Font normal = new Font("Helvetica", Font.PLAIN, 14);
  Font bold = new Font("Helvetica", Font.BOLD, 18);
  if (ini) {
    g.setFont(bold);
    g.drawString("This Java applet introduces", 20, 50);
    g.drawString("the team behind this application.", 20, 70);
    g.setFont(normal);
    g.drawString("Please press any button.", 20, 315);
    ini = false;
  }
  g.drawImage(pic, 10, 10, 271, 360, this);
  g.setFont(large);
  g.drawString(name, 295, 30);
  g.setFont(normal);
  g.drawString(ctry, 295, 50);
  g.setFont(bold);
  g.drawString(line1, 295, 100);
  g.drawString(line2, 295, 120);
}

public boolean action(Event e, Object arg) {
  if (guy == 99) {
    guy = 0;
  } else {
    if ("Previous".equals(arg)) {
      if (--guy == -1) {
        guy = 5;
      } else {
        name = guysname[guy];
        jpg = guyspix[guy];
        ctry = country[guy];
        line1 = job1[guy];
        line2 = job2[guy];
        pic = getImage(getCodeBase(), "pix/" + jpg);
        showStatus("Guy: "+ name + "," + ctry);
        repaint();
        return true;
      }
    } else if ("Next".equals(arg)) {
      if (++guy == 6) {
        guy = 0;
      } else {
        name = guysname[guy];
        jpg = guyspix[guy];
        ctry = country[guy];
        line1 = job1[guy];
        line2 = job2[guy];
        pic = getImage(getCodeBase(), "pix/" + jpg);
        showStatus("Guy: "+ name + "," + ctry);
        repaint();
        return true;
      }
    }
  }
}

The Web Travel Agency application was originally developed on OS/2 Warp; however, it is easily portable to Windows 95 and Windows NT. Few modifications were necessary to produce an Windows version, because the VisualAge DB2 interface on Windows 95 and NT is different from OS/2.

**Database 2 (DB2)**

On Windows NT, two services need to be started in order to use the Web Travel Agent. Apart from starting DB2, you must also start the DB2 security service, because DB2 otherwise cannot authenticate your user profile when you try to access the WTADB database. Use the Services icon in the Control Panel folder (Figure 321) to verify that both DB2 services are defined and started (see Figure 322). We recommend that you configure both services to start automatically at system startup.
If the security server is not started, and you connect with a user ID and password, you will get an SQL code of either 1042 or 1402. In either case, the corrective action is to start the security server, for example, through the command line:

```
NET START DB2NTSECSERVER.
```

**Figure 321. Windows NT Control Panel**

![Windows NT Control Panel](image1.png)

**Figure 322. Windows NT Services**

**Internet Connection Server**

While the HTTPD.CNF configuration file on OS/2 resides usually in the `\TCPIP\ETC` or `\MPTN\ETC` directory (run `SET ETC` from an OS/2 prompt to find the correct location), on Windows NT you find it in the `\WINDOWS` directory. The file looks no different otherwise, make the necessary changes as described in “Waikiki” on page 385.

**VisualAge**

The built-in database feature of VisualAge for Smalltalk uses the call level interface (CLI) to access DB2 on Windows NT. That requires a change to the VisualAge database connection specifications (see Figure 323).
The above change is reflected in the `DbConnectionAlias` method which is generated automatically. Additionally, the `startUp` method of the application needs to be modified to reflect the new database manager class. Instead of the `AbtDatabaseManager` class we have to use the `AbtCliDatabaseManager` class.
startUp
"The image is being loaded. Perform any necessary initialization.
Namely, bring up the WtaDb with the password and userid provided by a script.
This is a primitive aid to unattended operations."

"The following is from, 'VisualAge Features Class Guide', section 1.2.2.3
'Establishing Database Connections'."

"Connect to a database using a connection specification
and a logon specification. This example does not prompt
you with a database logon window."

| conSpec logonSpec |

conSpec := AbtDatabaseConnectionSpec
  forDbmClass: #AbtIbmCliDatabaseManager
  databaseName: 'WTADB'.

logonSpec := AbtDatabaseLogonSpec
  id: 'USERID'
  password: 'password'
  server: nil.

conSpec connectUsingAlias: 'DbConnectionAlias'
  logonSpec: logonSpec.

conSpec promptEnabled: false.

^AbtDbmSystem activeDatabaseConnection
Web Connection Classes

The Web Connection feature provides programming interfaces through which advanced users can access low-level features from Smalltalk. These interfaces are not necessary for most Web Connection applications, but can be helpful in some specialized situations. By using the Smalltalk interfaces, you can access detailed CGI information, control the creation of session data, and generate HTML.

Accessing CGI Request Information

Several classes are involved in handling an incoming CGI request. When a Web Connection part is instantiated as part of a request, an AbtCgiLink-Transaction object is created and associated with the part. The AbtCgiLink-Transaction instance contains references to two other objects, an AbtCgiLinkRequest object that models the CGI request data, and an AbtCgiLinkResponse object that models the CGI response data. These classes provide methods you can use to access various data associated with the CGI request. Following is a description of these classes and their methods.
Session Data

**Instance Methods for AbtAppBldrHtmlPage**

This class is the superclass of all Web Connection parts:

**transaction**

Answers the instance of `AbtCgiLinkTransaction` associated with the request that caused the part to be instantiated.

**Instance Methods for AbtCgiLinkTransaction**

**request**

Answers the instance of `AbtCgiLinkRequest` associated with the transaction.

**response**

Answers the instance of `AbtCgiLinkResponse` associated with the transaction.

**Instance Methods for AbtCgiLinkRequest**

**cgiVars**

Answer a `LookupTable` that contains CGI variable names as keys, and the values of those variables as values. The content received via standard input from the CGI program is available in the CGI variable XCGILINK_CONTENT_DATA.

**formDict**

Answer a `LookupTable` with form data fields as keys, and the value of those form data fields as the values. Note that the values will always be an `OrderedCollection` of `Strings`.

**Instance Methods for AbtCgiLinkResponse**

**content: aString1 contentType: aString2**

Set the content and Content-type of the response. `aString1` should be a `String` or `ByteArray`. `aString2` should be a valid MIME type (such as text/html).

**Session Data**

If your part does not need to use session data, you can implement the class method `canGenerateSessionKey` to return `false`. If you do this, a session key will not be generated when the part is requested.
The *AbtCgiLinkTransaction* class implements two methods to return session data and the session key associated with the transaction.

**sessionData**

Answers the session data object associated with the transaction.

**sessionDataKey**

Answers a string which is the key associated with this session data.

---

**Generating HTML from Smalltalk**

The Web Connection feature provides a stream class called *AbtHtmlWriteStream*. This stream is a subclass of *WriteStream* and implements the following instance methods:

**nextPutAll: aString**

Writes *aString* to the stream, but writes HTML entities for the characters `<`, `>`, `"`, and `&`.

**nextPutAllLiteral: aString**

Writes *aString* to the stream without translations of special characters to HTML entities.

---

**Error Handling with AbtCgiLinkServer Class Methods**

**errorHandler: aClass**

When an exception occurs during the processing of a request, the following processing occurs:

- The class specified by the *errorHandler: method is used to generate the output for the page. This class should be a normal Web Connection part. The processing that occurs for this page is the same as if the page had been requested normally.
- After the error page is generated and sent back to the server, the exception is signaled, which will generally bring up a debugger. In packaged run-time images, the exception is not signaled.

**errorHandler**

Answers the Web Connection part class set by the *errorHandler: method.
notFoundHandler: aClass

When a request is received that corresponds to a part not in the current image, the class specified by the notFoundHandler: method is used to generate the output for the page. This class should be a normal Web Connection part. The processing for this page is the same as if the page had been requested normally.

notFoundHandler

Answers the Web Connection part class set by the notFoundHandler: method.

---

## Codepage and Execution Control

### Class Methods for AbtCgiLinkServer

**externalCodePage: aString**

HTML is defined to use the ISO 8859-1 character set. However, on some platforms and environments on which VisualAge runs, this is not the character set in use. This can cause problems primarily for national characters, which are defined at different code points in the native code page and the ISO 8859-1 character set. To alleviate this problem, all input coming into VisualAge is translated from ISO 8859-1 (named ’ibm-819’ in VisualAge) into the native code page. Likewise, output is translated from the native code into ISO 8859-1.

The externalCodePage: method can be used to change the code page from and to which input and output is translated. This string should be in the same form as returned from the following method:

```
Smalltalk
AbtNLSCoordinator currentCharacterSet
```

To disable the translation, the current character set as returned above should be used as the parameter on the externalCodePage: method.

**externalCodePage**

Answers the string set by the externalCodePage method.
Class Methods for AbtAbtBldrHtmlPage

transactionMustBeSerialized

Each request received from the server is processed in a new Smalltalk process, which implies that multiple requests may be processed concurrently within an image. If this causes a problem because the processing your part does is not reentrant, the transactionMustBeSerialized method can be implemented on your part to return true. The CGI Link server ensures that at most one Web Connection part that answers false to transactionMustBeSerialized is running at a time.
Related Publications

The publications listed in this section are considered particularly suitable for a more detailed discussion of the topics covered in this redbook.

International Technical Support Organization Publications

For information on ordering these ITSO publications see “How To Get ITSO Redbooks” on page 473.

- VisualAge: Concepts and Features, GG24-3946
- VisualAge and Transaction Processing in a Client/Server Environment, GG24-4487
- AS/400 Application Development with VisualAge for Smalltalk, SG24-2535
- VisualAge: Building GUIs for Existing Applications, GG24-4244
- VisualAge for Smalltalk Distributed, SG24-4521
- VisualAge for Smalltalk and SOMObjects, SG24-4390
Redbooks on CD-ROMs

- OO Programming with Client Access for OS/400 and ODBC using VisualAge for Smalltalk, SG24-4718
- Object-Oriented Application Development with VisualAge for C++ for OS/2, SG24-2593
- Client Server Programming with VisualAge C++ for OS/400, SG24-4660
- Programming with VisualAge for C++ for Windows, SG24-4782
- IBM VisualAge for Cobol for OS/2: Primer, SG24-4605
- IBM VisualAge for Cobol for OS/2: Object-Oriented Programming, SG24-4606
- Visual Modeling Technique, Object Technology using Visual Programming, SG24-4227

Redbooks on CD-ROMs

Redbooks are also available on CD-ROMs. **Order a subscription** and receive updates 2-4 times a year at significant savings.

<table>
<thead>
<tr>
<th>CD-ROM Title</th>
<th>Subscription Number</th>
<th>Collection Kit Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>System/390 Redbooks Collection</td>
<td>SBOF-7201</td>
<td>SK2T-2177</td>
</tr>
<tr>
<td>Networking and Systems Management Redbooks Collection</td>
<td>SBOF-7370</td>
<td>SK2T-6022</td>
</tr>
<tr>
<td>Transaction Processing and Data Management Redbook</td>
<td>SBOF-7240</td>
<td>SK2T-8038</td>
</tr>
<tr>
<td>AS/400 Redbooks Collection</td>
<td>SBOF-7270</td>
<td>SK2T-2849</td>
</tr>
<tr>
<td>RISC System/6000 Redbooks Collection (HTML, BkMgr)</td>
<td>SBOF-7230</td>
<td>SK2T-8040</td>
</tr>
<tr>
<td>RISC System/6000 Redbooks Collection (PostScript)</td>
<td>SBOF-7205</td>
<td>SK2T-8037</td>
</tr>
<tr>
<td>Application Development Redbooks Collection</td>
<td>SBOF-7290</td>
<td>SK2T-8042</td>
</tr>
<tr>
<td>Personal Systems Redbooks Collection</td>
<td>SBOF-7250</td>
<td>SK2T-8042</td>
</tr>
</tbody>
</table>

Other Publications

These publications are also relevant as further information sources:

- VisualAge for Smalltalk User’s Guide, SC34-4518
- VisualAge for Smalltalk User’s Reference, SC34-4519
- VisualAge for Smalltalk Programmer’s Guide to Building Parts for Fun and Profit, SC34-4496
- Introduction to Object-Oriented Programming with IBM Smalltalk, SC34-4491
- IBM Smalltalk Programmer’s Reference, SC34-4493
- IBM Smalltalk User’s Guide, SC34-4536
Other Publications

Other Publications

- *Effective C++: 50 Specific Ways to Improve Your Programs and Designs*, by S. Meyers, Addison-Wesley, 1992
How To Get ITSO Redbooks

This section explains how both customers and IBM employees can find out about ITSO redbooks, CD-ROMs, workshops, and residencies.

This information was current at the time of publication, but is continually subject to change. The latest information may be found at URL http://www.redbooks.ibm.com.

How IBM Employees Can Get ITSO Redbooks

Employees may request ITSO deliverables (redbooks, BookManager BOOKs, and CD-ROMs) and information about redbooks, workshops, and residencies in the following ways:

- **PUBORDER** — to order hardcopies in United States
- **GOPHER link to the Internet**
  - type GOPHER.WTSCPOK.ITSO.IBM.COM
- **Tools disks**
  To get LIST3820s of redbooks, type one of the following commands:
  
  TOOLS SENDTO EHONE4 TOOLSZ REDPRINT GET SG24xxxx PACKAGE
  TOOLS SENDTO CANVM2 TOOLS REDPRINT GET SG24xxxx PACKAGE (Canadian users only)
  
  To get lists of redbooks:

  TOOLS SENDTO USDIST MKTTOOLS MKTTOOLS GET ITSOCAT TXT
  TOOLS SENDTO USDIST MKTTOOLS MKTTOOLS GET LISTSERV PACKAGE
  
  To register for information on workshops, residencies, and redbooks:

  TOOLS SENDTO WTSCPOK TOOLS ZDISK GET ITSOREGI 1996
  
  For a list of product area specialists in the ITSO:

  TOOLS SENDTO WTSCPOK TOOLS ZDISK GET ORGCARD PACKAGE

- **IBM Direct Publications Catalog on the World Wide Web**
  
  
  IBM employees may obtain LIST3820s of redbooks from this page.

- **REDBOOKS category on INEWS**

- **Online** — send orders to: USIB6FPL at IBMMAIL or DKIBMBSH at IBMMAIL
How Customers Can Get ITSO Redbooks

Customers may request ITSO deliverables (redbooks, BookManager BOOKs, and CD-ROMs) and information about redbooks, workshops, and residencies in the following ways:

- **Online Orders** (Do not send credit card information over the Internet) — send orders to:
  
  In United States: IBMMAIL
  In Canada: caibmbkz at ibmmail
  Outside North America: dkiibmbsh at ibmmail
  Internet: usib6fpl@ibmmail.com
  lmannix@vnet.ibm.com
  bookshop@dk.ibm.com

- **Telephone orders**
  
  United States (toll free) 1-800-879-2755
  Canada (toll free) 1-800-IBM-4YOU
  Outside North America (long distance charges apply)
  (+45) 4810-1320 - Danish
  (+45) 4810-1420 - Dutch
  (+45) 4810-1540 - English
  (+45) 4810-1670 - Finnish
  (+45) 4810-1220 - French
  Outside North America
  (+45) 4810-1320 - Danish
  (+45) 4810-1420 - Dutch
  (+45) 4810-1540 - English
  Outside North America
  (+45) 4810-1220 - French

- **Mail Orders** — send orders to:
  
  IBM Publications
  Publications Customer Support
  P.O. Box 29570
  Raleigh, NC 27626-0570
  USA
  IBM Direct Services
  Sortemosevej 21
  DK-3450 Allerød
  Denmark

- **Fax** — send orders to:
  
  United States (toll free) 1-800-445-9269
  Canada 1-403-267-4455
  Outside North America (+45) 48 14 2207
  (long distance charge)
  Outside North America
  (+45) 48 10 2207

- **1-800-IBM-4FAX (United States) or (+1) 415 855 43 29 (Outside USA)** — ask for:
  
  Index # 4421 Abstracts of new redbooks
  Index # 4422 IBM redbooks
  Index # 4420 Redbooks for last six months

- **Direct Services** - send note to softwareshop@vnet.ibm.com

- **On the World Wide Web**
  
  Redbooks Home Page http://www.redbooks.ibm.com

- **Internet Listserver**
  
  With an Internet E-mail address, anyone can subscribe to an IBM Announcement Listserv. To initiate the service, send an E-mail note to announce@webster.ibmlink.ibm.com with the keyword subscribe in the body of the note (leave the subject line blank).
Special Notices

This publication is intended to help VisualAge for Smalltalk and VisualAge for C++ developers to create World Wide Web applications. The information in this publication is not intended as the specification of any programming interfaces that are provided by VisualAge for Smalltalk, VisualAge for C++, Database 2, or IBM Internet Connection Server. See the PUBLICATIONS section of the IBM Programming Announcement for VisualAge for Smalltalk, VisualAge for C++, Database 2, and IBM Internet Connection Server for more information about what publications are considered to be product documentation.

References in this publication to IBM products, programs or services do not imply that IBM intends to make these available in all countries in which IBM operates. Any reference to an IBM product, program, or service is not intended to state or imply that only IBM’s product, program, or service may be used. Any functionally equivalent program that does not infringe any of IBM’s intellectual property rights may be used instead of the IBM product, program or service. Information in this book was developed in conjunction with use of the equipment specified, and is limited in application to those specific hardware and software products and levels.

IBM may have patents or pending patent applications covering subject matter 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 the IBM Director of Licensing, IBM Corporation, 500 Columbus Avenue, Thornwood, NY 10594 USA.

The information contained in this document has not been submitted to any formal IBM test and is distributed AS IS. The use of this information or the implementation of any of these techniques is a customer responsibility and depends on the customer’s ability to evaluate and integrate them into the customer’s operational environment. While each item may have been reviewed by IBM for accuracy in a specific situation, there is no guarantee that the same or similar results will be obtained elsewhere. Customers attempting to adapt these techniques to their own environments do so at their own risk.

References in this publication to IBM products, programs or services do not imply that IBM intends to make these available in all countries in which IBM operates. Any reference to an IBM product, program, or service is not intended to state or imply that only IBM’s product, program, or service may be used. Any functionally equivalent program that does not infringe any of IBM’s intellectual property rights may be used instead of the IBM product, program or service. Information in this book was developed in conjunction with use of the equipment specified, and is limited in application to those specific hardware and software products and levels.
IBM may have patents or pending patent applications covering subject matter 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 the IBM Director of Licensing, IBM Corporation, 500 Columbus Avenue, Thornwood, NY 10594 USA.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact IBM Corporation, Dept. 600A, Mail Drop 1329, Somers, NY 10589 USA.

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The information contained in this document has not been submitted to any formal IBM test and is distributed AS IS. The use of this information or the implementation of any of these techniques is a customer responsibility and depends on the customer’s ability to evaluate and integrate them into the customer’s operational environment. While each item may have been reviewed by IBM for accuracy in a specific situation, there is no guarantee that the same or similar results will be obtained elsewhere. Customers attempting to adapt these techniques to their own environments do so at their own risk.

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

- CICS
- DB2
- IBM
- Multimedia Presentation Manager/2
- Presentation Manager
- VisualAge
- WebExplorer
- DATABASE 2
- DB2/2
- IMS
- OS/2
- PS/2
- WebConnection

The following terms are trademarks of other companies:

- UNIX is a registered trademark in the United States and other countries licensed exclusively through X/Open Company Limited.
- Microsoft, Windows, Windows NT, and the Windows 95 logo are trademarks or registered trademarks of Microsoft Corporation.
- Sun, Java, JDBC, and HotJava are trademarks of Sun Microsystems, Inc.
- Netscape is a registered trademark of Netscape Communications Corporation.
- Lotus is a registered trademark of Lotus Development Corporation.

Other trademarks are trademarks of their respective companies.
Glossary

A

abstract class. A class that provides common behavior across a set of subclasses but is not itself designed to have instances that work. An abstract class represents a concept; classes derived from it represent implementations of the concept. For example, IControl is the abstract base class for control view windows; the ICanvas and IListBox classes are controls derived from IControl. An abstract class must have at least one pure virtual function. See also base class.

access. A property of a class that determines whether a class member is accessible in an expression or declaration.

accessor methods. Methods that an object provides to define the interface to its instance variables. The accessor method to return the value of an instance variable is often called a get method or getter method, and the accessor method to assign a value to an instance variable is called a set method or setter method.

access plan. The control structure produced during compile time that is used to process SQL statements encountered when the program is run.

action. In VisualAge, a function or operation that a part can perform upon receiving a message. Actions enable a part’s public interface to give other parts access to its behaviors. Compare with event and attribute.

advanced program-to-program communication (APPC). (1) IBM’s architected solution for program-to-program communication, distributed transaction processing, and remote database access. A transaction program (TP) using the APPC API can communicate with other TPs on systems that support APPC. (2) An implementation of the Systems Network Architecture (SNA) logical unit (LU) 6.2 protocol that enables interconnected systems to communicate and share the processing of programs.

anonymous FTP. Using the FTP function of the Internet anonymously by not logging in with an actual, secret login ID and password. Often permitted by large, host computers that are willing to share openly some of the files on their system to outside users who otherwise would not be able to log in. See also FTP.

API. Application program interface. An architected functional interface supplied by an operating system or other software system. The interface enables an application program written in a high-level language to use specific data or functions of the underlying system.

application. (1) The use to which an information processing system is put; for example, a payroll application or an order-entry application. (2) A collection of defined and extended classes that provides a reusable piece of functionality. An application contains and organizes functionally related classes. It also can contain subapplications and specify prerequisites.

application manager. (1) A team member who is responsible for the overall state of an application. An application manager coordinates the activities of the application’s developers and assigns ownership of classes to team members. (2) The browser from which users can create, delete, manage, or configure applications in their image.

application program interface (API). An architected functional interface supplied by an operating system or other software system. The interface enables an application program written in a high-level language to use specific data or functions of the underlying system.

Archie. An Internet tool for finding files stored on anonymous FTP sites. You need to know the exact file name or a substring of it.

argument. A data element included as part of a message. Arguments provide additional information that the receiver can use to per-
form the requested operation. Binary messages and keyword messages take arguments. In a keyword message, a colon (:) following a keyword indicates that an argument is required.

**ARPANet.** (Advanced Research Projects Agency Network), the precursor to the Internet. Developed in the late 1960s and early 1970s by the US Department of Defense as an experiment in wide-area-networking that would survive a nuclear war. See also Internet.

**array literal.** A literal that is an indexed sequence of literals. The symbol # precedes this sequence and parentheses enclose the sequence. For example, #(5 7 9) is an array of three integers.

**ASCII.** (American Standard Code for Information Interchange), this is the world-wide standard for the code numbers used by computers to represent all the upper and lower-case Latin letters, numbers, punctuation, etc. There are 128 standard ASCII codes each of which can be represented by a 7-digit binary number, 0000000 through 1111111.

**attribute.** In VisualAge, data that represents a property of a part. (For example, a customer part could have a name attribute and an address attribute.) Attributes enable a part’s public interface to give other parts access to its properties. An attribute can itself be a part, with its own behavior and attributes. Compare with event and action.

**attribute-to-attribute connection.** A connection from an attribute of one part to an attribute of another part. When one attribute is updated, the other attribute is updated automatically. See also connection.

**attribute-to-script connection.** A connection from an attribute of a part to a script. The connected attribute receives its value from the script, which can make calculations based on the values of other parts. See also connection.

**attribute-to-member function connection.** A connection from an attribute of a part to a member function. When the attribute undergoes a state change, then the member function is called.

**authority.** The right to do something on the system or to use an object, such as a file or document, in the system.

**authorization list.** A list that gives a group of users one or more types of access to objects (such as files or programs) or data in the objects (such as records in a file). It consists of a list of two or more user IDs and their authorities for system resources.

**B**

**backbone.** A high-speed line or series of connections that forms a major pathway within a network. The term is relative, as a backbone in a small network is likely to be much smaller than many nonbackbone lines in a large network. See also Network.

**bandwidth.** The transmission capacity of the lines that carry the Internet’s electronic traffic. Historically, it’s imposed severe limitations on the ability of the Internet to deliver all that we are demanding it deliver, but fiber-optic cables will ensure that bandwidth soon will be essentially limitless and free.

**bandwidth.** How much stuff you can send through a connection, usually measured in bits per second. A full page of English text is about 16,000 bits. A fast modem can move about 15,000 bits in one second. Full-motion full-screen video would require roughly 10,000,000 bits/second, depending on compression. See also 56kB Line, Bps, Bit, T-1.

**base class.** A class from which other classes or parts are derived. A base class may itself be derived from another base class. See also abstract class.

**baud.** In common usage the baud rate of a modem is how many bits it can send or receive per second. Technically, baud is the number of times per second that the carrier signal shifts value; for example a 1200 bit/second modem actually runs at 300 baud, but it moves 4 bits per baud. See also bit, modem.
behavior. (1) The set of external characteristics that an object exhibits. (2) The abstract class that provides common behavior for Class and Metaclass objects.

bind. The process by which the output from the SQL precompiler is converted to a usable structure called an access plan. This process is the one during which access paths to the data are selected and some authorization checking is performed.

bit. (binary digit) A single digit number in base 2, in other words, either a 1 or a zero. The smallest unit of computerized data. Bandwidth is usually measured in bits per second. See also bandwidth, bps, byte, kilobyte, megabyte.

block. A Smalltalk object consisting of one or more statements, enclosed in brackets [ ], passed as arguments or used as the receiver of messages that implement control flow. Blocks can define temporary variables for their own use.

Bps. (bits per second) A measurement of how fast data is moved from one place to another. A 28.8 modem can move 28,800 bits per second. See also bandwidth, bit.

breakpoint. A point in a computer program where the execution may be halted.

browser. (1) A window that supports one or more programming activities, such as creating new classes or methods, modifying existing classes or methods, or viewing library members. (2) Software that enables users to browse through the cyberspace of the World Wide Web. See also Client, URL, WWW.

build. An action that invokes the WorkFrame Build tool. The Build tool manages the project's make file, as well as build dependencies between projects in a project hierarchy.

byte. A set of bits that represent a single character. Usually there are 8 bits in a byte, sometimes more, depending on how the measurement is being made.

C

call level interface (CLI). A callable application program interface (API) for database access, which is an alternative to an embedded SQL application program interface. In contrast to embedded SQL, CLI does not require pre-compiling or binding by the user, but instead provides a standard set of functions to process SQL statements and related services at run time.

canvas. Canvases are windows with a layout algorithm that manage child windows. The canvas classes are a set of window classes that allow you to implement dialog boxes. The different canvases can manage the size and position of child windows, provide moveable split bars between windows, and support the ability to scroll in a window. The canvases include the base class, ICanvas, and its four subclasses: IMulticellCanvas, ISetCanvas, ISplitCanvas, and IViewPort.

cascaded messages. Multiple messages sent to the same receiver object. The messages are separated by a semicolon (;).

category. (1) On the VisualAge Composition Editor, a selectable grouping of parts represented by an icon in the leftmost column. Selecting a category displays the parts belonging to that category in the next column over. See also parts palette. (2) A logical association of a group of methods within a class, with a name assigned by the class developer.

CGI Link. A stand-alone executable program that receives incoming CGI requests and routes them to the VisualAge application. CGI Link runs on the HTTP server, which does not have to be the same as the machine running the VisualAge application.

CGI Link configuration file. A text file that defines the information CGI Link needs in order to route incoming requests to the VisualAge application. The CGI Link configuration file must be set up for your server.
**CGI Link server.** A Smalltalk process within the VisualAge image that receives requests from the CGI Link program. The CGI Link server must be running in order for a Web Connection application to work.

**CGI Link session data.** A Web Connection nonvisual part that holds a persistent data object. You can use CGI Link Session Data to store an application-specific object that remains valid from one CGI query to the next, for the duration of a session.

**CGI query.** A special kind of HTTP request from a client browser requesting that a server-based program be run. A CGI query specifies the name of the program to run, along with any input parameters. See also Common Gateway Interface.

**change-event symbol.** In VisualAge, the code used to signal that an attribute has changed in value.

**character literal.** A literal that is any single character preceded by a dollar sign ($).

**class.** The specification of an object, including its attributes and behavior. Once defined, a class can be used as a template for the creation of object instances. “Class,” therefore, can also refer to the collection of objects that share those specifications. A class exists within a hierarchy of classes in which it inherits attributes and behavior from its superclasses, which exist closer to the root of the hierarchy. See also class extension, definition class, inheritance, polymorphism, visible class.

**class definition.** The definition of a class, containing:

- The class name
- The type of class
- The immediate superclass for the class
- The variables: instance, class, and class instance
- The pool dictionaries the class uses

**class developer.** A team member who develops and changes classes. The team member who created an edition of a class is that edition’s class developer. Contrast with class owner.

**Class Editor.** The editor used to specify the names of files that the Visual Builder writes to when the user generates default code. This editor can also be used to do the following:

- Enter a description of the part
- Specify a different .vbb file in which to store the part
- See the name of the part’s base class
- Modify the part’s default constructor
- Enter additional constructor and destructor code
- Specify a .lib file for the part
- Specify a resource DLL and ID to assign an icon to the part
- Specify other files to be included when the application is built.

Compare to Composition Editor and Part Interface Editor.

**class extension.** An extension to the functionality of a class defined by another application. The extension consists of one or more methods that define the added functionality or behavior. These methods cannot modify the existing behavior of the defined class; they can only add behavior specific to the application that contains the extended class.

**class hierarchy.** A tree structure that defines the relationships among classes. A class has subclasses down the hierarchy from itself and superclasses up the hierarchy from itself. The methods and variables of a class are inherited by its subclasses.

**class identifier (CLSID).** In Windows 95 and Windows NT, the globally unique identifier for an object. The Win32 system registration database uses CLSIDs to distinguish all OLE objects available on a system. Also commonly referred to as a globally unique identifier (GUID).

**class instance variable.** Private data that belongs to a class. The defining class and each subclass maintain their own copy of the data. Only the class methods of the class can reference the data directly. Changing the data in one class does not change it for the other classes in the hierarchy. Contrast with class variable.
**class method.** A method that provides behavior for a class. Class methods are usually used to define ways to create instances of the class. Contrast with instance method.

**class owner.** Team member responsible for the integrity of that class in an application edition. The class owner is responsible for releasing class versions. Contrast with class developer.

**class variable.** Data that is shared by the defining class and its subclasses. The instance methods and class methods of the defining class and its subclasses can directly reference this data. Changing the data in one class changes it for all of the other classes. Contrast with class instance variable.

**client.** A software program that is used to contact and obtain data from a server software program on another computer, often across a great distance. Each client program is designed to work with one or more specific kinds of server programs, and each server requires a specific kind of client. A Web browser is a specific kind of client. See also browser, server.

**client area object.** An intermediate window between a frame window (IFrameWindow) and its controls and other child windows.

**client object.** An object that requests services from other objects.

**client/server.** The model of interaction in distributed data processing in which a program at one location sends a request to a program at another location and awaits a response. The requesting program is called a client, and the answering program is called a server.

**collection.** (1) In Smalltalk, a set of elements in which each element is an object. (2) In SQL, a set of objects created by the SQL/400 licensed program that consists of and logically classifies a set of objects, such as tables, views, and indexes.

**Collection Class Library.** A C++ class library that provide basic functions for collections, and can be used as base classes.

**comment.** A set of characters enclosed in double quotation marks. Smalltalk ignores comments and does not execute them.

**Common Gateway Interface.** A standard protocol through which a Web server can execute programs running on the server machine. CGI programs are executed in response to requests from Web client browsers.

**Common User Access (CUA).** An IBM architecture for designing graphical user interfaces that uses a set of standard components and terminology.

**component.** A functional grouping of classes and related files within a product.

**composite part.** A part that contains other parts; it can also contain data and behavior of its own. For example, a user interface view is a composite part composed of subparts such as entry fields, push buttons, and text.

**Composition Editor.** In VisualAge, a view that is used to build a graphical user interface and to make connections among parts.

**compound document.** A means for integrating arbitrary or unstructured data from different sources into one centralized location.

**Compound Document Framework.** A starting point for creating a server a container document that is OLE-enabled. The framework stores compound documents using the OLE-structured storage specification (docfiles).

**Compound Object Model (COM).** The underlying model for all OLE services. It consists of a variety of APIs and object interfaces that allow container components to communicate and interact with one another.

**concrete class.** A subclass of an abstract class that is a specialization of the abstract class.

**configuration.** A description of a group of components that identifies, for each component, the component edition or version that is part of the group.

**connection.** In VisualAge, a formal, explicit relationship among parts. Connections define the ways in which parts communicate with one another. Making connections is the basic technique used for building any VisualAge
application. See also attribute-to-attribute, attribute-to-script, event-to-action connection and event-to-script.

construction from parts. A software development technology in which applications are assembled from reusable and existing software components known as parts.

constructor. A special class member function that has the same name as the class and is used to construct and possibly initialize objects of its class type.

custom logic connection. A connection that causes customized C or C++ code to be run. This connection can be triggered either when an attribute's value changes or when an event occurs.

cyberspace. Term originated by author William Gibson in his novel Neuromancer, the word cyberspace is currently used to describe the whole range of information resources available through computer networks.

data abstraction. A data type with a private representation and a public set of operations. The C++ language uses the concept of classes to implement data abstraction.

data base. (1) A systematized collection of data that can be accessed and operated upon by an information processing system. (2) A collection of information such as tables, views, and indexes.

data base manager. A VisualAge or IBM Smalltalk database component that models a database management system in order to provide the interface between an application and the database management system.

data member. Private data that belongs to a given object and is hidden from direct access by all other objects. Data members can only be accessed by the member functions of the defining class and its subclasses.

data model. A combination of the base classes and parts shipped with the product and the classes and parts a user saves and creates. They are saved in a file named vbbase.vbb.

data object. A storage area used to hold a value.

DB2 Call Level Interface (CLI). The DB2 call level interface is an alternative SQL interface for the DB2 family of products and takes full advantage of DB2 capability. This implementation closely follows industry standards, such as X/Open™, to enhance application portability. Currently, the DB2 Call Level Interface functions are compatible with ODBC 2.0, and contain DB2 specific APIs to help exploit DB2 capability.

debugger. A software tool used to detect, trace, and eliminate errors in computer programs or other software.

derivation. The creation of a new or abstract class from an existing or base class.

derived class. A class that inherits from a base class. Derived classes allow the addition of new data members and member functions to the class. You can manipulate a derived class object as if it were a base class object. The derived class can override virtual functions of the base class. Synonym for child class and subclass.

destructor. A special class member function that has the same name as the class and is used to destroy class objects.

dictionary. In Smalltalk, an unordered collection whose elements are accessed by an explicitly assigned external key. See also pool dictionary.
**distributed application.** A workstation application that runs in cooperation with programs running on other processes or machines. Client/server applications are a subset of distributed applications.

**domain name.** The unique name that identifies an Internet site. Domain names always have two or more parts, separated by dots. The part on the left is the most specific, and the part on the right is the most general. A given machine may have more than one domain name but a given domain name points to only one machine. Usually, all of the machines on a network will have the same thing as the right-hand portion of their domain names, for example, gateway.mynetwork.com, mail.mynetwork.com, and so on. It is also possible for a domain name to exist but not be connected to an actual machine. This is often done so that a group or business can have an Internet e-mail address without having to establish a real Internet site. In these cases, some real Internet machine must handle the mail on behalf of the listed domain name. See also IP Number.

**dynamic link library (DLL).** A file containing data and code objects that can be used by programs or applications during loading or at run time but are not part of the program’s executable (.EXE) file.

**E**

**edition.** In the VisualAge or IBM Smalltalk team programming environment, a software component that is subject to further change. A software component can have one or more editions, identified by a timestamp stating the date and time of the edition’s creation. Many changes can be made to a single edition of a class. In contrast, every change to a method creates a new edition of that method.

**e-mail.** (Electronic mail) Messages transmitted over the Internet from user to user. E-mail can contain text, but also can carry with it files of any type as attachments.

**encapsulation.** The hiding of a software object’s internal data representation. The object provides an interface that queries and manipulates the data without exposing its underlying structure.

**Ethernet.** A very common method of networking computers in a LAN. Ethernet will handle about 10,000,000 bits/second and can be used with almost any kind of computer. See also bandwidth, LAN.

**event.** A representation of a change that occurs to a part. The events on a part’s public interface enable other interested parts to receive notification when something about the part changes. For example, a push button generates an event signaling that it has been clicked, which might cause another part to display a window. Compare to action and attribute.

**event-to-action connection.** A connection that causes an action to be performed when an event occurs. See also connection.

**event-to-member function connection.** A connection from an event of a part to a member function. When the connected event occurs, the member function is executed.

**event-to-script connection.** A connection that causes a script to run when an event occurs. See also connection.

**expansion area.** The section of a multicell canvas between the current cell grid and the outer edge of the canvas. Usually, this area is bounded by the rightmost column gridline and the bottommost row gridline.

**expression.** In Smalltalk, the syntactic representation of one or more messages. An expression can consist of subexpressions representing the receiver and arguments of the message. The expression can also cause the assignment of its result to one or more variables.

**F**

**feature.** (1) A major component of a software product that can be ordered separately. (2) In VisualAge, an action, attribute, or event that is
available from a part’s public interface and that other parts can connect to. See also action, attribute, and event.

field. A group of related bytes (such as name or amount) that are treated as a unit in a record.

file. (1) A generic term for the object type that refers to a database file, a device file, or a save file. The system-recognized identifier for the object type is *FILE. (2) In the hierarchical file system, a piece of related information (data), such as a document. (3) In SQL, the term is generally referred to as a table.

firewall. A combination of hardware and software that protects a local area network (LAN) from Internet hackers. It separates the network into two or more parts and restricts outsiders to the area outside the firewall. Private or sensitive information is kept inside the firewall.

first-in first-out (FIFO). A queuing technique in which the next request to be processed from a queue is the request of the highest priority that has been on the queue for the longest time.

form. An HTML element that can include entry fields, push buttons, and other user-interface controls through which users can enter information. Sometimes called a fill-in form.

FQDN. (Fully Qualified Domain Name) The official name assigned to a computer. Organizations register names, such as ibm.com or utulsa.edu. They then assign unique names to their computers, such as watson5.ibm.com or tornado.cs.utulsa.edu.

framework. A library of classes, intended for reuse, that fall within a particular domain (for example, a communications framework or a graphics framework).

free-form surface. In VisualAge, the large open area of the Composition Editor window. The free-form surface holds the visual parts contained by the views a user builds and representations of the nonvisual parts that an application includes.

FTP. (File Transfer Protocol) The basic Internet function that enables files to be transferred between computers. You can use it to download files from a remote, host computer, as well as to upload files from your computer to a remote, host computer. See Anonymous FTP.

full attribute. An attribute that has all of the behaviors and characteristics that an attribute can have: a data member, a get member function, a set member function, and an event identifier.

function. In the Common Widgets subsystem, a method that enables a widget to perform an action.

g

garbage collection. A Smalltalk process for periodically identifying unreferenced objects and deallocating their memory.

gateway. A host computer that connects networks that communicate in different languages. For example, a gateway connects a company’s LAN to the Internet.

GET. One of the methods used in HTTP requests. A GET request is used to retrieve data from an HTTP server. See also POST.

GIF. (Graphics Interchange Format) A graphics file format that is commonly used on the Internet to provide graphics images in Web pages.

global variable. A variable that any method in any object can access.

graphical user interface (GUI). A type of interface that enables users to communicate with a program by manipulating graphical elements rather than by entering commands. Typically, a graphical user interface includes a combination of graphics, pointing devices, menu bars, overlapping windows, and icons.
handles. Small squares that appear on the corners of a selected visual part in the visual builder. Handles are used to resize parts.

header file. A file that contains system-defined control information that precedes user data.

heap storage. An area of storage used for allocation of storage whose lifetime is not related to the execution of the current routine. The heap consists of the initial heap segment and zero or more increments.

host. (1) A computer that “hosts” outside computer users by providing files, services or sharing its resources. (2) Any computer on a network that is a repository for services available to other computers on the network. It is quite common to have one host machine provide several services, such as WWW and USENET. See also Node, Network.

host variable. A variable in an SQL statement used for substituting data values into the statement at execution time.

HTML (hypertext markup language). The basic language that is used to build hypertext documents on the World Wide Web. It is used in basic, plain ASCII-text documents, but when those documents are interpreted (called rendering) by a Web browser such as Netscape, the document can display formatted text, color, a variety of fonts, graphic images, special effects, hypertext jumps to other Internet locations and information forms.

HTTP (hypertext transfer protocol). The protocol for moving hypertext files across the Internet. Requires a HTTP client program on one end, and an HTTP server program on the other end. HTTP is the most important protocol used in the World Wide Web (WWW). See also Client, Server, WWW.

HTTP request. A transaction initiated by a Web browser and adhering to HTTP. The server usually responds with HTML data, but can send other kinds of objects as well.

hypertext. Text in a document that contains a hidden link to other text. You can click a mouse on a hypertext word and it will take you to the text designated in the link. Hypertext is used in Windows help programs and CD encyclopedias to jump to related references elsewhere within the same document. The wonderful thing about hypertext, however, is its ability to link—using HTTP over the Web—to any Web document in the world, yet still require only a single mouse click to jump clear around the world.

I

icon. A small pictorial representation of an object.

image. A Smalltalk file that provides a development environment on an individual workstation. An image contains object instances, classes, and methods. It must be loaded into the Smalltalk virtual machine in order to run.

index. A set of pointers that are logically arranged by the values of a key. Indexes provide quick access and can enforce uniqueness on the rows in a table.

inheritance. A relationship among classes in which one class shares the structure and behavior of another. A subclass inherits from a superclass.

instance. An object that is a single occurrence of a particular class. An instance exists in memory or external media in persistent form. Compare with persistent object.

instance method. In Smalltalk, a method that provides behavior for particular instances of a class. Messages that invoke instance methods are sent to particular instances, rather than to the class as a whole. Contrast with class method.

instance variable. Private data that belongs to an instance of a class and is hidden from direct access by all other objects. Instance variables can be accessed only by the instance methods of the defining class and its subclasses.
Internet. The vast collection of interconnected networks that all use the TCP/IP protocols and that evolved from the ARPANET of the late 1960s and early 1970s. By July of 1995, the Internet was connecting roughly 60,000 independent networks into a vast global net.

intranet. A private network inside a company or organization that uses the same kinds of software that you would find on the public Internet, but that is only for internal use. As the Internet has become more popular, many of the tools used on the Internet are being used in private networks, for example, many companies have Web servers that are available only to employees.

IP. (Internet Protocol) The rules that provide basic Internet functions. See TCP/IP.

IP Number. An Internet address that is a unique number consisting of four parts separated by dots, sometimes called a dotted quad. (For example: 198.204.112.1) Every Internet computer has an IP number and most computers also have one or more domain names that are plain language substitutes for the dotted quad.

ISDN. (Integrated Services Digital Network) A set of communications standards that enable a single phone line or optical cable to carry voice, digital network services and video. ISDN is intended to eventually replace our standard telephone system.

iterative development. A software development process that allows progress in stages. At the end of each stage, the result is verified by end users. Through such verification, requirements are dynamically identified and refined while the product is under development.

Java. Java is a new programming language invented by Sun Microsystems that is specifically designed for writing programs that can be safely downloaded to your computer through the Internet and immediately run without fear of viruses or other harm to your computer or files. Using small Java programs (called applets, Web pages can include functions such as animations, calculators, and other fancy tricks. We can expect to see a huge variety of features added to the Web using Java, since you can write a Java program to do almost anything a regular computer program can do, and then include that Java program in a Web page.

JPEG. (Joint Photographic Experts Group) The name of the committee that designed the photographic image-compression standard. JPEG is optimized for compressing full-color or gray-scale photographic-type, digital images. It doesn’t work well on drawn images such as line drawings, and it does not handle black-and-white images or video images.

K

kbps. (kilobits per second) A speed rating for computer modems that measures (in units of 1024 bits) the maximum number of bits the device can transfer in one second under ideal conditions.

keyword. A predefined word reserved for the C and C++ languages, that may not be used as an identifier.

keyword message. A message that takes one or more arguments. A keyword is an identifier followed by a colon (:). Each keyword requires one argument, and the order of the keywords is important. 'hello' at: 2 put: $H is an example of a keyword message; at: and put: are keyword selectors, 2 and $H are the arguments. See also message.

kilobyte. A thousand bytes. Actually, usually 1024 (2^10) bytes. See also byte, bit.

J

LAN. Local area network. A computer network located on a user’s establishment within a limited geographical area. A LAN typically consists of one or more server machines providing services to a number of client workstations. See also Ethernet.
legacy code. Existing code that a user might have. Legacy applications often have character-based, non-graphical user interfaces; usually they are written in a non-object-oriented language, such as C or COBOL.

library. A shared repository represented by a single file. It stores source code, object (compiled) code, and persistent objects, including editions, versions, and releases of software components.

listserv. An Internet application that automatically serves mailing lists by sending electronic newsletters to a stored database of Internet user addresses. Users can handle their own subscribe/unsubscribe actions without requiring anyone at the server location to personally handle the transaction.

literal. An object that can be created by the compiler. A literal can be a number, a character string, a single character, a symbol, or an array. All literals are unique: Two literals with the same value refer to the same object. The object created by a literal is read-only; it cannot be changed.

literal text. Text in an HTML Text part that is passed to the client browser exactly as entered. You can use literal text to code HTML tagging that is not directly supported by the Web Connection parts.

load. A system operation that links the compiled code for a software component from a library into an active image. Loading also performs other operations that enable the component to run, such as linking prerequisites.

loaded. The state of the mouse pointer between the time one selects a part from the parts palette and deposits the part on the freeform surface.

locale. The definition of the subset of a user’s environment that depends on language and cultural conventions.

Login. The account name used to gain access to a computer system. Not kept secret (unlike password).

main part. The part that users see when they start an application. This is the part from which the main() function C++ code for the application is generated. The main part is a special kind of composite part. See also part and subpart.

mangling. The encoding during compilation of identifiers such as function and variable names to include type and scope information. This information is stored in object files and executables. The pre linker uses this information to ensure type-safe linkage.

megabyte. A million bytes. A thousand kilobytes. See also byte, bit, kilobyte.

member. (1) A data object in a structure or a union. (2) In C++, classes and structures can also contain functions and types as members.

member function. An operator or function that is declared as a member of a class. A member function has access to the private and protected data members and member functions of objects of its class.

member function call. A communication from one object to another that requests the receiving object to execute a member function. A member function call consists of a member function name that indicates the requested member function and the arguments to be used in executing the member function. The member function call always returns some object to the requesting object as the result of performing the member function. Synonym for message.

member function name. The component of a member function call that specifies the requested operation.

message. In Smalltalk, a communication from one object to another that requests the receiving object to execute a method. A message consists of a reference to the receiving object, followed by a selector indicating the requested method, and (in many cases) arguments to be used in executing the method. There are three types of messages: binary, keyword, and unary.
method. Executable code that implements the logic of a particular message for a class. In VisualAge, methods are also called scripts. See also class method, instance method, private method, and public method.

MIME. (Multipurpose Internet Mail Extensions) A set of Internet functions that extend normal e-mail capabilities and enable nontext computer files to be attached to e-mail. Nontext files include graphics, spreadsheets, formatted word-processor documents, sound files, and so on. Files sent by MIME arrive at their destination as exact copies of the original so that you can send fully formatted word-processing files, spreadsheets, graphics images and software applications to other users via simple e-mail. Besides email software, the MIME standard is also universally used by Web servers to identify the files they are sending to Web clients, in this way new file formats can be accommodated simply by updating the browsers’ list of pairs of MIME types and appropriate software for handling each type. See also browser, client, server.

model. A nonvisual part that represents the state and behavior of a real-world object, such as a customer or an account. Contrast with view.

notebook. In VisualAge, a view that resembles a bound notebook, containing pages separated into sections by tabbed divider pages. A user can turn the pages of a notebook or select the tabs to move from one section to another.

notification framework. A set of classes that implement the notifier/observer protocol. The notification framework is the base of the construction from parts technology (Visual Builder).

O

object. (1) The basic building block in Smalltalk development. An object is anything that exhibits behavior. All code and data in Smalltalk must be part of an object. (2) On the AS/400 system, an object is a named storage space consisting of a set of characteristics that describes itself and, in some cases, data. Some examples of objects are programs, files, and libraries.

object class. A template for defining the attributes and member functions of an object. An object class can contain other object classes. An individual representation of an object class is called an object.

object factory. A nonvisual part capable of dynamically creating new instances of a specified part. For example, during the execution of an application, an object factory can create instances of a new class to collect the data being generated.

object-oriented programming. A programming methodology built around objects and based on sending messages back and forth between those objects. The basic concepts of object-oriented programming are encapsulation, inheritance, and polymorphism.

object persistence. A characteristic that enables objects to exist beyond the time in which their creating application runs. One use of object persistence is sharing objects among programmers in a development environment.

observer. An object that receives notification from a notifier object.
**operation.** A member function or service that can be requested of an object.

**Object Request Broker (ORB).** A CORBA term designating the means by which objects transparently make requests and receive responses from objects, whether they are local or remote.

**overloading.** An object-oriented programming technique that allows redefinition of functions and most standard C++ operators when the functions and operators are used with class types.

**P**

**parameter.** A data element included as part of a message to provide information that the object might need. In Smalltalk, generally referred to as an argument.

**parameter connection.** A connection that satisfies a parameter of an action or member function by supplying either an attribute’s value or the return value of an action, member function, or custom logic. The parameter is always the source of the connection.

**parent class.** The class from which another part or class inherits data, member functions, or both.

**part.** A self-contained software object with a standardized public interface, consisting of a set of external features that allow the part to interact with other parts. The parts on the VisualAge parts palette can be used as templates to create instances of objects.

**part event.** A representation of a change that occurs to a part. The events on a part’s interface enable other interested parts to receive notification when something about the part changes. For example, a push button generates an event signaling that it has been clicked, which might cause another part to display a window.

**part event ID.** The name of a part static-data member used to identify the notification that is being signaled.

**part interface.** A set of external features that allows a part to interact with other parts. A part’s interface is made up of three characteristics: attributes, actions, and events.

**Part Interface Editor.** An editor that the application developer uses to create and modify attributes, actions, and events, which together make up the interface of a part. Compare to Class Editor and Composition Editor.

**parts palette.** In the VisualAge Composition Editor, an organized collection of visual and nonvisual parts used in building composite parts for an application. The parts palette is organized into categories. Application developers can add parts to the palette for use in defining applications or other parts.

**password.** A code used to gain access to a locked system. Good passwords contain letters and nonletters and are not simple combinations.

**PATH_INFO.** A CGI variable, usually transmitted to the CGI program in the form of an environment variable. The PATH_INFO variable contains all path information from the URL following the name of the CGI executable. For a Web Connection application, this information is the same as the VisualAge part name.

**persistent object.** Instances stored outside of the image. A persistent object must be loaded into virtual or real storage before it can process messages sent to it.

**polymorphism.** The ability of different objects to respond to the same message in different ways. This means that different objects can have very different method implementations for the same message. An object can send a message without concern for its underlying implementation.

**pool dictionary.** A dictionary object whose keys define variables that can be shared by multiple classes. All methods for a class can access the variables in a pool dictionary if the class declares the pool dictionary as part of its scope.
Port. (1) A place where information goes into or out of a computer, or both. For example, the serial port on a personal computer is where a modem would be connected. (2) On the Internet port often refers to a number that is part of a URL, appearing after a colon (:) right after the domain name. Every service on an Internet server listens on a particular port number on that server. Most services have standard port numbers; Web servers normally listen on port 80. Services can also listen on nonstandard ports, in which case the port number must be specified in a URL when accessing the server. (3) Refers to translating a piece of software to bring it from one type of computer system to another. See also domain name, server, URL.

POST. One of the methods used in HTTP requests. A POST request is used to send data to an HTTP server. See also GET.

pragma. An implementation-defined instruction to the compiler.

preferred features. A subset of the part’s features that appear in a pop-up connection menu. Generally, they are the features used most often.

primary part. In a composite part constructed with the VisualAge Composition Editor, the subpart whose public interface is fully exposed on the public interface of the composite part. The primary part is transparently visible to parts outside the composite part and is the subpart with which most interaction will take place.

primitive part. A basic building block of other parts. A primitive part can be relatively complex in terms of the function it provides.

private method. In VisualAge or IBM Smalltalk, a method that is not part of the system API but is provided as part of the internal functioning of the system. Developers using a method marked as private may have to modify their code in order for it to work with a future release of VisualAge or IBM Smalltalk. Contrast with public method. Individual application development projects can use the public and private designations as a means of organizing their code.

process. In Smalltalk, a sequence of actions described by expressions and performed by the system’s virtual machine.

program. (1) One or more files containing a set of instructions conforming to a particular programming language syntax. (2) A self-contained, executable module. Multiple copies of the same program can be run in different processes.

promote features. Make features of a subpart available to be used for making connections. This applies to subparts that are to be included in other parts, for example, a subpart consisting of three push buttons on a canvas. If this example subpart is placed in a frame window, the features of the push buttons would have to be promoted to make them available from within the frame window.

protected. Pertaining to a class member that is only accessible to member functions and friends of that class, or to member functions and friends of classes derived from that class.

protocol. (1) The set of all messages to which an object will respond. (2) Specification of the structure and meaning (the semantics) of messages that are exchanged between a client and a server. (3) Computer rules that provide uniform specifications so that computer hardware and operating systems can communicate. It’s similar to the way that mail, in countries around the world, is addressed in the same basic format so that postal workers know where to find the recipient’s address, the sender’s return address and the postage stamp. Regardless of the underlying language, the basic protocols remain the same.

prototype. A function declaration or definition that includes both the return type of the function and the types of its arguments.

proxy. An application gateway from one network to another for a specific network application like Telnet of FTP, for example, a firewall’s proxy Telnet server performs authentication of the user and then lets the traffic flow through the proxy as if it were not there. Function is performed in the firewall and not in the client workstation, causing more load in the firewall. Compare with socks.
public interface. A set of external features that enable a part to interact with other parts. A part’s public interface is made up of three characteristics: attributes, actions, and events.

Public Interface Editor. A VisualAge view used to create and modify attributes, actions, and events, which together make up the public interface of a part.

public method. A method that is provided as part of the VisualAge or IBM Smalltalk API. Public methods are designed to function with future releases of VisualAge or IBM Smalltalk and with operating systems supported by VisualAge or IBM Smalltalk. Contrast with private method. Individual application development projects can use the public and private designations as a means of organizing their code.

Q

query specification. A database query definition. All queries issued to a database by VisualAge or IBM Smalltalk must be defined by a query specification.

quick form. In the VisualAge Composition Editor, a menu option that enables application developers to quickly create a default view for a part.

R

receiver. The object that receives a message. Contrast with sender.

record. A group of related data, fields, or words, treated as a unit, such as name, address, and telephone number.

RecordStructure. An object that contains information about the format, structure, and types of the data it contains.

RecordStructure classes. Classes that provide a flexible, open architecture for converting Smalltalk objects to and from data types of other languages.

release. A system operation on a component that changes its containing component’s configuration. Releasing a component adds its released edition or version to the configuration for its containing component. When a containing component is loaded into an image, the released editions or versions of the components it contains are also loaded.

repository. (1) An organized, shared body of information that can support business and data-processing activities. (2) In VisualAge or IBM Smalltalk, the multiuser library that stores components such as applications, classes, and methods created by application developers. It stores source code, object code, and persistent objects.

reset button. A type of push button that can appear on a form. A reset button restores all input fields to their default states.

resource file. A file that contains data used by an application, such as text strings and icons.

return value. An object or data type that a receiver object passes to a sender object in response to a message.

router. A network device that enables the network to reroute messages it receives that are intended for other networks. The network with the router receives the message and sends it on its way exactly as received. In normal operations, they do not store any of the messages that they pass through.

runtime type information (RTTI). An extension to the C++ language that allows to retrieve the type of an object at run time, and to perform safe casting operations.

S

script. A series of Smalltalk statements that implement an action for a part. Scripts are equivalent to Smalltalk methods.

Script Editor. A VisualAge view that enables a developer to implement part behavior by writing Smalltalk scripts (methods). The Script
Editor shows all the objects that can be referenced by a method and all previously defined methods of a part's class.

**selection handles.** In the Composition Editor, small squares that appear on the corners of a selected visual part. Selection handles are used to resize parts.

**selector.** The component of a message that specifies the requested operation. There are three kinds of selectors: binary, keyword, and unary.

**sender.** An object that sends a message to another object. On the level of code implementation, the sender is considered to be the sending method within the class or instance that issues the message. Contrast with receiver.

**server.** (1) A computer that provides services to multiple users or workstations in a network; for example, a file server, print server, or mail server. (2) An object that performs one or more tasks on behalf of a client. The server can be a computer (a file server), a specific process on a server, or a distributed object. A single server machine could have several different server software packages running on it, thus providing many different servers to clients on the network. See also client, network.

**service.** A specific behavior that an object is responsible for exhibiting.

**session.** A series of CGI queries that come from the same client and belong to the same logical sequence. A session is identified by a unique session key, which is generated by VisualAge. A session begins when a client initially connects (without a session key) and ends when a specified timeout period has elapsed since the last connection.

**session key.** A unique string, generated automatically by VisualAge, that identifies a session. All requests and replies carry the session key in hidden HTML data fields.

**settings view.** A view of a part that provides a way to display and set the attributes and options associated with the part.

**Smalltalk (ST).** (1) A complete programming environment for developing object-oriented applications. Smalltalk is a pure implementation of object-oriented concepts; every entity in the environment is an object. (2) The name of the programming language that the Smalltalk programming environment supports. (3) In the IBM Smalltalk programming environment, the name of the System Dictionary containing the global variables.

**socks.** Software to intercept and redirect all TCP/IP requests at the firewall. It handles data to and from applications such as Telnet, FTP, Mosaic, and Gopher. Provides users in a secured network access to resources outside the network by directing data through the firewall. Firewall users must use client programs specifically designed to work with the sockd server.

**SQL Editor.** An interactive tool for creating structured query language (SQL) statements. It consists of a set of dialogs that prompt the user for information about database tables and use that information to generate SQL statements.

**sticky.** In the Composition Editor, the mode that enables an application developer to add multiple parts of the same class (for example, three push buttons) without going back and forth between the parts palette and the free-form surface.

**stored procedure.** A procedure stored in a database system that contains SQL and other control statements.

**structure.** A construct that contains an ordered group of data objects. Unlike an array, the data objects within a structure can have varied data types.

**structured query language (SQL).** A language used to access relational databases.

**subapplication.** An application contained by another application. Using subapplications, one can organize the classes of an application into a tree of subapplications or isolate the parts of an application that are platform-specific.
subclass. A class that inherits behaviors and specifications (in other words, methods and variables) from another class. Contrast with superclass.

submit button. A type of push button that can appear on a form. A submit button initiates a connection to the HTTP server and sends a CGI query, using the data from the input fields as parameters.

subpart. A part that is embedded within a composite part.

superclass. A class from which another class inherits behaviors and specifications (in other words, methods and variables). Contrast with subclass.

symbol. In Smalltalk, an object that represents a string used as a name within the system. A symbol literal is a sequence of characters preceded by the pound sign (#) with no embedded blanks, such as #George or #messageSelector.

Systems Network Architecture (SNA). The description of the logical structure, formats, protocols, and operational sequences for transmitting information units through, and controlling the configuration and operation of, networks.

T

TCP/IP. (Transmission Control Protocol/Internet Protocol) The basic programming foundation that carries computer messages around the globe via the Internet. The suite of protocols that defines the Internet. Originally designed for the UNIX operating system, TCP/IP software is now available for every major kind of computer operating system. To be truly on the Internet, your computer must have TCP/IP software.

team programming. Development of a system, program, or application suite by a team of two or more programmers or application developers.

tear-off attribute. An attribute that an application developer has exposed to work with as though it were a standalone part.

Telnet. An Internet protocol that lets you connect your PC as a remote workstation to a host computer anywhere in the world and to use that computer as if you were logged on locally. You often have the ability to use all of the software and capability on the host computer, even if it's a huge mainframe.

template. A family of classes or functions where the code remains invariant but operates with variable types.

template class. A class instance generated by a class template.

template function. A function generated by a function template.

temporary variable. A variable whose scope is limited to the Smalltalk method or block in which it is defined. A temporary variable takes an assigned value.

thread. A unit of execution within a process.

tool bar. In the VisualAge Composition Editor, the strip of icons along the top of the free-form surface. The tool bar contains tools to help construct composite parts. These tools are also available through the Tools pull-down menu of the Composition Editor window.

Transcript window. The main controlling window in Smalltalk.

U

uniform resource locator (URL). A standard identifier for a resource on the World Wide Web, used by a Web browser to initiate a connection. The URL includes the communications protocol to use, the name of the server, and path information identifying the object to be retrieved on the server. A URL looks like this:

http://www.matisse.net/seminars.html.br
or telnet://well.sf.ca.us.br
or news:new.newusers.questions.br
unloaded. The state of the mouse pointer before a user selects a part from the parts palette and after the user deposits a part on the free-form surface. In addition, a user can unload the mouse pointer by pressing the Esc key.

user interface (UI). The hardware, software, or both that enables a user to interact with a computer. In VisualAge, user interface normally refers to the visual presentation with which a user interacts and its underlying software.

user profile. A file that contains the user’s password, the list of special authorities assigned to a user, and the objects the user owns. It is used by the system to verify the user’s authorization to read or use objects, such as files or devices, or to run the jobs on the system. Each user profile must have a unique name.

visible class. A class that another class can subclass or refer to by name in a method. Visible refers to the scope in which the class name can be used. For a class in a given application, visible classes include:
- All classes defined in the same application
- All public classes defined in any subapplication
- All prerequisite classes, including prerequisites of prerequisites to the lowest level

visible part. A part that has a visual representation at run time. Visual parts, such as windows, push buttons, and entry fields, make up the user interface of an application. Compare to view. Contrast with nonvisual part.

W

WAN. (Wide Area Network)— Any internet or network that covers an area larger than a single building or campus. See also Internet, LAN, network.

widget. An object that provides a user-interface abstraction; for example, a scrollbar widget. Widgets support obtaining input from the user and displaying output to the user.

window. (1) A rectangular area of the screen with visible boundaries in which information is displayed. Windows can overlap on the screen, giving the appearance of one window being on top of another. (2) In the VisualAge Composition Editor, a part that can be used as a container for other visual parts, such as push buttons.

World Wide Web. (WWW) (W3) (the Web) An Internet client-server distributed information and retrieval system based upon HTTP that transfers hypertext documents across a varied array of computer systems. The Web was created by the CERN High-Energy Physics Laboratories in Geneva, Switzerland in 1991. CERN boosted the Web into international prominence on the Internet.
## List of Abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AIX</td>
<td>Advanced Interactive eXecutive</td>
</tr>
<tr>
<td>APAR</td>
<td>authorized program analysis report</td>
</tr>
<tr>
<td>API</td>
<td>application programming interface</td>
</tr>
<tr>
<td>CAE</td>
<td>Client Application Enabler</td>
</tr>
<tr>
<td>CERN</td>
<td>Conseil Europeen pour la Recherche Nucleaire</td>
</tr>
<tr>
<td>CGI</td>
<td>Common Gateway Interface</td>
</tr>
<tr>
<td>CICS</td>
<td>customer information control system</td>
</tr>
<tr>
<td>CLI</td>
<td>call level interface</td>
</tr>
<tr>
<td>CLSID</td>
<td>class identifier</td>
</tr>
<tr>
<td>COFF</td>
<td>common object file format</td>
</tr>
<tr>
<td>CORBA</td>
<td>Common Object Request Broker Architecture</td>
</tr>
<tr>
<td>CRC</td>
<td>class responsibility collaborators</td>
</tr>
<tr>
<td>DBMS</td>
<td>database management system</td>
</tr>
<tr>
<td>DB2</td>
<td>Database 2</td>
</tr>
<tr>
<td>DDCS</td>
<td>distributed database connection services</td>
</tr>
<tr>
<td>DDL</td>
<td>database definition language</td>
</tr>
<tr>
<td>DLL</td>
<td>dynamic link library</td>
</tr>
<tr>
<td>DNS</td>
<td>domain name server</td>
</tr>
<tr>
<td>DSOM</td>
<td>distributed system object model</td>
</tr>
<tr>
<td>ESA</td>
<td>enterprise systems architecture</td>
</tr>
<tr>
<td>FAT</td>
<td>File Allocation Table</td>
</tr>
<tr>
<td>FTP</td>
<td>File Transfer Protocol</td>
</tr>
<tr>
<td>GUI</td>
<td>graphical user interface</td>
</tr>
<tr>
<td>HPFS</td>
<td>high performance file system</td>
</tr>
<tr>
<td>HTML</td>
<td>Hypertext Markup Language</td>
</tr>
<tr>
<td>HTTP</td>
<td>Hypertext Transfer Protocol</td>
</tr>
<tr>
<td>HTTPD</td>
<td>Hypertext Transfer Protocol daemon</td>
</tr>
<tr>
<td>IBM</td>
<td>International Business Machines Corporation</td>
</tr>
<tr>
<td>IDC</td>
<td>International Data Corporation</td>
</tr>
<tr>
<td>IDE</td>
<td>integrated development environment</td>
</tr>
<tr>
<td>IOC</td>
<td>IBM Open Class Library</td>
</tr>
<tr>
<td>IMS</td>
<td>information management system</td>
</tr>
<tr>
<td>ISO</td>
<td>International Standardization Organization</td>
</tr>
<tr>
<td>ISP</td>
<td>Internet service provider</td>
</tr>
<tr>
<td>ITSO</td>
<td>International Technical Support Organization</td>
</tr>
<tr>
<td>JDBC</td>
<td>Java Database Connectivity</td>
</tr>
<tr>
<td>JPEG</td>
<td>Joint Photographic Experts Group</td>
</tr>
<tr>
<td>LAN</td>
<td>local area network</td>
</tr>
<tr>
<td>MIME</td>
<td>Multipurpose Internet Mail Extensions</td>
</tr>
<tr>
<td>Abbreviation</td>
<td>Description</td>
</tr>
<tr>
<td>--------------</td>
<td>------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>MLE</td>
<td>multiline entryfield</td>
</tr>
<tr>
<td>MVS</td>
<td>multiple virtual storage</td>
</tr>
<tr>
<td>NCSA</td>
<td>National Center for Super-Computer Research</td>
</tr>
<tr>
<td>NLS</td>
<td>National Language Support</td>
</tr>
<tr>
<td>NT</td>
<td>new technology</td>
</tr>
<tr>
<td>ODBC</td>
<td>Open Database Connectivity</td>
</tr>
<tr>
<td>OLE</td>
<td>Object Linking and Embedding</td>
</tr>
<tr>
<td>OMF</td>
<td>object module format</td>
</tr>
<tr>
<td>OMG</td>
<td>Object Management Group</td>
</tr>
<tr>
<td>OLTP</td>
<td>on-line transaction processing</td>
</tr>
<tr>
<td>OMT</td>
<td>object modeling technique</td>
</tr>
<tr>
<td>OOP</td>
<td>object-oriented programming</td>
</tr>
<tr>
<td>OOSE</td>
<td>object-oriented software engineering</td>
</tr>
<tr>
<td>OS/2</td>
<td>Operating System/2</td>
</tr>
<tr>
<td>PM</td>
<td>Presentation Manager</td>
</tr>
<tr>
<td>RAD</td>
<td>rapid application development</td>
</tr>
<tr>
<td>RDBMS</td>
<td>relational database management system</td>
</tr>
<tr>
<td>RDD</td>
<td>responsibility-driven design</td>
</tr>
<tr>
<td>RTF</td>
<td>rich text format</td>
</tr>
<tr>
<td>SGML</td>
<td>standard generalized mark-up language</td>
</tr>
<tr>
<td>SHTTP</td>
<td>Secure Hypertext Transfer Protocol</td>
</tr>
<tr>
<td>SOM</td>
<td>system object model</td>
</tr>
<tr>
<td>SQL</td>
<td>structured query language</td>
</tr>
<tr>
<td>SSL</td>
<td>secure sockets layer</td>
</tr>
<tr>
<td>TCP/IP</td>
<td>Transmission Control Protocol/Internet Protocol</td>
</tr>
<tr>
<td>URL</td>
<td>uniform resource locator</td>
</tr>
<tr>
<td>VBB</td>
<td>Visual Builder binary</td>
</tr>
<tr>
<td>VBE</td>
<td>Visual Builder export</td>
</tr>
<tr>
<td>VMT</td>
<td>visual modeling technique</td>
</tr>
<tr>
<td>WTA</td>
<td>web travel agent</td>
</tr>
<tr>
<td>WWW</td>
<td>world wide web</td>
</tr>
<tr>
<td>WYSIWYG</td>
<td>what you see is what you get</td>
</tr>
</tbody>
</table>

World Wide Web Programming with VisualAge for C++ and Smalltalk
Index

A
abbreviations 495
acronyms 495
action
  executeQuery 354
  generateHTML 57
  packageQueryExecute 357
  transferRequest 53
  transferRequestWith 53
  triggering 52
administrator
  authentication 366
  cleanup 109
  database cleanup 110
  database statistics 376
  DB2 WWW configuration 402
  functions 365
  home page 368
  implementation 365
  Internet connection server 397
  reset database 376
  search customer information 370
  tasks 109
alignment of parts 37
applet
  generated HTML 35
application
  accessing session data 48
  administrator 109
  analysis 105
  analysis and design 57
  architecture 159
  building 37
  business logic 58
  command-line arguments 424
  components 32
  connections 49
  creating parts 57
  data entry controls 43
  database access 31
  debugging 59
  design 105
  development cycle 116
  development process 57
  distributing servers 30
  flow 33, 37
  image size 59
  interface 42
  make executable 420
modeling 110
on the WWW 32
packaging 32, 59, 419
performance 32
prerequisite 420
preserve state information 32
program logic 45
prototype 111
response time 51
single page 47
startup parameters 424
testing 37, 59
VMT 110
web 37
Windows NT port 459
writing macros 92
application models 19
attribute
cgiVars 52
formData 52
promotion of 274
resultTable 354
transferredObject 53
authentication 100, 459

B
background 259
backward links 9
banner 260
basic pages 38
basic parts 38
bibliography 469
buttons on HTML Table 59

C
call level interface 460
CDROM 386
CERN 5
CGI 6
  client/server 20
data manipulation 15
database access 92
DB2 WWW 94
directory 34
dynamic documents 12
form action 43
customize 355
embedded 357
hidden connections 356
reusable 41
reusing parts 351
typical use 41
CompuServe xi
CONFIG.SYS 385
connection
custom logic 85
execution order of 279, 293, 323
connectionless transport 4
constructor 480
controlling part placement 37
CRC cards
WtaAboutView 449
WtaAttractionProcessor 448
WtaAttractionView 448
WtaCarClassView 437
WtaCarSelectionProcessor 436
WtaCarSelectionView 436
WtaCommitter 441
WtaComponentSelectionProcessor 432
WtaComponentSelectionView 432
WtaCreditCardInfoView 443
WtaCustomer 445
WtaCustomerAuthenticationProcessor 444
WtaCustomerAuthenticationView 444
WtaCustomerIdProcessor 440
WtaCustomerIdView 440
WtaDeveloperView 449
WtaFlight 434
WtaFlightSelectionProcessor 433
WtaFlightSelectionView 433
WtaHotel 435
WtaHotelSelectionProcessor 435
WtaHotelSelectionView 434
WtaLocation 449
WtaNewCustomerIdProcessor 441
WtaNewCustomerIdView 440
WtaPackageConfirmationView 443
WtaPackageCreationController 430
WtaPackageCreationProcessor 431
WtaPackageCreationView 431
WtaPackageDetailsView 447
WtaPackageFailedView 438
WtaPackageListProcessor 446
WtaPackageListView 446
WtaPackagePricer 438
WtaPackagePurchaseProcessor 439
WtaPackagePurchaseView 439
WtaPersonalInfoProcessor 445
WtaPersonalInfoView 444
WtaRequestedCar 437
WtaRequestedFlightSeats 434
WtaRequestedHotelRooms 436
WtaRequestedPackage 440
WtaWelcomeProcessor 430
WtaWelcomeView 430
creating a page 39
customer
DDL 451
detail information 373
identification 108
interactions 106
listing 109
personal data modification 108

d
Data Access Builder 340
access method 341
automate link to application 343
class type 341
generated classes 342, 345
invoking 340
Data Access Builder part
data identifier 343, 346
database
access 91
authentication 100
call level interface 460
cascaded deletes 370
CGI 6
cleanup 110
connection specifications 460
controller 128
DDL 451
defining keys 132
definition 451
design 129, 370
directory 340
initialization 49
internal structure 101
Java applet 21
keys 132
Lotus Notes 24
macro definition 95
manager 341, 461
object persistence 130
owner 341
primary key 343
protecting directories 100
referential integrity 133
reset 376
single row query 359
statistics 376
type 340
Index

data 43, 45
generated HTML 35
input section 96
macro variables 93
nesting 43
pages 38
parts 42
reset button 43
retrieving data from 45
setting up action 42
submit button 43
user-interface controls 43
form data, retrieving 45
form parts 42
forms 5
FTP 8

g
Gartner group 18
generalization 113
generating HTML from Smalltalk 465
GET 10, 43, 77, 81
GoServe 5

h
handling errors 465
hidden entry field 256, 278, 298
host name 380
hotel
  application description 106
  availability 119
  capacity 107
  DDL 453
  event trace diagrams 123
  preferences 118
  price 119
  reservation 107
  room category 107
  room selection 117
  state transition diagrams 125
  use case 117
HPFS 392
HTML 5
  architecture 58
  basic pages 38
  CGI process 13
  coding 44
  composite 41, 351, 354
  control generation 57

  CRC cards 120
  creating pages 57
  dynamic 12
  dynamically creating 6
  environment variables 13
  extensions 44
  flowed layout 37
  font size 39
  form 13, 260
  form pages 38
  frames 19
  generated 34
  generation for parts 34
  heading levels 39
  highlighting 39
  images 40
  input section 95, 96, 371
  invoking DB2 WWW 94
  Java 455
  lists 59
  minimum tagging 39
  parts 31
  passing variables to SQL 93
  push buttons 43
  quick 351
  rendered 37
  report section 95, 99
  static 12, 19, 91
  table 59, 354
  tables 19
  text color 39
  user attribute 259
  viewing source 101
  visual parts 34
HTML Form Data 45
  accessing values 46
  alternative access 52
  application connections 49
  attributes 46
  CGI request 52
  part specification 46
  processing 46
  retrieving 45
  setting up 46
  single page application 47
HTML page
  adding 39
  adding images 40
  adding text 39
  basic 38
  control HTML generation 57
  creating 39
  edit-time fonts 40
  fonts 40
  lifetime 47
link to 40
minimum tagging 39
multiple 39, 56
new instance 47
primary 57
reusing parts 41
run-time fonts 40
title bar 39
HTML Page Wrapper 53
HTML Table
buttons on 59
graphics 59
HTML table
generated HTML 35
HTML table column
generated HTML 35
HTML Table Column.*
setting up
HTML Table.*
setting up with Quick HTML
HTML Text
adding 39
formatting 39
literal 59
HTTP 5
application flow 33
authorization 9
CGI Link 30
client/server 20
configuring 396
distributing server 30
Java applet 21
location header 14
middleware 23
port number 398
response headers 10
standard port 8
starting server 395
HTTP header 9
HTTP request 33
HTTP server
setting up 378
HTTPD.CNF 367, 397, 460
hypergraphic link 41
hypertext links 8, 12, 39, 40
ICollectionViewListbox 64, 266, 267, 280
ICollectionViewListbox 67
IComboBox 66
.IContainerControl 67, 88
IDatastore 264, 268
IDate 263, 264
IEntryField 67, 70
IFrameWindow 64, 67, 71, 259
IGroupBox 67
IHeadControl 67, 73, 260
ImgControl 67, 72, 260, 274
IListBox 68
image
adding 40
file formats 41
generated HTML 35
linking from 41
IMessageBox 68
IMle 68
IMultiCellCanvas 68, 70, 260, 274
inheritance 113
INotebook 68
installing Web Connection 378
interactive Web model 20
Internet
address 7
applications 19
business 18
CGI 11
database access 92
dynamically generating 41edit-time 40
format 41
ITSO x
location 40
Lotus Notes 24
opportunity 18
part 40
performance 18
run-time 40
security 18
VisualAge directions 17
Internet Connection Server 5
configure 396
directives 397
ETC directory 460
home directory 398
installing 392
protecting 399
requirements 392
starting server 395
Internet PC 20
interprocess communication 253
IPersistentObject 345
IPOManager 346
IProfile 255, 263, 267, 268, 277, 298
IPushButton 69, 260, 274
 IRadioButton 69
 Is Displayed qualifier 345
 ISetCanvas 69, 280
 ISplitCanvas 69
 IStandardNotifier 267
 IStaticText 69, 70, 265, 274, 280, 311
 ITime 263
 ITSO
 FTP server x
 Internet x
 World Wide Web x
 IVBFactory 85, 268, 277
 IVBStringPart 268, 280
 IVSequence 346

J
Java
 applet 20, 455
 CGI program 23
 check values on forms 22
 client 22
 data-access classes 24
 database access 21
 distributed builder 23
 distributed Web model 20
 proxy 23
 server interaction 21
 source 455
 visual builder 23
 JDBC 24

L
link
 defining 40
 hypertext 8
 specifying location 40
 link specifications 45
 linking 34
 literal 44, 59
 Lotus Notes 6, 24

M
macro
 administrative functions 100
 contents 94

define section 95
define variables 94
definition 94
footer information 98
header information 98
hidden variables 101
input section 96
naming sections 96
report section 99
SQL message subsection 98
SQL report subsection 97
SQL section 96
subsections 96
termination 99
testing 93
unnamed sections 99
variables 93
makefile 343
META group 18
method
carPreferences 179, 206
carPrice 206
checkDaysInMonth 173
checkIfDateIsPassed 174
checkInput 213, 218, 238
checkWhichButtonPressed 237
commitCar 226
commitFlight 226
commitHotel 225
commitPackage 225
commitPackageController 224
commitRequestedCar 226
commitRequestedFlightSeats 226
commitRequestedHotelRooms 225
DbCconnectionAlias 461
DbConnectionAlias 250, 461
flightPreferences 178, 205
flightPrice 206
formatLocations 172, 239
formatMoney 206
generatePackageId 224
hotelPreferences 178, 205
hotelPrice 205, 209
initializeRequestedPackage 169
packaging application 421
priceCar 201
priceFlight 201
priceHotel 202
pricePackage 200
processCancel 181
processCar 181
processCarClass 194
processCarSelectionForm 198
processContinue 181
processCustomer 215, 219, 238

Index 503
processDateStrings 174
processDuration 174
processFlight 181, 191
processFlightSelectionForm 190
processHotel 180, 186
processHotelSelectionForm 185
processLocationName 175, 242
processNumberOfRooms 186
processNumberOfSeats 190
processPackageListForm 231
pumpUpFlight 233
pumpUpHotel 233
pumpUpRequestedCar 233
pumpUpRequestedFlightSeats 232
pumpUpRequestedHotelRooms 232
pumpUpRequestedPackage 231
queryDb 214, 219
queryForHotelsByLocation 184
queryForLocations 188
queryForPackages 229
reinitializeRequestedPackage 231
remove unused 422
routeToService 215
sayEndDate 178
sayStartDate 178
showAddress 236
showName 236
showPhoneNumber 237
startUp 461, 462
storeServiceTypeAndRoute 168
totalPrice 207
updateDb 220, 238
MIME headers 14
MIME-type 9, 10, 399
modeling 110

N
naming convention 256
NCSA 5
nesting forms 43
nonvisual part
  connecting to 49
  handle CGI request 52
nonvisual parts
  connecting to 49
  using 45
NULL 293, 294, 311

O
object
dynamic model 113, 123
event trace diagram 113
flattening 130
hierarchy 113
inheritance 113
interactions 110
mapping to tables 131
model 111, 113
notation 111
OMT 113
persistence 130
services 112
state changes 110
state transition diagram 115
static model 113, 126
streaming out 130
transferring 53
OMT 113
one-step packager 59

P
package
  application description 106
  complete old 368
  component selection 118
  confirmation 108
  creating 105
  DDL 451
  deletion 109
  inquiry 109
  price 119
  search 375
packaging 59, 419
packaging a run-time application 419
page router 53
parsing CGI programs 15
part
  basic 38
  behavior 36
  CGI Link 51
  composite 351
  construction from 23
  containing form 46
  data access 24
data debugging 59
  encapsulating query 354
  form 42
  hidden composite connections 356
hierarchy 121
HTML 31
HTML composite 41
HTML table 59
HTML table column 59
image 40
java proxy 24
link 44
link to 40
literal text 59
lookupTable 52
nonvisual 23, 33, 45
not requiring form 38
parent 356
persistence 47
placement 43
primary 57
promote feature 354
query 59
reusable 41
reusing 351
settings 65
testing 50, 59
type 57
visual 23, 33, 34
visual appearance 58
Web Connection 30
Part Interface Editor 268
password 341
PATH_INFO 380
Perl 11, 22, 32
persistent object 345
persistent storage
database based 254
flat-file based 254
profile-file based 254
pipe 254
PMTYPE:PM 63, 391
port 7, 380, 398, 425
POST 10, 43, 77, 81, 86
pragma 9, 343
preferences 40
preserving session data 47
problem domain 116
profile file 255
programming interfaces, low-level 463
prototype 120
proxy 9
push button
label 44
reset 43
submit 43
Q
query
AllCars 249
calling 99
CarPriceWhereClass 249
defining 95
encapsulating 354
flightQuery 249
getAllCustomers 249
getCustomerNameAndId 250
getHotelAllColumnsAllRows 250
getHotelsForALocation 250
getLocationAllColumnsAllRows 250
getLocationsFromDb 250
getPackage 250
multiple row 352
part 59
reusable 351
single row 359
specifications 352
SQL 95
QUERY_STRING variable 15
Quick HTML
result table 59
with single-row queries 359
quick HTML 351
R
referential integrity 133
referer 9
registry 255
reset button 43
reset push button 310
response headers 10
reuse 351
REXX 11
router 83, 302
routing requests to different locations 380
routing requests to different pages 53
run-time graphic 41
run-time image, creating 420
run-time setup 424
S
secure sockets layer 21, 100
serializing requests 467
session 47
definition 47
null
uniform resource identifier 10
Uniform Resource Locator 5
format 34
UNIX 32
URL 5
case sensitive 8
description 7
elements 8
format 34
Internet connection server 397
link 44
link to 40
port 7
referer 9
sections 7
submit request 13
use cases 110, 112, 116, 126
user ID 341
user interface analysis 120
user-interface controls 38

variable
conditional 95
DATABASE 95
hidden 101
list 95
override 99
special 100
SQL_CODE 99
VBBASE.VBB 90
VBE file 342
VBGENCODE
environment variable 74

visual part
adding text 39
behavior 36
building 37
composite 41, 351
flowed layout 37
packaging 420
placement 37
resize 37
reusable 41
reusing 351
user-interface controls 43
Web Connection category 37

W
Waikiki 62
Web browser
accessing applications from 34
animation 20
application flow 33
clients 8
communications session 21
description 4
HTML generation 31

VisualAge
access Web application 31
application flow 33, 37
application testing 37
call level interface 460
CGI Link 49
CGI programming 22
controls 22
creating parts 57
DB2 interface 459
debugging 59
distributed applications 25
distributed Web 22
distributing servers 30
enterprise distributed Web 23
file-in 354
image part 41
interactive Web 22
Java 455
Lotus Notes support 24
multiple TCP/IP addresses 425
one-step packager 59
packaged application 32
packaging 59, 419
palette 33
parts 161
quick HTML 354
reusing parts 351
run-time executable 32
run-time files 424
session key 47
SQL generator 353
support xi
Web application models 21
Web Connection feature 29
Web development 17, 32
VMT 105, 110, 113
Java 20
local file 7
push button labels 44
rendering 37
request 9
retrieve CGI variables 52
run-time graphic 41
session identification 47
sound 20
submit request 13, 33
supported image format 41
title bar 39
Web Connection parts 31
Web Connection application
accessing 34
packaging for runtime 419
sample, building
setting up at runtime 424
starting at runtime 424
Web Connection feature
application flow 33
architecture 33
category 30, 34
components 30
composition editor 31
defining links 44
description 29
introduction 30
Java subpart 455
nonvisual parts 33
packaging 419
part type 57
parts 30
performance 32
run-time files 424
setting up 378
tables 59
testing application 37
testing installation 382
testing parts 50
unsupported HTML extensions 44
using GUI parts 31
visual parts 33, 37
Web page
creating 39
HTML generated 39
routing requests 53
Web server 5, 6
administrator authentication 366
application flow 33
auditing 9
authorization 9
CGI 11
client/server 20
configuring 396
daemon 33
DB2 WWW 402
dynamic documents 12
executing transactions 6
firewall 101
home directory 398
hostname 398
incoming requests 31
interpreting response 14
Java applet 21
logging 9
multiple requests 49
off-load tasks 21
OS/2 Warp 5
packaged application 59
parsing request 12
performance 18
port number 380, 398
protecting 399
protecting directories 100
public-domain 5
receive request 33
remote administration 6
response generation 15
return directives 9
scalability 19
smalltalk process 49
starting 395
stateless connection 49
static documents 12
UNIX 5
updating files 6
Windows NT 5
call level interface 460
database connection specifications 460
DB2 interface 459
DB2 security service 459
wrapper 53
WTA
administrative functions 365
administrative tasks 109
administrator 109
analysis 105
application architecture 159
assumptions 107
authentication 366
car reservation 107
cleanup process 109
customer interactions 106
customer listing 109
database 134
database cleanup 110
database definition 451
database design 129, 132
database statistics 376
DDL 451
description 106
design 105
event trace diagrams 123
flight reservation 106
goal 105
hotel reservation 107
implementation 161
package confirmation 108
package deletion 109
package inquiry 109
part hierarchy 121
personal data modification 108
reset database 376
Smalltalk implementation 159
travel agent interactions 109
use cases 116
view descriptions 122
Windows NT port 459

WtaAboutView
CRC card 449
description 242

WtaAttractionProcessor
CRC card 448
description 240

WtaAttractionView
CRC card 448
description 239

WtaCarClassView
CRC card 437
description 194, 290

WtaCarSelectionProcessor
CRC card 436
description 197, 291

WtaCarSelectionView
CRC card 436
description 192, 288

WtaCommitter
CRC card 441
description 222, 315

WtaComponentSelectionProcessor
CRC card 432
description 179, 277

WtaComponentSelectionView
CRC card 432
description 175, 274

WtaCreditCardInfoView
CRC card 443
description 220, 314

WtaCustomer
CRC card 445

WtaCustomerAuthenticationProcessor
CRC card 444
description 213, 305

WtaCustomerAuthenticationView
CRC card 444
description 211, 305

WtaCustomerIdProcessor
CRC card 440
description 211, 304

WtaCustomerIdView
CRC card 440
description 209, 303

WTADB database 264, 340

WtaDeveloperView
CRC card 449
description 243

Java applet 455

WtaExceptionView
description 245

WtaFlight
CRC card 434

WtaFlightSelectionProcessor
CRC card 433
description 189
index 285

WtaFlightSelectionView
CRC card 433
description 285

WtaHotel
CRC card 435

WtaHotelSelectionProcessor
CRC card 435
description 184, 281

WtaHotelSelectionView
CRC card 434
description 279

WtaLocation
CRC card 449

WtaNewCustomerIdProcessor
CRC card 441
description 217, 311

WtaNewCustomerIdView
CRC card 440
description 216, 309

WtaPackageConfirmationView
CRC card 443
description 226, 323

WtaPackageCreationController
CRC card 430
description 168

WtaPackageCreationProcessor
CRC card 431
description 172, 267

WtaPackageCreationView
CRC card 431
description 170, 264

WtaPackageDetailsView
CRC card 447
description 234
WtaPackageFailedView
 CRC card 438
description 303
WtaPackageListProcessor
 CRC card 446
description 230, 325
WtaPackageListView
 CRC card 446
description 227, 324
WtaPackagePricer
 CRC card 438
description 199, 293
WtaPackagePurchaseProcessor
 CRC card 439
description 207, 302
WtaPackagePurchaseView
 CRC card 439
description 202, 298
WtaPersonalInfoProcessor
 CRC card 445
description 237, 336
WtaPersonalInfoView
 CRC card 444
description 235
WtaRequestedCar
 CRC card 437
WtaRequestedFlightSeats
 CRC card 434
WtaRequestedHotelRooms
 CRC card 436
WtaRequestedPackage
 CRC card 440
WtaWelcomeProcessor
 CRC card 430
description 167, 263
WtaWelcomeView
 CRC card 430
description 258