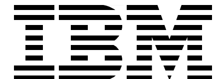
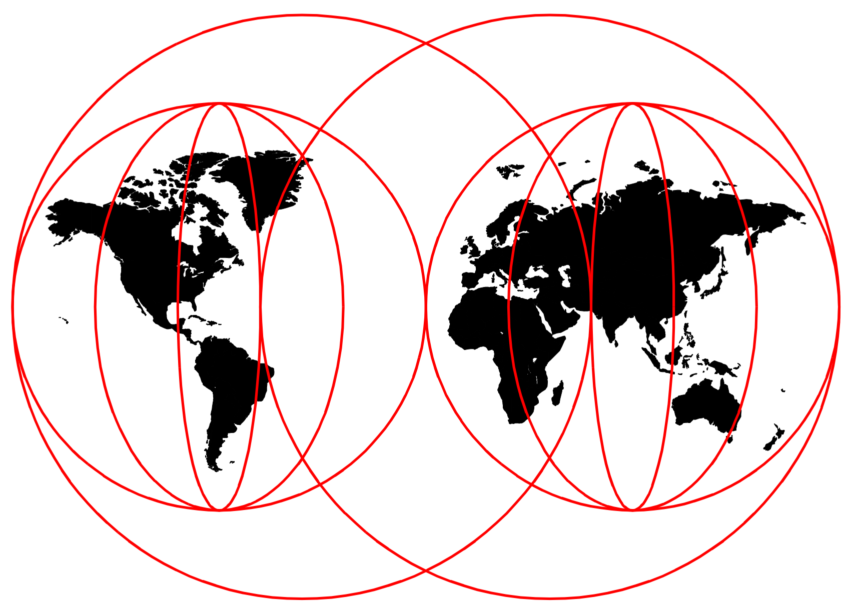


VisualAge for Java Enterprise Version 2: Data Access Beans - Servlets - CICS Connector

Olaf Graf, Avril Kotzen, Osamu Takagiwa
Ueli Wahli



International Technical Support Organization

http://www.redbooks.ibm.com

SG24-5265-00

SG24-5265-00

IBM

International Technical Support Organization

VisualAge for Java Enterprise Version 2: Data Access Beans - Servlets - CICS Connector

December 1998

Take Note!

Before using this information and the product it supports, be sure to read the general information in Appendix B, "Special Notices" on page 375.

First Edition (December 1998)

This edition applies to Version 2.0 of VisualAge for Java Enterprise, for use with the OS/2, Windows 95, or Windows NT operating system.

Sample Code on the Internet:

The sample code for this redbook is available as sg245265.zip file on the ITSO home page on the Internet:

```
ftp://www.redbooks.ibm.com/redbooks/SG245265
```

Download the sample code and read Appendix A.6, "Installation of the Redbook Samples" on page 373.

Comments may be addressed to:
IBM Corporation, International Technical Support Organization
Dept. QXXE Building 80-E2
650 Harry Road
San Jose, California 95120-6099

When you send information to IBM, you grant IBM a non-exclusive right to use or distribute the information in any way it believes appropriate without incurring any obligation to you.

© Copyright International Business Machines Corporation 1998.   All rights reserved
Note to U.S Government Users – Documentation related to restricted rights – Use, duplication or disclosure is subject to restrictions set forth in GSA ADP Schedule Contract with IBM Corp.

Contents

Figures xiii

Tables xix

Preface xxi
The Team That Wrote This Redbook xxii
Comments Welcome xxiii

Part 1. VisualAge for Java Enterprise Version 2 1

Chapter 1. Introduction 3
1.1 VisualAge for Java Version 2 Professional 4
Support for Java Development Kit 1.1.6 4
New Integrated Development Environment Features 4
New Visual Composition Editor Features 4
JavaBeans for Easy Access to Data 4
1.2 VisualAge for Java Version 2 Enterprise 5
Java Team Programming Support 5
Source Code Management Tools Integration 5
Open Tool Integrator APIs 5
Enterprise Toolkits for Workstation, AS/400, and OS/390 6
Enterprise Access Builders 6
Automated Object to Relational Mapping 7
Servlet Builder 7
IDL Development Environment 7
Support for SanFrancisco, Tivoli, Lotus, and Component Broker 8
AIX Development Environment 8

Chapter 2. Relational Database Access with Data Access Beans . 9
2.1 Overview 10
Data Access Beans versus Data Access Builder 11
2.2 Development Process with Data Access Beans 11
Loading the Data Access Bean Feature 11
Using the Select Bean 12
Development Process Step by Step 17
2.3 Building a Sample Application 19
Application Requirements 19
Development Process 20
Creating the Project and the Package 21

© Copyright IBM Corp. 1998

Creating the Sample Panel and the Select Bean 21
Building the User Interface 41
Improving the Select Bean 45
Run the Application 46
2.4 Summary 46

Chapter 3. Enterprise Application Development with Servlets . 47
3.1 Server-Side Applications 48
Common Gateway Interface 48
Servlets 48
What Are Servlets? 49
Servlet Creation Tools 51
Web Server Consideration 52
3.2 Inside Servlets 52
Simple Servlet 53
Invoking a Servlet in HTML 54
Invoking a Servlet with Parameters 56
HttpServlet 57
Complex Servlets 58
3.3 Servlet Builder Overview 59
How Do Servlet Builder Beans Work? 59
Advantages of the Servlet Builder 60
Visual Servlet 61
Servlet Builder Visual Beans 62
Servlet Builder Nonvisual Beans 65
Form Data 65
Cookie Wrapper 66
Session Data Wrapper 67
Run Configuration 67
Invoking Another Servlet 67
3.4 Creating Visual Servlets 68
Simple Servlet 69
Server-Side Include Servlet 71
Counter Servlet 71
Passing Data to the Servlet 73
3.5 Advanced Servlet Techniques 75
Advanced HTML Tags 75
Servlet Chaining 77
Keeping and Passing Data between Servlets 81
Servlet Branch 84
Condition Control 85
Disable Caching of Generated HTML 87
Servlet with JDBC 88

Chapter 4. CICS Access with the CICS Connector 91
4.1 The Enterprise Access Builder 92
4.2 Connectors 92
4.3 The CICS Connector 93
CICS Connector Installation 93
CICS Connector Classes. 94
4.4 CICS Universal Clients 95
Communcation Protocols 95
Client Customization 96
Client Functions 98
4.5 CICS Transaction Gateway. 100
What the CICS Transaction Gateway Provides 101
How the CICS Transaction Gateway Accesses CICS 101
4.6 A Discussion Review 103
4.7 Accessing Enterprise Data 103
Overview 104
Structure Description. 105
Records and the Java Record Framework. 105
Record Bean Generation 109
Commands. 111
Navigators. 117
Business Objects. 119
Mappers. 119
Executing the Command 123
4.8 A Review of Accessing Enterprise Data 124

Part 2. Implementing the ATM Application .125

Chapter 5. ATM Application Requirements and ATM Database 127
5.1 ATM Application Requirements 128
5.2 ATM Database Implementation 130
Sample Data of ATM Tables 133

Chapter 6. ATM Application Business Model 137
6.1 Application Design. 138
Application Layers 138
Application Layer Architecture 138
6.2 Business Object Layer 140
Business Logic Classes. 142
Testing the Business Objects. 152
6.3 Application Controller 154
Persistence Layer Interface 154
Controller Interface 155

Controller and Persistence Interfaces 157
Implementing the Controller 158
6.4 Persistence Layer. 162

Chapter 7. ATM Application Persistence Using Data Access Beans 163
7.1 Persistence Layer Design 164
7.2 Database Access with ATM Database Beans 165
PIN Validation 165
List of Accounts. 174
Debit and Credit Transactions. 175
Transaction History 177
7.3 Business Object Creation with ATM Database Beans 178
PIN Validation 178
List of Accounts. 180
Debit and Credit Transactions. 182
Transaction History 184
7.4 Implementing the Persistence Interface 186
AtmDB Bean 186
Testing the Implementation of the Persistence Interface 188
7.5 Preparation for Servlet Usage. 189

Chapter 8. Swing GUI for ATM Application 191
8.1 Design of the GUI Application 192
Application Controller 193
Panel Design 193
8.2 Implementation of the Application Panels. 195
Card Panel 195
PIN Panel 196
Select Account Panel 198
Transaction Panel. 200
ATM Applet. 202
8.3 Running the ATM GUI Applet 204

Chapter 9. ATM Application Using Servlets 205
9.1 Create a Skeleton Controller Servlet 206
9.2 Servlet Views 206
Card Servlet 206
PIN Servlet 209
Account Servlet. 212
Transaction Servlet 216
Thank You Servlet 220
9.3 Application Flow Design 221
9.4 Implementing the Controller Servlet 223

Preparation for Testing 223
Initialization 223
Customer Verification 225
PIN Verification 226
Account Selection 227
Deposit Transaction 228
Withdraw Transaction 229
Query Transaction History 230
Termination and Restart 231
Disable Caching of the Output HTML 233
Controller Servlet Total Design 234
9.5 Testing the ATM Servlet Application 235
Built-in HTTP Server 235
Using the WebSphere Application Server 235
Using the ATM Servlet Application with DB2 236
9.6 Deploying Servlets 236

Chapter 10. ATM Application with the CICS Connector 239
10.1 A Review of the ATM Application Design 240
The Persistence Interface 240
10.2 Task Overview 242
Conventions. 242
Only a Subset of the Interface Methods 242
CICS Infrastructure Assumptions 242
CICS Programs 242
Tasks Implemented 243
10.3 CICS Infrastructure Requirements 243
CICS Server Resources. 243
CICS Client Configuration and Startup 244
Starting the CICS Transaction Gateway 245
10.4 Initial Creation of AtmCICS Class 245
10.5 ATM Header for the COMMAREA 245
10.6 CICS Transaction to Retrieve an ATM Card 246
CICS COBOL Program ATMCARDI 247
Card Record Bean 248
Card Command 250
Building a Navigator to Execute the CICS Transaction 252
Implement the extGetCard Method. 254
10.7 Using Mappers 255
Input Mapper for Card 255
Output Mapper for Card 256
Create a Command with Mappers 257
Execute the CICS Transaction with Mappers 257

Change the AtmCICS Class to Use the Mappers 259
10.8 Test the CICS Card Transaction. 260
Prepare Test Output for Card Transaction. 260
Testing Card Transaction with a Scrapbook Script 261
Testing without CICS. 262
10.9 Discussion Review 262
10.10 CICS Transaction to Retrieve Accounts 263
CICS COBOL Program ATMACCNT 263
Accounts Record Bean 264
Accounts Input Mapper 266
Accounts Command 266
Navigator to Execute the CICS Accounts Transaction. 267
Implement the extGetAccounts Method 270
10.11 Testing the CICS Accounts Transaction. 270
Prepare Test Output for Accounts Transaction 270
Testing the Accounts Transaction with a Scrapbook Script 271
10.12 Testing the ATM Application with CICS 272
Testing the Real Application 272
10.13 Using an Advanced Navigator 273
Design of a Navigator. 273
Implementation of the Navigator 274
Testing the Navigator. 276
10.14 Implementation of the Back-End Programs. 277
10.15 Conclusion 277

Chapter 11. ATM Application Using MQSeries. 279
11.1 A Brief Overview of MQSeries 280
Messages and Queues. 280
MQSeries Objects 281
MQSeries Clients and Servers. 286
11.2 MQSeries Version 5 287
11.3 About MQSeries and Java. 287
MQSeries Client for Java. 287
MQSeries Bindings for Java 288
The MQSeries Java Programming Interface 289
11.4 Implementing the ATM Application with MQSeries. 291
11.5 MQSeries Queue Manager and Objects 292
Create a Queue Manager 292
Define MQSeries Objects 292
Command File to Start the Queue Manager. 294
11.6 Importing MQSeries into VisualAge for Java 294
11.7 Create an MQAccess Bean 295
Sample MQSeries Package 295

MQAccess Bean. 295
Implement the MQAccess Methods 297
11.8 ATM MQSeries Design Choices 304
Conforming to the ATM Model 305
Unit of Work Considerations 305
11.9 ATM Request Classes 306
AtmRequest Class. 306
Card Request. 308
Accounts Request 308
11.10 ATM Response Classes 309
AtmResponse Class 309
Card Response 310
Accounts Response 311
11.11 ATM Access Classes. 312
Card Access Class 312
Account Access Class 314
11.12 Persistence Interface with MQSeries 315
AtmMQ Class 315
11.13 Adding Additional Transactions 318
Create a Class for the MQSeries Request 319
Create a Class for the MQSeries Response 320
Create a Transaction-Specific Access Class 320
Modify the AtmMQ Class. 322
Create a Back-End Application Program 324
11.14 Back-End Programs. 324
Java Back-End Server Program 324
Testing the ATM MQSeries Server 330
Testing the ATM Application with MQSeries. 331
CICS COBOL Back-End Programs 332

Chapter 12. Deployment of the ATM Application Implementations
. . . . 333
12.1 Deployment of Applications. 334
Prerequisites for Applications 334
Exporting an Application from VisualAge for Java 334
Deployment Process for Applications 335
12.2 Deployment of Applets. 336
Exporting Applets from VisualAge for Java 336
Deployment Process for Applets 336
12.3 Deployment of Servlets 338
Deployment of Servlets for Lotus Domino Go Webserver 339
Target Location. 340
Class Path Setting for Web Server 340

12.4 Deployment of Applications with Swing 344
12.5 Tailoring the Web Browser 344

Chapter 13. High-Performance Compiler and Remote Debugger 345
13.1 High-Performance Compiler 346
Compiler Options 346
Base Java Classes 346
Swing Classes 347
Execution 347
13.2 Compiling the ATM Application 347
Export the ATM Application 347
Compile the ATM Application 348
Compile the Data Access Beans 348
Compile the DB2 JDBC Drivers 349
Remove the Object Files 349
13.3 Run the Compiled ATM Application 350
13.4 Alternative Compile Approach 351
13.5 Remote Debugger 353
Reasons for Remote Debugging 353
Running the Remote Debugger 353
13.6 Remote Debugging of the ATM Application 354
13.7 Debugging a Compiled Program 357

Appendixes 359

Appendix A. Installation, Setup, and Configuration 361
A.1 Setup of VisualAge for Java Enterprise Version 2 362
A.2 Setup for Data Access Beans 363
A.3 Setup for the Servlet Builder 364
Web Server 364
A.4 Setup for the CICS Connector 365
Setup of the CICS Server and Client 365
COBOL Sample Programs 366
A.5 Setup for MQSeries on Windows NT 369
Installation Considerations 369
Queue Manager and Queue Setup 369
VisualAge for Java Setup 369
MQSeries CICS Bridge Program 369
A.6 Installation of the Redbook Samples 373

Appendix B. Special Notices 375

Appendix C. Related Publications 379
C.1 International Technical Support Organization Publications 380
C.2 Redbooks on CD-ROMs 381
C.3 Other Publications 381

How to Get ITSO Redbooks 383
How IBM Employees Can Get ITSO Redbooks 383
How Customers Can Get ITSO Redbooks 384
IBM Redbook Order Form 385

List of Abbreviations 387

Index 389

ITSO Redbook Evaluation 395

Figures

1. Select Bean Result Set in Memory 13
2. Development Process with Data Access Beans 18
3. Data Access Beans Sample Application 19
4. Placing a Select Bean on the Free-Form Surface 22
5. Query Property Editor of the Select Bean 24
6. Specification of the Database Access Class 25
7. Specification of the Connection Alias Definition 26
8. Query Property Editor Connection Page 28
9. Query Property Editor SQL Page 29
10. Define a new SQL Specification 30
11. SQL Assist SmartGuide Tables Page (required) 31
12. Schemas to View and Table Name Filters 32
13. SQL Assist SmartGuide Join Page (optional) 33
14. SQL Assist SmartGuide Condition Page (optional) 34
15. Search for Column Values in the Value Lookup Window 35
16. SQL Assist SmartGuide Columns Page (optional) 36
17. SQL Assist SmartGuide Sort Page (optional) 37
18. SQL Assist SmartGuide Mapping Page (optional) 39
19. SQL Assist SmartGuide SQL Page (optional) 40
20. Query Property Editor with New SQL Specification 41
21. Sample Application User Interface and Database Connection 43
22. Sample Application Logic to Display the Employee Photos 44
23. HTTP Transactions 49
24. HTTP Session and Servlet 50
25. Simple Servlet Source Code 53
26. Server-Side Include HTML File 55
27. Servlet with HTTP Server-Side Include 55
28. Server-Side Include Counter Servlet Source Code 56
29. HTML File with Form Invoking a Servlet 56
30. Servlet Processing with Form Data 57
31. Servlet with Post Processing 58
32. Servlet Class Hierarchy 60
33. Visual Servlet 61
34. Flow between Servlets 66
35. Adding the Servlet Builder Feature to the Workbench 68
36. Create Servlet SmartGuide 69
37. Servlet Palette and HTML Page in Visual Composition Editor 70
38. Code Property for a Server-Side Include Servlet 71

© Copyright IBM Corp. 1998

39. HTML Page with a Counter . . . 72
40. Servlet with Counter in Netscape Browser . . . 72
41. Interactive Servlet . . . 74
42. Style Sheet Specification . . . 76
43. JavaScript Invocation . . . 76
44. Servlet Chaining in Single Flow . . . 77
45. Sign On Servlet . . . 78
46. Data Entry Servlet . . . 78
47. Service Handler Specification for Form Action . . . 79
48. Conversion Servlet . . . 80
49. Data Entry Servlet with Cookie and Session Data . . . 83
50. Conversion Servlet with Cookie and Session Data . . . 83
51. Servlet Branch. . . . 84
52. Branch Form . . . 84
53. Condition Control Servlet . . . 85
54. Controller Servlet . . . 86
55. Post Servlet with Caching Disabled. . . . 87
56. Disabling Caching in a Visual Servlet . . . 87
57. Servlet with Data Access Bean . . . 89
58. Servlet with Data Access Bean Browser Result . . . 90
59. Client Initialization File Syntax . . . 97
60. Extract for Customized Client Initialization File . . . 98
61. CICS Transaction Gateway . . . 100
62. Record Bean Creation . . . 107
63. SmartGuide for COBOL Record Type . . . 108
64. Generate Records SmartGuide. . . . 109
65. Generate Records SmartGuide: Changing Properties . . . 110
66. Command Construction . . . 112
67. Command Editor: Initial View . . . 114
68. Command Editor with Communication, Input, and Output Beans . . . 116
69. Visual Composition Editor View of a Command . . . 116
70. Navigator . . . 117
71. Construction of a Navigator . . . 118
72. Mapper Editor . . . 121
73. Command Editor with Mappers . . . 122
74. Visual Composition Editor View of Command with Mappers . . . 123
75. ATM Application Layers and Implementations . . . 126
76. ATM Application Panels and Flow. . . . 129
77. Relationships among the ATM Tables . . . 130
78. ATM Database Data Definition Language . . . 132
79. ATM Database Sample Data Load. . . . 135
80. Layers of the ATM Application . . . 139
81. Object Model of the ATM Business Object Layer . . . 141

82. Defining an Event with an Event Listener (First Page) 144
83. Defining an Event with an Event Listener (Second Page) 144
84. Scrapbook Script for Testing the Business Model 153
85. Controller and Persistence Interfaces 157
86. Select Statement for Customer Information and PIN Validation . . . 166
87. Specification of the Database Access Class 167
88. Connection Alias Definition for the ATM Application 168
89. New SQL Specification 169
90. SQL Assist SmartGuide: ATM Table Specification. 170
91. SQL Assist SmartGuide: ATM Join Specification. 171
92. SQL Assist SmartGuide: Condition Specification 172
93. SQL Assist SmartGuide: Columns Specification 173
94. SQL Assist SmartGuide:View and Test the SQL Statement 174
95. Select Statement to Retrieve Accounts of a Card 175
96. Select Statement to Update the Account Balance. 175
97. Select Statement to Retrieve All Transactions 176
98. Select Statement to Retrieve the Transactions of an Account 177
99. Visual Composition of PinCustInfo Bean 178
100. Visual Composition of Accounts Bean 180
101. Visual Composition of UpdateBalance Bean 182
102. Visual Composition of Transactions Bean 184
103. Scrapbook Script for Testing the AtmDB Bean. 188
104. Updated Connection Specification for the ATM Application 189
105. ATM Application Panels 192
106. GUI Application with Application Controller 193
107. Visual Composition of the Card Panel 195
108. Visual Composition of the PIN Panel 196
109. Visual Composition of the Select Account Panel 198
110. Visual Composition of the Transaction Panel 200
111. Visual Composition of the ATM Applet 202
112. Card Servlet View 206
113. Card Servlet Design 207
114. PIN Servlet View. 209
115. PIN Servlet Design 210
116. Account Servlet View 212
117. Account Servlet Design. 213
118. Transaction Servlet View 216
119. Transaction Servlet Design 217
120. Thank You Servlet View. 220
121. Servlet Application Flow 221
122. Initializing the Controller Servlet 224
123. Customer Verification. 225
124. PIN Verification 226

125. Account Selection 228
126. Deposit Transaction 229
127. Withdraw Transaction 230
128. Query Transaction History. 231
129. Termination and Restart 232
130. Disabling Caching for the ATM Servlets 233
131. Controller Servlet Total Design 234
132. Client Listener Definition. 244
133. Checking the Return Code of the CICS Transaction 246
134. COMMAREA of the ATMCARDI Program 247
135. Card Record Type Creation: Class and COBOL File 248
136. Card Record Type Creation: COMMAREA Selection 249
137. Card Record Bean Generation 250
138. CardCommand with CICSConnectionSpec Properties 251
139. Visual Composition of CICSCardNavigator Class 253
140. Implementation of extGetCard 254
141. Mapper Editor for Input Record Mapping 256
142. Command Editor with Mappers. 257
143. Visual Composition of the CICSCardMapperAccess Class. 258
144. Extracting Objects from a Command. 258
145. Implementation of Enhanced extGetCard. 260
146. Scrapbook for CICS Card Transaction Testing 261
147. COMMAREA of the ATMACCNT Program. 264
148. Edit of Accounts Record Type. 265
149. Classes Generated from Accounts Record Type 265
150. Visual Composition of CICSAccountsNavigator 267
151. Code Listing of createAccountsVector Method 269
152. extGetAccounts Method 270
153. Scrapbook Script for CICS Accounts Transaction Testing 271
154. Scrapbook Script for CICS Application Testing 272
155. Visual Composition of the Navigator 274
156. Scrapbook Script for Advanced Navigator. 276
157. Method to Invoke the Advanced Navigator 277
158. Local Queue Attributes. 282
159. Message Flow across a Channel. 284
160. Two-way Message Channel Communication. 284
161. MQI Channel Definition 285
162. Use of an MQI Channel 285
163. MQSeries Client to Server Flow. 286
164. MQSeries Objects for ATM Application. 293
165. MQSeries Startup Command File 294
166. connectToQmgr Method 298
167. disconnectFromQmgr Method 298

168. openQueue(String, int) Method 299
169. openQueue(String) Method 300
170. closeQueue Method 300
171. putRequestMessage Method. 301
172. putRequestMessage Method for ATM Requests 302
173. retrieveSpecificMessage Method 303
174. getHeader Method. 307
175. toString Method 307
176. Request Trigger Method 308
177. getCard Method. 310
178. getAccounts Method 311
179. Visual Composition of Card Access Class 313
180. Visual Composition of Accounts Access Class 314
181. Properties of MQAccess Beans 316
182. Visual Composition of AtmMQ Class. 317
183. extGetCard Method. 318
184. extGetAccounts Method 318
185. Visual Composition of Update Balance Access Class 322
186. Visual Composition of AtmMQ with Update Balance. 323
187. extUpdateBalance Method 323
188. Visual Composition of the ATM MQSeries Server Class 325
189. Scrapbook Script for Testing the ATM MQSeries Server 330
190. Deployment Process for Applications. 335
191. Deployment Process for Applets. 337
192. Deployment Process for Servlets in Lotus Domino Go Webserver. . . 339
193. WebSphere Application Server: Administration 341
194. WebSphere Application Server: Manage Configuration 342
195. WebSphere Application Server: Servlet Configuration Basic Page . . 343
196. Remote Debugger: Session Control Window 354
197. Remote Debugger: Source Window 355
198. Remote Debugger: Source Window with Breakpoint 356
199. Remote Debugger: Program Monitor Window. 356

Tables

1. DBNavigator Push Buttons . . . 16
2. Mapping between SQL Data Types and Java Classes . . . 38
3. Web Servers Supporting Servlets . . . 52
4. Methods of the ServerRequest Class . . . 54
5. Methods of the ServerResponse Class . . . 54
6. Servlet Builder Visual Beans . . . 63
7. Servlet Builder Visual Beans for Forms . . . 64
8. Servlet Builder Nonvisual Beans . . . 65
9. Client Environment Variables . . . 96
10. Customer Table . . . 130
11. Card Table . . . 131
12. Account Table . . . 131
13. Transaction Table . . . 131
14. Customer Table Sample Data . . . 133
15. Card Table Sample Data . . . 133
16. Account Table Sample Data . . . 134
17. Transaction Table Sample Data . . . 134
18. ATM Persistence Interface Methods . . . 155
19. ATM Application Controller Methods . . . 156
20. ATM Application Controller Events . . . 156
21. ATM Database Beans . . . 165
22. GUI Beans in Card Servlet . . . 208
23. GUI Beans in PIN Servlet . . . 210
24. GUI Beans in Account Servlet . . . 213
25. GUI Beans in Transaction Servlet . . . 218
26. ATM Persistence Interface Methods . . . 241
27. MQAccess Bean Properties . . . 296
28. MQAccess Bean Method Features . . . 297
29. Methods of the AtmRequest Class . . . 307
30. Card Request Class . . . 308
31. Accounts Request Class . . . 309
32. Methods of the AtmResponse Class . . . 309
33. Card Response Class . . . 310
34. Accounts Response Class . . . 311
35. Properties of the Card Access Class . . . 312
36. Properties of the Account Access Class . . . 314
37. AtmMQ Methods Implemented for ATM Application . . . 317
38. Update Balance Request Class . . . 319

© Copyright IBM Corp. 1998

39. Properties of the Update Balance Access Class 322
40. Methods of the ATM MQSeries Server Class 325
41. Servlet Deployment Specifications 340
42. High-Performance Compiler Options 346
43. CICS COBOL Programs. 366
44. Redbook Sample Code 373
45. Packages of the Redbook Sample Applications. 374

Preface

VisualAge for Java Enterprise Version 2 is the second generation of the award-winning VisualAge for Java product. Version 2 extends the connectivity to enterprise data and applications through a set of new enterprise access builders:

- Data access beans for easy access to relational databases
- Servlet Builder for visual constructions of servlets
- E-Connectors to connect to enterprise transactions servers, including CICS and Encina
- Persistence Builder to automate the mapping of a business model into a relational database

Version 2 also includes a high-performance compiler, a remote debugger, team programming support, the Java Foundation Classes (Swing), an IDL development environment, support for SAP/R3, Lotus Notes, Component Broker, Tivoli, San Francisco Framework, and an open tool integration API.

In this book we concentrate on data access beans, the Servlet Builder, and the CICS Connector. We also touch on the high-performance compiler and debugger.

The book demonstrates a practical approach to using VisualAge for Java Enterprise Version 2. A sample ATM bank application is used throughout the book to illustrate the use of the new enterprise access builders.

© Copyright IBM Corp. 1998

The Team That Wrote This Redbook

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

Olaf Graf is an AD Technical Support Specialist working for IBM Techline Germany. He provides pre- and post-sales technical support to customers across Europe, the Middle East, and Africa. His areas of expertise include application development, databases, and VisualAge products. He is a Sun Certified Java JDK 1.1 Programmer. Before joining CSG, a subsidiary of IBM Germany, two years ago, Olaf worked as a software developer on a research project entitled "Sleep Analysis by Means of Neural Networks." Olaf holds a degree in Electrical Engineering from the Technical University of Ilmenau, Germany.

Avril Kotzen is an IT specialist in South Africa. She provides product support for CICS and MQSeries. Her roles include designing end-to-end integration solutions. Avril has recently worked on a Lotus Notes to CICS integration solution using MQSeries.

Osamu Takagiwa is an advisory IT Specialist working for IBM Japan Systems Engineering Co., Ltd. He provides technical support to customers in Japan. Osamu is a Sun Certified Java JDK 1.1 Programmer and an IBM Certified VisualAge for Java Developer. He wrote an entry-level book for VisualAge for Java in Japanese. Osamu has recently worked on a server-side Java solution using WebSphere Application Server and VisualAge for Java.

Ueli Wahli is a Consultant AD Specialist at the IBM International Technical Support Organization in San Jose, California. Before joining the ITSO 14 years ago, Ueli worked in technical support at IBM Switzerland. He writes extensively and teaches IBM classes worldwide on application development, object technology, VisualAge products, data dictionaries, and library management. Ueli holds a degree in Mathematics from the Swiss Federal Institute of Technology. His e-mail address is *wahli @ us.ibm.com.*

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

- Derek Carter and Becky Nin, IBM Santa Teresa Laboratory
- Dean Williams and Sheldon Wosnick, IBM Toronto Laboratory
- Scott Rich, Lawrence Smith, and Joe Winchester, IBM RTP Laboratory
- Hanspeter Nagel and Emma Jacobs, ITSO San Jose Center
- Maggie Cutler, ITSO San Jose Center, for her outstanding editing work

Comments Welcome

Your comments are important to us!

We want our redbooks to be as helpful as possible. Please send us your comments about this or other redbooks in one of the following ways:

- ❑ Fax the evaluation form found in “ITSO Redbook Evaluation” on page 395 to the fax number shown on the form.
- ❑ Use the electronic evaluation form found on the Redbooks Web sites:

  For Internet users `http://www.redbooks.ibm.com`
  For IBM Intranet users `http://w3.itso.ibm.com`

- ❑ Send us a note at the following address:

  `redbook@us.ibm.com`

Part 1 VisualAge for Java Enterprise Version 2

In Part 1 we describe the new functions of VisualAge for Java Enterprise Version 2.

We give an introduction of all the new functions and then concentrate on three key features:

- ❑ Data Access Beans
- ❑ Servlet Builder
- ❑ CICS Connector

© Copyright IBM Corp. 1998

1 Introduction

In this chapter we give a short overview of the new function of VisualAge for Java Version 2. In subsequent chapters we describe some of the new function in detail.

VisualAge for Java Version 2 is available as two products:

- ❑ VisualAge for Java Version 2 Professional
- ❑ VisualAge for Java Version 2 Enterprise

© Copyright IBM Corp. 1998

1.1 VisualAge for Java Version 2 Professional

VisualAge for Java Version 2 Professional provides the new function listed here. **The redbook, *Programming with VisualAge for Java Version 2*, SG24-5264, provides a detailed description of the new function of the Professional edition**.

Support for Java Development Kit 1.1.6

VisualAge for Java 2.0 supports the JDK 1.1.6. This support includes Swing 1.0.2, inner classes, and anonymous classes, and the Java Native Interface (JNI). **For more details on Swing, see Chapter 8, “Swing GUI for ATM Application.”**

New Integrated Development Environment Features

The integrated development environment (IDE) provides:

- Advanced coding tools such as automatic formatting, automatic code completion, and fix-on-save
- Context-sensitive help
- Advanced debugging tools such as conditional breakpoints and both multiple and incremental program debug
- Support for JavaDoc output
- Enhanced searching capabilities

New Visual Composition Editor Features

The Visual Composition Editor provides:

- Visual programming support for Swing beans
- Wizards for string externalization to assist in building multilanguage applications
- Complete support for object serialization
- Ability to import GUIs built in other Java IDEs

JavaBeans for Easy Access to Data

New data access beans give your Java application the power to access relational data from any Java Database Connectivity (JDBC) enabled database and make it available on the Web. **For more details, see Chapter 2, “Relational Database Access with Data Access Beans” and Chapter 7, “ATM Application Persistence Using Data Access Beans.”**

1.2 VisualAge for Java Version 2 Enterprise

VisualAge for Java Version 2 Enterprise provides the new function listed here.

Java Team Programming Support

The ultimate quality of your Java applications depends on how well you manage your development process. VisualAge for Java includes a built-in source code and version control system that provides you with a complete audit trail of your project and helps in recovery from undesired code changes.

In addition to source code and version control, Enterprise Edition users also get a fully integrated team development environment that improves productivity and reuse levels for any size team. Each developer gets a personalized workspace that is integrated with a collaborative repository providing fine-grained versioning of individual components, change identification, and impact analysis across multiple projects. This tight integration avoids time-consuming switching between the repository and the development environment and gives every developer instant “live access” to a library of reusable components.

For more details on the team support, consult the redbook, *VisualAge for Java Version 2 Team Support*, SG24-5245.

Source Code Management Tools Integration

If you are developing on the Windows platform, you can also check in or check out your VisualAge for Java code to or from either VisualAge TeamConnection, ClearCase, or PVCS.

Open Tool Integrator APIs

Advanced users and commercial software developers who need to extend VisualAge for Java can use the tool integrator API to:

- Add third-party tools that are launched from within the IDE
- Store and retrieve components from the integrated repository
- Add JavaBeans to the Visual Composition Editor's parts palette

Enterprise Toolkits for Workstation, AS/400, and OS/390

The increasing popularity of Java as a server language has placed new requirements for application scalability on Java development shops. Enterprise Edition 2.0 is ready to meet those requirements with a new high-performance compiler for Java that maximizes the execution speed of your server code.

We have also filled VisualAge for Java's toolkit with cross-platform debugging, testing, and performance analysis tools that are accessed from your development workstation and that target applications built to run on OS/2, Windows NT, AIX, OS/400, and OS/390. Plus, the VisualAge for Java remote debugger tests and debugs interpreted Java, compiled Java, and C++ on multiple platforms, giving you a true multitier development environment. New for S/390 developers is JPort, which prescreens Java programs to ensure OS/390 portability and profiles OS/390 Java applications to detect performance bottlenecks.

For more details, see Chapter 13, "High-Performance Compiler and Remote Debugger."

Enterprise Access Builders

Extending existing enterprise application servers to the Web is a critical success factor for leading-edge IT shops. VisualAge for Java Enterprise Edition provides a collection of Enterprise Access Builders that give you access to enterprise systems such as relational data, CICS transactions, or SAP R/3 applications from your Java programs.

VisualAge for Java's unique approach lets you access multiple systems from a single Java application and uses a consistent programming interface across diverse enterprise systems to reduce your learning curve, maximize your productivity, and increase the run-time performance of your applications.

VisualAge for Java's Enterprise Access Builders include:

- Access Builder for CICS including CICS ECI, CICS EPI, CICS EXCI. **For more details, see Chapter 4, "CICS Access with the CICS Connector" and Chapter 10, "ATM Application with the CICS Connector."**
- Access Builder for Encina using the DCE Encina Lightweight Client
- Access Builder for SAP R/3 using SAP R/3 Business Objects
- Access Builder for Data for JDBC access to enterprise data
- Access Builder for J2C++ for access to C++ programs
- Access Builder for Remote Method Invocation (RMI) for creating distributed Java applications

Automated Object to Relational Mapping

A new Enterprise Access Builder for Persistence provides a set of tools that automate the task of mapping the persistent state of Java objects to relational databases. These tools generate a layer of code that implements all of the JDBC access calls necessary to insert, update, or retrieve the data for an object from an SQL database.

The programming model used to create the persistent Java objects is based on the industry standard Enterprise JavaBeans (EJB) Architecture. The Enterprise Access Builder for Persistence and its generated Java code will be upward compatible to support full EJBs in a future release of VisualAge for Java. This functionality will coincide with the delivery of IBM’s Enterprise Java Server (EJS) environments, such as IBM’s WebSphere Application Server and Component Broker.

Servlet Builder

Enterprise Edition users can now use visual programming techniques to create and test servlets. With Servlet Builder, a wide variety of custom and off-the-shelf business objects can be Web-enabled and used within the VisualAge for Java reusable parts library. When Servlet Builder is used along with the IBM WebSphere Application Server, site builders can test and debug a combination of pages built using pure HTML, compiled Java Server Pages (JSP), and Servlet Builder visual servlets.

For more details, see Chapter 3, “Enterprise Application Development with Servlets” and Chapter 9, “ATM Application Using Servlets.”

IDL Development Environment

The Enterprise Edition provides an integrated development environment for CORBA-based applications. Interface Definition Language (IDL) descriptors can be stored in the repository, together with the Java source code that is generated using IDL-to-Java compilers. Products that implement the CORBA standard can be invoked from the VisualAge for Java IDE to develop and test Java applications that communicate using Internet Inter-ORB Protocol (IIOP).

For more information, consult the redbook, *Using VisualAge for Java Enterprise Version 2 to Develop CORBA and EJB Applications*, SG24-5276.

Support for SanFrancisco, Tivoli, Lotus, and Component Broker

IBM has a rich portfolio of Java-based solutions, and VisualAge for Java is the tool of choice for developing many of these systems:

- Enterprise Edition includes SanFrancisco wizards for building applications from the SanFrancisco Business Application Components.
- VisualAge for Java can also be used to build Java-based business productivity applications using the Lotus eSuite components and to develop, debug, and test Lotus Notes Agents.
- Version 2.0 includes new Tivoli Beans used to make Java applications "ready to manage" with Tivoli's enterprise management software. Tivoli lets you easily track activities such as version upgrade, daily use monitoring, and operation and distribution to target systems. VisualAge for Java's complete IDL environment can be used to create and manage applications that can communicate with CORBA business objects, such as those deployed on IBM's Component Broker application server.

AIX Development Environment

Now, with Version 2.0, you can use AIX 4.2 and 4.3 workstations as your development platform.

2 Relational Database Access with Data Access Beans

In this chapter we look at new ways of accessing data outside VisualAge for Java Version 2, specifically, how to access relational databases through data access beans. The data access bean feature is included in both VisualAge for Java Professional and Enterprise Version 2. We provide detailed information about this new feature that facilitates retrieving and updating existing SQL databases.

In Version 2 of VisualAge for Java access to relational databases is provided in three ways:

- ❑ Data access beans (covered in this chapter)
- ❑ Persistence framework (a topic for a future redbook)
- ❑ Data Access Builder (unchanged from VisualAge for Java Version 1)

Now you can decide for yourself how sophisticated, but also how complex your relational database development with VisualAge for Java should be.

© Copyright IBM Corp. 1998

2.1 Overview

Using data access beans is the fastest, nonprogramming way of building SQL queries accessing existing databases. Just open the Visual Composition Editor, place a Select bean on the free-form surface, specify the database connection and data you need, and you are ready.

Most functionality is predefined. For example, your application can add, update, and delete rows, commit or rollback database transactions, handle multiple connections, lock rows, make the access read only, and specify how many rows are fetched into memory (cache).

An SQL Assist SmartGuide helps you to visually specify the data you need. You can select one or more tables, join tables, define search conditions, restrict the number of columns, sort the result set, and change the mapping between the SQL types in the database and the Java types in the application.

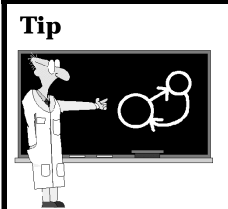
In addition, an SQL Editor lets you enter SQL statements manually. Use this method when you need to compose very special or very sophisticated database queries.[1]

After you have defined your database access using a Select bean, you can place a DBNavigator bean into your visual application. The DBNavigator bean incorporates buttons that navigate the result set of a query and perform various relational database operations.

The Select bean fits into the JTable model of the new Java Foundation Classes (JFC, called Swing). This relationship between retrieved SQL data and the Swing table model makes it very easy to develop attractive user interfaces for Java applets and applications, based on standard Java classes.

With data access beans you have direct access to any database for which a JDK 1.1 compliant JDBC driver exists, for example, DB2 Universal Database (UDB) Version 5.[2] Alternatively, you can use an Open Database Connectivity (ODBC) driver together with the JDBC-ODBC bridge that comes with the Sun JDK.

[1] SQL Editor generated code, in contrast to SQL Assistant generated code, cannot handle SQL parameters (host variables) because the statement is not parsed. The developer or the application code can add the parameter definitions manually to the generated methods to enable passing of values into the host variables.
[2] You can also use DB2 V2.12, plus the latest CSD.

Tip	
	We assume that you are familiar with the basic concepts of JDBC. A good start for beginners is the redbook, *Application Development with VisualAge for Java Enterprise*, SG24-5081, and the *Sun JDBC Guide: Getting Started* that is part of the JDK documentation.

Data Access Beans versus Data Access Builder

In comparison to the Data Access Builder that was introduced in VisualAge for Java Enterprise Version 1, data access beans use a distinct approach for the development process:

- With Data Access Builder you invoke a separate tool to create and manage plenty of database-specific access beans. Data visualization works best with database-specific table beans, which are also created.
- Data access beans are available in the beans palette of the Visual Composition Editor. You have to customize *one* data access bean, the Select bean. Data visualization is completely based on Sun JDK classes; the DBNavigator bean is optional.

Basically, Data Access Builder provides the same functionality as data access beans, in a more complex and sophisticated way. In addition, Data Access Builder generated beans can perform some database actions beyond the scope of data access beans. You can use DB2 stored procedures, you have background thread support to execute long-running methods asynchronously, and there is explicit support to handle primary keys, that is, a column (or a group of columns) that uniquely identifies each row.

2.2 Development Process with Data Access Beans

In this section we describe in detail how to use data access beans.

Loading the Data Access Bean Feature

Before you can use data access beans you have to add the data access bean feature to the Workbench. Use the Quick Start menu (F2), select *Features -> Add feature*, and select *IBM Data Access Beans 1.0* in the dialog that is displayed. This action loads the project into the Workbench and adds the data access beans to the beans palette of the Visual Composition Editor.

You also have to make suitable JDBC drivers available to the Workbench. You can either load the driver classes into a Workbench project or add a zip or jar file containing the driver classes to the Workbench's class path. The class path is specified in the Resources page of the *Window -> Options* dialog. The DB2 JDBC drivers are contained in d:\SQLLIB\JAVA\db2java.zip.

Using the Select Bean

The Select bean, available from the database category in the beans palette of the Visual Composition Editor, provides the base functionality to deal with all kinds of relational data.

Retrieving the Result Set

To use the Select bean you have to specify both a connection alias and an SQL statement,[3] which identify the database as well as the data you want to retrieve (see "Specify the Connection Alias" on page 25 and "Make an SQL Specification" on page 29).

Whenever you start a database transaction, first you have to execute the SQL statement specified for the Select bean, using the execute method (see Figure 1 on page 13, step 1), to retrieve a result set. All methods to display, navigate, insert, update, or delete data expect this, nonempty, result set. All changes are made in memory and then applied to the database.

The result set is automatically closed by the SQL statement that generated it when that SQL statement is closed, reexecuted, or used to retrieve the next result from a sequence of multiple results.

You can define parameters, or host variables, in your SQL statement, and specify the parameters at run time. You must set the parameters before you invoke the execute method. The Select bean has two properties for each parameter defined. One property is the parameter in its specified data type, the other property is a string representation of the parameter. Therefore, you can make a property-to-property connection between the text property of an entry field and the string representation of the parameter to invoke the setParameter method.

[3] The VisualAge for Java documentation uses the phrase *SQL statement*, and we will use the phrase in this book. However, for all readers who know SQL, the SQL statement is a SELECT statement.

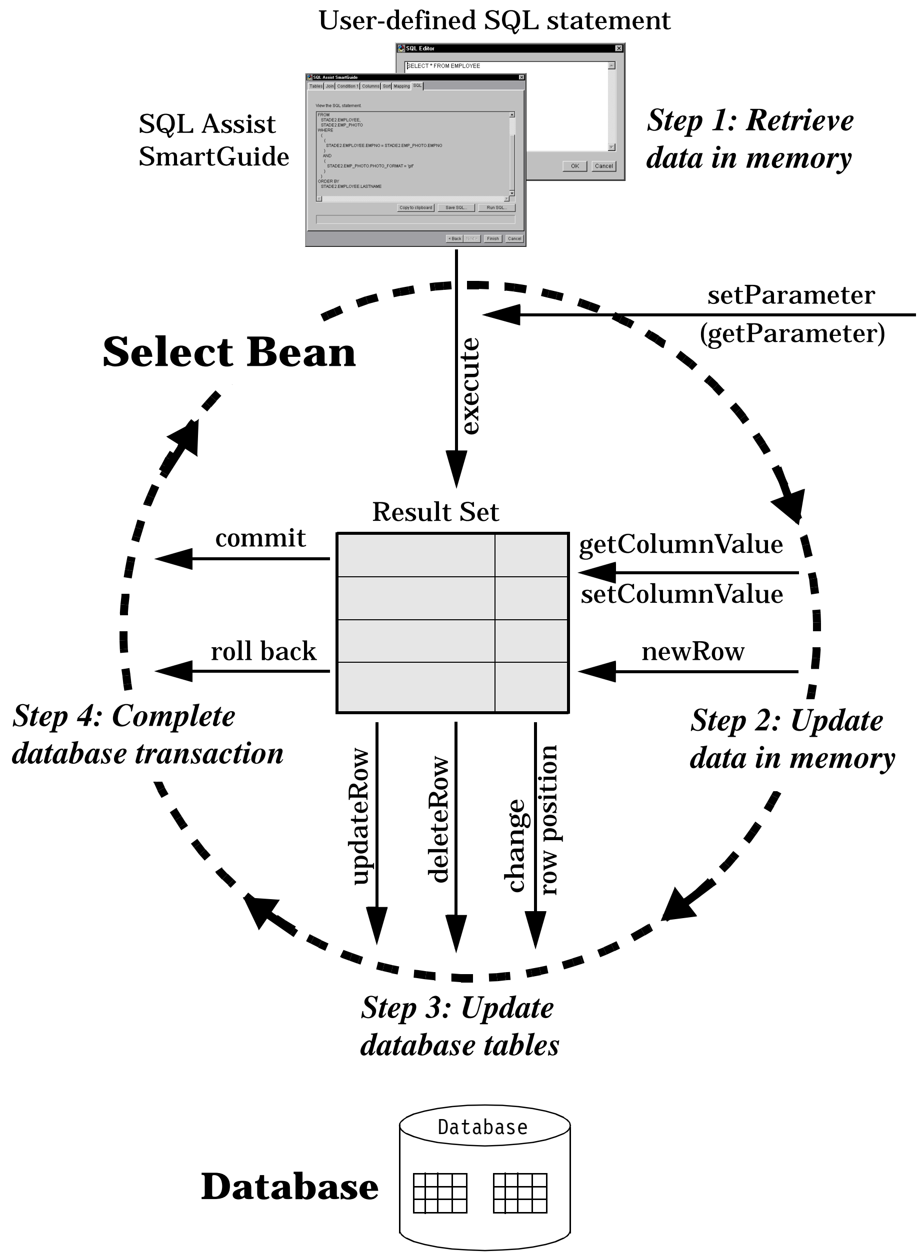


Figure 1. Select Bean Result Set in Memory

Displaying and Updating the Result Set

The Select bean has two bound properties for each data column in the SQL specification. One property is the data column in its specified data type, the other is a String representation of the data column.

Once you have retrieved the result set (Figure 1, step 2), you can display the data by making a property-to-property connection between a user interface component, such as the text property of an entry field, and a property of the Select bean. Such a connection invokes the getColumnValue method to update the user interface when the Select bean property changes.

This property-to-property connection works in both directions. In the same way as you view the contents of the result set, you can update the result set, changing the value in the connected user interface component to invoke the setColumnValue method.

Many of the Select bean methods are designed to operate on the current row of the result set. When an SQL statement is executed using a Select bean, the first row of the result set is the current row. The Select bean includes methods to change the current row, for example, one method makes the next row in the result set the current row. Each column value property for the Select bean is a bound property, if the current row is changed, the data displayed in any interface component connected to the bound property is updated.

Updating the Database

Rows that have been changed in the result set are marked for update of the database as soon as the current row position changes.

The Select bean also provides predefined methods to insert, update, and delete data in the database (Figure 1, step 3), and to commit or roll back changes to the database (Figure 1, step 4).

There are various ways to use these methods. For example, one way to implement updates in an application is to make an event-to-method connection between an appropriate user interface component, such as a push button, and the updateRow, deleteRow, or newRow method of the Select bean.

The updateRow method applies changes to the database, and the deleteRow method deletes data in the database, both on the basis of data in the current row of the result set. To insert a new row, invoke the newRow method. Then use the setColumnValue method to set values for its columns. The new row is not inserted into the database until you move to another row or invoke the updateRow method.

You can specify that all database updates are automatically committed for each SQL query, or you can call the update as well as the rollback methods of the Select bean directly to implement a more sophisticated transaction control.

For more information refer to the VisualAge for Java Online Reference (IBM Tool APIs, Data Access Beans, Package com.ibm.ivj.db.uibeans).

Using the DBNavigator Bean

The DBNavigator bean, available from the database category in the beans palette of the Visual Composition Editor, provides a set of buttons to perform relational database operations for the associated Select bean (see Table 1). The DBNavigator bean is a Swing component and requires the Swing class library.

To use the DBNavigator bean, add it to your user interface components and edit its properties. Among other things you can specify which buttons will be displayed; however, you cannot control the order of buttons in the display.

To associate the DBNavigator bean with the Select bean, create a property-to-property connection between the *this* property of the Select bean and the *model* property of the DBNavigator bean.

You can also use the DBNavigator bean to update data in a relational database, although the DBNavigator bean does not provide an Update button. Change the displayed value, as appropriate, in a user interface component that is connected to the pertinent column value in the result set. If the connection specifies an event to trigger the propagation of the updated value, the value will be set in the result set. Then click on a DBNavigator button, such as Next or Last, that changes the currentRow property value. The values of the current row in the result set are updated in the database before the currentRow property value is changed.

Table 1. DBNavigator Push Buttons

Button	Remark
Retrieve the Result Set	
Execute	Connects to the database, using the connection specified in the connection alias for the associated Select bean, and executes the SQL statement for the associated Select bean.
Refresh	Executes the SQL statement for the associated Select bean. It is designed to reexecute an SQL statement that was previously executed. The button does not reconnect to the database. If the SQL statement is changed after its initial invocation, the initial version of the query is executed.
Navigate the Result Set	
First	Sets the currentRow property of the associated Select bean to the *first* row in the result set.
Previous	Sets the currentRow property of the associated Select bean to the *previous* row in the result set.
Next	Sets the currentRow property of the associated Select bean to the *next* row in the result set.
Last	Sets the currentRow property of the associated Select bean to the *last* row in the result set.
Modify the Result Set	
Insert	Inserts a new, blank row in the result set at the position specified by the currentRow property of the associated Select bean. An associated user interface component will display blanks.
Delete	Deletes the current row in the result set of the associated Select bean. An associated user interface component will display the next row, or the previous row when the deleted row was the last one.
Commit	Commits any uncommitted changes to the database made by the associated Select bean or by any other Select bean that shares the connection alias with the associated Select bean.
Rollback	Rolls back any uncommitted changes to the database made by the associated Select bean or by any other Select bean that shares the connection alias with the associated Select bean.

Development Process Step by Step

Follow these high-level steps to construct an applet or application with data access beans (see Figure 2 on page 18):

1. Create a project and package (or use an existing one) in the Workbench, and open the Visual Composition Editor for a created class (for example, an applet).
2. Place a Select bean from the *Database* category of the Beans Palette on the free-form surface of the Visual Composition Editor (1).
3. Edit the properties of the Select bean according to your requirements:
   - ❑ Specify characteristics of the Select bean, that is, bean name, how many rows are fetched into memory (cache), whether a lock is immediately acquired for the row, and whether database update is allowed (2).
   - ❑ Define characteristics of the database query (query property):
     - Specify a new or existing database access class (3).
     - Specify the connection alias. Identify the characteristics of the database connection, that is, connection name, JDBC driver name, database name, TCP/IP name and port number of the database server, whether database updates are automatically committed, user ID, password, and additional connection parameters. Saving the connection alias generates a connection method in the database access class (4).
     - Make an SQL specification. The query editor uses the connection specification to interact with the database catalog (5). Specify the SQL statement to retrieve the result set. Saving the SQL specification generates an SQL query method in the database access class (6). Methods for adding, updating, and deleting a row, setting parameters, and getting column values are predefined in the Select bean.

   Depending on how you want to separate different database transactions in your applet or application, you can use one or more database access classes. Each class can hold methods for multiple connections and SQL statements.
4. Design the application's GUI, add a DBNavigator bean from the *Database* category of the *Beans Palette*, if needed (7).
5. Connect the visual components (entry fields, push buttons, DBNavigator) of the application's GUI to the Select bean (8). Now, a user can use the DBNavigator bean to execute the SQL statement specified in the Select bean and display the result in the entry fields (9).

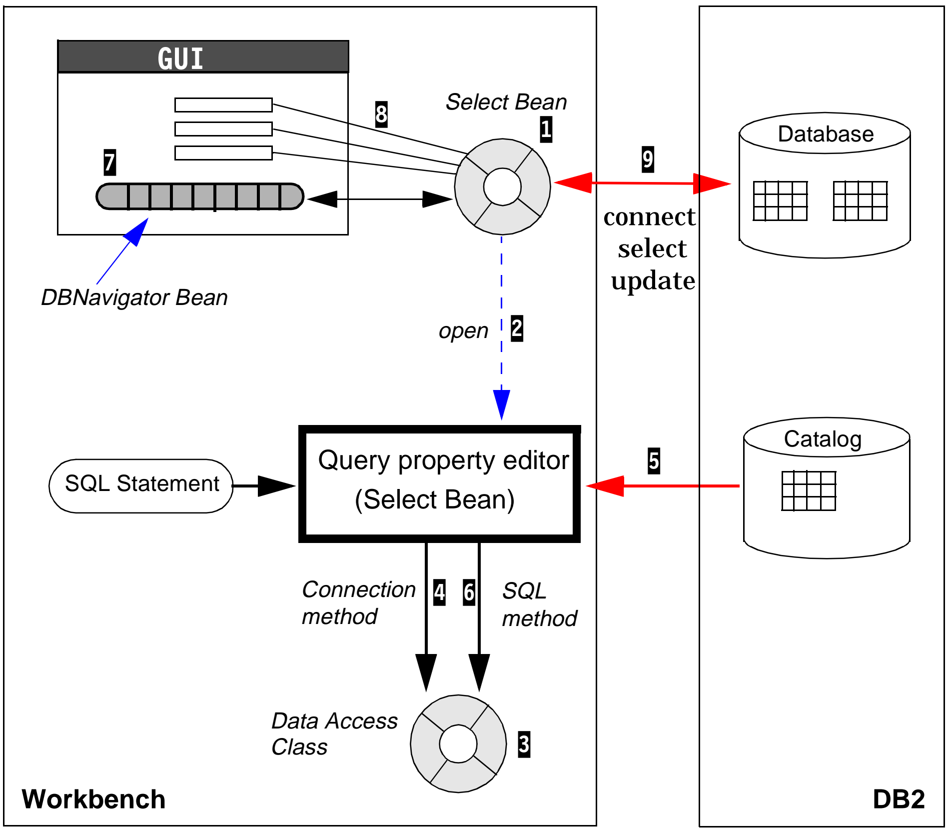


Figure 2. Development Process with Data Access Beans

You can have more than one Select bean in your applet or application. Select beans can share a database connection using the same connection alias. They can also share the SQL specifications of the database access class.

2.3 Building a Sample Application

The fastest way to demonstrate how data access beans work is an example. Now let's start building an application using data access beans.

Application Requirements

The application we are building retrieves all photos saved in gif format from the DB2 SAMPLE database and displays each photo, one after the other, in a window. A table below the photo lists the names of all employees. A DBNavigator bean on top of the application lets you perform all database-specific actions.

You can select an employee and view his or her picture. To initiate the database connection, click on the execute button of the DBNavigator, then navigate between the photos by using the other buttons of the DBNavigator bean (Figure 3).

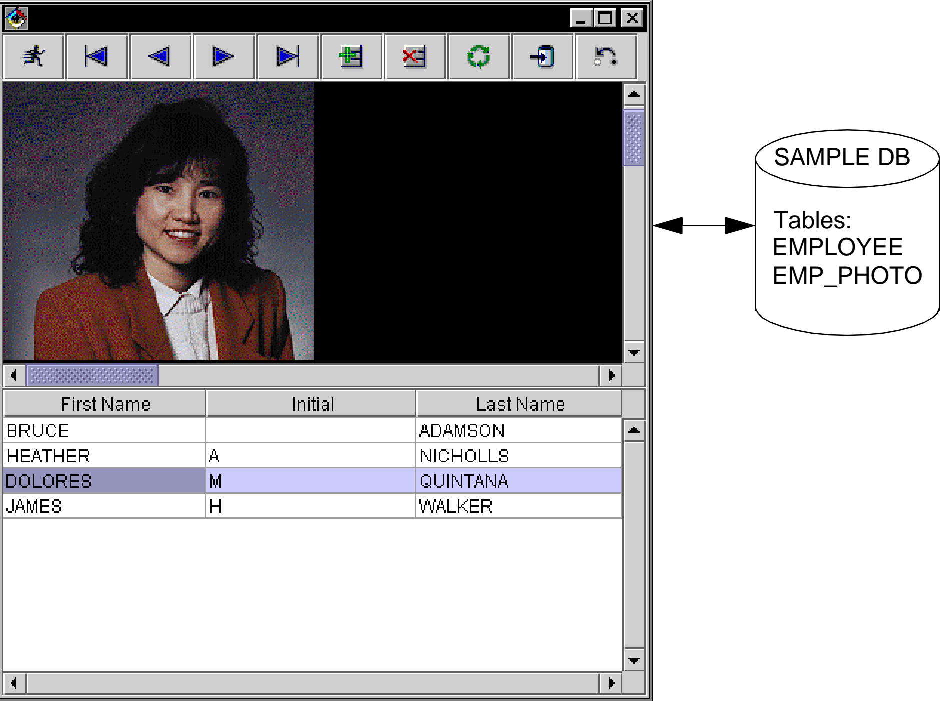


Figure 3. Data Access Beans Sample Application

Although the example is simple, it combines the functionality of data access beans with the new Swing support of VisualAge for Java Version 2.

With Swing, you can develop efficient GUI components that have exactly the "look and feel" that you specify. For example, a program that uses Swing components can be designed in such a way that it will execute without modification on any kind of computer and will always look and feel just like a program written specifically for the particular computer on which it is running.

From an architectural point of view, the Swing component set extends, but does not replace, the Abstract Windowing Toolkit (AWT). The class hierarchy is similar in some ways to the AWT hierarchy but has more than twice as many components as the AWT.

We cover only components of the Swing component set we need for our sample application. For an introduction to the Swing support of VisualAge for Java Version 2 we recommend the IBM redbook entitled *Programming with VisualAge for Java Version 2*, SG24-5264.

We use the JTable bean to create the list of names and show how you can customize the Select bean so that it fits better with the JTable model. A JLabel displays the photos, or an error message, if necessary. A JPanel works as container for all other components. Later the JPanel can be placed in a JApplet or combined with additional components to build an application.

Remember, you can always divide an application into three modules: the user interface, the business logic, and the data store. For simplicity, we wrap the data store into the Select bean, making our application pure object-oriented, and we place the Select bean together with the business logic in the user interface module.

Development Process

Before you start, please verify that you have loaded the IBM Data Access Beans feature and have installed the JDBC support as described in "Setup for Data Access Beans" on page 363.

Keep in mind that the SQL language is not case sensitive. For instance, whenever you specify a database table name, like EMPLOYEE, you can also use employee or Employee. For clarity we use upper-case letters for databases, tables, and columns, and lower-case letters for SQL language elements, such as *select, update, insert, from*, *where, and,* and *order by.*

To build the sample application perform the following steps:

- ❑ Create a project and a package for the sample application.
- ❑ Create a sample panel and a Select bean that retrieves the required data from the SAMPLE database.
- ❑ Create the user interface of the sample application, using beans such as JPanel, JTable, JButton, JScrollPane, ImageIcon, and DBNavigator. Add the logic to convert the picture data returned from the database into a Java image.
- ❑ Subclass the Select bean to improve its standard behavior.
- ❑ Run the application and view the results.

We assume that you are familiar with the basic functionality of VisualAge for Java, especially with the Workbench and the Visual Composition Editor. Otherwise, refer to the VisualAge for Java Online Help.

Creating the Project and the Package

Create a project named **ITSO VAJ Enterprise Book V2** in the Workbench and add a package named **itso.entbk2.sample.databean**.

All our samples will be stored in this project, and we will create individual packages for each example.

Creating the Sample Panel and the Select Bean

Add a class named SamplePanel derived from com.sun.java.swing.JPanel to the itso.entbk2.samples.databean package. This class will keep a Select bean for the database access and, later, it will also include the user interface. Select the package in the Workbench, select *Add -> Class...* in the *Selected* menu and create the new class. Alternatively, use the context (pop-up) menu of the selected package.

Open the Visual Composition Editor for the SamplePanel class. In the *Beans Palette,* switch to the *Database* category and place a Select bean on the free-from surface of the Visual Composition Editor. Rename the bean SampleDB (Figure 4).

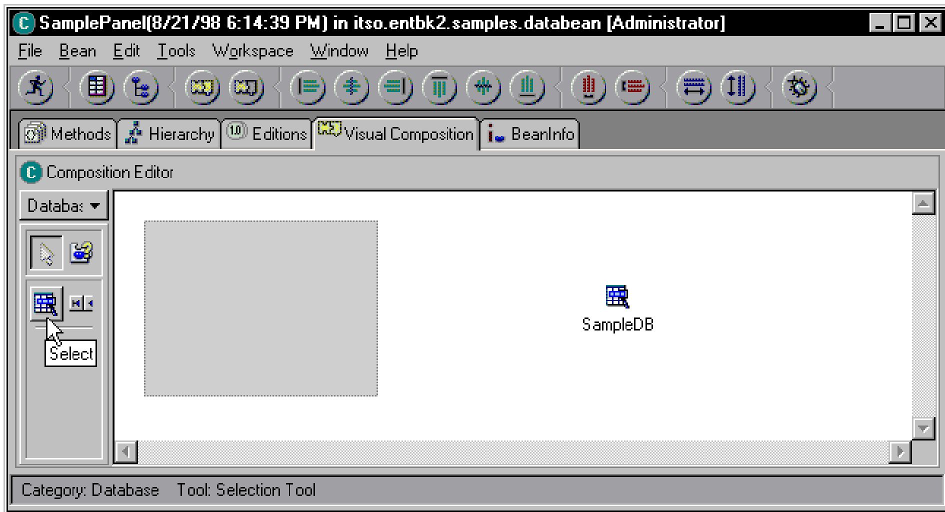


Figure 4. Placing a Select Bean on the Free-Form Surface

Select Bean Properties

With the Select bean, you specify properties pertinent to relational database access, for example, when a lock should be acquired for a row in a table. You also specify a query property that contains the connection alias and an SQL specification. When you later execute the SQL statement, it returns a result set.

Open the Properties window of the Select bean by double-clicking on it and check the *Show Expert Features* checkbox to display all features of the Select bean. You will find the following properties; the description comes directly from the VisualAge for Java help text:

beanName

Specifies the name of the Select bean instance. It must follow standard naming rules for beans. The default name is *Selectn*, where *n* is the number of Select beans with default names; for example, the first default name is *Select1*.

currentRow

Specifies the current row of the result set. A value of -1 indicates that there is no current row, that is, an SQL statement has not yet been executed or the result set is empty.

currentRowInCache

Specifies the current row in cache. A value of -1 indicates that there is no current row, that is, an SQL statement has not yet been executed or the result set is empty.

fillCacheOnExecute

Specifies whether all the rows of the result set are fetched into memory (cache) or only a subset of the result set. A value of *true* means that all the rows of the result set are fetched, up to a maximum number of rows. The maximum number of rows is the *maximumRows* value, or the product of the *packetSize* value multiplied by the *maximumPacketsInCache* value—whichever is smaller. Suppose a result set is 1000 rows, *fillCacheOnExecute* is *true*, *maximumRows* is *100*, *packetSize* is *10*, and *maximumPacketsInCache* is *50*. Executing an SQL statement fetches 100 rows into the cache, that is, the value of *maximumRows*.

A value of *false* means that only the number of rows in the result set needed to satisfy the SQL statement are fetched into the cache. For example, if a result set is 1000 rows, but the application displays only 10 rows, only 10 rows are fetched into the cache.

The default value is *true*.

lockRows

Specifies whether a lock is immediately acquired for the row. A value of *true* means a lock is immediately acquired for the current row. A value of *false* means that a lock is not acquired for the row until an update request is issued. The default value is *false*.

maximumPacketsInCache

Specifies the maximum number of packets allowed in the cache. A packet is a set of rows. A value of *0* means that there is no maximum. The default value is *0*.

maximumRows

Specifies the maximum number of rows that can be fetched into the cache. A value of *0* means that there is no maximum. The default value is *0*.

packetSize

Specifies the number of rows in a packet. A value of *0* means that there is no maximum. The default value is *0*.

query

Specifies the connection alias and SQL specification for the Select bean.

readOnly

Specifies whether updates to the data are allowed. A value of *true* means that updates are disallowed even if the database manager would permit them. A value of *false* means that updates are allowed, provided that the database manager permits them. The default value is *false*.

You can change some property values now, or create connections to change the values at run time. For example, depending on how big the result set is and how fast the database connection, you should decide how many rows you want to cache. Be aware that caching a big result set increases the time it takes to update a database view. However, when you want to change a lot of rows, or you have a slow database connection, caching may accelerate your work.

At this time, use default values. Therefore, you only have to specify the connection alias and the SQL statement. Select the query property, then click on the rectangle on the right side of the query entry.

The Query property editor opens (Figure 5). You will see two pages, a Connection page to specify the connection alias, and an SQL page to specify the SQL statement.

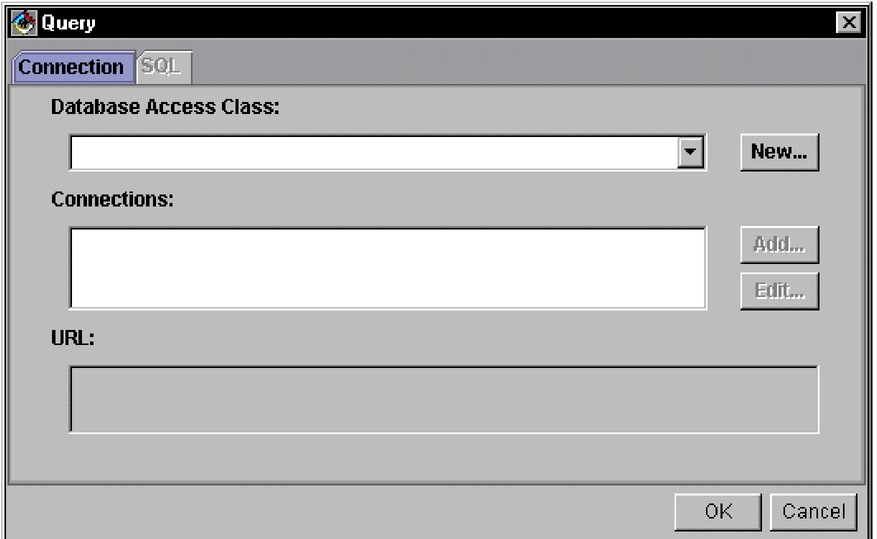


Figure 5. Query Property Editor of the Select Bean

Specify the Connection Alias

A connection alias defines database connection characteristics for the Select bean. The Connection page also identifies the database access class to hold the definition. One database access class can handle more than one connection alias. Use this page to create a new connection alias or select an existing one.

Multiple Select beans can use the same connection alias. In this case, they share the database connection associated with that connection alias. If one Select bean commits updates to a database, it commits all uncommitted updates made by any Select bean sharing the database connection.

The Database Access Class combo box lists all database access classes that exist in the workspace.

For this example, create a new database access class by clicking on the *New...* button. The New Database Access Class window opens and you enter itso.entbk2.samples.databean as the name of the package and SampleDB as the name of the database access class (Figure 6).

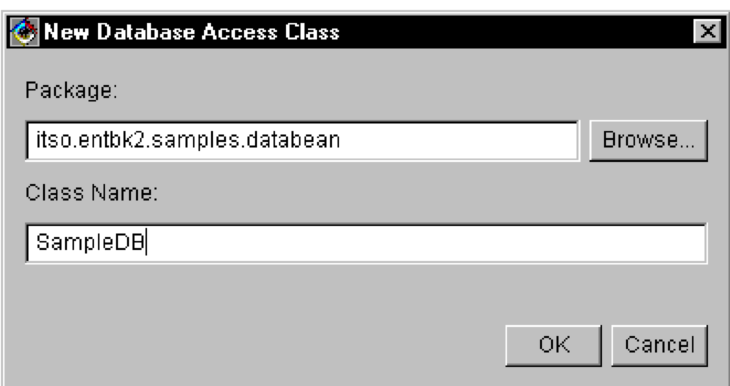


Figure 6. Specification of the Database Access Class

The Connections field (Figure 5) lists all connection aliases currently defined in the selected database access class. Because the class is new, the list is empty at the moment.

Click on the *Add...* button to open the Connection Alias Definition window (Figure 7) and enter the following information:

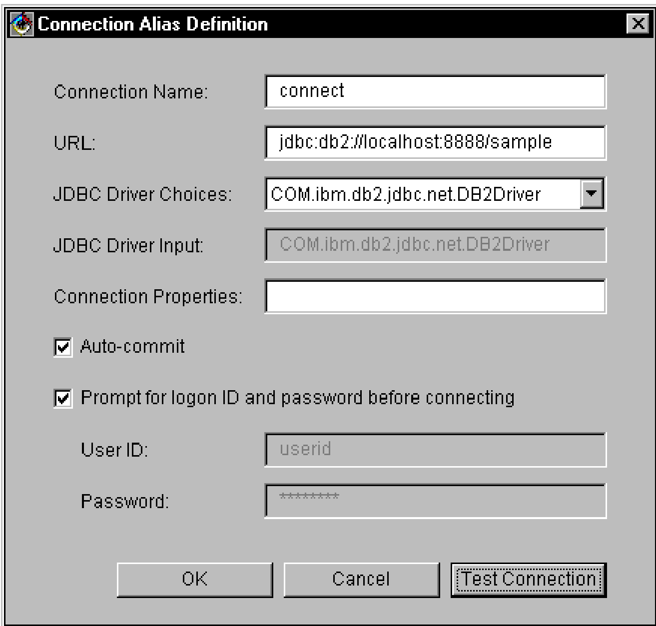


Figure 7. Specification of the Connection Alias Definition

Connection Name

Name of the connection alias. It must be a valid Java method name, usually starting with a lower-case letter.

URL

The URL for the database connection identifies the data source for the connection and provides information to locate the database.[4]

For example, the URL specification using:

- The DB2 application JDBC driver to access a local database named *sample* is:

  jdbc:db2:sample

- The DB2 network JDBC driver to access a database named *sample*, on a remote server named *dbserver*, through port number *8888* is:

  jdbc:db2://dbserver:8888/sample

In the first case, you must run the DB2 Client Application Enabler on the same machine where your application runs.

[4] To familiarize yourself with the JDBC driver concept, read the *Sun JDBC Guide: Getting Started* which comes with the Sun JDK documentation.

In the second case, you must start both the DB2 database manager and the DB2 JDBC daemon on the remote server. From the command line, call:

- *db2start*, to start the DB2 database manager
- *db2jstrt port*, to start the DB2 JDBC Applet Server for a specified TCP/IP port, for example, db2jstrt 8888.

JDBC Driver Choices

The predefined JDBC drivers are:

- COM.ibm.db2.jdbc.app.DB2Driver, the DB2 JDBC application driver
- COM.ibm.db2.jdbc.net.DB2Driver, the DB2 JDBC network driver
- sun.jdbc.odbc.JdbcOdbcDriver, the JDBC-ODBC driver bridge
- oracle.jdbc.driver.OracleDriver, the Oracle JDBC driver
- com.sybase.jdbc.SybDriver, the Sybase JDBC driver

JDBC Driver Input

Name of a JDBC driver class, if not listed in the *JDBC Driver Choices* field.

Connection Properties

Any properties to be passed in the database connection request, other than the user ID and password. Specify the properties in the following format: prop=value;prop=value;... where, *prop* is the name of the property, and *value* is the value of the property.

In the following example, three properties are passed:

```
proxy=localhost:8888;a=1;b=2
```

Auto-commit

Database updates are automatically committed for each SQL query if checked (default).

Prompt for logon ID and password before connecting

You are prompted for the user ID and password when a connection to the database is required at development and run time.

User ID

The user ID for the database connection request. (This user ID is also displayed in the prompt dialog if the *Prompt for logon ID and password before connecting* checkbox is checked.)

Password

The password for the database connection request (This password is also displayed in the prompt dialog if the *Prompt for logon ID and password before connecting* checkbox is checked.)

With data access beans you can build a database connection to any database management system (DBMS) that supports JDBC. In the same way as you access your DBMS to read the database catalog at development time, you access it to retrieve the data at run time.

At the same time you create a connection alias you can test the connection. Just click on the *Test Connection* button to test the database connection using the specifications made. You can find errors, for example, an unavailable JDBC driver, as soon as possible.

Click on the *OK* button to finish your work. The query builder defines the new connection alias, generates a new static method called *connect* in the SampleDB class, and adds the connection alias to the Connections list (Figure 8).

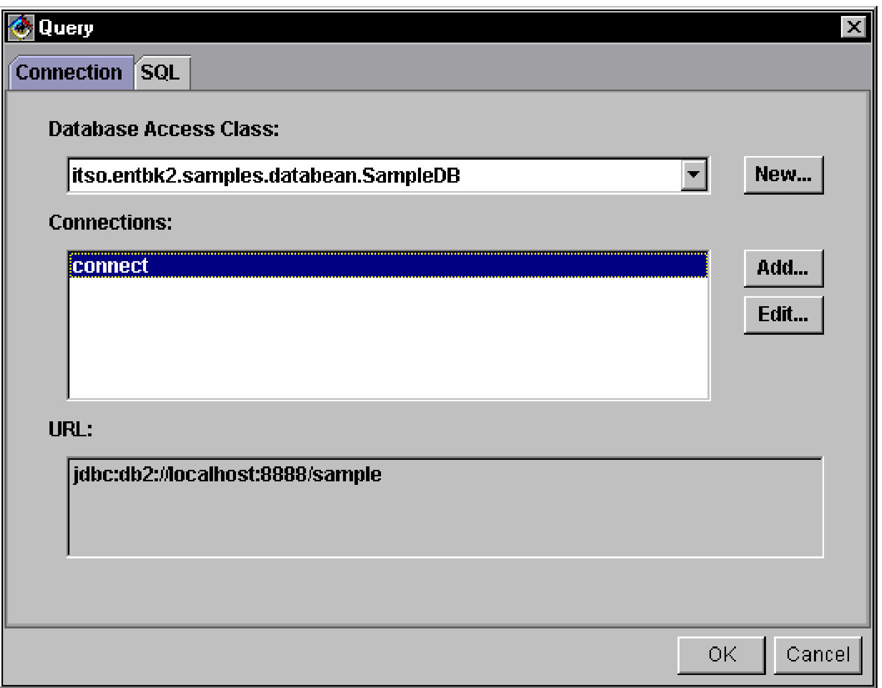


Figure 8. Query Property Editor Connection Page

Make an SQL Specification

Specifying the connection alias enables the SQL tab to switch to the SQL specification page. This page lets you compose the SQL statement to retrieve the result set. In fact, creating the SELECT statement is the only time that you will ever come in touch with SQL code.

When you define an SQL specification, you also identify a database access class to hold the definition. You normally use the same database access class for both the connection alias and SQL specification. In addition, different Select beans can use the same SQL specification to share the result set.

All database access classes defined in the workspace are listed in the Database Access Class field. As you can see, one of these classes is the itso.entbk2.samples.databean.SampleDB class that you just created. Because our sample application is small, we decide to keep the specification for the connection alias and the SQL statement together (Figure 9).

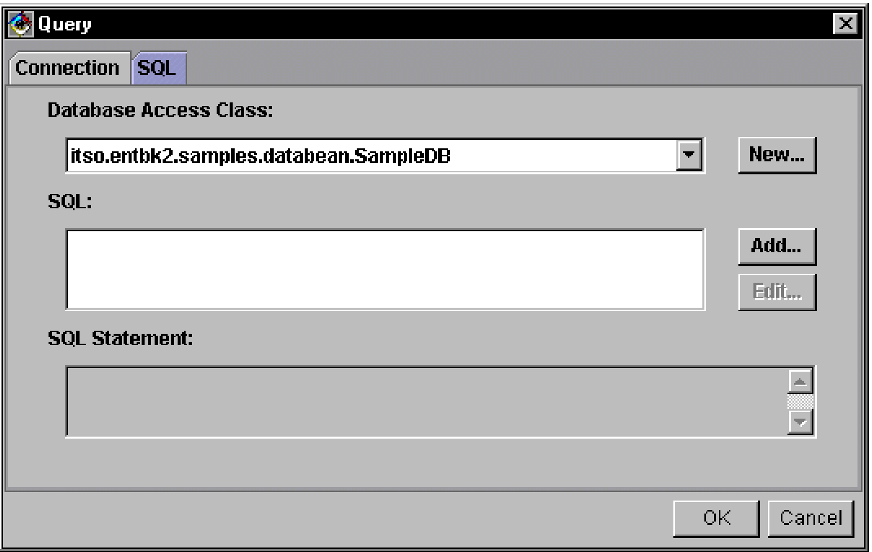


Figure 9. Query Property Editor SQL Page

The SQL field lists all SQL specifications that can be found in the selected database access class. At the moment this list is empty, because no specifications have been made.

Click on the *Add...* button to add a new SQL specification. The New SQL Specification window opens. Enter *getSampleData* as the SQL Name (Figure 10).

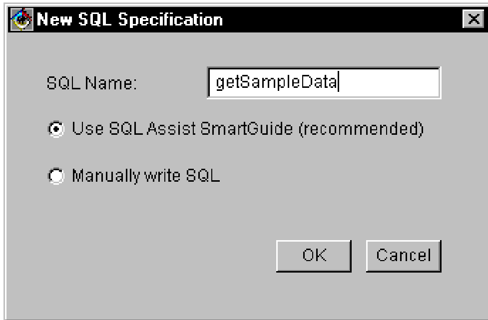


Figure 10. Define a new SQL Specification

The SQL name must be a valid Java method name; it should start with a lower-case letter.

Then choose *Use SQL Assist SmartGuide*. The SmartGuide lets you graphically compose the SQL statement without SQL skills.

Although in this book every SQL statement is composed using the SmartGuide, we also list the appropriate SQL code, so that readers with previous experience in SQL will have a good understanding of what really happens and everyone can avoid mistakes. Alternatively, you can select *Manually write SQL* to enter the SQL statement manually, but then you might as well stop reading this chapter.

Click on the *OK* button to open the SQL Assist SmartGuide. Although you can use this tool without knowing SQL, you should at least know how your database is organized, the information you require, and how to get that information from the database. Then you will find that the SQL Assist SmartGuide is a good tool that provides visual control over the code you create and a powerful test environment for looking up values in the database, running the generated SQL code whenever you want, and finding errors early.

Figure 11 shows the SQL Assist SmartGuide. Below we explain the different pages of the SmartGuide.

Tables Page

For the sample application, we have to retrieve the names of the employees from the EMPLOYEE table and the photos from the EMP_PHOTO table, both assigned with the creator ID of the sample database as schema name.

The Table name field lists the tables that are accessible in the sample database identified by the connection alias *connect*. Make sure that both the

EMPLOYEE and the EMP_PHOTO tables are checked (Figure 11). These tables will appear in the FROM clause of the SQL statement.

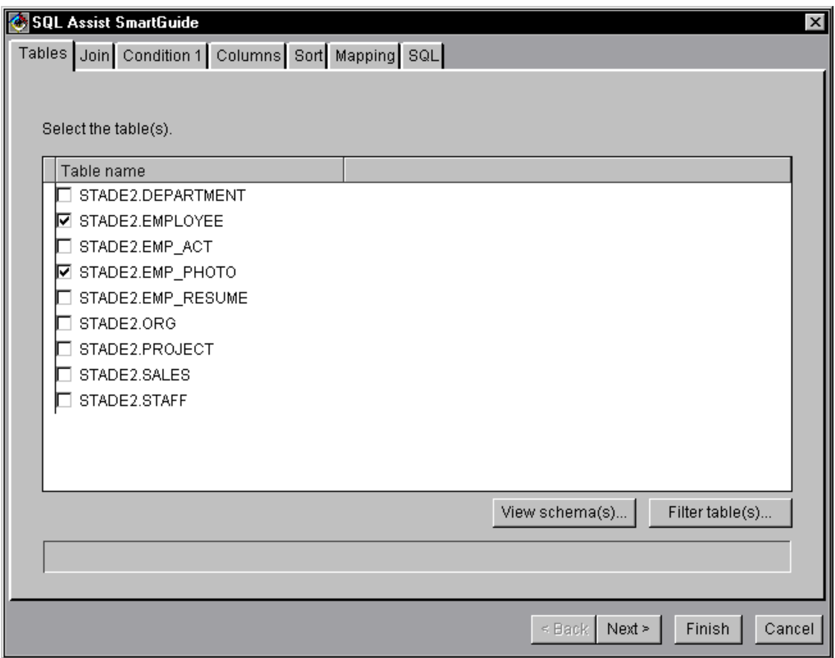


Figure 11. SQL Assist SmartGuide Tables Page (required)

You can control the table names displayed in the list by clicking on either the *View Schema(s)...* or *Filter table(s)...* button.

The *View Schema(s)...* button opens the Schema(s) to View window (Figure 12). In this window you specify which schemas are to be shown. We recommend removing the SYSCAT, SYSIBM, and SYSSTAT schemas, which are DB2 internal tables and are not necessary for the example.

Clicking on the *Filter table(s)...* button opens the Table Name Filter window (Figure 12). You can enter the following information for the filter:

❑ Filtering characters for the table name. The filtering characters are case-sensitive. These characters limit the display to table names beginning with those characters. For example, if you enter EMP, only table names that begin with EMP are listed, such as the EMPLOYEE table. The % character is a wildcard character. Use it to position the filter. For example, specifying %ID requests the display of all table names that end with the characters ID. The specification N%ID requests the display of all tables names that begin with N and end with ID.

- ❑ Table type. This information determines the type of tables that will be displayed in the Tables page. You can specify alias tables, system tables, user tables, or views by checking its checkbox in the Table type field. You can check multiple table types.

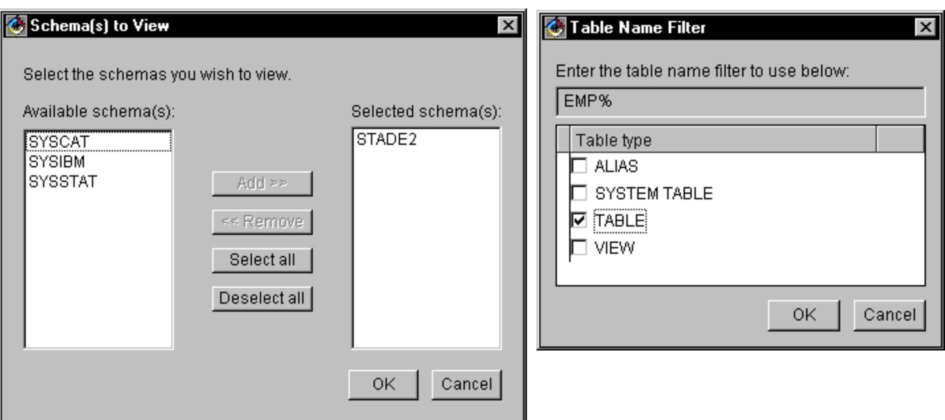


Figure 12. Schemas to View and Table Name Filters

The sample database is very small, so you can ignore both windows. However, if you were to manage dozens of databases and hundreds of tables, you could see how such inconspicuous features would accelerate your work.

Join Page

We only want the names of the employees for whom a photo exists in the database, and vice versa. From a database point of view, you have to specify an inner join between the EMPNO column in both the EMPLOYEE and the EMP_PHOTO tables.

The Join page (Figure 13) displays the columns of both tables; remember, you marked the tables in the Tables page. Select EMPNO in the EMPLOYEE table and EMPNO in the EMP_PHOTO table. A line appears connecting both columns to indicate the requested join. You can see whether the requested join is invalid, for instance, because of a mismatch in the data type of the columns, or you request the same join twice. The control buttons are also enabled. Click on the *Join* button. A red join line indicates that the join is successful.

By default, the *Join* button creates an inner join; do not change it. Your join will appear in the WHERE clause of the SQL statement.

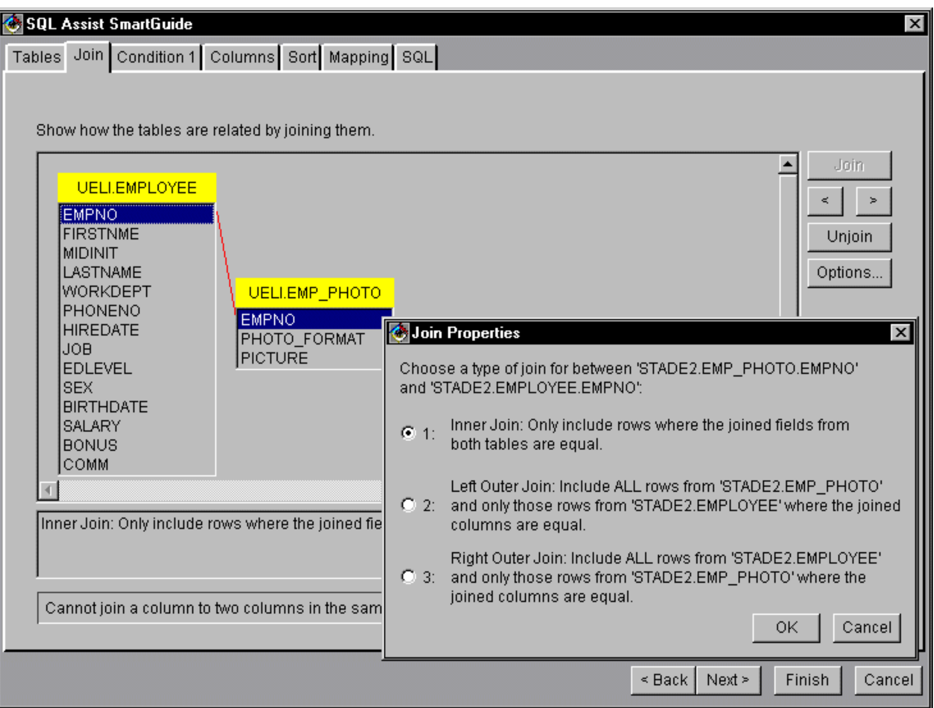


Figure 13. SQL Assist SmartGuide Join Page (optional)

If you want to see what other types of joins are available, click on the *Options* button to get the Join Properties window:

1. *Inner Join*—This is a request for rows where the values in the joined columns match.
2. *Left Outer Join*—This is a request for an inner join and any additional rows in the left table (as viewed on the Join page) that are not already included in the inner join.
3. *Right Outer Join*—This is a request for an inner join and any additional rows in the right table (as viewed on the Join page) that are not already included in the inner join.

If you have selected more than two tables, you can request additional joins in the same way as the initial join. You can join other displayed columns in the same tables or in other tables. Navigate between multiple joins by clicking on > or <. The selected join is indicated by a red join line.

Condition Page

We want to restrict retrieval of photos to only those that are in gif format. This format is directly supported by the Sun JDK, so no additional code is necessary to display the photos.

You need a search condition. On the Condition 1 page, select the EMP_PHOTO table in the Selected table(s) field and PHOTO_FORMAT in the Columns field. Then select *is exactly equal to* in the Operator field and enter *gif* in the Values field (Figure 14).

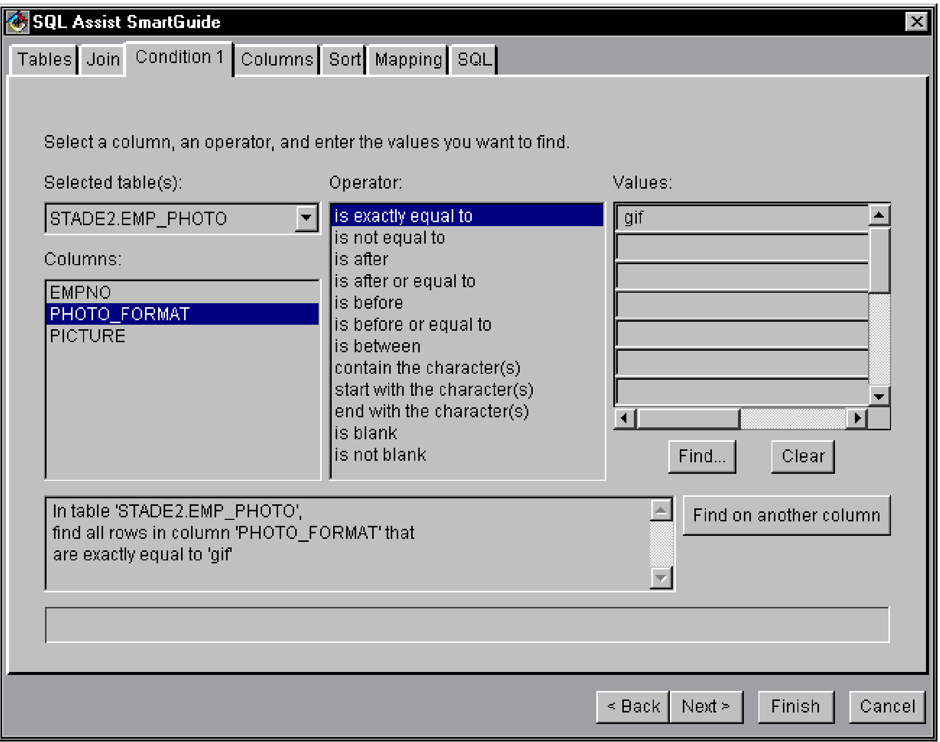


Figure 14. SQL Assist SmartGuide Condition Page (optional)

Alternatively, you can use the *Find...* button to find appropriate values for a search condition (see Figure 15).

You can look for values that include a specific character string or you can display all values in the column. The % character is used as a wildcard character. For example, specifying A% searches for values that begin with the character A. Specifying %1 searches for values that end with the character 1. The specification A%1 searches for values that begin with the character A and end with the character 1.

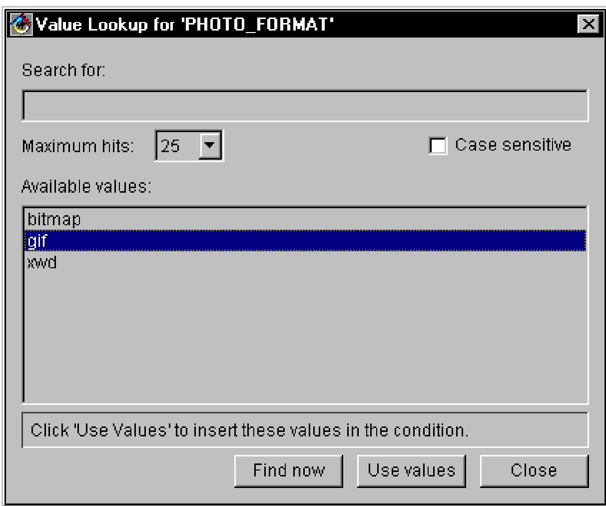


Figure 15. Search for Column Values in the Value Lookup Window

Check the *Case sensitive* checkbox if you want to search for the characters in upper or lower case, exactly as entered in the Search for field. Select a value from the Maximum hits list to control the maximum number of values returned for the search.

Results are displayed in the Available values list. Select an appropriate value or values from the list, and click on *Use values*. The selected values are added to the Values list on the Condition page.

Take time to explore the tool, for instance, see which other picture formats are stored in the database. Perhaps you can enter the statement in the SQL editor faster, especially if you are well-versed in SQL. But, are you really sure that you never will make a spelling mistake? Now you can be sure.

You can also specify parameters, called host variables in SQL, in the Values list to pass the value to the SQL statement at run time. A parameter is specified in the format *:parm*, where parm is the parameter name. The dialog converts your name to upper case while you type. For example, :EMPID is a valid specification for a parameter named EMPID.

If you want to add a second condition, click on the *Find on another column* button. A second Condition window is displayed (the tab for the page is labeled Condition 2). Repeat the process until you specify all the search conditions for the query.

The search conditions supplement any joins specified on the Join page, that is, the joins and the search conditions appear in the WHERE clause of the SQL statement.

Columns Page

We need the information from the FIRSTNME, MIDINIT, and LASTNAME column of the EMPLOYEE table, and from the PICTURE column of the EMP_PHOTO table.

Select the tables in the Selected table(s) field and the column you want in the Columns field, then click on the *Add>>* button to add the columns. Repeat until you have added all columns.

As you will see later, it is important to add the columns in the order in which they are listed. Use the *Move up* and *Move down* buttons, if necessary (Figure 16).

The columns you specify in the Columns page will appear in the SELECT clause of the SQL statement. If you do not specify a column, an * will appear in the SELECT clause, that is, all columns are selected.

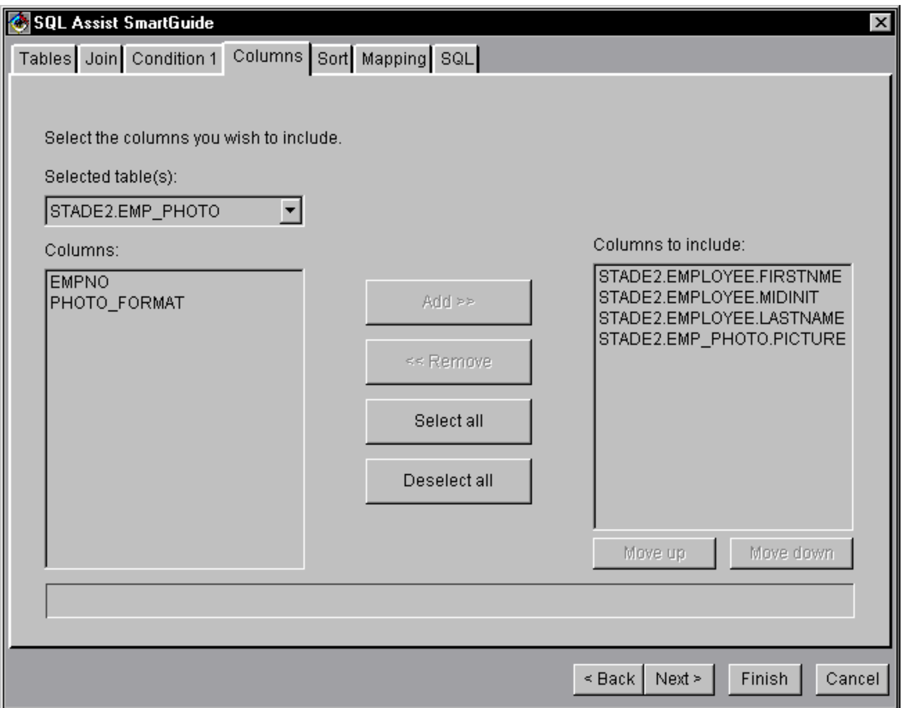


Figure 16. SQL Assist SmartGuide Columns Page (optional)

Sort Page

We want the order of a person in the name list to be based purely on an alphabetical ordering of his or her last name. Specify the order by identifying the EMPLOYEE.LASTNAME column to be used as a sort key (ascending order). Select the EMPLOYEE table, the LASTNAME column, and the sort order, *Ascending*. Then click on the *Add>>* button (Figure 17).

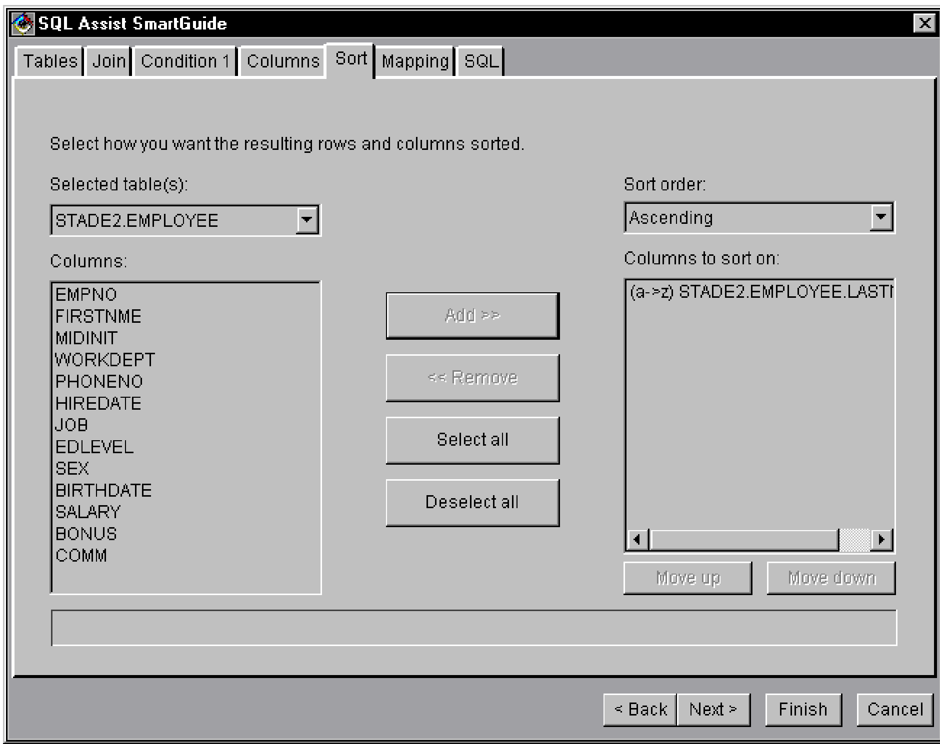


Figure 17. SQL Assist SmartGuide Sort Page (optional)

You can also specify a descending order for a column, or you can specify multiple columns. Each column is used as a separate sort key. The rows of the result set are ordered by the value in the selected column, that is, by the value of the sort key. If you specify more than one sort column, the rows of the result set are ordered by the value of the first sort column, then by the value of the second sort column, and so forth.

The sorting specification will appear in the ORDER BY clause of the SQL statement.

Mapping Page

On the Mapping page you can remap the data retrieved from a table column to a different SQL data type, and thus, to a different Java class, as shown in Table 2.

Table 2. Mapping between SQL Data Types and Java Classes

SQL Type	Java Class
CHAR	java.lang.String
VARCHAR	java.lang.String
LONG VARCHAR	java.lang.String
INTEGER	java.lang.Integer
TINYINT	java.lang.Integer
SMALLINT	java.lang.Short
DECIMAL	java.math.BigDecimal
NUMERIC	java.math.BigDecimal
BIT	java.math.Boolean
BIGINT	java.lang.Long
REAL	java.lang.Float
FLOAT	java.lang.Double
DOUBLE	java.lang.Double
BINARY	byte[]
VARBINARY	byte[]
LONGVARBINARY	byte[]
DATE	java.sql.Date
TIME	java.sql.Time
TIMESTAMP	java.sql.Timestamp

The JDBC driver attempts to convert the underlying data to the specified Java type and returns a suitable Java value. Check this page when you are experiencing problems with data conversion. Leave this page untouched for our example (Figure 18).

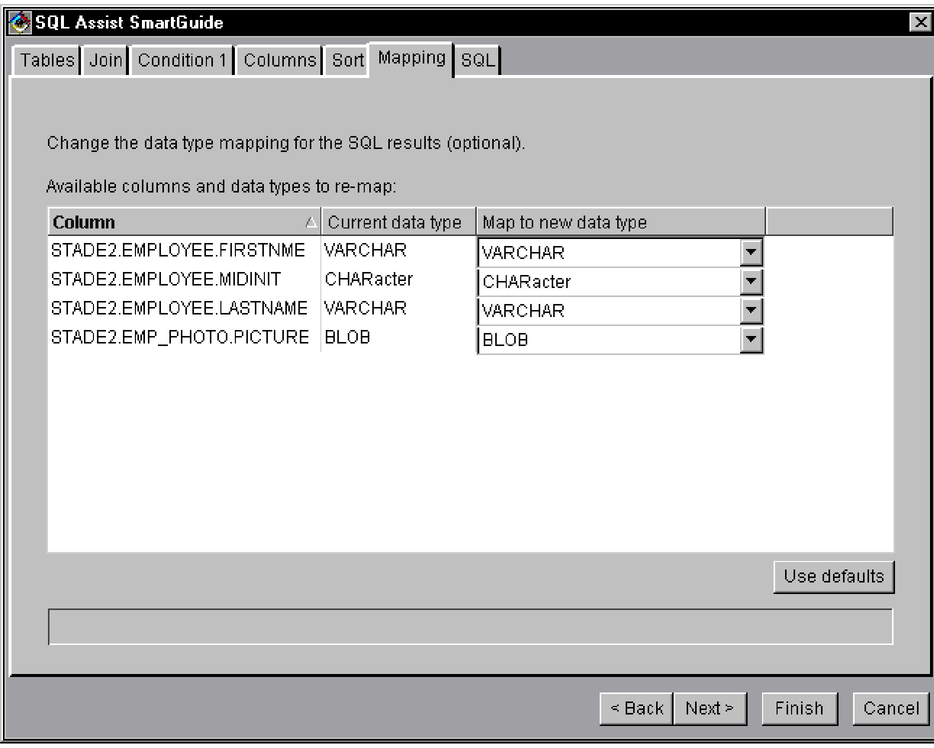


Figure 18. SQL Assist SmartGuide Mapping Page (optional)

SQL Page

The SQL page displays the final SQL statement (Figure 19). When your statement looks exactly the same as the displayed statement, you know that half the work is done:

```
SELECT
   <schema>.EMPLOYEE.FIRSTNME,
   <schema>.EMPLOYEE.MIDINIT,
   <schema>.EMPLOYEE.LASTNAME,
   <schema>.EMP_PHOTO.PICTURE
FROM
   <schema>.EMPLOYEE,
   <schema>.EMP_PHOTO
WHERE
   ((<schema>.EMPLOYEE.EMPNO = <schema>.EMP_PHOTO.EMPNO)
AND
   (<schema>.EMP_PHOTO.PHOTO_FORMAT = 'gif'))
ORDER BY
   <schema>.EMPLOYEE.LASTNAME
```

Instead of *<schema>* you will find the name representing the creator ID of the sample database.

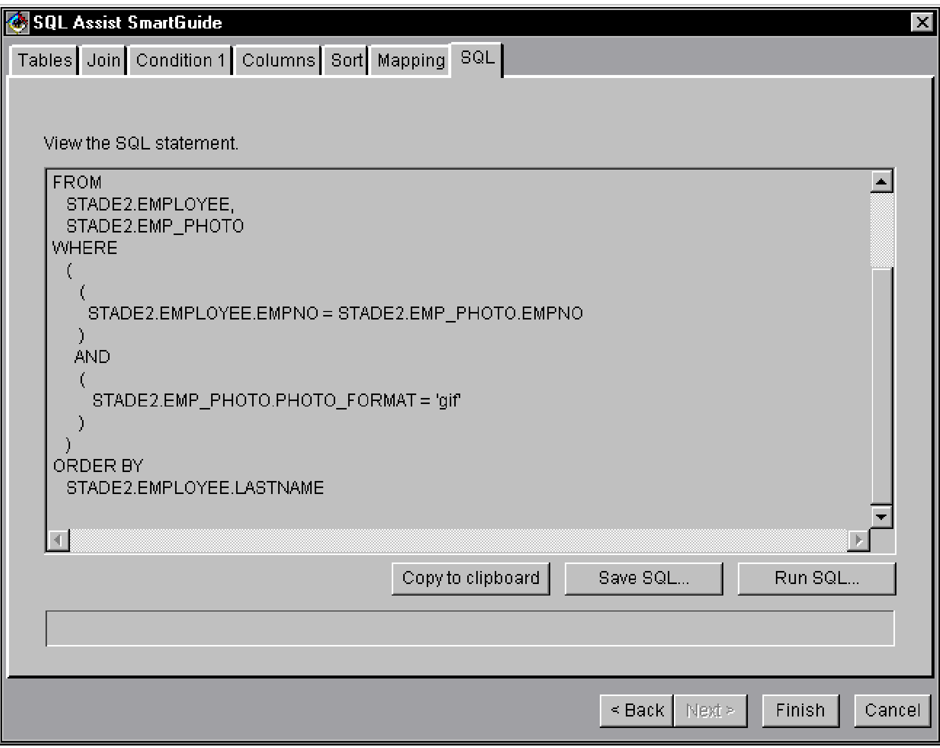


Figure 19. SQL Assist SmartGuide SQL Page (optional)

You can generate your SQL method, copy it to the clipboard, or save it to a file. We recommend that you test the statement before finishing. SQL code can be checked only by your underlying database management system. You may generate the wrong SQL code without detecting the mistake. Years later, somebody will call your code and get unexpected behavior. Now it takes only minutes to fix the problem, later it could take ages. Do you really want to risk that?

Click on the *Finish* button to generate the code. Generation defines an SQL specification, creates a new static method, *getSampleData,* in the *itso.entbk2.samples.databean* package, and adds the SQL specification to the SQL list (Figure 20). Click on *OK* to return to the Visual Composition Editor.

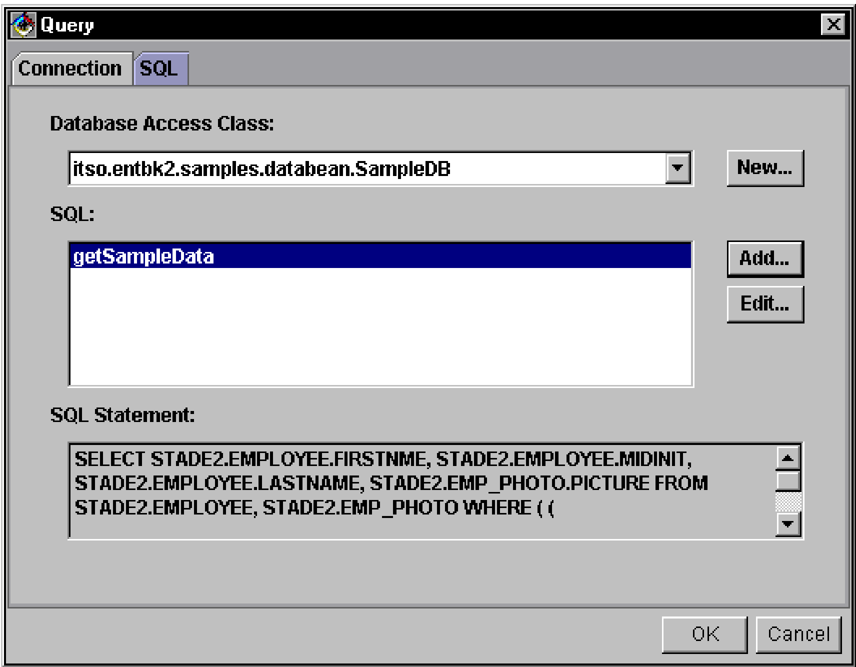


Figure 20. Query Property Editor with New SQL Specification

Building the User Interface

You have finished and tested the database part. Now it is time to design the user interface. This interface is based on Swing and the DBNavigator bean from the Database category of the Beans palette. Keep in mind that the DBNavigator bean uses a Swing component and requires the JFC library.

When building a Swing user interface, it is not always easy to select the user interface components in the Visual Composition Editor to change their arrangement or properties. Open the Beans List window (*Tools -> Beans List*) and select the components from there.

Perform the following steps to design the user interface:

- ❑ Select the SamplePanel, open the Properties window, and set the *layout* property to BorderLayout. Then select the *constraints* property and set both height and width to 400.
- ❑ Add a JPanel bean from the Swing category to the center region of the SamplePanel, rename it to DataPanel, and set the *layout* property to GridLayout with 2 rows and 1 column.

❑ Place a JScrollPane in the DataPanel, rename it to PhotoScrollPane, and set the following properties:

  - verticalScrollBarPolicy: VERTICAL_SCROLLBAR_ALWAYS
  - horizontalScrollBarPolicy: HORIZONTAL_SCROLLBAR_ALWAYS

❑ Add a JLabel to the PhotoScrollPane, change the bean name property to PhotoLabel, and remove the text in the text property. Later the PhotoLabel will show the photos and, if necessary, any error messages.

❑ Select the JTable bean in the Swing category of the Beans palette and add it to the DataPanel. This adds a JTable inside a JScrollPane. Rename the bean names to NamesTable and NamesScrollPane. We use the NamesTable to display the list of employees.

❑ To complete your user interface, drop a DBNavigator bean from the Database category to the north region of the SamplePanel and change the bean name to DBNavigator.

After placing all the visual components, you have to add connections:

❑ Connect SampleDB *this* to DBNavigator *model*. This plugs the Select bean named SampleDB into the DBNavigator bean model and allows you to connect to the database and navigate in the result set by clicking one of the *Execute*, *First*, *Previous*, *Next,* or *Last* buttons (1).

  All other buttons have no functionality in our sample application, in fact, most of them throw an exception and are shown only for completeness. Feel free to disable these buttons in the Properties window of the DBNavigator bean.

❑ Display all error messages in the PhotoLabel. Create an event-to-method connection from DBNavigator *exceptionOccurred* to PhotoLabel *text*. Make sure the *Pass event data* checkbox is checked in the Properties window of the connection (2).

Figure 21 shows the resulting GUI. To test the database connection, click on the *Execute* button. You should be able to connect to the database without any error message. If unsuccessful, check that both DB2 database manager and DB2 JDBC daemon are running and that you can connect to the database with the specified user ID and password. Check the Console window for additional information.

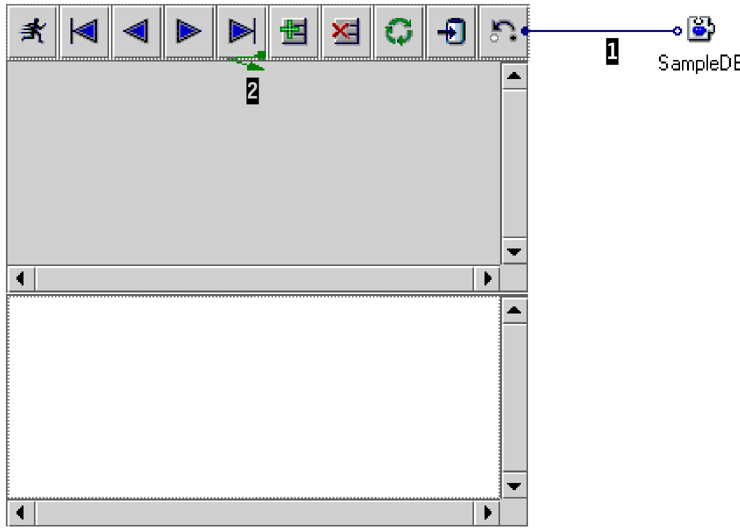


Figure 21. Sample Application User Interface and Database Connection

If your database access was successful, you can start developing the logic to convert the image data retrieved from the database and display the photos in PhotoLabel.

The SampleDB bean has a bound property containing the data from the PICTURE column, and the PhotoLabel has another property called *icon*. It should be very easy to display the photos by simply defining a property-to-property connection, provided that both properties can be directly converted.

The run-time type of the source property is byte[] because the original data is stored as a BLOB (see Table 2 on page 38). The PhotoLabel bean needs an icon of type com.sun.java.swing.ImageIcon. In our application, we have one of the rare situations where handwritten code is needed to convert the data.[5]

In the Visual Composition Editor (see Figure 22):

❑ Tear off the *EMP_PHOTO.PICTURE (Object)* property of the SampleDB bean. You will get a variable of type java.lang.Object. Rename this variable to Photo (3).

❑ Add a factory from the Other palette category to the free-form surface and change the type to com.sun.java.swing.ImageIcon, then rename the bean name to ImageIcon (4).

[5] A complete visual design would be possible if VisualAge for Java would allow you to create variables that are one of the simple types, for example, int, float, or byte[].

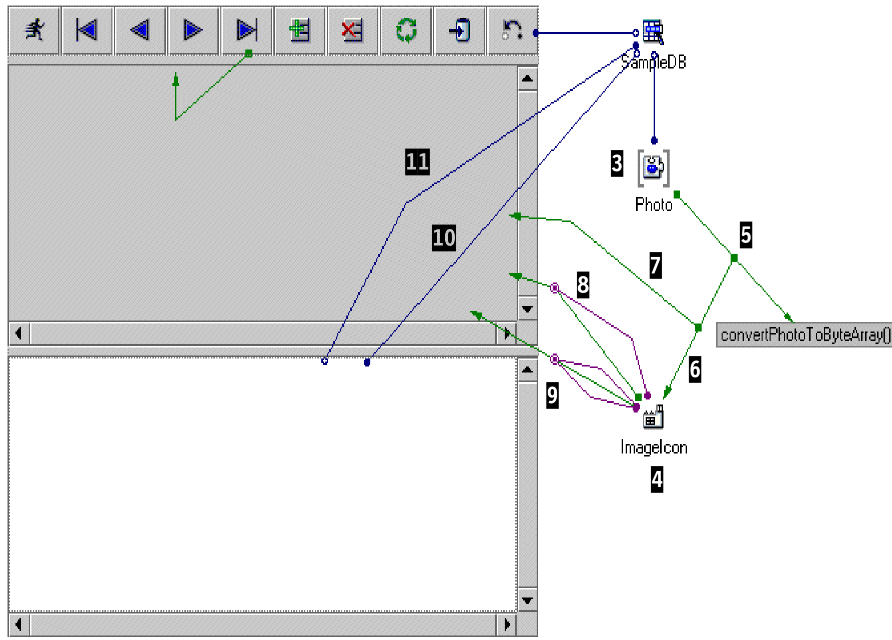


Figure 22. Sample Application Logic to Display the Employee Photos

Next, connect the beans:

- Create an event-to-code connection from Photo *this* to the free-form surface of the Visual Composition Editor (5), mark the *Pass event data* checkbox and enter this code:

```
public byte[] convertPhotoToByteArray(java.lang.Object photo) {
    return (byte[])photo;
}
```

- Connect the *normalResult* of the previous connection to the constructor *ImageIcon(byte[])* of the ImageIcon factory to create a new ImageIcon (6). To display an error message, connect *exceptionOccurred* of this connection to PhotoLabel *text* and mark the *Pass event data* checkbox in the Properties window of the connection (7).
- To pass the ImageIcon to the PhotoLabel bean, connect the ImageIcon *this* event to the *setIcon* method of PhotoLabel (8). Pass *this* of the ImageIcon as a parameter.
- Connect the ImageIcon *this* event to the *setSize* method of PhotoLabel to initiate resizing the PhotoLabel and PhotoScrollPane (9). This guarantees that the photos are correctly shown. Pass the properties *iconHeight* and *iconWidth* of PhotoIcon as parameters *height* and *width*.

Test the application again. You should see photos of four employees. Navigate among the photos, using the DBNavigator bean.

If everything works, you can create two additional connections to display the names of the employees:

- ❑ Connect the SampleDB *this* property to the NamesTable *model* property (**10**).
- ❑ Connect the NamesTable *selectedRow* property to the SampleDB *currentRow* property. Note that *currentRow* is an expert feature (click on the *Show expert features* checkbox). Change the source event of the connection to *mousePressed* (**11**).

Test the final application. Whenever you select one of the employees in the JTable, you should see his or her photo.

Improving the Select Bean

At this time, the application has two disadvantages. You cannot read the headings of the columns in the JTable, and you cannot remove the PICTURE column that displays the object address. To get a really nice-looking application you have to make some improvements.

Create a new class, SampleSelect, subclass of com.ibm.ivj.db.uibeans.Select and add it to the itso.entbk2.samples.databean package. Then add two methods to the class:

```
public int getColumnCount() {
   return super.getColumnCount()-1;
}
```

and

```
public String getColumnName(int column) {
   switch (column) {
      case 0 :
         return "First Name";
      case 1 :
         return "Initial";
      case 2 :
         return "Last Name";
      default :
         return super.getColumnName(column);
   }
}
```

To change from the original Select bean to the SampleSelect, select *Morph into...* from the context menu of the SampleDB bean and morph the Select bean to an itso.entbk2.samples.databean.SampleSelect bean. Make sure your Bean Type is class, and not variable. All connections should still be valid. Save and generate the code.

Run the Application

Test your new application. The three columns of the table should now display nice headings and data.

2.4 Summary

In this chapter we describe the function of the data access beans and show with a sample application how easy it is to access a relational database with data access beans.

We show in some detail how to build a database access class with a connection to the database and an SQL select statement to retrieve relational data.

For an additional application with data access beans, see Chapter 7, "ATM Application Persistence Using Data Access Beans."

3 Enterprise Application Development with Servlets

In this chapter we describe servlets and the Servlet Builder, which is a new function of VisualAge for Java Version 2. We describe how servlets work in general and we explain how to create a servlet with the Servlet Builder.

© Copyright IBM Corp. 1998

3.1 Server-Side Applications

To use Web-based applications more efficiently, server-side applications are the most important. There are basically two approaches to server-side applications, Common Gateway Interface (CGI) programs and servlets.

Common Gateway Interface

CGI programs have performed this function for a long time. A CGI program is a server-side program that is invoked by the Web server to generate an HTML file and send it to the client. User input is added by the Web browser to the URL statement and sent to the Web server. CGI programs are usually written in C, C++, Perl, or other compiled languages. Java is also available for CGI programming, but Java servlets provide an even better approach. The major drawback to a CGI program is that the program is loaded and terminated for each interaction.

Servlets

A servlet is the best solution for creating a server-side application running on a Web server. The servlet was introduced by Sun Microsystems, and its usage is growing fast. A servlet is similar to a CGI program or Active Server Pages (ASP), but it is based on Java, so servlet programming is platform independent.

To use servlets, you need a Web server that supports the servlet model. Current Web server vendors are trying to support servlets, and many of the famous Web servers, such as Sun Java Web server and Lotus Domino Go Webserver already support servlets. Several vendors also released a servlet plugin that enables the servlet function on the Web server. IBM WebSphere Application Server is such a plugin and supports Lotus Domino Go Webserver, Microsoft Internet Information Server, and Apache. The client machine does not require Java; a simple browser with HTTP support is fine.

Sun provides the Java Servlet Development Kit (JSDK) to create servlets. JSDK is freely distributed, but it only includes a run-time library and a testing tool.

With VisualAge for Java Version 2, IBM has released an easy servlet creation tool, the Servlet Builder. With the Servlet Builder and the Visual Composition Editor, you can create servlets visually and debug them in the VisualAge for Java development environment.

What Are Servlets?

As mentioned, servlets are server-side Java programs. Servlets are invoked by the Web server, and they communicate with a client through HTTP. The client performs standard Web browsing. The Web server receives a request that invokes a servlet. The servlet processes the request and generates an HTML reply that is returned to the client.

Because servlets run on a Web server, they have no user interface. Servlets receive data values from an HTML form, and they return an HTML reply. Servlets have no security limitation; they can access files and databases on the server or connect to other systems within the enterprise. For example, a servlet can invoke a CICS transaction for part of its processing.

A servlet is loaded into a Web server only once, when the first user invokes the servlet. After that, the Web server calls the servlet to process the user request. For visual servlets, the Web server creates a new instance of the servlet to use. This instance is removed when the servlet finishes its process.

HTTP Session

Let's look at basic HTTP transactions (Figure 23).

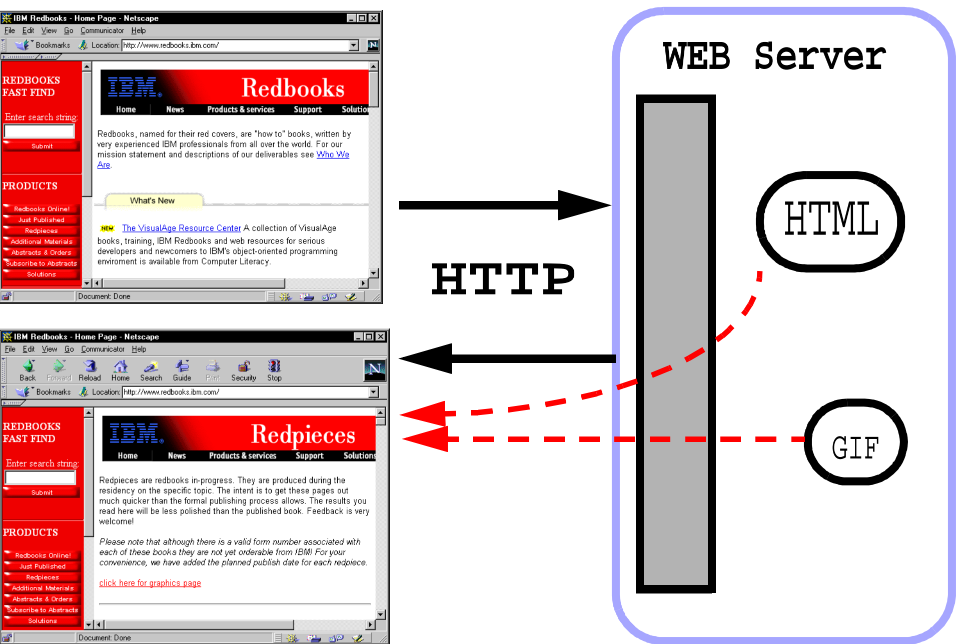


Figure 23. HTTP Transactions

The Web browser sends URLs to the server through an HTTP session, and the Web server sends back an HTML file. Once the Web browser receives the HTML file, it reads it and requests more information from the Web server as additional HTTP transactions, for example, to retrieve images or other required items. If the user clicks on a link, the Web browser sends the URL through another HTTP session. These interactions create a big problem for an application that uses several continuous pages. Each HTTP request is separate, and such an application cannot keep some data unique to the user.

To keep data across multiple HTTP transactions, application programs are used on the Web server, either CGI programs or servlets.

Servlet Processing

Figure 24 shows the servlet programming model.

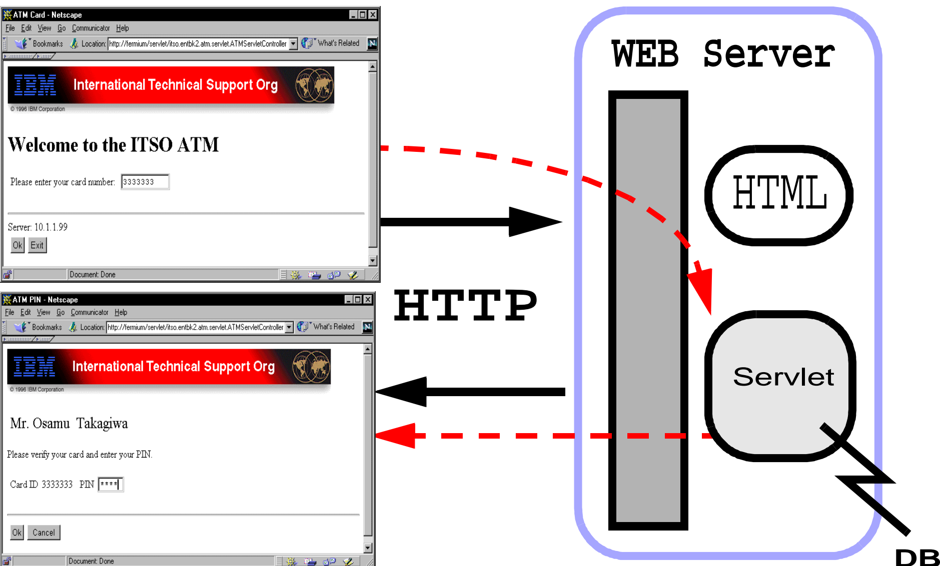


Figure 24. HTTP Session and Servlet

Servlet or CGI?

Similarities and differences between servlets and CGI programs are:

- Both the servlet and the CGI program can hold user requests and generate different HTML files, depending on the user request.
- Servlets are platform independent because they are written in Java. CGI programs are usually written in other programming languages, although Java is supported for CGI as well.

- For CGI programs, the Web server creates a process for every request by the client, but a servlet is loaded once and is then ready to run. After the first request, the servlet is not removed from the Web server until it is unloaded. Therefore a servlet is faster than a CGI program.
- A servlet can use multiple threads. It is easy to split jobs by using threads. For example, with JDBC it takes a long time to connect to a database. However, you can create the connection as another thread and keep it running. When the servlet that uses the database is started, it can access the database immediately, using the database connection thread.
- Servlets are secure and can use standard Java security. If the Web server is secure, servlets are secure too.
- Servlets and CGI programs work with a lightweight client. Only a simple HTML browser is required (Java applets do require JDK-specific browsers).
- Servlets are scalable and can use all Java functions.
- Both the servlet and the CGI program support cookies and can keep user information on the client PC.

Servlet Creation Tools

What do you need to create servlets? The JSDK is good for studying how servlets works but not quite good enough for real production of an application. In "Inside Servlets" on page 52 we describe how to create a servlet with the JSDK.

The latest JSDK is Version 2.0, but (at the time of writing) only Sun's Java Web Server supports Version 2.0. Use JSDK 1.1 for the following reasons:

- To use JSDK 2.0, you have to use the beta version of JDK 1.2. As you know, beta code is not good for production.
- Many Web servers and plugins support Version 1.1 only.
- VisualAge for Java Enterprise Version 2 is based on JDK 1.1.6 and supports JSDK 1.1, which is included in the product.
- VisualAge for Java provides a servlet test environment, and you do not require a real Web server for testing.
- VisualAge for Java contains an easy-to-use servlet creation tool, the Servlet Builder.

Why Servlet Builder?

The Servlet Builder uses JSDK 1.1 and provides a servlet development environment using visual construction. Now you can construct your servlet

Web page layout through drag and drop of HTML elements. The Servlet Builder components are:

- A SmartGuide (Create Servlet) to generate a skeleton of a servlet
- A palette of visual beans for HTML elements and servlets (cookies, session data)
- The Visual Composition Editor for visual constructions of HTML pages and forms and for connecting servlet data and events to other Java beans for processing and enterprise access
- A test environment with a built-in Web server for debugging servlets in the Workbench

Web Server Consideration

To run servlets on a Web server, make sure the Web server contains a servlet run-time facility. Table 3 lists some Web servers that support servlets.

Table 3. Web Servers Supporting Servlets

Server Name	Vendor	JSDK Version
IBM WebSphere Application Server	IBM	1.1
Lotus Domino Go Webserver	Lotus/IBM	1.1
Sun Java Web Server	Sun	2.0

3.2 Inside Servlets

Let's see how servlets work and how to develop them. Servlets can be invoked in several ways. The simplest way is to specify a URL in the Web browser:

```
http://your.web.server.com/servlet/package.YourServletClass
```

This is similar to a URL request for an HTML page. To invoke a servlet you should specify the second word as servlet and add a name of the servlet class with its package name. For such a request, the Web server loads the servlet (the first time only), instantiates the servlet, and runs it.

So what is a servlet class? A servlet class implements the javax.servlet.Servlet interface. The JSDK provides two implementations, the GenericServlet and the HttpServlet. When you create a servlet, you subclass either of these two implementations, depending on what level of API you want to use. This will become clear when we describe the simple examples.

After the servlet is loaded, the Web server invokes the init method of the servlet. This is only done once and not for every invocation.

For each invocation of the servlet, some API-specific methods are called. This is where the logic of the servlet is written and the result HTML is generated.

Simple Servlet

Let's study a simple servlet (Figure 25). This servlet sends a short message to the client. To invoke this servlet, use this URL:

```
http://[Your Server Address]/servlet/Simple
```

```
import javax.servlet.*;
import java.io.*;

public class SimpleGeneric extends GenericServlet {

   public void init( ServletConfig cfg ) throws ServletException {
      super.init(cfg);
   }
   public void service (ServletRequest req, ServletResponse res)
                     throws ServletException,IOException {
      res.setContentType("text/html");
      ServletOutputStream so = res.getOutputStream();
      so.println("<html>");
      so.println("</html>");
      so.println("<body>");
      so.println("Welcome to the Servlet world!");
      so.println("</body>");
      so.close();
   }
   public void destroy() {}
}
```

Figure 25. Simple Servlet Source Code

GenericServlet

The javax.servlet.* packages are included in the JSDK and the javax.servlet.GenericServlet class is a skeleton class for a standard servlet.

Init and Destroy Methods

The init and destroy methods are called only once when the servlet is loaded and unloaded. The init method is a good place to activate a thread for database connections.

Service Method

The service method is an abstract method and must be implemented when subclassing from the GenericServlet class. (An alternative using the HttpServlet class is discussed in "HttpServlet" on page 57.)

The service method processes the user request and return. The servlet request object contains parameters provided by the client. Our sample does not process any input and only returns the "Welcome to the Servlet world!" message to the client in HTML format. To respond, open a stream on the ServletResponse object and write to it. To receive a parameter, use the getParameter method of the ServletRequest object.

Tables 4 and 5 list the usable methods of the ServerRequest and the ServerResponse classes.

Table 4. Methods of the ServerRequest Class

Method	Description
ServletInputStream getInputStream()	Returns a stream handle to receive binary data from the client
BufferedReader getReader()	Returns a handle to receive text data from the client with proper encoding
String[] getParameterValues(String)	Returns values of a named parameter from the client HTML form
String getRemoteAddr()	Returns client's IP address
String getRemoteHost()	Returns client's host name
String getServerName()	Returns server's host name

Table 5. Methods of the ServerResponse Class

Method	Description
ServletOutputStream getInputStream()	Returns a stream handle to send binary data to the client
PrintWriter()	Returns a handle to send text data to the client with proper encoding

Invoking a Servlet in HTML

This example invokes a servlet from an HTML file. This type of HTML file has a special .SHTML extension (Figure 26).

```
<HTML> <BODY> <CENTER> Welcome to My Home Page! <P>
You are the <SERVLET CODE=Counter> </SERVLET>
th visitor! <BR> </BODY> </HTML>
```

Figure 26. Server-Side Include HTML File

When the Web server gets such a request, it invokes the servlet and then merges the output of the servlet with the HTML and sends it to the browser. This technique is called *server-side include* (SSI).

Here is the servlet tag syntax:

```
<SERVLET CODE=class name with package name>
   <PARAM NAME=Parameter1 VALUE=One>
   <PARAM NAME=Parameter2 VALUE=Two>
</SERVLET>
```

The syntax of the servlet tag is similar to that of the applet tag. In the code field, specify the full servlet class name and add parameter data through the PARAM tag.

Counter Servlet

The counter servlet (Figure 27) is not much different from the simple servlet (Figure 25 on page 53). The counter servlet has a permanent variable counter that is initialized in the init method. In the service method, the counter is increased and written as HTML to be merged with the existing SHTML file. (Note that the counter is reset to zero if the servlet is unloaded.)

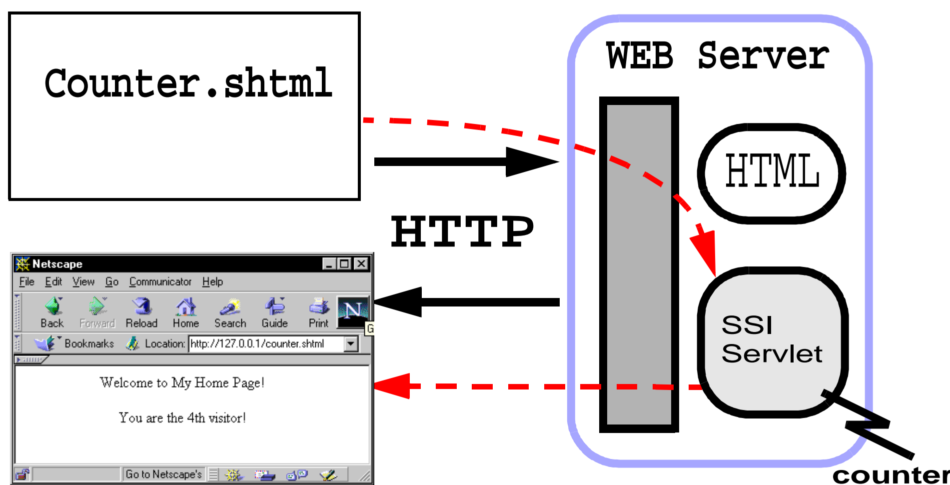


Figure 27. Servlet with HTTP Server-Side Include

Figure 28 shows the source code of the counter servlet.

```
import javax.servlet.*;
import java.io.*;

public class Counter extends GenericServlet {
   private static int count;
   public void init( ServletConfig cfg ) throws ServletException {
      super.init(cfg);
      count = 0;
   }

   public void service (ServletRequest req, ServletResponse res)
                throws ServletException,ioException {
      ServletOutputStream so = res.getOutputStream();
      so.println("<B>");
      so.println(count++);
      so.println("</B>");
      so.close();
   }
}
```

Figure 28. Server-Side Include Counter Servlet Source Code

Invoking a Servlet with Parameters

This sample shows you how to pass the data from the client to the servlet. As with a CGI program, when the user clicks on a button in a form, the servlet is invoked with user input data.

HTML forms provide a way to submit user input data. In the form tag, you can specify the target servlet name. The entry fields become parameter values, and a button (usually Submit) invokes the servlet. Figure 29 shows a sample HTML file invoking a servlet with a user-entered parameter. Note that the submit button displays the text Send.

```
<HTML> <BODY>
<FORM method="Post" action="http://www.xxx.com/servlet/PostServlet">
   <CENTER> Please let me know your name!
   <INPUT type="text" name="name"> <BR>
   <INPUT type="submit" value="Send" name="Send">
</FORM>
</BODY> </HTML>
```

Figure 29. HTML File with Form Invoking a Servlet

Entry fields are coded as <input type="text" ...>, and the Submit button is coded as <input type="submit" ...>. Clicking on the Submit button sends the form data to the Web server. The Web server invokes the servlet that is specified in the action value of the form (Figure 30).

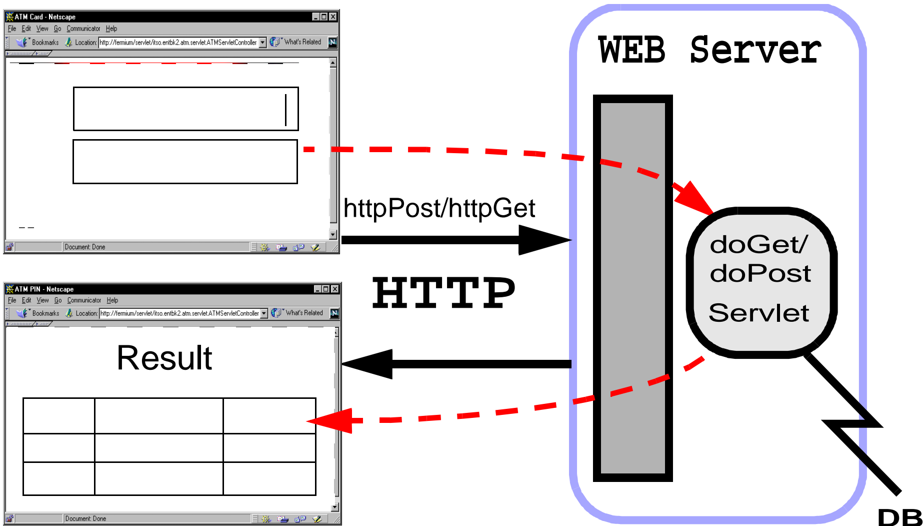


Figure 30. Servlet Processing with Form Data

Get or Post?

The method value of the form tag can be get or post. Get is the HTTP get, and post is the HTTP post. Both methods send data to the servlet. So what is different? Get sends all parameters in the URL, which has a length limitation and causes security problems because the data is viewable. Post sends the data in the HTTP entity body. It takes two steps to send the form data: contacting the Web server, and sending the data when it is requested by the server. Post might be a little bit slower than get but is recommended for large forms. (Post is not well supported by the JDSK on Windows 95/98.)

HttpServlet

The easiest way to code a servlet with get or post processing is to create a subclass of HttpServlet. HttpServlet itself is a subclass of GenericServlet. This class handles the HTML get and post requests by providing doGet and doPost methods. The service method invokes the proper method according to the get or post specification in the HTML form. In the servlet code you implement either the doGet or doPost method (or both to be completely

flexible). To get the user data from the HTML form, use the getParameterValues method of the ServerRequest object.

Servlet Post Processing

Figure 31 shows the code of the post servlet. This example retrieves the name entered by the user and sends back a thank you message. All of the names are accumulated in a vector.

```
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
import java.util.*;

public class PostServlet extends HttpServlet {

  private Vector users;

   public void init( ServletConfig cfg ) throws ServletException {
      super.init(cfg);
      users = new java.util.Vector();
   }
   public void doPost (HttpServletRequest req, HttpServletResponse res)
             throws ServletException, IOException {
      String str = req.getParameterValues("name")[0];
      users.addElement(str);
      res.setContentType("text/html");
      ServletOutputStream so = res.getOutputStream();
      so.println("<HTML><BODY>Thank you ");
      so.println( str );
      so.println("</BODY></HTML>");
      so.close();
   }
}
```

Figure 31. Servlet with Post Processing

Complex Servlets

Now you know the basics about servlets and how to create some simple servlets. Creating a small example is always easier than creating a real application.

Without VisualAge for Java, you have to design the output HTML without a WYSIWYG editor and the processing without visual composition. Using VisualAge for Java, you can create servlets visually. Let’s take a look at the Servlet Builder of VisualAge for Java Version 2.

3.3 Servlet Builder Overview

The VisualAge for Java Servlet Builder consists of four parts:

- The **Create Servlet SmartGuide** generates a visual servlet. A visual servlet inherits from com.ibm.ivj.servlet.http.VisualServlet, which inherits from javax.servlet.http.HttpServlet.
- **Servlet Builder beans** consist of visual beans and nonvisual beans. Visual beans are wrappers of HTML elements, and nonvisual beans are wrapper of Sun JSDK classes.
- The **Visual Composition Editor** provides an environment for constructing an HTML page, using Servlet Builder visual beans, and an application flow, using Servlet Builder nonvisual beans, and connecting the beans into an application.
- A **test environment** with a built-in Web server enables the debugging of servlets in VisualAge for Java with a real Web browser.

How Do Servlet Builder Beans Work?

The Web server invokes a visual servlet object to process a user request. A visual servlet (VisualServlet class) works like an HttpServlet, but its construction is much simplified by the Servlet Builder.

You create an HTML page with an input form, using the Visual Construction Editor and Servlet Builder visual beans. The Servlet Builder constructs a JavaBean that contains the data of the input form. This FormData bean has a bound property for each input field, and an event representing the Submit button. The form also specifies the name of the servlet to be invoked.

You create an HTML output page as the result of the servlet, using the Visual Construction Editor. In some instances the input and output HTML pages are the same; in other instances one servlet generates the input page and form, and another servlet processes the form and generates the output HTML page. You use the FormData bean in the servlet that generates the output. You connect its data and events to other beans for application logic and database or transaction access.

For testing, VisualAge for Java provides a tool based on the Sun Servlet Runner. A Web server is started inside VisualAge for Java to test and debug the servlet. A browser is started to receive the servlet's output and to submit further input to the same or other servlets. Because the Web server and the servlets run inside VisualAge for Java, you can debug all the code.

Advantages of the Servlet Builder

The Servlet Builder covers the weak points of the GenericServlet and the HttpServlet classes by providing:

- ❑ Visual Composition Editor for input and output forms
- ❑ Event-oriented programming, using connections
- ❑ Full capability of JavaBeans
- ❑ Session data and cookie data to keep client state
- ❑ Creation of reusable servlet beans
- ❑ Database access or connections to other enterprise resources
- ❑ Testing and debugging environment

Figure 32 shows the class hierarchy of servlet classes. For the GenericServlet the service method must be implemented. For the HttpServlet either doGet or doPost methods (or both) must be implemented. The VisualServlet is constructed using the Visual Construction Editor and no special methods must be implemented. The parameters are placed into a form data bean by the system, and the output HTML is generated from HTML beans.

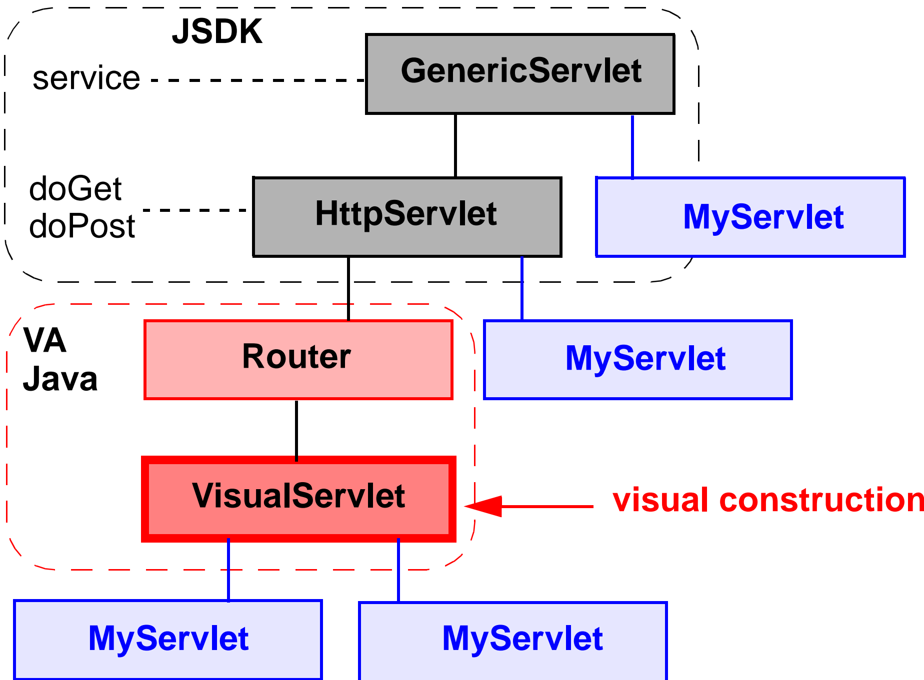


Figure 32. Servlet Class Hierarchy

Visual Servlet

The visual servlet enables you to construct the input and output Web pages using the Visual Composition Editor. You can drag and drop HTML objects such as text, forms, buttons, and entry fields. You can drop any other nonvisual JavaBeans, such as database access beans and cookie data, on the free-form surface. Then you connect the beans by events or properties.

To test the servlets simply click on the *Run* button in the tool bar. You do not have to write a configuration file for the Servlet Runner or be concerned about loading and unloading the servlet. An HTTP server and a browser are started. The Console window shows which Servlet is running and its messages. You can see which thread is running, and you can put breakpoints anywhere in the code. You can debug your servlets in the same way as you debug applets.

Each visual servlet is processed as a new instance. The Web server caches the initial instance and creates a new instance when the service method is called. This process is handled by the Router class, which inherits from HttpServlet and is the parent of VisualServlet.

Figure 33 shows the basic construction of a visual servlet.

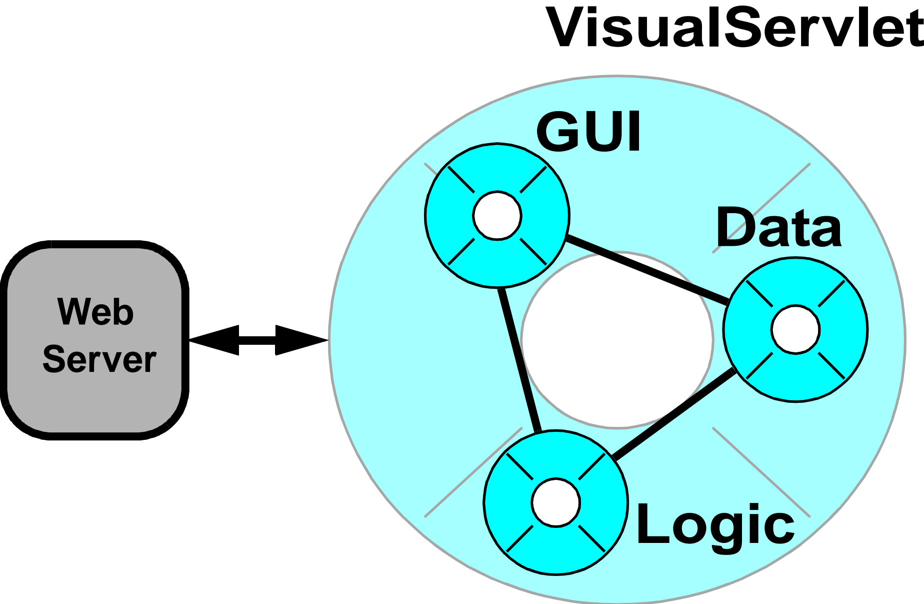


Figure 33. Visual Servlet

Servlet Builder Visual Beans

You cannot use Java's AWT or Swing classes to create the output of a servlet because the result is an HTML page. The Servlet Builder visual beans represent HTML elements and generate the HTML dynamically at run time.

To construct the visual part of a servlet, use Servlet Builder visual beans in the Visual Composition Editor. You can edit an HTML page visually just as you can edit a Java applet. The HTML page uses a unique layout technique that simulates a browser. The display of the page is not exactly the same as what you will eventually see in a browser because each browser has its own technique for displaying the details of HTML elements.

HTML Page

The container that represents an HTML page or part of a page is the HtmlPage bean. The HTML page can generate the complete output page or a part of a page that is embedded into another servlet or HTML file, using the servlet bean or tag. All other beans are placed inside an HtmlPage bean.

HTML Elements

The palette of the Servlet Builder contains two sets of visual beans. Most of the beans can be put anywhere on the HTML page (see Table 6 on page 63); however, beans that store data to be sent to the Web server can only be placed into an HTML form (see Table 7 on page 64).

HTML Tables

There are two HTML table beans: HtmlTable and HtmlResultTable. You can add and remove rows and columns at design time through a pop-up menu or at run time through application logic. HtmlTable is a static table and can be anywhere on the HTML page. HtmlResultTable is a useful bean to display data retrieved by an SQL query result. The HtmlResultTable is based on the same table models as the Swing JTable bean.

HTML Button

The HTMLPushButton bean can be used in three ways through its property sheet:

- ❑ Submit button—used to submit the form.data that is passed to the servlet
- ❑ Reset button—used to clear the data in the form (servlet is not invoked)
- ❑ Button—used to activate a script in HTML (servlet is not invoked)

Table 6. Servlet Builder Visual Beans

Icon	Bean Name	HTML	Description
Basic	HTMLPage	<HTML>	Represents an HTML page or part of an HTML page
Basic	HTMLText	NONE	Text data
Basic	HTMLImage	<IMAGE>	Sets a URL to display an image
Basic	HTMLRule	<HR>	Horizontal rule
Basic	HTMLLineBreak	 	Adds a line break to force the next element to a new line
Basic	HTMLParagraph	<P>	Starts a new paragraph
Basic	HTMLScript	<SCRIPT>	Adds a script or JavaScript code to the HTML
Basic	HTMLStyleSheet	<STYLE>	Adds a predesigned style to the HTML
Basic	HTMLServlet	<SERVLET>	Embeds the output of another servlet
Basic	HTMLApplet	<APPLET>	Adds an applet in the HTML
Basic	HTMLEmbed	<EMBED>	Adds an object such as audio in the HTML
Basic	HTMLTable	<TABLE>	Arranges HTML objects in matrix style

Table 7. Servlet Builder Visual Beans for Forms

Icon	Bean Name	HTML	Description
Form	HTMLForm	<FORM>	Container for HTML form objects
Form	HTMLHiddenInput	<INPUT>	Invisible form object to keep some data to be sent back to the server
Form	HTMLPushButton	<INPUT type=submit> (or other)	Push button to send the form data to the servlet specified for the form
Form	HTMLCheckBox	<INPUT type=checkbox>	Adds a checkbox in the HTML form (data is sent to the server)
Form	HTMLRadioButtonSet	<INPUT type=radio>	Adds a radio button in the HTML form (data is sent to the server)
Form	HTMLEntryField	<INPUT type=text>	Adds an entry field in the HTML form (data is sent to the server)
Form	HTMLTextArea	<TEXTAREA>	Adds a text area in the HTML form (data is sent to the server)
Form	HTMLList	<SELECT>	Adds a list box in the HTML form (data is sent to the server)
Form	HTMLDropDownList	<SELECT size=1>	Adds a drop-down list in the HTML form (data is sent to the server)
Form	HTMLResultTable	<TABLE>	Adds a table that represents a query result in the HTML form
Form	HTMLResultColumn	<TR><TH>	Adds a column to the result table in the HTML form

Servlet Builder Nonvisual Beans

To implement the application server-side logic you can use any JavaBeans. The Servlet Builder provides three nonvisual beans to handle the servlet data (Table 8).

Table 8. Servlet Builder Nonvisual Beans

Icon	Bean Name	Description
	FormData	Contains user input data and generates an event representing the Submit button
	CookieWrapper	Saves or retrieves data in a cookie that is stored on the client. Value is a string identified by a key. An expiration data can be specified.
	SessionDataWrapper	Saves or retrieves data in a session bean that is stored on the server. Value is any object, for example a JavaBean.

Form Data

Every input element has to be placed inside an HTML form bean. A property of the form is used to select the method for passing parameters as get or post (see "Get or Post?" on page 57). The action of the form is another property and allows you to link to a URL for processing by any tool or select a servlet from the list of existing servlets to process the data.

The **FormData** bean is generated by the Servlet Builder when you save the visual servlet class from the Visual Composition Editor. The name of the generated class is *YourServlet*FormData. The FormData bean is the key part for the application flow of control.

The FormData bean contains properties for all input values on the HTML page, such as entry field strings, selected list items, checkboxes, and any other data elements placed inside an HTML form of your visual servlet. The FormData bean only fires events and has input properties if the current request corresponds to the matching form.

Visual Composition with the FormData Bean

The FormData bean contains the data from the client. To handle this data in the Visual Composition Editor, put the FormData bean on the free-form surface and connect events and properties.

When you put the FormData bean next to your visual HTML page, it seems that it gets its values directly from the form in the page, but that is not the case. You can connect form data properties to the visual beans to prepare the output HTML page sent to the client. You cannot get the user data values through connections from the visual HTML fields; they are placed into the FormData bean automatically when the user submits the form.

What you are designing visually is the output page, not the input page. Note that in many cases the input page is not the same as the output page. You will use the FormData bean generated from an input form in the servlet that processes the data and displays it in an output page (Figure 34).

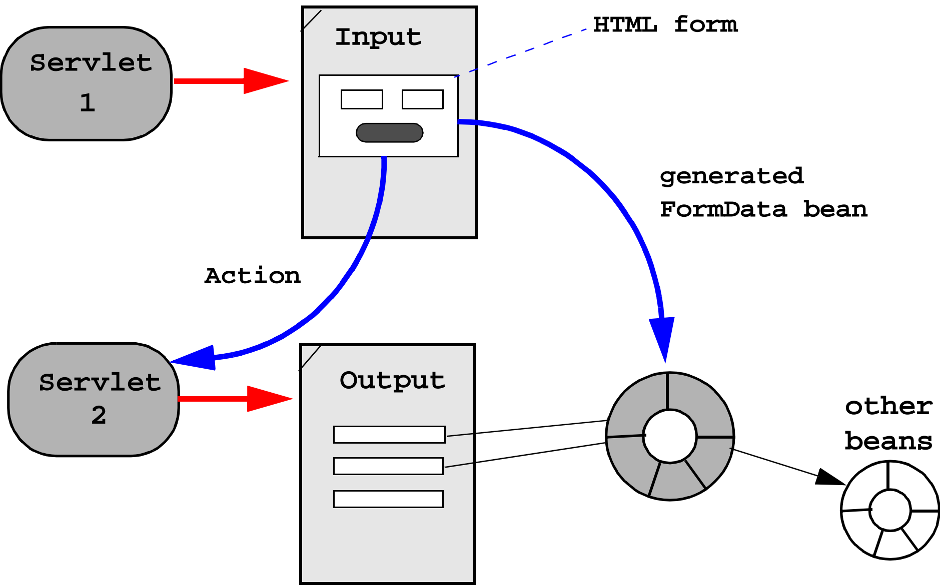


Figure 34. Flow between Servlets

Cookie Wrapper

A cookie is a data object stored on the client machine. A cookie is useful for keeping data during a session or a specified time frame. The servlet can save the data to the client hard drive as a file if the user accepts it. Cookies have been used before servlets existed to save data. Once a cookie is saved on the client, a servlet can access the cookie data by request. This is useful for keeping client access information handy, for example, to inquire about the progress of the application. Cookies can live longer than the servlet; the cookie data is available until its expiration date or until it is deleted from the client machine.

The **CookieWrapper** bean is a nonvisual bean to be used with the Visual Composition Editor. The CookieWrapper bean has two properties: name and value. The value is a string. You can keep multiple CookieWrapper beans with individual name and value pairs. Each CookieWrapper bean is stored as a cookie on the client machine.

Session Data Wrapper

Session data is similar to a cookie but is stored on the server. Session data is only available as long as the servlet is loaded in the Web server.

The **SessionDataWrapper** bean works like the CookieWrapper bean. It contains two properties: name and value. The value of the SessionDataWrapper bean is any object. You can keep multiple SessionDataWrapper beans with individual name and value pairs. A cookie that points to the session data is generated automatically.

The advantages of the SessionDataWrapper bean are:

- The SessionDataWrapper bean keeps any object (a cookie keeps strings).
- There is no network transfer of data because the object is on the server.
- The data is secure for each user session.

Run Configuration

To test the servlet, simply click on the *Run* button in the tool bar. VisualAge for Java has a built-in Web server from the Sun JSDK Servlet Runner. If you need to configure or would like to use WebSphere Application Server, you have to edit configuration file. The file name is:

```
\IBMVJava\ide\project_resources\IBM Servlet ....... class libraries\
   com\ibm\ivj\servlet\runner\configuration.properties
```

To change the browser for a test, specify a browser in the configuration file or change the browser used by the Help facility in the *Windows -> Options* dialog. You can change the server host name or address, port, and directory:

```
URL 127.0.0.1:8080                 <=== localhost address
Directory /../serunner
```

Invoking Another Servlet

Each visual servlet has a TransferToServiceHandler property where you can specify another servlet to be invoked. To use a servlet as a nonvisual router or controller, set the *isTransferring* property to true and, after processing, use the *TransferToServiceHandler* property to invoke another servlet.

3.4 Creating Visual Servlets

In this section we describe how to create four basic visual servlets with the Servlet Builder of VisualAge for Java:

- ❑ A simple servlet that only displays a message
- ❑ A server-side include servlet
- ❑ A servlet that uses a variable to keep data as long as the servlet is loaded in the Web server
- ❑ An interactive servlet that sends back the data entered by the user

Loading the Servlet Builder

Before you can create any servlets with VisualAge for Java you have to load the Servlet Builder into the Workbench. If you cannot see the **IBM Servlet Builder class libraries** project in your workbench, use *Quick Start* (F2) and *Add Feature* to add the Servlet Builder to your Workbench (Figure 35).

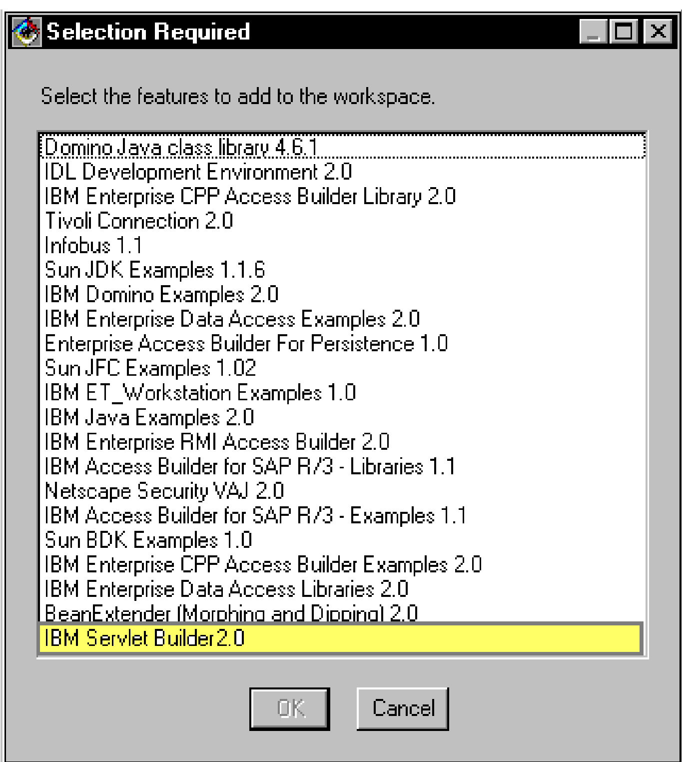


Figure 35. Adding the Servlet Builder Feature to the Workbench

Simple Servlet

It is time to create a simple servlet with VisualAge for Java. Use *Quick Start* and select *Servlet*. Select *Create Servlet* then click on the *OK* button. Fill in all fields in the SmartGuide window (Figure 36). We use the **itso.entbk2.sample.servlet** package for these exercises.

Choose *Simple* from the template list to create an HTML page.

Click on *Next* and move to next page. You do not have to modify this page. The servlet name will be your class name, and the form data class is generated as ***ClassName*FormData**. The radio buttons allow you to select a server-side include; for now leave this option as a stand-alone page. Click on *Finish* to generate the servlet.

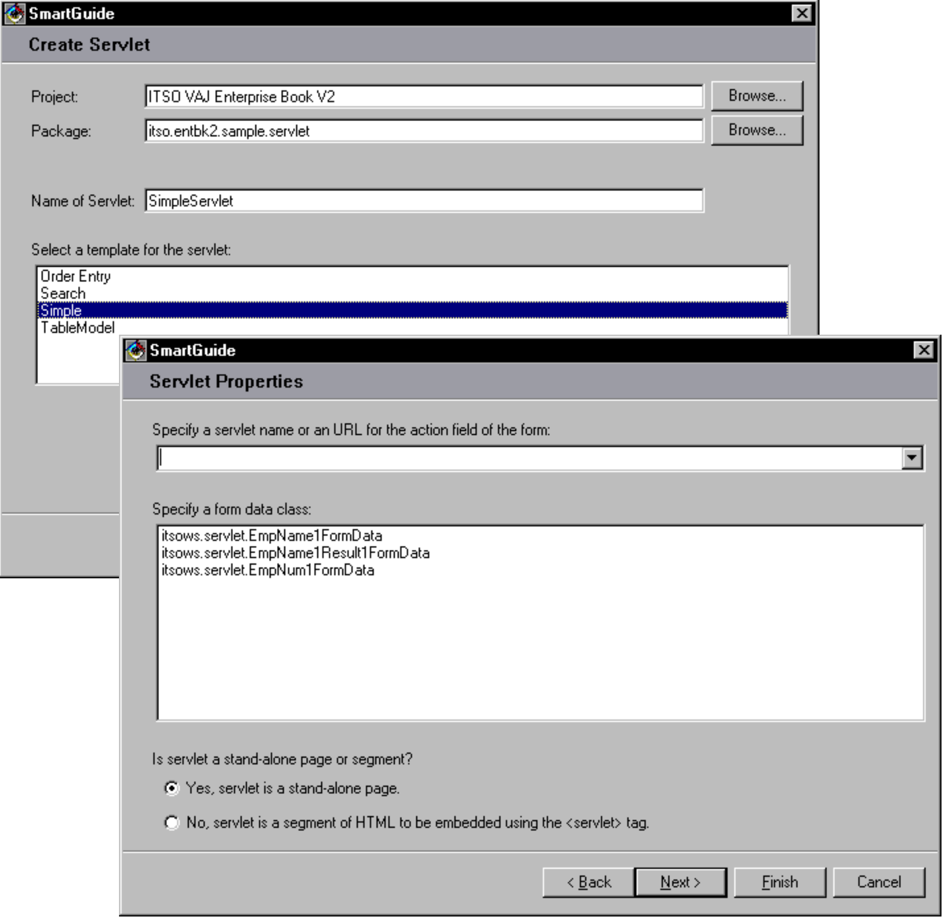


Figure 36. Create Servlet SmartGuide

The Visual Composition Editor opens with an HTML page. Select the Servlet palette (Figure 37) and add an HTMLText bean to the page.

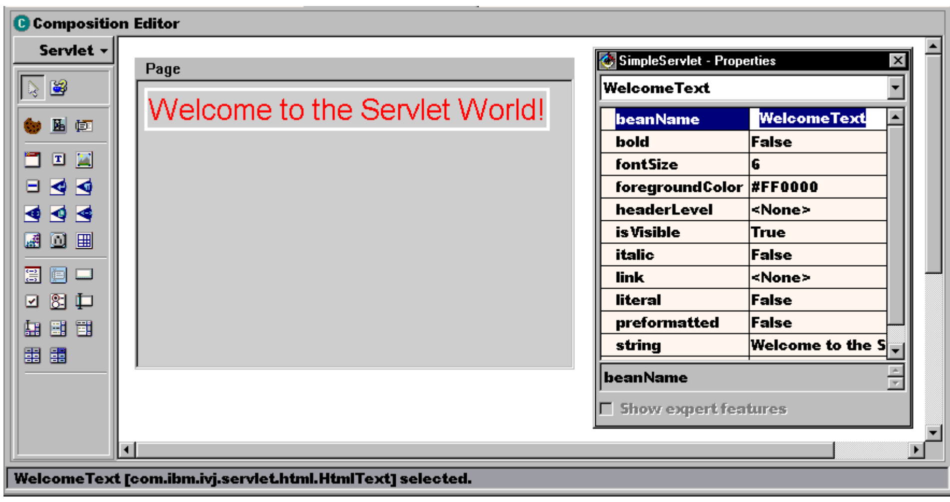


Figure 37. Servlet Palette and HTML Page in Visual Composition Editor

Edit the properties of the HTMLText bean:

- ❑ Change the bean name to WelcomeText.
- ❑ Change the font size to 6.
- ❑ Change the foreground color (for fun).
- ❑ Change the HTML text string to *Welcome to the Servlet World!*

Testing the Servlet

To test this servlet, click on the *Run* button in the tool bar. The built-in HTTP server is started and displays this message in the Console window:

```
ServletRunner starting with settings:
  port = 8080
  backlog = 50
  max handlers = 100
  timeout = 5000
  servlet dir = .
  document dir = .
  servlet propfile = .\servlet.properties
```

The servlet is started from the Web browser. You can change the browser from the Help option of the *Window -> Options* dialog. Error messages or debug output of the servlet are displayed in the Console window.

Server-Side Include Servlet

Do you remember the difference between an SSI servlet and a standard servlet? The standard servlet returns a full HTML page, whereas the SSI servlet generates only a part of the HTML page.

VisualAge for Java generates the HTML tags. To set the tags to be generated by an SSI servlet, either change the *generateBodyElement* property of the HTML page bean to *false*, or select *No, servlet is a segment of HTML to be embedded using the <servlet> tag* on the second page of the Create Servlet SmartGuide (Figure 36 on page 69).

Create an SSI servlet named SSIServlet with two HtmlRules and the text *Imbedded from SSI Servlet* between the rules. You should not need any specific instructions for this task.

To use the SSI servlet in another servlet, drop an HTMLServlet bean onto the primary HTMLPage of the SimpleServlet and change the *code* property of the embedded servlet to specify the name of the SSI servlet (Figure 38).

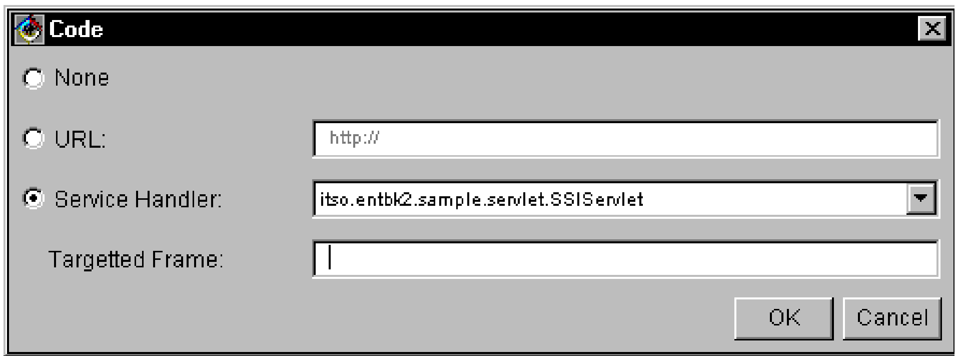


Figure 38. Code Property for a Server-Side Include Servlet

See Figure 39 for the layout of the SimpleServlet with the embedded SSIServlet.

Counter Servlet

For every user request, an instance of the servlet is created. This instance is destroyed when processing has finished. However, the servlet can keep a class variable that can be used with all instances of one servlet class.

Let’s add a counter to the simple servlet. To hold the counter value, the servlet must have a static property that we connect to an HTML text:

- ❑ Open the BeanInfo page of SimpleServlet.
- ❑ Create a bound read-only *int* property named *counterValue.*
- ❑ Go to the Method page and deselect all methods.
- ❑ Edit the modifier of *fieldCounterValue* variable to *static.*
- ❑ Select the getCounterValue method and change the return statement:

```
return fieldCounterValue++;
```

- ❑ Go to Visual Composition Editor and add a paragraph and two HTML text fields to the page. Change the text fields to *Counter:* and *count.*
- ❑ Connect the *counterValue* property (of the servlet) to the *string* property of the count text field (Figure 39).

Figure 39. HTML Page with a Counter

Save the servlet and test it. Click the *Reload* button in your browser and verify that the counter is counting the visitors (Figure 40).

Figure 40. Servlet with Counter in Netscape Browser

Passing Data to the Servlet

Now let’s implement an interactive servlet that can handle user input and respond to the user after performing some work. This sample, named InteractiveServlet, has an entry field, a push button, and an image field. The servlet generates the response HTML with the image requested by the user.

❑ Create a Servlet named InteractiveServlet, using the SmartGuide with the Simple template.

❑ Add a header level 1 text of *Interactive Servlet*.

❑ Add an HtmlForm after a paragraph.

❑ Place an HTMLTable into the form. The table has two rows and columns by default.

❑ Place an HTMLText into the top left cell of the table and set the string to *Image file:*.

❑ Place an HTMLEntryField into the top right cell, name it ImageSource*,* and set the size to 30.

❑ Place an HTMLPushButton into the bottom right cell, name it GetButton, and set its string value to *Get Image*.

❑ Select the cell with the push button and change the align property to *Right*.

❑ Add a third row (use the Beans List to select the second row, then select *Add Row Below* in the context menu).

❑ Put an HTMLImage into the bottom right cell and name it Image.

Once you have designed the page layout, save the servlet (*Bean -> Save*) to generate the FormData bean. Use the FormData bean to handle the input data.

❑ Place a FormData bean on the free-form surface. Select the InteractiveServletFormData bean in the dialog (1).

❑ Connect the *imageSourceString* property (HtmlEntryField) of the FormData bean to the *source* property of the Image (HtmlImage) in the table (2).

❑ Connect the *imageSourceString* property to the *string* property of the entry field in the input form to redisplay the selected image name (3).

❑ Connect the *GetButtonPressed* event of the FormData bean to the *transferToServiceHandler* method of the servlet itself (free-form surface) and pass the *this* of the servlet as a parameter (4).

Note that you cannot connect from the *Get Image* button in the HTML page. The button does not have an action event. You must use the properties and events in the FormData bean for any action to be performed.

The servlet displays the image entered by the user. Test the servlet and enter the full file name of any image that you can access on your machine. Figure 41 shows the finished servlet.

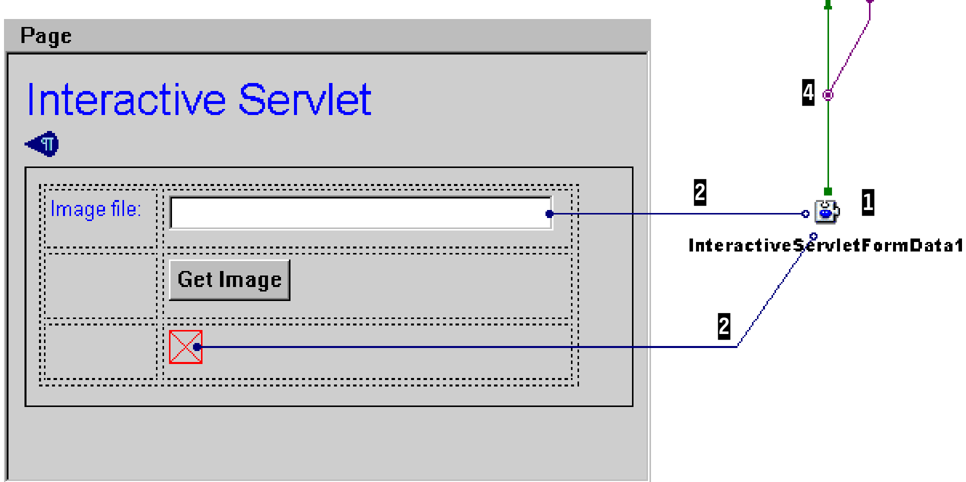


Figure 41. Interactive Servlet

3.5 Advanced Servlet Techniques

In this section we cover advanced servlet topics. HTML has more tags than are available through the visual beans. We start by explaining how to use extra HTML tags and then move to implement cooperation among multiple servlets. We conclude by discussing how to implement servlets in a multitier application environment.

Advanced HTML Tags

Some advanced HTML tags are applet, embed, style sheet, and JavaScript.

Applet

If you want to embed an applet in the servlet form, use the HTMLApplet bean. This bean has properties that are required to construct the applet tag, such as *code*, *codebase*, *parameters*, *size*, and *archives*. Properties are reflected in the generated applet tag. You cannot see the applet in the Visual Composition Editor where you develop the servlet. You can see its layout only in its own development environment.

Embed

Embed is similar to an applet but is used to embed other objects, such as sound or movies. Use the HTMLEmbed bean and set the *source* property to point to the URL of the target object.

Style Sheet

You can define a page style, using the HTMLStyleSheet bean. A style sheet contains fonts and colors of various heading and body elements. To keep your pages in the same formatting style, you can point with the URL property to a predefined HTMLStyleSheet bean. Additional specifications can be added through the *styles* property (Figure 42) and the *extraAttributes* property.

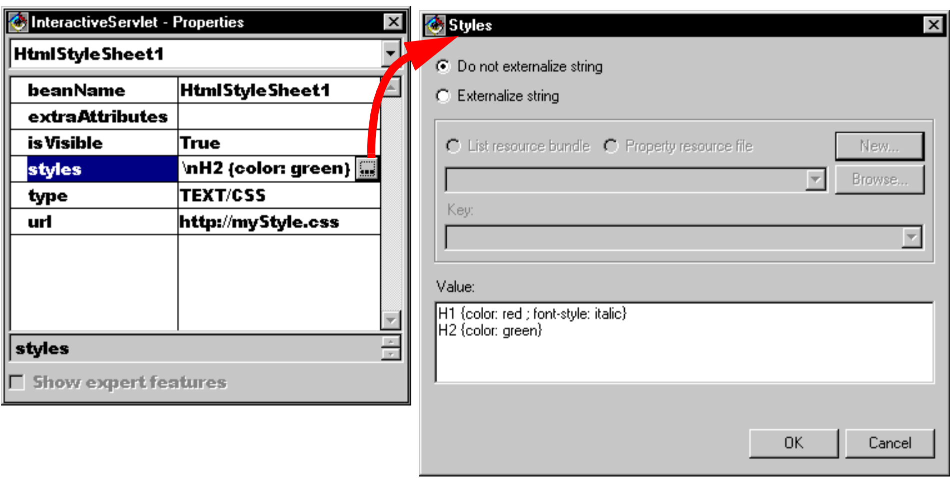


Figure 42. Style Sheet Specification

JavaScript

If you want your page to be more active, for example, for input validation or animation, you can use JavaScript. JavaScript can handle list or button events and set or edit the fields of forms. Therefore, if the process does not need data from the server, JavaScript is a good solution for processing because it is much faster than invoking code on the server.

To use JavaScript, drop an HTMLScript bean into the HTML page and write the script in the string property. To activate the script from a form, write code such as *onSelect=...* into the *extraAttribute* property of the form (Figure 43).

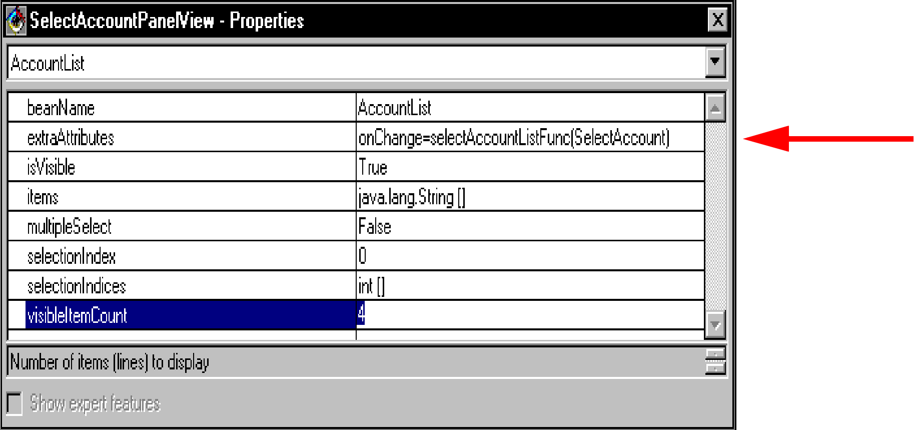


Figure 43. JavaScript Invocation

Servlet Chaining

To create a real application, it is not enough to have one form and one process. Applications usually have several forms and a continuous process. In this section we describe how to create an application with multiple forms.

Let's think about a servlet with a single flow. Assume a data entry application. The user must log on and enter the data, and a result is calculated. For this application we require three forms:

- ❑ Sign on form
- ❑ Data entry form
- ❑ Conversion form (result)

Here is a scenario. The first form is the sign on form. The user enters the logon ID and clicks the *SignOn* button. The servlet processes the logon and displays the entry form. The user enters the data and clicks on the *Process* button. The servlet calculates and displays the conversion form (Figure 44).

Figure 44. Servlet Chaining in Single Flow

Step by Step Instructions

Follow these steps to create the sign on servlet (Figure 45):

- ❑ Create a visual servlet named AppSignOn.
- ❑ Use an HTMLText bean as header level 1 with the string *Sign On Panel: Enter your ID.*
- ❑ Place an HTMLForm below the text.
- ❑ Place an HTMLTable into the form and delete the second column.
- ❑ Place an HTMLEntryField in the left cell of the table and name it UserID.
- ❑ Place an HTMLButton in the right cell and name it SignOn and set the string to *Sign On.*

Now you have the sign on servlet. Save it and generate the AppSignOnFormData bean.

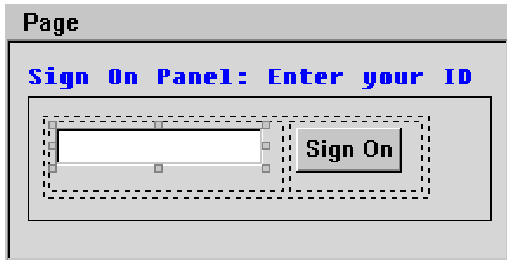


Figure 45. Sign On Servlet

The data entry servlet greets the user and presents a data entry form. The Dollar amount entered by the user is passed to the conversion servlet to calculate the equivalent value in Japanese Yen. Follow these steps to create the data entry servlet (Figure 46):

❑ Create a visual servlet named AppEntry.

❑ Add a header level 1 that reads *Currency Conversion*.

❑ Add a table with two text fields that read *Welcome* and *??*. Name the second field userID (it will display the sign on ID from the first servlet).

❑ Add a form with a table of one row and three columns. Put the text *Dollar amount:* into the left column, an entry field named Dollar into the middle, and a push button named Calculate into the right column. Set the text of the button to *Calculate Yen*.

❑ Put the AppSignOnFormData generated from the sign on servlet on the free-form surface.

❑ Connect the *userIDString* property of the AppSignOnFormData bean to the *string* property of the userID text field. This action copies the user ID entered in the sign on servlet to the output of the data entry servlet.

Save the data entry servlet to generate the AppEntryFormData bean.

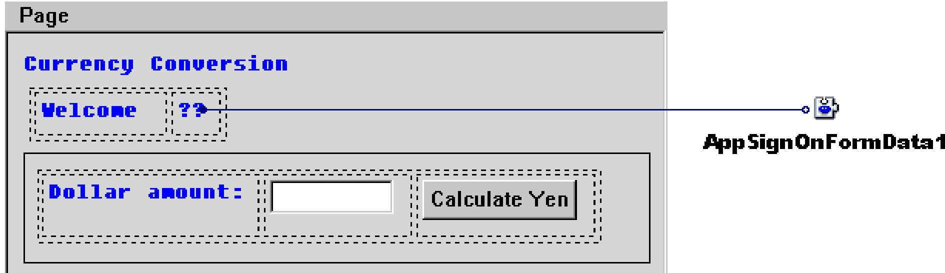


Figure 46. Data Entry Servlet

The data entry servlet gets data from the sign on servlet and passes data to the conversion servlet. How does the sign on servlet invoke the data entry servlet? Yes, you have to relate them:

- ❑ Open the sign on servlet again.
- ❑ Double-click on the form, select the *action* property, and click on the small button in the edit field.
- ❑ Select Service Handler and choose the AppEntry servlet in the drop-down list (Figure 47). Click on *OK* and save the servlet.

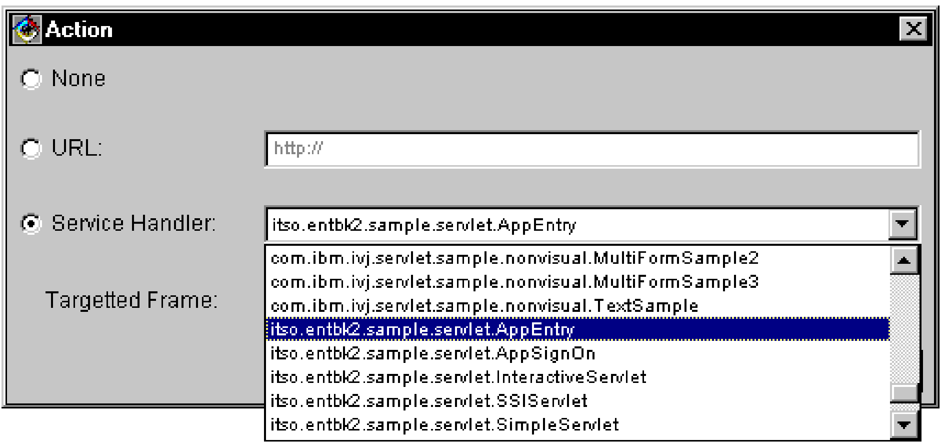


Figure 47. Service Handler Specification for Form Action

The conversion servlet is next. The Dollar value is passed from the data entry servlet and converted into Yen. To perform the calculation we use an Event-to-Code connection. Follow these steps to create the conversion servlet (see Figure 48):

- ❑ Create a visual servlet named AppConversion.
- ❑ Put a header level 1 that reads *Currency Conversion Result*.
- ❑ Add a table with two rows and columns. Set the *border* property to 2.
- ❑ Place two text fields that read *Dollar amount* and *Yen amount* into the first row.
- ❑ Put two text field into each cell in the second row. Use *$@*, *xxx*, *Y@*, and *yyy*, where @ stands for one blank character.
- ❑ Change the align property of the two lower cells to *Right*.
- ❑ Add a thank you text field below the table.
- ❑ Add the AppEntryFormData bean to the free-form surface (1).

❑ Connect the *dollarString* property of the AppEntryFormData bean to the *string* property of the xxx field (2).

❑ Connect the *calculatePressed* event of the AppEntryFormData bean to Event-to-Code (3). Create the calculateYen method that converts a Dollar input into a Yen output:

```
public String calculateYen(String dollar) {
   return new String(new Integer(Integer.parseInt(dollar)*120).toString());
}
```

❑ Connect the *dollar* parameter of the connection to the *dollarString* property of the AppEntryFormData bean (4).

❑ Connect the *normalResult* of the event-to-code connection to the *string* property of the yyy field (5).

❑ Save the servlet.

❑ To invoke this servlet from the data entry servlet, open the AppEntry servlet and set the form's action to the AppConversion servlet.

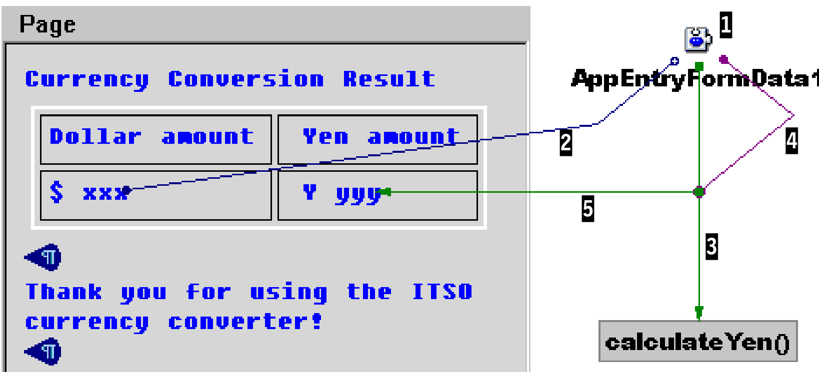


Figure 48. Conversion Servlet

Test the three servlets by starting the sign on servlet. All servlets should work. Start with the sign on servlet and calculate a few Dollar to Yen conversions.

You have learned how to create chained servlets by following these important concepts:

❑ To invoke the next Servlet, use the *action* property of a form.

❑ To access user-entered data or events, use the FormData bean.

Keeping and Passing Data between Servlets

The sign on, data entry, and conversion servlets work in a continuous way, but they do not keep such data as the user ID. The data entered in the sign on servlet is passed to the data entry servlet but is not forwarded to the conversion servlet. Each transaction is based on HTTP, and every instance of a servlet is removed after its processing. How can you keep the data? You have three ways of keeping the data:

- ❑ Use a static variable. It is not feasible, however, to keep a user-unique value in a static variable, because you can have multiple users connected to the same servlet. Note that static variables are of limited value because the servlet may be unloaded, or there may be multiple instances of the servlet in a multiserver solution with load balancing.
- ❑ Keep the data in each form. You can use a hidden field in the form to pass a value from one servlet to another. The hidden field is invisible to the user, who is not aware that such a value is passed along. An experienced user can find the value by looking at the HTML source.
- ❑ Use a cookie or session data. A cookie can be rejected by the browser or the user.

We have already used a static variable for the counter servlet (see “Counter Servlet” on page 55). Let’s implement solutions with a hidden field and with a cookie or session data.

Hidden Field

Follow these steps to implement a hidden field to pass the user ID from the entry servlet to the conversion servlet:

- ❑ Open the data entry servlet (see Figure 49 on page 83).
- ❑ Place an HTMLHiddenInput bean into the form and name it userIDhidden (1).
- ❑ Connect the *userIDString* property of the AppSignOnFormData to the *string* property of the hidden field (2).
- ❑ Save the data entry servlet to regenerate the code and FormData bean.
- ❑ Open the conversion servlet (see Figure 50 on page 83).
- ❑ Place a text field before the thank you text and set the text to ??. Change *Thank you ...* to *, thank you ...* (the user ID will go in front of the comma).
- ❑ Connect the *userIDhidden* property of the AppEntryFormData to the *string* property of the ?? text (3). (If you cannot see the *userIDhidden* property, use the context menu and select *Refresh Interface*.)

Save and test the servlets. Check out the HTML source of the data entry servlet and look for the hidden input field with the user ID.

This is a small sample with only one variable. For just a small amount of data a hidden field is just fine, but what if many forms require the same data or there is a lot of data? In such cases a cookie or session data is a better solution. Let's modify the example to utilize a cookie or session data.

Cookie

To use a cookie in a servlet, choose a CookieWrapper bean from the palette. This bean has a cookieName and a cookieValue property. In most cases a constant is used as the cookieName and a dynamic value is assigned to the cookieValue.

To store the userID in a cookie and pass it to the conversion servlet follow these steps:

- ❑ Open the data entry servlet (Figure 49).
- ❑ Place a CookieWrapper bean on the free-form surface. Open the bean, set the bean name to *userIDcookie* and the cookieName to *userID*.
- ❑ Connect the *userIDString* property of the AppSignOnFormData to the *cookieValue* property of the cookie (4).
- ❑ Open the conversion servlet (Figure 50).
- ❑ Place a CookieWrapper bean on the free-form surface. Open the bean, set the bean name to *userIDcookie* and the cookieName to *userID*.
- ❑ Connect the *cookieValue* property to the *string* property of a text (5).

Now you can save the userID data to a cookie and have it retrieved automatically in another servlet. For testing, tailor your browser to prompt you when a cookie is stored. In Netscape Navigator Version 4, open *Edit -> Preferences* and on the *Advanced* page set the *Warn me* checkbox in the Cookies section.

Session Data

A cookie can only save a string to its value property. If you would like to save an object, use the SessionDataWrapper instead of the CookieWrapper.

Lets us store a Date object in a SessionDataWrapper and pass it from the data entry servlet to the conversion servlet.

- ❑ Open the data entry servlet.
- ❑ Place a SessionDataWrapper bean on the free-form surface. Open the bean, set its name to *dateSessionData* and the propertyName to *date*.

❑ Connect the *signOnPressed* event of the AppSignOnFormData to the *propertyValue* method of the session data bean (6). To store the current date, create a parameter-from-code connection to a new *getDate* method:

```
public java.util.Date getDate() {
   return new java.util.Date();
}
```

❑ Save the servlet (Figure 49).

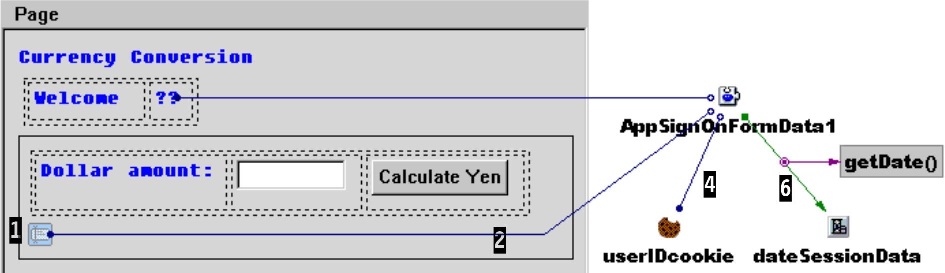


Figure 49. Data Entry Servlet with Cookie and Session Data

❑ Open the conversion servlet.

❑ Place a SessionDataWrapper bean on the free-form surface. Open the bean, set its name to *dateSessionData* and the propertyName to *date*.

❑ Connect the *propertyValue* property to the *string* of any HTML text (7).

❑ Save the servlet (Figure 50).

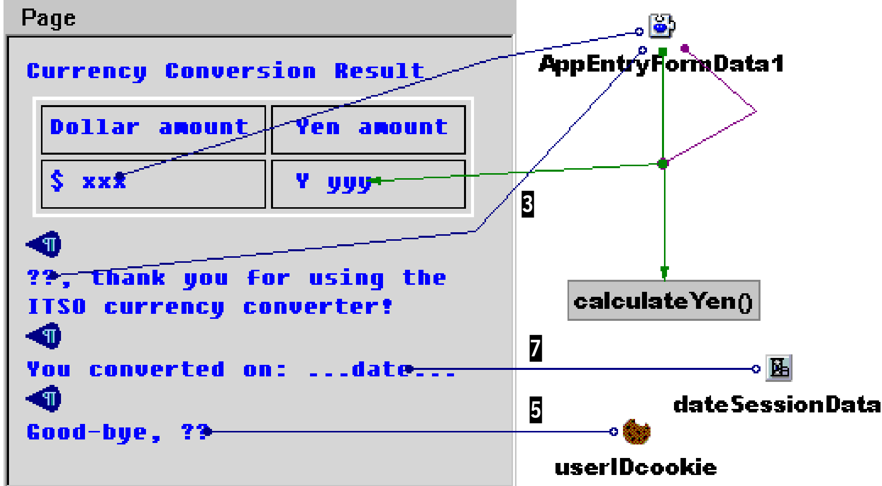


Figure 50. Conversion Servlet with Cookie and Session Data

❑ Test the servlets. If you have enabled the cookie warning, you notice that two cookies are stored: the session data pointer and the user ID cookies.

Servlet Branch

Would you like to have a branch in your application? Many applications have a selection menu and branch to multiple servlets like a fork (Figure 51).

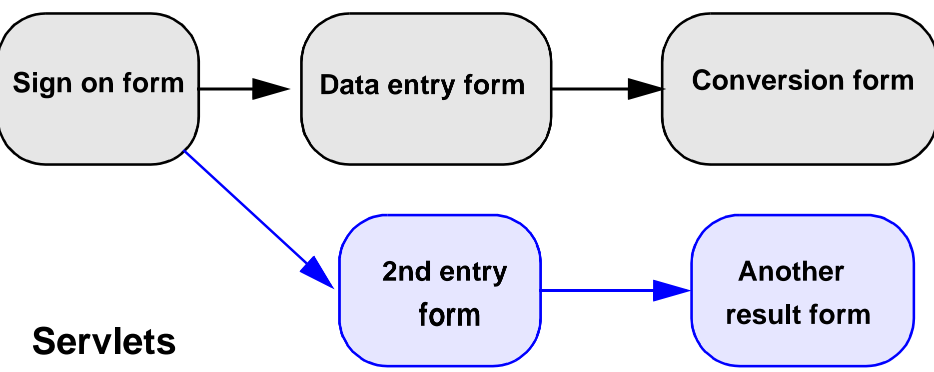


Figure 51. Servlet Branch

It is not complicated to implement a branch. The HTML form has a target servlet specification, so you can put multiple forms on one servlet page and specify the targets for each form (Figure 52).

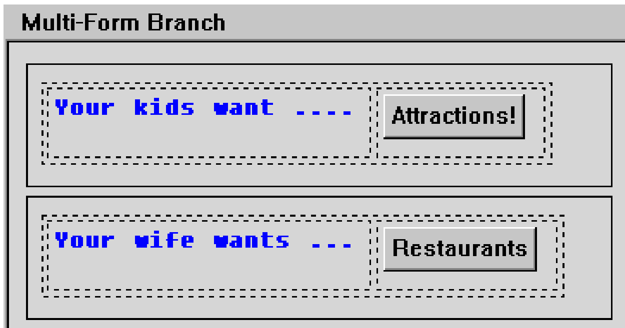


Figure 52. Branch Form

Condition Control

What about condition control? In our currency conversion application, the user ID should be validated; if it is not valid, an error message should be displayed without proceeding to the data entry servlet (Figure 53).

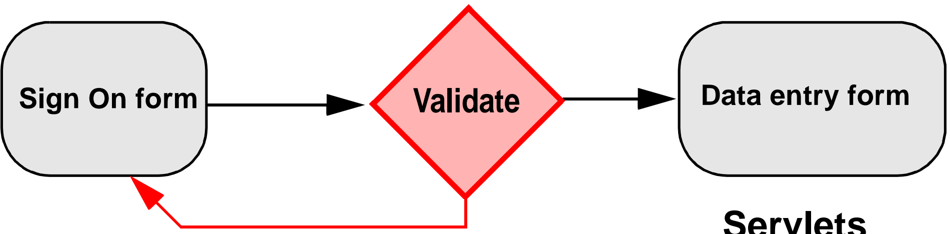


Figure 53. Condition Control Servlet

To implement a conditional branch, an invisible servlet is useful. An invisible servlet is called a *router* or *controller*. Note that there is a Router class in the servlet hierarchy (see Figure 32 on page 60), but for a different purpose. A nonvisual servlet also extends from the visual servlet class but does not have any GUI pages. The sign on servlet invokes the controller servlet, which validates the user ID. If the user ID is valid, the controller servlet invokes the data entry servlet. If the user ID is invalid, the sign on servlet is invoked again to display an error message.

When the Web server invokes a servlet, an HTML response is expected. The controller servlet has to make sure that it passes control to a servlet that creates the HTML response. The controller servlet itself does not produce any HTML.

Let's implement a message field in the sign on servlet and a controller servlet that validates the user ID:

- ❑ Open the sign on servlet.
- ❑ Add a text field with an initial value of *Welcome!,* and name the bean message. Set the foreground color to *red* and italic to *true*.
- ❑ Promote the *string* property of the message field. This creates a property named *messageString* (with getMessageString and setMessageString methods) and makes the message field accessible to the controller.
- ❑ Create the controller servlet as a visual servlet named AppRouter. Delete the Page bean that is generated.
- ❑ Connect the *requestReceived* event of the servlet to the *isTransferring* property and set the parameter value to *true*. We specify here that the servlet is transferring control to another servlet.

❑ Add two beans of types AppSignOn and AppEntry. One of these two servlet beans will get control. Add a FormData bean and select the AppSignOnFormData class.

❑ Implement the user ID validation. Connect the *signOnPressed* event of the AppSignOnFormData bean to a new *validate* method (event-to-code) (1) The validate method accepts the user ID *ITSO* and any IDs starting with the letter *U*:

```
public void validate(String userid) throws Exception {
   if (!userid.equals("ITSO") &
       !userid.startsWith("U")) throw new Exception();
}
```

❑ Set the *userid* parameter for the validate method from the *userIDString* property of the AppSignOnFormData (2).

❑ Connect the *exceptionOccurred* event of the validate connection to the *messageString* property of the AppSignOn servlet and set the parameter to *Invalid user ID, please reenter!* (3)

❑ Connect the same *exceptionOccurred* event to the *transferToServiceHandler* property of the controller servlet and pass the *this* property of the AppSignOn servlet as a parameter (4)

❑ Connect the *normalResult* event of the validate connection to the *transferToServiceHandler* property of the controller servlet and pass the *this* property of the AppEntry servlet as a parameter (5).

❑ Save the controller servlet (Figure 54).

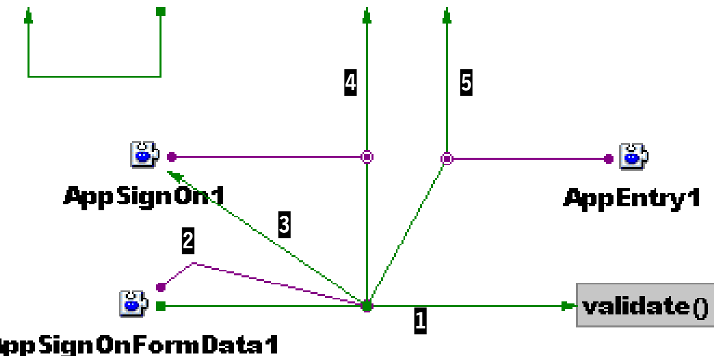


Figure 54. Controller Servlet

❑ Open the sign on servlet and change the action of the form to point to the new controller servlet.

❑ Save the sign on servlet and test the application flow.

Disable Caching of Generated HTML

The HTML output of a servlet is cached by Web browsers and displayed when the servlet is invoked again. In many cases this can lead to wrong results, and the user must force the reload and repost of the data to the servlet.

Caching of HTML can be disabled through the HTML header. To disable caching of the generated HTML of the post servlet (see “Servlet Post Processing” on page 58), change the doPost method as shown in Figure 55.

```
   public void doPost (HttpServletRequest req, HttpServletResponse res)
               throws ServletException, IOException {
      String str = req.getParameterValues("name")[0];
      users.addElement(str);
      res.setContentType("text/html");
      res.setHeader("Pragma","no-cache");
      res.setHeader("Cache-Control","no-cache");
      res.setDateHeader("Expires",0);
      ServletOutputStream so = res.getOutputStream();
      so.println("<HTML><BODY>Thank you ");
      so.println( str );
      so.println("</BODY></HTML>");
      so.close();
   }
```

Figure 55. Post Servlet with Caching Disabled

In a visual servlet, you first have to access the response object so that you can issue the setHeader and setDateHeader methods. The best place to disable caching in a visual servlet is the initialize method (Figure 56).

```
private void initialize() {
   // user code begin {1}
   // user code end
   initConnections();
   ...
   // user code begin {2}
   javax.servlet.http.HttpServletResponse resp = getResponse();  // <====
   resp.setContentType("text/html");
   resp.setHeader("Pragma","no-cache");          // no caching
   resp.setHeader("Cache-Control","no-cache");
   resp.setDateHeader("Expires",0);              // expires immediately
   // user code end
}
```

Figure 56. Disabling Caching in a Visual Servlet

Servlet with JDBC

It is very easy to access a relational database by using JDBC from a servlet. You do not have to worry about the Java security architecture because the database access is on the server. A servlet can use JDBC as a local application. In this section we describe how to use data access beans in a servlet.

Data access beans are capable of retrieving, updating, deleting, and inserting rows in a relational table. We combine the simplicity of data access beans with the HTML result table bean of the Servlet Builder. The Select bean (of the data access beans) implements the table model, and that is exactly what the HTML result table expects.

We describe here only a very simple scenario. See Chapter 9, “ATM Application Using Servlets” for more complex database access from servlets.

For our example we want to display the names and a few other attributes of the employees of a specified department from the DB sample database:

- ❑ Create a new servlet named EmpOfDept (Figure 57).
- ❑ Place a form on the page with a text, an entry field (deptnum), a push button (RetrieveButton), and an HTML result table in the form.
- ❑ Save the servlet to generate the EmpOfDeptFormData bean.
- ❑ Add the EmpOfDeptFormData bean and a Select bean (database palette).
- ❑ Open the Select bean and specify the connection and SQL statement. Refer to “Creating the Sample Panel and the Select Bean” on page 21 for detailed instructions. Use the same *SampleDB* data access class, the same connection, and generate a new SQL statement named *getEmpOfDept*:

```
SELECT <s>.EMPLOYEE.EMPNO, <s>.EMPLOYEE.FIRSTNME, <s>.EMPLOYEE.LASTNAME,
   <s>.EMPLOYEE.PHONENO, <s>.EMPLOYEE.JOB, <s>.EMPLOYEE.SEX
   FROM <s>.EMPLOYEE WHERE ( ( <s>.EMPLOYEE.WORKDEPT = :DEPT ) )
```

- ❑ Connect the *deptnumString* (of FormData) to the *string* of the deptnum entry field to redisplay the value that was entered (❶).
- ❑ Connect the *retrieveButtonPressed* event (of FormData) to the *Parm_DEPT_String* property (of Select) to set the host variable in the SQL statement (❷). Pass the *deptnumString* property as a parameter.
- ❑ Connect the *retrieveButtonPressed* event to the *execute* method (of Select) to run the SQL statement before generating the HTML output (❸).
- ❑ Connect the *this* property of the Select bean to the *tableModel* property of the HTML result table (❹). The Select bean and the HTML result table are compatible with the table model of Swing.
- ❑ Save the servlet and test.

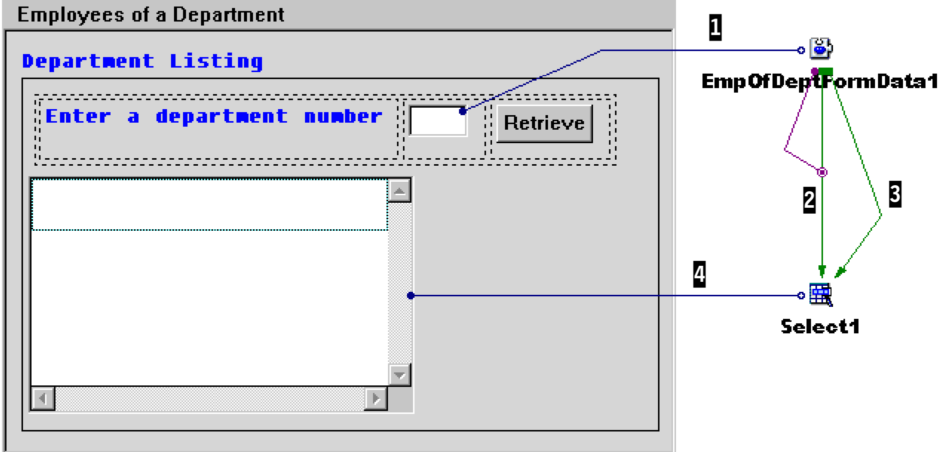


Figure 57. Servlet with Data Access Bean

The Select bean, in conjunction with the HTML result table, makes it quite easy to display SQL data in a servlet. The only visible problem is the table headings with their long names, such as USERID.EMPLOYEE.EMPNUM.

As long as you only retrieve data with the Select bean you can easily change the headings in the method generated for the SQL statement. Open the getEmpOfDept method of the SampleDB data access class and change the code in the addColumn specifications:

```
public static StatementMetaData getEmpOfDept() throws Throwable {
  String name = "itso.entbk2.sample.databean.SampleDB.getEmpOfDept";
  String statement = "SELECT .....................)";
  StatementMetaData aSpec = null;
  try{
      aSpec = new StatementMetaData();
      aSpec.setName(name);
       aSpec.setSQL(statement);
      aSpec.addTable("USERID.EMPLOYEE");
      aSpec.addColumn("Number", 1,1);
      aSpec.addColumn("Firstname", 12,12);
      aSpec.addColumn("Lastname", 12,12);
      aSpec.addColumn("Phone", 1,1);
      aSpec.addColumn("Job", 1,1);
      aSpec.addColumn("Sex", 1,1);
      aSpec.addParameter("DEPT", 1, 1);
   } catch(java.lang.Throwable e){throw e;}
   return aSpec;
}
```

With the corrected table headings the output in the browser is quite nice (Figure 58).

Department Listing

Enter a department number A00 Retrieve

Number	Firstname	Lastname	Phone	Job	Sex
000010	CHRISTINE	HAAS	3978	PRES	F
000110	VINCENZO	LUCCHESSI	3490	SALESREP	M
000120	SEAN	O'CONNELL	2167	CLERK	M

Figure 58. Servlet with Data Access Bean Browser Result

4 CICS Access with the CICS Connector

VisualAge for Java Enterprise provides the Common Connector Framework, the CICS Connector, the Enterprise Access Builder Library, and the Java Record Library as building stones for accessing CICS transactions from Java applications and servlets.

The Common Connector Framework provides a consistent means of interacting with enterprise resources, such as CICS transactions, from any Java execution environment. Enterprise resource management products have developed IBM e-business Connectors, or prebuilt beans, that use the Common Connector Framework.

The CICS Connector is such an IBM e-business Connector and is provided with VisualAge for Java Enterprise Version 2.

In this chapter we take a look at the CICS Connector and the supporting frameworks. We examine what is needed to implement an interaction where data is sent to a CICS application program, which in turn responds by sending output data.

© Copyright IBM Corp. 1998

4.1 The Enterprise Access Builder

The Enterprise Access Builder consists of frameworks and tools that allow Java applications to access existing host applications and data. An interaction with a back-end system often involves sending input data to a host application, which would respond with some output data. Such an interaction is often called a *transaction* in back-end systems. We reserve the term transaction, however, for a more global meaning, namely, a series of interactions with back-end systems that are committed as a whole.

In the Enterprise Access Builder, an interaction is called a *command*. Commands dictate the data that gets passed to and from the back-end system in a single interaction. *Records* are used as input and output for the commands. Records define the layout of the input and output data within the back-end system.

More complex interactions, consisting of a sequence of interactions, are represented by *navigators*. A navigator encapsulates a sequence of commands. A *business object* is an object representing an entity in the application space and contains a number of Java properties. The relationship between commands (or navigators) and business objects are defined through objects called *mappers*. Mapper objects are classes that map record properties within commands and navigators to business objects and move the values of these properties from business objects to a record before the host interaction, and from records to business objects after the interaction.

The Enterprise Access Builder provides the Common Connector Framework that is consistent across different host applications and facilitates communication between a client and host program. For Version 2 of VisualAge Java, the Enterprise Access Builder supports CICS and Encina DELight transactions. In this book we cover only the CICS Connector.

4.2 Connectors

The Common Connector Framework provides a consistent means of interacting with enterprise resources from any Java execution environment.

The Common Connector Framework can be used to:

- Extend existing enterprise data and applications to any Java execution environment
- Reduce the learning curve required to access multiple back-end resources

- Integrate coordination and management of enterprise resources at run time
- Improve performance by sharing and reusing client/server connections.

The Common Connector Framework consists of interface or class definitions that provide the framework for each connector. The three main interfaces are:

- The Communication interface, which defines the common communication interface for all connectors. It allows you to connect, disconnect, and pass data to a host system.
- The ConnectionSpec interface, which defines the common connection specification interface for all connectors. It allows you to create a new communication configured with properties specific to the host system.
- The InteractionSpec interface, which defines the host-specific properties for a single interaction with that host resource manager.

These interfaces are implemented in the CICS Connector.

4.3 The CICS Connector

The CICS Connector provides a powerful way of accessing CICS servers from the Internet. An applet or servlet uses the CICS Connector classes to access the CICS Transaction Gateway through its own TCP/IP-based protocol, or through HTTP, HTTPS, or SSL protocols.

The CICS Transaction Gateway, a Java application residing on the Web server, uses the CICS Universal Client to communicate with the CICS server.

CICS Connector Installation

VisualAge for Java Enterprise Version 2 is packaged with the following:

- CICS Transaction Gateway Version 3, which contains the CICS Universal Client Version 3 (the CICS Client for Windows NT has been renamed CICS Universal Client for Windows NT)
- CICS Connector classes
- HTML documentation on the Gateway and the Universal Client

Features To Be Added to the Workspace

Once the CICS Connector has been installed, the following features should be added to your workspace:

- ❑ IBM Common Connector Framework
- ❑ IBM Enterprise Access Builder Library
- ❑ CICS Connector
- ❑ IBM Java Record Library

CICS Connector Classes

The CICS Connector classes are supplied in packages in the *Connector CICS* project.

In keeping with the Common Connector Framework, the CICS Connector classes include a CICS ConnectionSpec, an external call interface (ECI) InteractionSpec, and an external presentation interface (EPI) InteractionSpec. These classes are found in package com.ibm.connector.cics.

The value of the URL property of the CICSConnectionSpec is set to the IP address or host name of the CICS Transaction Gateway. The value of the CICSServer property of the CICSConnectionSpec is set to the name of the CICS server as specified in the CICS client initialization file (CICSCLI.INI).

The CICSInteractionSpec is used to indicate which program has to be run at the CICS server. It describes the call to be made at the CICS server.

The ECI allows a non-CICS application to call a CICS program in a CICS server. The CICS program cannot perform terminal I/O but can access and update all other CICS resources. Data is exchanged by means of a communication area (COMMAREA).

The EPI allows a non-CICS application program to be viewed as a 3270 terminal by the CICS server system to which it is connected. For more information about the ECI and EPI, see “CICS Universal Clients” on page 95.

Compatibility with Previous Gateway and Client Software

Before we installed the CICS Transaction Gateway and the CICS Universal Client, we worked with the CICS Gateway for Java and CICS Client for NT Version 2 in our test environment. These had successfully been tested with the CICS Access Builder supplied with VisualAge for Java Enterprise Version 1.

We were unable to run the Adder ECI CICS Connector sample provided with VisualAge for Java Enterprise Version 2 with this configuration. Only after we had installed the CICS Transaction Gateway and the Universal Client were we able to successfully run the Adder sample in package com.ibm.ivj.eab.sample.eci.adder.

There are therefore potential migration issues if the CICS Connector is used. Please note the difference in name between the CICS Gateway for Java and the CICS Transaction Gateway. This change affects package names. The packages relevant to the CICS Gateway for Java start with ibm.cics.jgate, for example, ibm.cics.jgate.client. The packages relevant to the CICS Transaction Gateway start with com.ibm.ctg, for example, com.ibm.ctg.client.

When using the CICS Connector make sure that the com.ibm.ctg.* packages are added to your workspace. These are found in the Connector CICS project.

When planning to migrate, consult the section on migration issues in the "Planning before Installation" chapter of the *CICS Transaction Gateway Administration* manual.

4.4 CICS Universal Clients

This discussion is taken from the Overview chapter of the *CICS Universal Clients Administration* manual. For more information about the CICS Universal Clients, refer to that manual, which comes with the CICS Connector.

CICS Universal Clients allow users to access transactions and programs on the entire family of CICS application servers.

The CICS Universal Clients family comprises:

- IBM CICS Universal Client for OS/2
- IBM CICS Universal Client for Windows 98
- IBM CICS Universal Client for Windows NT
- IBM CICS Universal Client for AIX
- IBM CICS Universal Client for Solaris

Communcation Protocols

CICS Universal Clients can communicate through the following protocols:

- Network Basic Input/Output System (NetBIOS)
- Transmission Control Protocol/Internet Protocol (TCP/IP)
- Advanced Program-to-Program Communication (APPC)

CICS Universal Client for OS/2, Windows 98, and Windows NT can also communicate with CICS for MVS/ESA through the IBM TCP62 protocol mapper, which allows APPC applications to communicate over a TCP/IP network.

CICS Universal Clients can communicate with multiple CICS servers. The client initialization file determines the parameters for client operation and identifies the associated servers and protocols used for communication.

Client Customization

Client customization can be done through modifying the contents of three .ini files. Default .ini files are supplied with the CICS Universal Clients in the Client \BIN directory:

- CICSCLI.INI - the client initialization file
- CICSKEY.INI - the keyboard mapping file
- CICSCOL.INI - the color mapping file

We recommend that you use different names if you create your own customized versions of these files. You should reference your customized files through the environment variables shown in Table 9.

Table 9. Client Environment Variables

File	Environment Variable
Client initialization file	CICSCLI
Keyboard mapping file	CICSKEY
Color mapping file	CICSCOL

Client Initialization File

There is a relationship between the client initialization file and a property value of one of the beans used by the CICS Connector. It is for this reason that we discuss this file here.

The client initialization file contains configuration information used to inform the client of the servers it can connect to, and the necessary communication protocols that are used to access the servers.

The client initialization file is structured into sections, each containing a set of parameters specific to that section. The sections are:

- ❑ At most one Client section.
- ❑ One or more Server sections. The first server definition is used as the default server for the client.
- ❑ One or more Driver sections. There is one Driver section for each unique protocol referenced in the Server sections.

The syntax is shown in Figure 59 and explained in the *CICS Universal Client for Windows Administration* manual.

```
[Client=* | applid
         [MaxBufferSize=nn]
         [TerminalExit=name]
         [TraceFile=filename]
         [LogFile=filename]
         [ApiTrace=Y|N]
         [MaxServers=nnn]
         [MaxRequests=nnnnn]
         [PrintCommand=command]
         [PrintFile=filename]
         [DumpMemSize=nn]
         [DumpFile=filename]
         [CPName=name]
         [CPIPAddressMask=mask]
         [DomainNameSuffix=suffix]
         [DceCellDirectory=Y|N]
         [EnablePopups=Y|N]
       ]
       Server=Servername
         NetName=Applid | LUName | HostName | IPAddress
         Protocol=ProtocolName
         [Description=desc]
         [UpperCaseSecurity=Y|N]
         [InitialTransid=transid]
         [ModelTerm=name]
         [Port=nnnn]
         [Adaptor=0|1|2|3]
         [LocalLUName=LUName]
         [Modename=modename]
         [LUAliasNames=Y|N]
         [SnaSessionLimit=nnn]
         [SnaMaxRUSize=nnnn]
         [SnaPacingSize=nnn]
         [LUIPAddressmask=suffix]
         [EndPoint=portnumber]
         [TcpKeepAlive=Y|N]

       Driver=ProtocolName
         Drivername=drivername
```

Figure 59. Client Initialization File Syntax

Of special interest here is the value of the *Server* keyword. Figure 60 is an extract from the customized file used during the development of the samples in this book.

```
Server = ATMTCP                 ; name for the server
   Description = TCP/IP Server  ; description for the server
   Protocol = TCPIP             ; Protocol is TCP/IP
   NetName = bosporus           ; server's  TCP/IP address
   Port = 0                     ; Use the default TCP/IP CICS port
```

Figure 60. Extract for Customized Client Initialization File

The server name of ATMTCP is referenced in the ServerName property of the CICSConnectionSpec bean that is discussed in "CICS Connector Classes" on page 94 and in "Constructing a Command" on page 112.

Client Functions

In this section we summarize the functions provided by the CICS Universal Client.

3270 Terminal Emulation

CICS 3270 emulation enables a client workstation to function as a 3270 display or printer for CICS applications, without needing a separate 3270 emulator product.

External Call Interface

The ECI enables a non-CICS client application to call a CICS program synchronously or asynchronously, as a subroutine. The client application communicates with the server CICS program through the COMMAREA. The COMMAREA is passed to the CICS server on the call, and the CICS program typically populates it with data accessed from files or databases, which is then returned to the client for manipulation or display.

The ECI is the recommended interface for developing new client/server applications. The ECI call structure easily separates the presentation logic (usually in the client) from the business logic in the CICS application. For example, the ECI can be used with mainframe CICS applications that are already separated into business logic (in the application-owning region) and presentation logic (in the terminal-owning region). The business logic can remain unaltered when the presentation logic is developed.

The examples discussed in this book use the ECI.

External Presentation Interface

The EPI enables client applications to start and converse with a legacy 3270 CICS application running on the CICS server. The CICS application sends and receives 3270 data streams (for example, a CICS basic mapping support (BMS) transaction) to and from the client application as though it were conversing with a 3270 terminal. The client application captures these data streams and, typically, displays them with a non-3270 presentation product, such as GUI or multimedia software.

The EPI is therefore a method of enhancing an existing CICS application by adding a graphical or other modern interface. The CICS application itself does not need to be altered.

External Security Interface

The ESI enables client applications to verify that a password matches the password recorded by an external security manager for a specified user ID.

3270 Client Printer Support

With CICS 3270 client printer support a printer terminal can be defined on the client workstation. Thus CICS applications running on the server can direct output to the client-attached printer.

CICS 3270 client printer support uses CICS 3270 emulation functions.

Client Control

CICS Universal Clients provide commands or icons to:

- Start or stop the client process
- Turn the client trace on or off
- Specify the client initialization file to be used
- Set up security by specifying user IDs and passwords for a CICS server
- List connected servers
- Enable and disable the display of messages
- Control terminal emulation
- Control client printer operation

CICS Telnet Support

CICS Universal Clients provides a command, CICSTELD, to start a CICS Telnet daemon. You can use any Telnet 3270 client to access a CICS server through the CICS Telnet daemon.

4.5 CICS Transaction Gateway

This discussion is taken from the Overview chapter of the *CICS Transaction Gateway Administration Version 3* manual. For more information on the CICS Transaction Gateway, refer to that manual, which comes with the CICS Connector. Additional information is also available in the redbook *Revealed! CICS Transaction Gateway with More CICS Clients Unmasked*, SG24-5277.

The IBM CICS Transaction Gateway provides secure, easy access from Web browsers and network computers to business-critical applications running on a CICS Transaction Server or TXSeries server using standard Internet protocols.

The CICS Transaction Gateway is provided for the OS/2, Windows NT, AIX, and Solaris platforms. The CICS Transaction Gateway is also provided for Windows 95 and 98, but on these platforms it can only be used for development purposes and not for production.

Figure 61 shows how a Web client can access CICS programs through the CICS Transaction Gateway.

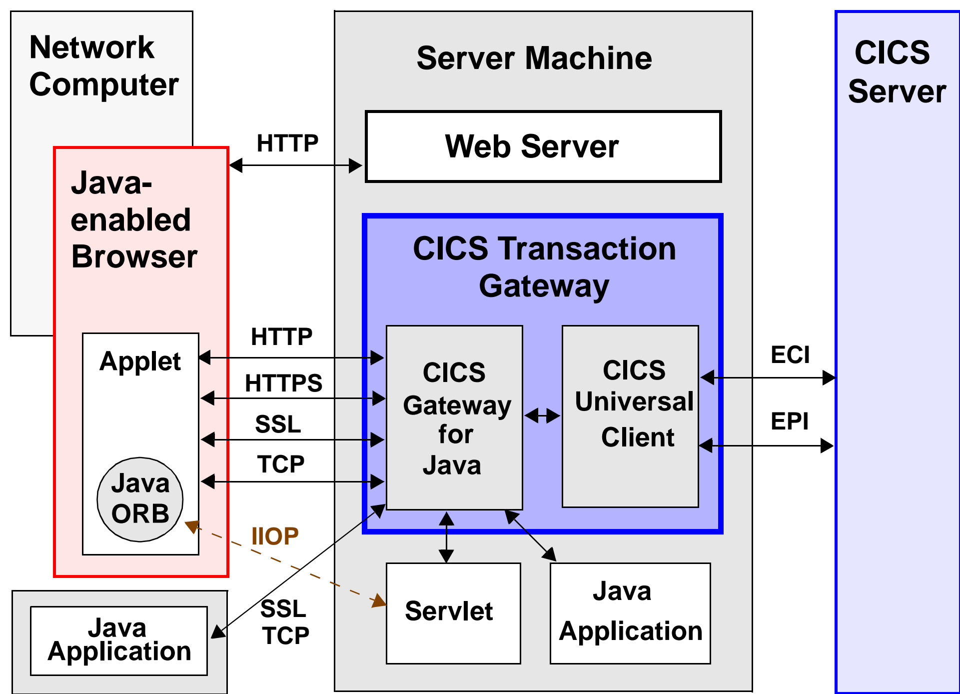


Figure 61. CICS Transaction Gateway

What the CICS Transaction Gateway Provides

The CICS Transaction Gateway provides:

- A Java gateway application that is usually resident (for security reasons) on a Web server machine. It communicates with CICS applications running in CICS servers through the ECI or EPI provided by the CICS Universal Clients. This Java application was previously available in the IBM CICS Gateway for Java.
- A CICS Universal Client that provides the ECI and EPI, as well as the terminal emulation function. See "CICS Universal Clients" on page 95.
- A CICS Java class library that includes classes that provide an API and are used to communicate between the Java gateway application and a Java application (applet or servlet). The JavaGateway class is used to establish communication with the gateway process and uses Java's sockets protocol. The ECIRequest class is used to specify the ECI calls that flow to the gateway. The EPIRequest class is used to specify the EPI calls that flow to the gateway. These Java classes were previously available in the IBM CICS Gateway for Java.
- A terminal servlet that enables you to use a Web browser as an emulator for a 3270 CICS application running on any CICS server. The terminal servlet can be used with a Web server or a servlet engine that provides support equivalent to JSDK 1.1 or later. This is an enhanced version of the function that was provided by the CICS Internet gateway in IBM CICS Clients Version 2.
- A set of Java EPI beans for creating Java front ends for existing CICS 3270 applications, without any programming.

The CICS Transaction Gateway can concurrently manage many communication links to connected Web browsers and can control asynchronous conversations to multiple CICS server systems. The multithreaded architecture of the CICS Transaction Gateway enables a single Gateway to support multiple concurrently connected users.

How the CICS Transaction Gateway Accesses CICS

Access is discussed for both applets and servlets. This discussion is taken from the Version 3.0 *CICS Transaction Gateway Administration* manual.

Java Applet Access

For Java applets, access is achieved as follows:

- The Web browser or network computer requests an HTML page from the Web server using HTTP.

- The Web server returns the HTML page containing a tag identifying a Java applet.
- The browser starts requesting relevant Java classes from the Web server.
- The Web server returns Java classes, including CICS Transaction Gateway classes as requested.
- As classes are returned, the Java applet starts.
- For ECI, the Java applet creates an ECIRequest object containing ECI calls and sends it to the gateway using the JavaGateway.flow method. The ECIRequest supports most of the ECI calls.
- For EPI, the Java applet creates an EPIRequest object containing EPI calls and sends it to the gateway using the JavaGateway.flow method. The EPIRequest supports most of the EPI calls.
- The Gateway receives the request, unpacks it, and makes corresponding ECI or EPI calls to the CICS Universal Client.
- The CICS Universal Client passes the calls to the intended CICS server.
- The CICS server processes the call, including verification of the user ID and password if required, and passes control and user data to the CICS application program.
- When it has finished processing, the CICS application returns control and data back to CICS, which passes it back to the gateway.
- The gateway packs these results and returns them to the Java applet running on the Web browser.

Java Servlet Access

For Java servlets, access is achieved as follows:

- The Web server loads and initializes the servlet, either when the Web server starts or the first time a request is made to the servlet. The servlet may at this point create a JavaGateway object to connect to the CICS Transaction Gateway.
- When the servlet is invoked by an appropriate HTTP request, the Web server calls its Service method with details of the request.
- For ECI, the Java servlet creates an ECIRequest object containing ECI calls and sends it to the gateway using the JavaGateway.flow method. The ECIRequest supports most of the ECI calls.
- For EPI, the Java servlet creates an EPIRequest object containing EPI calls and sends it to the gateway using the JavaGateway.flow method. The EPIRequest supports most of the EPI calls.

- The Gateway receives the request, unpacks it, and makes corresponding ECI or EPI calls to the CICS Universal Client.
- The CICS Universal Client passes the calls to the intended CICS server.
- The CICS server processes the call, including verification of the user ID and password if required, and passes control and user data to the CICS application program.
- When it has finished processing, the CICS application returns control and data back to CICS, which passes it back to the gateway.
- The gateway packs these results and returns them to the Java servlet.
- When the servlet receives the results of the requests it has made to the CICS Transaction Gateway, it generates an HTTP response to be returned to the Web browser.

4.6 A Discussion Review

Thus far the discussion has focused on the CICS Connector as an implementation of the Common Connector Framework, the CICS Transaction Gateway, and the Universal CICS Clients, which provide the means for interacting from a Web client to a CICS server. Typically an interaction would consist of an input data stream that is passed to a program on the CICS server. When complete, this program would typically pass response data back.

How such data is formatted and how the flow is defined and controlled still needs to be discussed.

4.7 Accessing Enterprise Data

In this section we discuss at a high level the tasks required to get an end-to-end flow between a Web client and a CICS server. We do not describe in detail how to carry out each task; rather we introduce the tasks conceptually. For a more detailed discussion of exactly how to implement the tasks, see Chapter 10, “ATM Application with the CICS Connector.”

Overview

In the *VisualAge for Java Enterprise Concepts* documentation the page entitled "Accessing Enterprise Data: Overview" lists a typical task flow for accessing enterprise data with the Enterprise Access Builder:

- ❑ Create a structure description.
- ❑ Generate a dynamic record type. Use the SmartGuide that corresponds to your middleware to generate a record type for your source. The dynamic record type represents the data structure on the host. It can be used as generated or modified in the record editor.
- ❑ If necessary edit the dynamic record type. This is probably not necessary for simple COMMAREA design.
- ❑ Generate a record bean or class. Use the Generate Records SmartGuide to generate a record bean or class based on the record type you define as input.
- ❑ Construct commands. Define connection and interaction specifications and use them in conjunction with record beans to create commands. Commands define the flow of interactions with the host program.
- ❑ Construct a navigator. Use your commands to construct a navigator that combines the commands you created and defines the overall flow of interactions with the host program.
- ❑ Create business objects. Create classes that inherit from the IBusinessObject class. You can create either managed or unmanaged business object classes.
- ❑ Map the business object to the record bean. Use the mapper to map the properties between the business object and the record. Then, add the mapper to the record beans in the command.
- ❑ Connect business objects.
- ❑ Export business objects and supporting classes. If you wish to use your business objects in another environment, for example, IBM Component Broker or IBM WebSphere, export your business objects and associated classes.
- ❑ Create a client program using your business objects. Use the VisualAge for Java Visual Composition Editor to construct a client program that contains your business objects.
- ❑ Test your client application. Test run your client program either as an applet or a main program.
- ❑ Deploy your client application.

Structure Description

The structure description pertains to some sort of source representing an input or output data stream such as a CICS COMMAREA. Typically a structure description would be found in the server program that is called. Throughout this discussion it is assumed that the server program is written in COBOL.

Records and the Java Record Framework

The bulk of enterprise data continues to reside in either a database or a record-oriented store. This data is processed by existing line-of-business applications such a CICS transactions. Most of these applications are written in programming languages such as COBOL.

The Java application developer needs to work with such record-oriented data. Most of today's record data has been written by non-Java applications. Consequently record access from Java must be able to handle data conversions required by cross-language access. In order to assist with this task the Java record framework is provided.

Java Record Framework

The Java record framework describes and converts record data. The framework is usage-context independent. It is used as a base for record-oriented file input and output, as well as for record-based message-passing schemes.

A record is a logical collection of application data elements. These data elements are related by an application-level semantic that is stored and retrieved as a unit. When a record is retrieved, individual data elements can be accessed directly, typically through a native language structure, such as COBOL, C, or PL/I.

Custom and Dynamic Records

Two separate mechanisms for record data access are provided as part of the Java record framework. The first is an implementation of a dynamic descriptor structure. The second mechanism is intended for optimized access to records whose format is known ahead of time and does not change. Such a record is referred to as a *custom* record.

In general, records can have fixed or variable lengths. The Java record framework allows the manipulation of fixed or variable length records. A dynamic record is defined as a collection of separate data fields with descriptive field level information being captured as part of a run-time access

structure. Variable length records, whose format is not known ahead of time, have to be dynamic records. A custom record is used for fixed formats. Instead of a dynamic record descriptor, the custom record makes direct references to fields based on their field offsets relative to the record.

Although quite different in their usage and implementation, both styles of records implement a common set of record handling interfaces. Consequently, either record style can be used in higher-level frameworks based on the record support.

Support for Different Record Structure Groupings

The Java record framework supports nested structures, arrays, overlaid fields, and field alignment and packing.

A record can be viewed as a series of nested subrecords, which involves building up a composite record type descriptor. The framework supports such an approach.

The Java record framework supports array fields with an arbitrary number of dimensions. Individual array elements can be accessed directly.

Existing record applications allow the definition of field overlays. The framework supports this by defining a union of data fields, all anchored to the same offset within the record bytes (subject to individual component subfield alignment). The length of an overlaid field is the length of its largest component subfield (again, including any subfield alignment).

Record structures typically support the concepts of alignment and packing. In compiled languages, these determine the tradeoff between speed of access to record elements and required storage for the data record. The framework provides an implementation of packing and alignment in the supplied classes.

Framework Tools

The main purpose of the Java record framework is to provide run-time support for accessing application record data. The framework also defines a set of mechanisms for record builder tools to use. The builder-type tools are provided to assist in the construction of record-oriented structures.

For a more detailed discussion of the Java record framework refer to the Java record framework documentation supplied with VisualAge for Java Enterprise.

Record Creation Overview

Record beans contain properties that represent the fields in the server application with which a Java client application interacts. Record beans are generated from dynamic record types.

Dynamic Record Types

A dynamic record type is a representation of the field content of a record in an application. A field can be another dynamic record type, an array, or a simple field. Dynamic record types are created from source code. Currently there is support for COBOL, basic mapping support (BMS), and message format service (MFS) source. You can use the SmartGuide corresponding to your source to parse a local copy of a source file and generate a dynamic record type.

Dynamic record types can be modified in the record editor. Once you have defined the fields of your record type, you can use the Generate Records SmartGuide to generate a record bean or class. Figure 62 shows the steps to create a record bean.

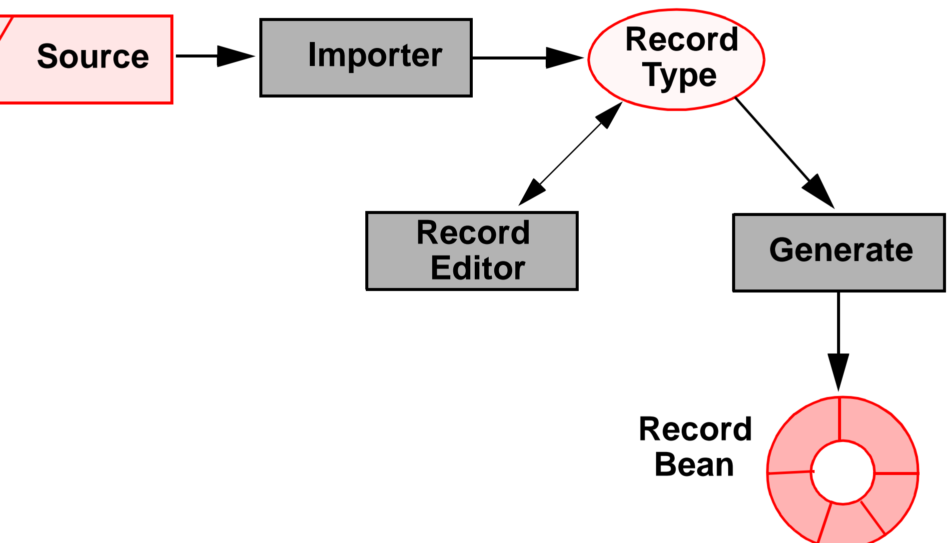


Figure 62. Record Bean Creation

SmartGuides for Dynamic Record Type Creation

To use the Java record framework you have to define and generate a record bean or class. Three record type SmartGuides are provided with VisualAge for Java Version 2: BMS, MFS, and COBOL. A record type is a representation

of the field contents of a record in a host application. Each SmartGuide parses a local copy of a source file and generates a record type. The SmartGuides can be accessed by selecting *Tools -> Records -> Create....Recordtype.*

A record type can also be hand-coded by creating a class that inherits from the DynamicRecord class.

Record types can be modified in the record editor.

Below we describe the COBOL Record Type SmartGuide, because it is used most often.

Cobol RecordType SmartGuide

To generate a record type from COBOL source, follow these steps:

- ❑ Download a local representation of your COBOL source file into the codepage of your current locale.
- ❑ In the Workbench, select a package to contain the generated record type.
- ❑ From the Selected menu, select *Tools -> Records -> Create COBOL RecordType* to open the COBOL RecordType SmartGuide (Figure 63).

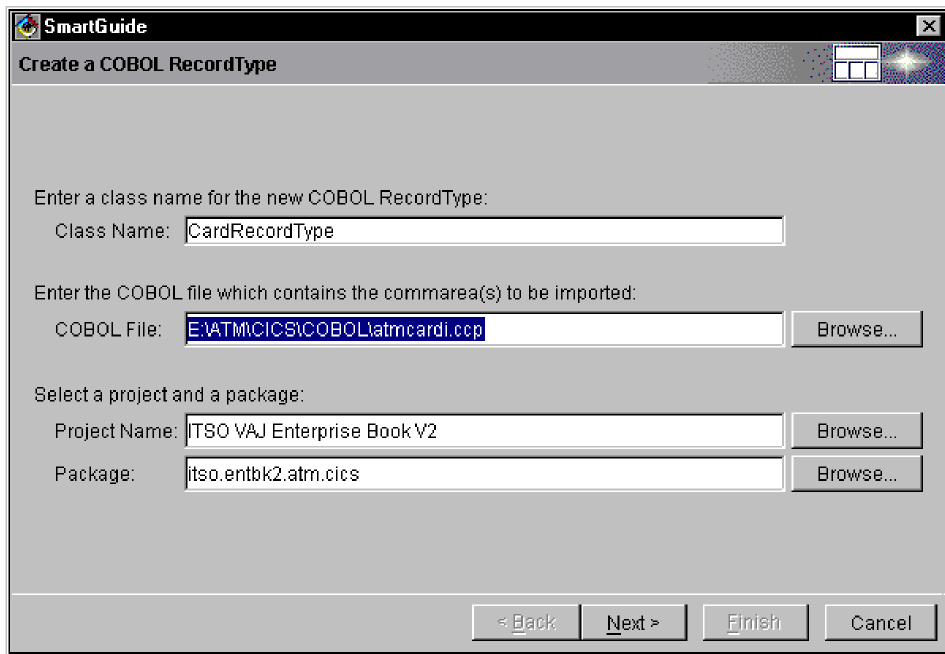


Figure 63. SmartGuide for COBOL Record Type

- ❑ In the Class Name field, specify a valid class name for the new COMMAREA RecordType.
- ❑ In the COBOL file field, specify the name of the COBOL program for which you want to generate a record type. You can also click on *Browse* to open a dialog in which you can select the COBOL file from a directory.
- ❑ Specify a project and package.
- ❑ Click on *Next*. The second page of the SmartGuide opens. From the Available list, select a level 1 COMMAREA from the COBOL source file you defined earlier. Click on > to add the COMMAREA to the Selected list.
- ❑ Click on *Finish*. A record type is generated into the package you selected.

Record Bean Generation

Once you have defined the fields of a record type, you can use the Generate Records SmartGuide to generate a record bean or class. To access this SmartGuide, select the record type you just created and select *Tools -> Record -> Generate Record*. The Generate Records SmartGuide opens (Figure 64).

Figure 64. Generate Records SmartGuide

In the SmartGuide you can choose direct or hierarchical field access. If the field names are not unique, hierarchical access to the field names forces the field names to be fully qualified. If the field names are unique, choose direct access.

You can also choose between dynamic records or custom records for the generated record beans or classes. Typically you would choose dynamic for variable records and custom when you know the exact record layout ahead of time.

Click on *Next* and specify character code set, encoding, and operating system options. The defaults are associated with MVS (Figure 65).

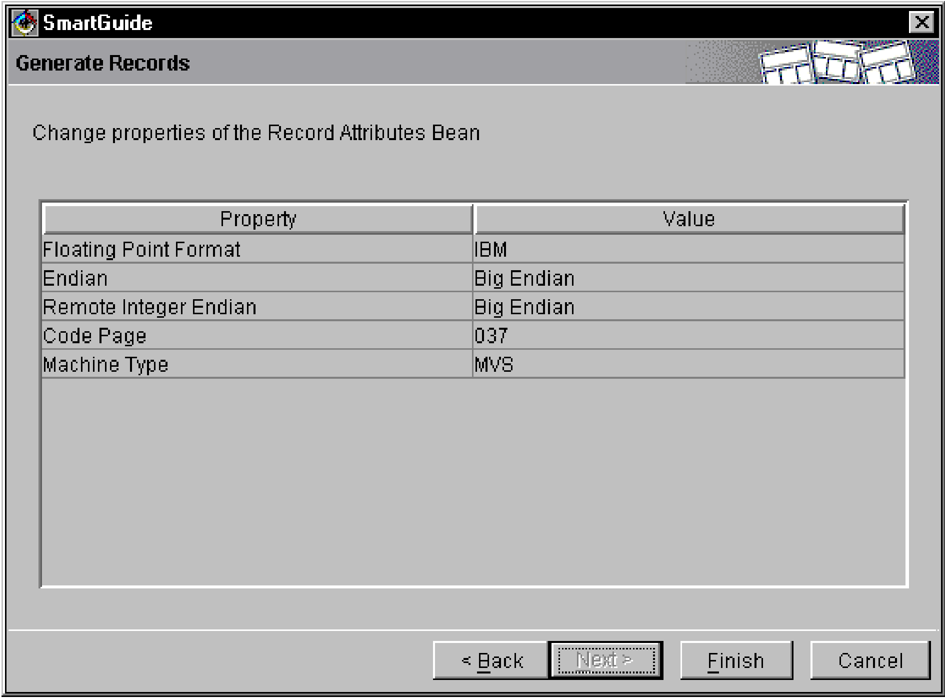


Figure 65. Generate Records SmartGuide: Changing Properties

The generated record bean or class is referenced in a command.

Commands

The interaction between Java applications and host-based systems involves input data being sent to the receiving application, followed by the application responding with output data. For example, when invoking a CICS program, fields within the input COMMAREA would be set, the CICS program would be invoked passing the input COMMAREA, and the CICS program would respond with the output COMMAREA. The Enterprise Access Builder allows for the specification of such an interaction through commands. A command within the Enterprise Access Builder consists of the input-interaction-output combination.

An Enterprise Access Builder command wraps a single interaction with a host system. On execution, an Enterprise Access Builder command:

- ❑ Takes its input data and sends the data to a host system, using a connector.
- ❑ Sets, as its output, the data returned by the host system.

Figure 66 shows the construction pattern of a command.

You need the following information when constructing a command:

- ❑ Input data

  The input data is required by the command to perform the execution. The input of a command is defined by a record bean.

- ❑ Output data

  The output data is the result of the command's execution. The output of a command is defined by one or more record beans.

- ❑ Connection information

  Connection information is specified in the ConnectionSpec and InteractionSpec beans. These beans are specific to the CICS Connector and implement the ConnectionSpec and InteractionSpec interfaces defined in the Common Connector Framework. These interfaces are discussed in "Connectors" on page 92.

  The ConnectionSpec bean is used to specify the connection to the CICS system. The InteractionSpec bean is used to specify the CICS program to be called.

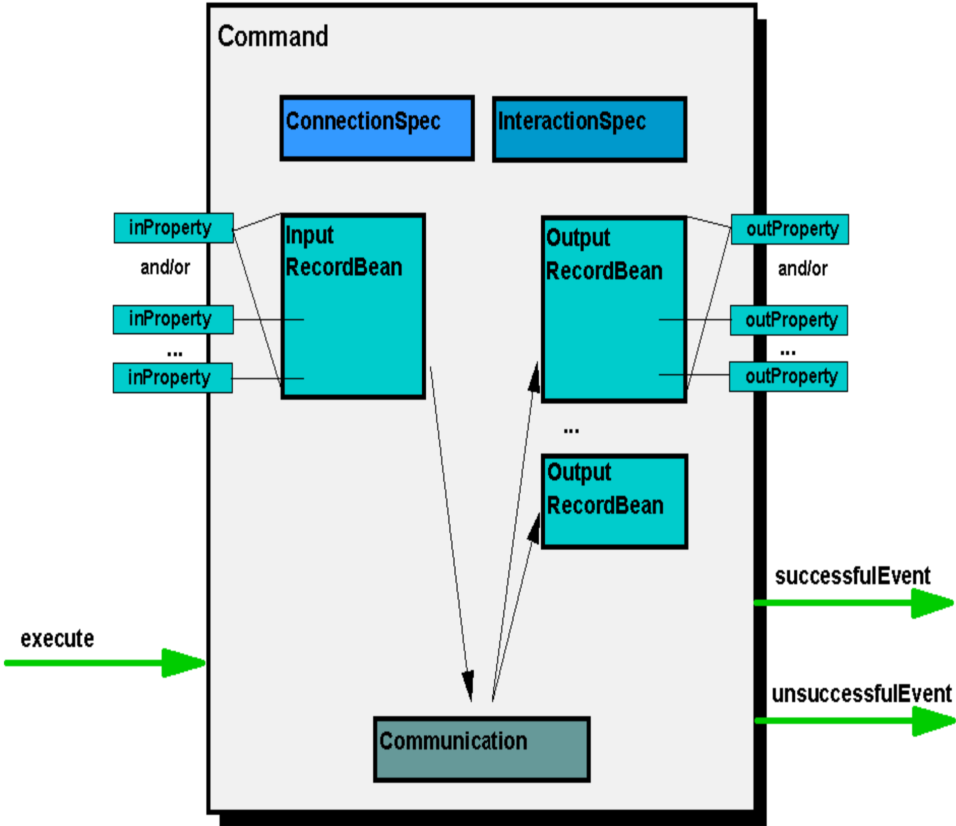


Figure 66. Command Construction

Constructing a Command

You construct a command with either the Visual Composition Editor or the Command Editor.

The Visual Composition Editor enables visual programming when composing a command. The Command Editor provides guidance for going through the steps of constructing a command and thus allows you to focus on the specific composition patterns of a command. Because both tools work on the same metadata, it is possible to switch from one tool to the other.

Using the Visual Composition Editor

Originally, when first exploring commands, we created a command using the Visual Composition Editor. All beans were associated with names of our own the details of the command. When using the Visual Composition Editor, you must adhere to a naming convention:

- ❑ Name the interaction specification bean *ceInteractionSpec.*
- ❑ Name the connection specification bean *ceConnectionSpec.*
- ❑ Name the input bean *ceInput* and the corresponding mapper bean *ceMapperCeInput.*
- ❑ If there is only one output that does not implement the IByteBuffer interface, name it *ceOutput* and the corresponding mapper, *ceMapperCeOutput.*
- ❑ If there is only one output bean implementing the IByteBuffer interface, name it *ceOutput1* and the corresponding mappe, *ceMapperCeOutput1*.
- ❑ If there are multiple output beans, name them *ceOutput1*, *ceOutput2*, ..., and the mappers, *ceMapperCeOutput1*, *ceMapperCeOutput2*, ...

We do not show how to use the Visual Composition Editor to construct a command. We found that the command editor allows for a more intuitive approach. However, this is an entirely personal view.

In addition, there are instances where the Visual Composition Editor has to be used. Once such case is when the input is defined as a bean variable. Refer to the documentation supplied with VisualAge for Java for details on how to construct a command with the Visual Composition Editor. The documentation on the sample ECI transactions is useful.

Using the Command Editor

To use the Command Editor, create a new class that is a subclass of com.ibm.ivj.eab.command.CommunicationCommand.

To launch the Command Editor select such a class and choose *Tools -> Command Editor*. Figure 67 shows the Command Editor dialog.

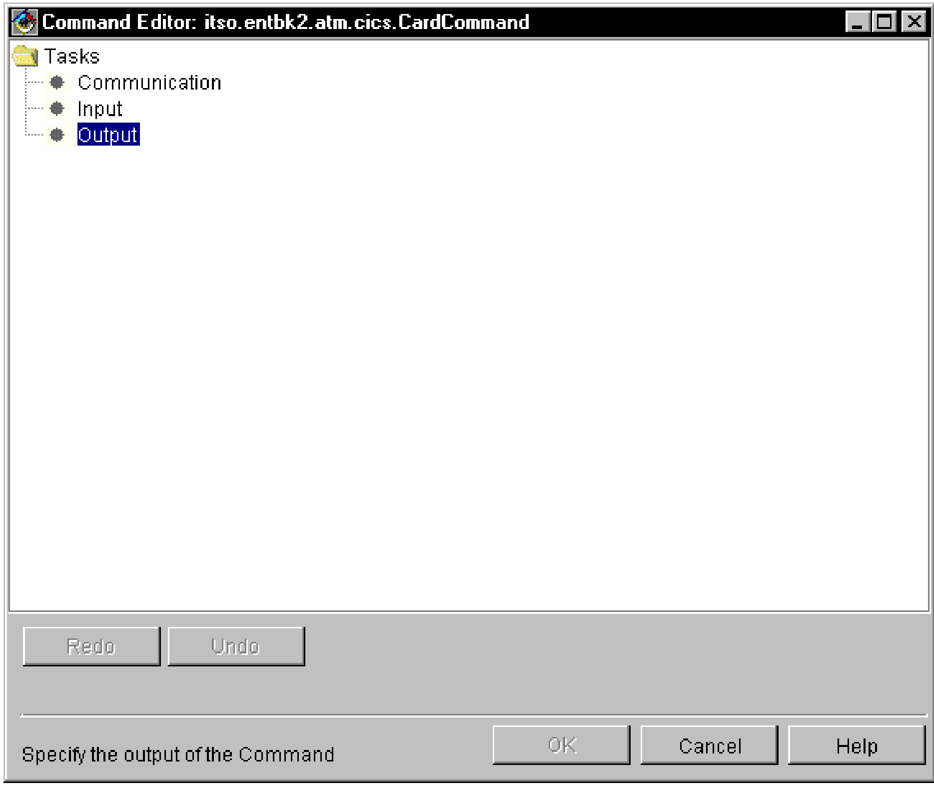


Figure 67. Command Editor: Initial View

Setting up Common Connector Framework Information

The ConnectionSpec and InteractionSpec properties must be specified. Follow these steps:

- Select *Communication* and right-click.
- A menu box pops up allowing the addition of ConnectionSpec and InteractionSpec.
- Select *Add ConnectionSpec*. A pop-up menu displays a choice of ConnectionSpecs. For CICS choose *CICSConnectionSpec*.
- Select *Add InteractionSpec*. A pop-up menu displays a choice of InteractionSpecs. For ECI choose *ECIInteractionSpec*.
- Right-click on the *ceConnectionSpec* and choose *Properties*. Specify the relevant CICSServer and URL and any other properties that must be changed. The CICSServer is associated with the server name specified in the client initialization file. See “Client Customization” on page 96. The

URL must point to the hostname or IP address where the CICS Transaction Gateway is running.

- ❑ Right-click on the *ceInteractionSpec* and choose *Properties*. Specify the relevant programName and any other pertinent properties. For example, specify *ATMCARDI* if this is the name of the CICS program to be called.

Specifying the Input Bean

- ❑ Select Input and right-click. A menu box pops up with options *Add IBytebuffer Bean* and *Add Input Bean*.
- ❑ Choose the type of input, for example, *Add Input Bean*. Specify the name of the record bean you want to use as input.

Specifying the Output Bean

- ❑ Select Output and right-click. A menu box pops up with options *Add IBytebuffer Bean* and *Add Output Bean*.
- ❑ Choose the type of output, for example, *Add Output Bean*. Specify the name of the record bean you want to use as output.

  The output can be associated with multiple beans.

Promoting Bean Features

- ❑ Right-click on the relevant bean. A menu box pops up with options *Property*, *Promote Bean Features*, *Delete*, and *Add Mapper*.
- ❑ Promote those bean features that need to be visible. (These are the features that you might want to connect later in the Visual Composition Editor.)

Adding Mappers

- ❑ To add a mapper, select *Add Mapper* and specify the name of the relevant mapper. The mapper class must be generated before doing this step. Mappers are discussed in “Mappers” on page 119. Essentially they allow for the mapping of a record to a business object.

Figure 68 shows a view of the Command Editor after the communication, input, and output beans have been added.

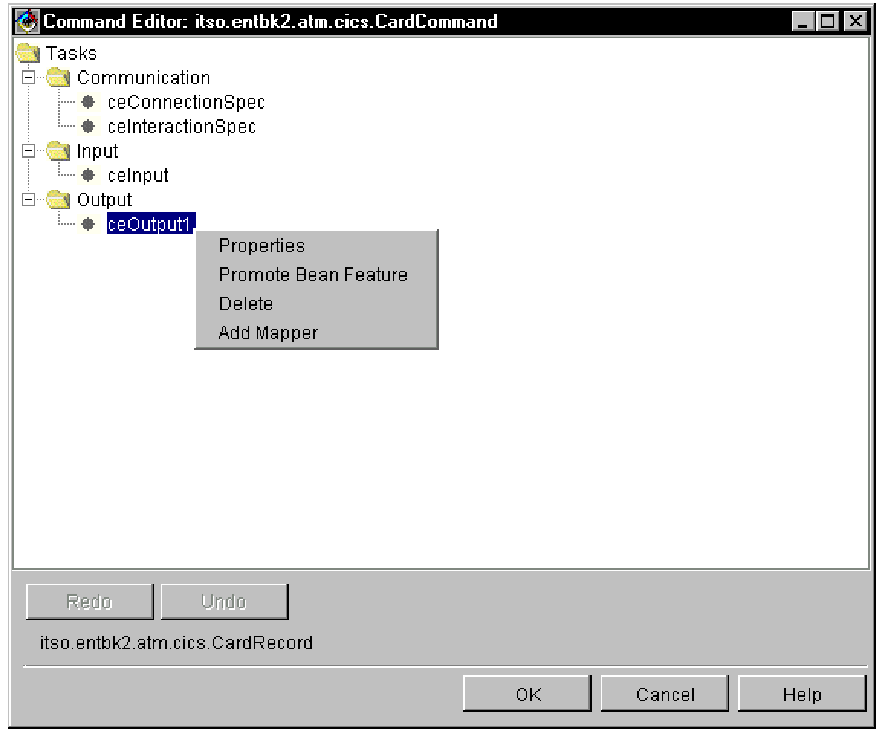


Figure 68. Command Editor with Communication, Input, and Output Beans

You can view a command in the Visual Composition Editor (Figure 69).

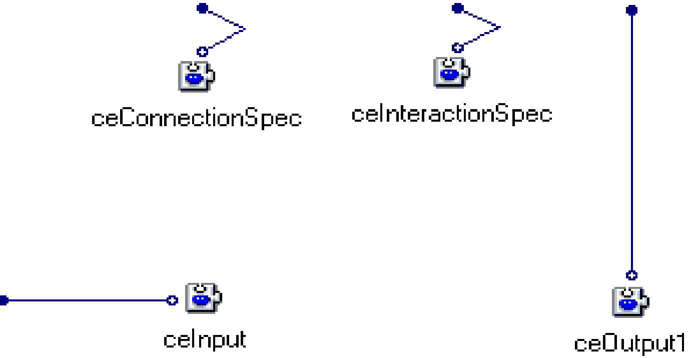


Figure 69. Visual Composition Editor View of a Command

Navigators

A navigator is a sequence of commands strung together to form a more complex interaction with the host system. Each command in the navigator is executed in the order specified. A navigator can be constructed in the Visual Composition Editor.

Connection information can be specified at the navigation level. This overrides the connection information supplied with the individual commands. Navigators can be strung together to form higher-level navigations.

Figure 70 shows that a navigator is made up of a set of commands.

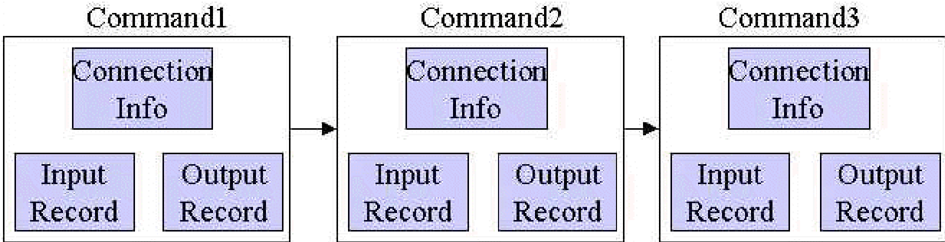


Figure 70. Navigator

Constructing a Navigator

This discussion is not very detailed as we designed only a few navigators. A navigator is constructed with the Visual Composition Editor as a subclass of com.ibm.ivj.eab.command.CommunicationNavigator.

To construct a Navigator:

- Set up a ConnectionSpec bean. This is optional. If not specified, the ConnectionSpec of the commands is used.
- Select the command bean for the first command in the sequence.
- Trigger this command by connecting the *internal*ExecutionStarting event of the navigator to the *execute* method of this first command.
- To handle unsuccessful execution, connect the *executionUnsuccessful* event of the command to the *returnExecutionUnsuccessful* method of the navigator.
- Select the command bean of the next command in the sequence.
- Invoke the next command by connecting the *executionSuccessful* event of the previous command to the *execute* method of the next command.

- Connect the *executionUnsuccessful* event of the next command to the *returnExecutionUnsuccessful* method of the navigator.
- Add more commands and trigger their execution from the previous command.
- For the last command, connect the *executionSuccessful* event to the *returnExecutionSuccessful* method of the navigator.

Figure 71 shows the construction of a navigator from the sample ATM application.

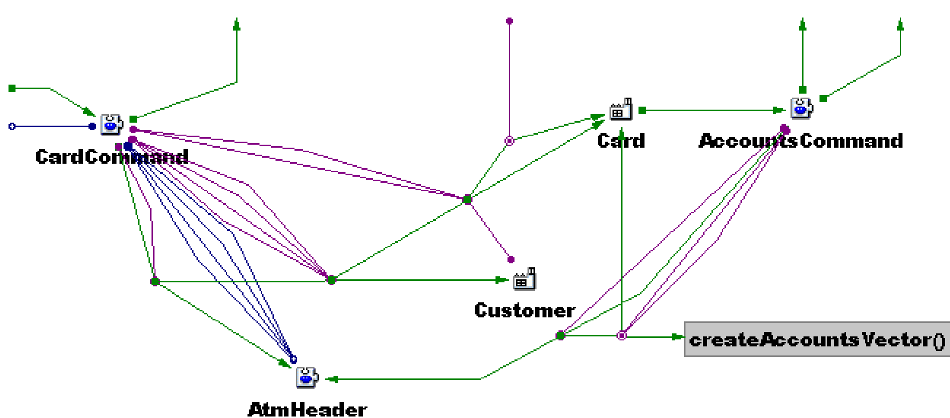


Figure 71. Construction of a Navigator

A navigator may contain only one command. Even in such cases a navigator can be very helpful because of the execute method and the events that it provides. Those facilities make visual construction of command processing quite simple.

For examples of navigators, see the following packages:

```
com.ibm.ivj.eab.sample.eci.mapper
com.ibm.ibj.eab.sample.eci.navigator
```

Business Objects

A business object can be used to map many different host interactions with many different host resources.

A business object is an object that is a subclass of either the BusinessObject class or the BusinessObjectKey class. A business object can contain any number of Java properties. There are two types of business objects:

- ❑ Managed business objects
- ❑ Unmanaged business objects

For more information about business objects refer to the online documentation about business objects supplied with the Visual Age for Java product.

In the ATM application, business objects have not been used. Instead records have been mapped to the beans associated with the ATM business model in the itso.entbk2.atm.model package.

Mappers

To exchange data between a business object and a command or navigator, the Enterprise Access Builder uses classes called mappers.

Generally, the documentation supplied with VisualAge for Java associates mappers with business objects. We, however, have successfully mapped to classes whose superclass is java.lang.Object. We did this because we did not want to reinvent the classes associated with the ATM application. This probably does not give the portability offered by business objects, but at least the mapping can be done. Although the ensuing discussion also refers to business objects, bear in mind that we did not map to business objects, that is, classes that implement the BusinessObject interface.

A mapper maps the properties of a record to or from the properties of a business object. Thus the input record of a command can have data from a business object automatically set within it before the command execution. A mapper also allows the properties of an output record of a command to automatically set the properties of a business object after the command is executed.

The Enterprise Access Builder supplies a mapper builder, which enables you to map a record to one or more business objects. The mapper builder generates a mapper bean that can be included in a command. In the Command Editor, the *Add Mapper* option is used to specify a mapper for the command.

The Mapper Builder

The record bean is used in the mapper builder as an input bean. Multiple business objects or classes can be specified as output beans.

To invoke the mapper builder select a record bean and *Tools -> Mapper Editor*.

The mapper editor is presented. The input bean is primed with the properties of the selected Record.

To complete the mapping follow these steps:

❑ Add business objects (classes) to the output bean. You can add more than one class.

❑ In the Output Bean pane, select a business object property that you want to map to a record bean property.

❑ In the Input Bean pane, select the record bean field or property that you want to map to the business object property.

❑ Click on the *Connection* button that corresponds to the desired direction of data flow between the selected properties. For the mapper of an input record, the source is the output bean (business object) and the target is the input bean (record). For the mapper of an output record, the source is the input bean (record) and the target is the output bean (business object).

❑ Complete the previous three steps for any additional connections you want to map.

❑ Click on *Apply*. You are prompted to supply project, package, and class names.

❑ When all the mappings are established, click on *OK* to generate the mapper bean.

Figure 72 shows the mapper editor for creating a mapping from the CardRecord bean to objects of the customer and card classes. This mapping was developed for the ATM application.

You can also use the mapper editor to edit existing mapper classes.

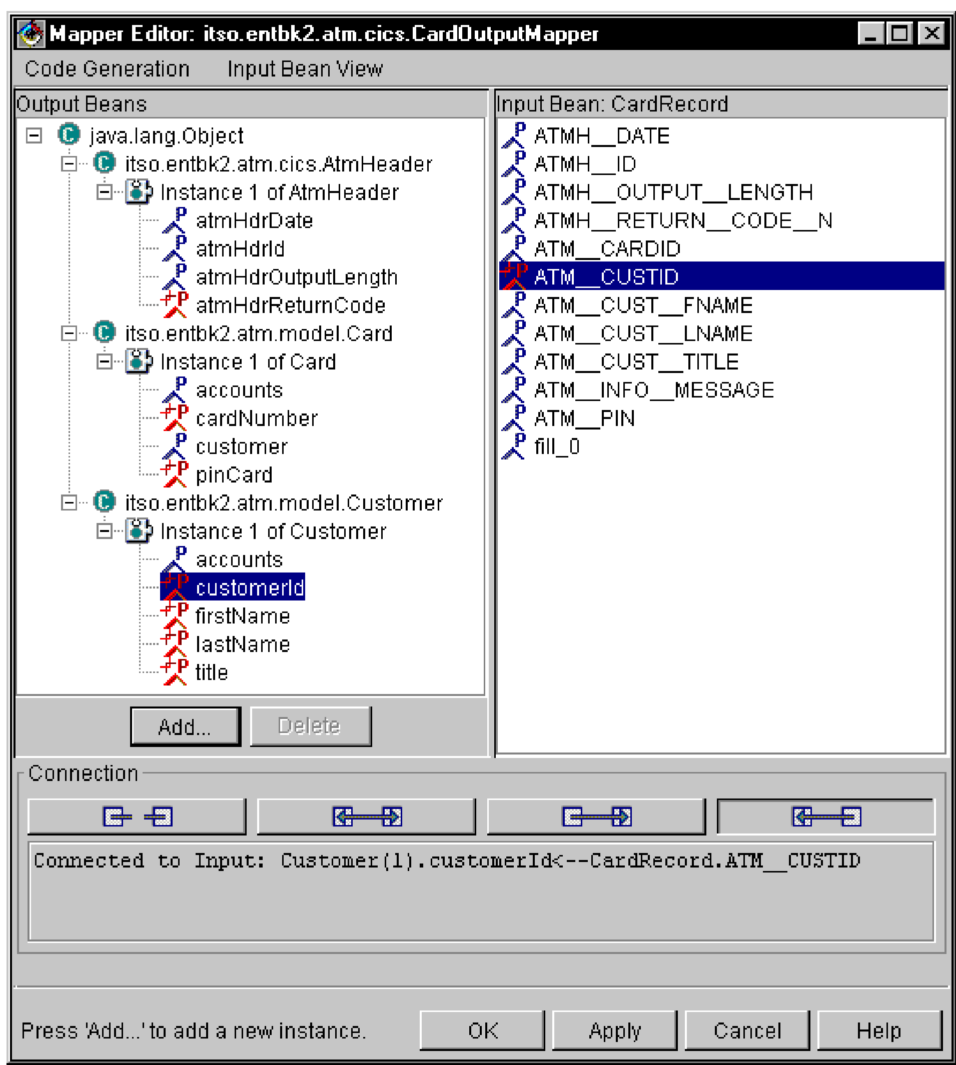


Figure 72. Mapper Editor

Including the Mapper in a Command

As mentioned in “Constructing a Command” on page 112, mapper beans can be added to a command.

Assuming that you have launched the Command Editor:

- ❑ Mark the record bean to be associated with the mapper.
- ❑ Right-click and select *Add Mapper*.
- ❑ Specify the name of the mapper class.
- ❑ Click on *OK*.

Figure 73 shows the Command Editor once mappers have been added.

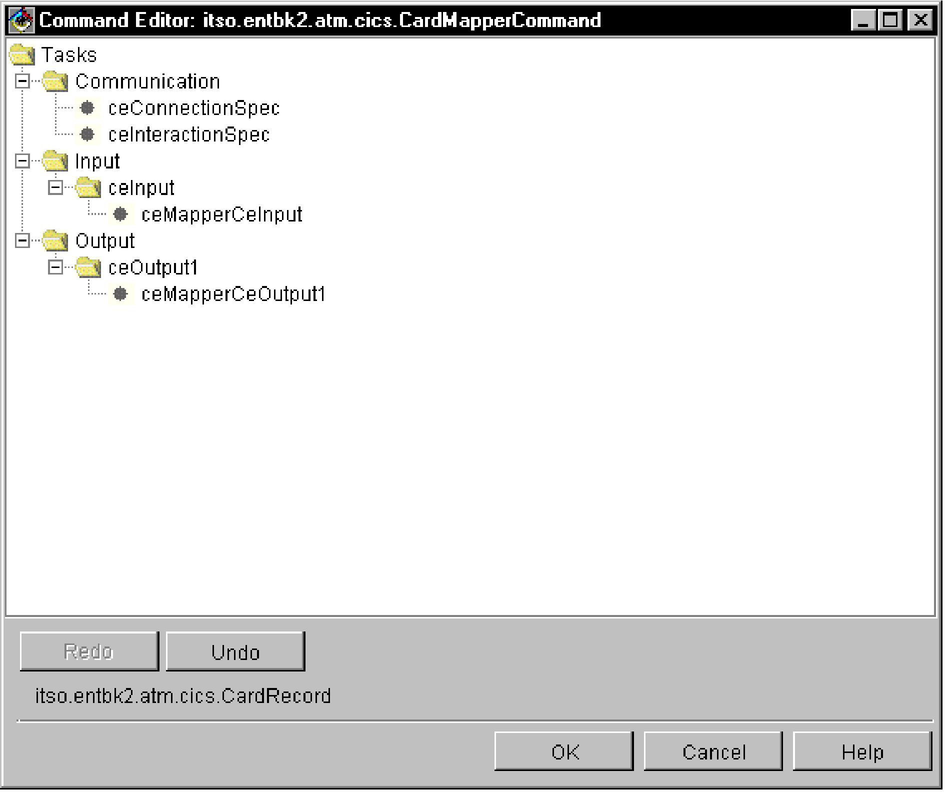


Figure 73. Command Editor with Mappers

Figure 74 shows the Visual Composition Editor view of a command with mapper beans.

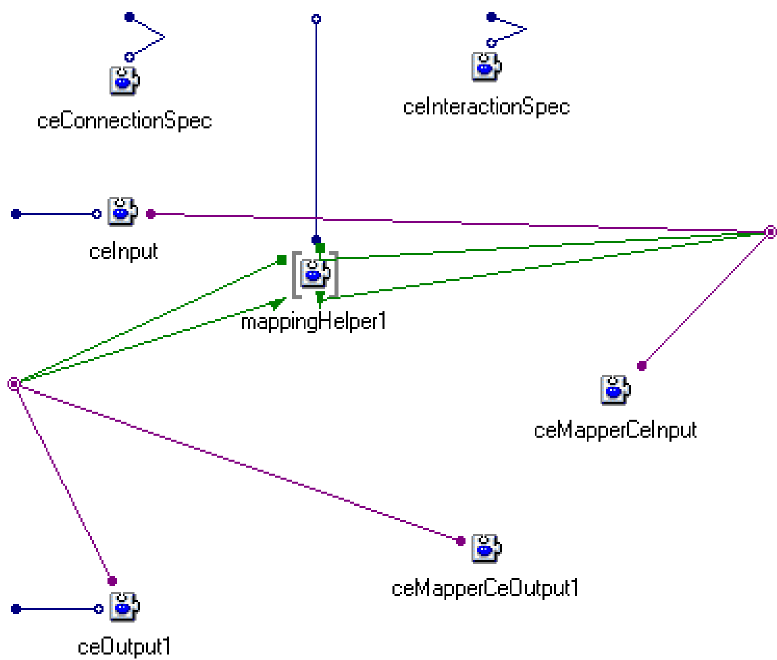


Figure 74. Visual Composition Editor View of Command with Mappers

Executing the Command

To use the command you create a class that invokes the command's execute method. Use the Visual Composition Editor.

The general principle is to have an event triggered within the application. Connect this event in the Visual Composition Editor to the *execute* method of the command bean. Then connect the *executionSuccessful* and *executionUnsuccessful* events of the command bean to application actions.

A sample execution of a command from a client application is described in Chapter 10, "ATM Application with the CICS Connector."

4.8 A Review of Accessing Enterprise Data

In “Accessing Enterprise Data” on page 103, the classes and beans needed to access enterprise data with CICS programs are discussed.

Essentially classes are needed to define the flow from a Java class to a CICS program using a COMMAREA.

The COMMAREA is represented by a record bean. A record bean is associated with both input and output. The connectivity to CICS and the program that is to be called is specified in the CICSConnectionSpec and InteractionSpec beans, respectively.

A command is used to collate input, output, and communication beans. A set of commands can be strung together to form a navigator. In the navigator the sequence of execution between commands is defined.

A record can be mapped to business objects through a mapper bean. The mapper bean is added to a command. With a mapper bean, a record that represents the COMMAREA is mapped to one or more business objects or application classes. Data from the business objects (or application classes) is moved to the COMMAREA record before command execution, and data from the COMMAREA is moved to business objects after command execution.

In Chapter 10, “ATM Application with the CICS Connector” we show how the CICS Connector is used to connect the ATM application to CICS programs running on a CICS server.

Part 2 Implementing the ATM Application

In Part 2 we introduce the ATM application (see Figure 75 on page 126) and then implement it using the features of VisualAge for Java Enterprise Version 2 described in Part 1:

- Data access beans
- Servlet Builder
- CICS Connector

We also show how to use MQSeries to connect the ATM application to a CICS or other server.

We describe how to deploy VisualAge for Java applets, applications, and servlets to various environments.

We conclude with a short description of the Java high-performance compiler and remote debugger.

© Copyright IBM Corp. 1998

Figure 75 shows the user interfaces, business model, application controller, and persistence servers we use in this book for the ATM application.

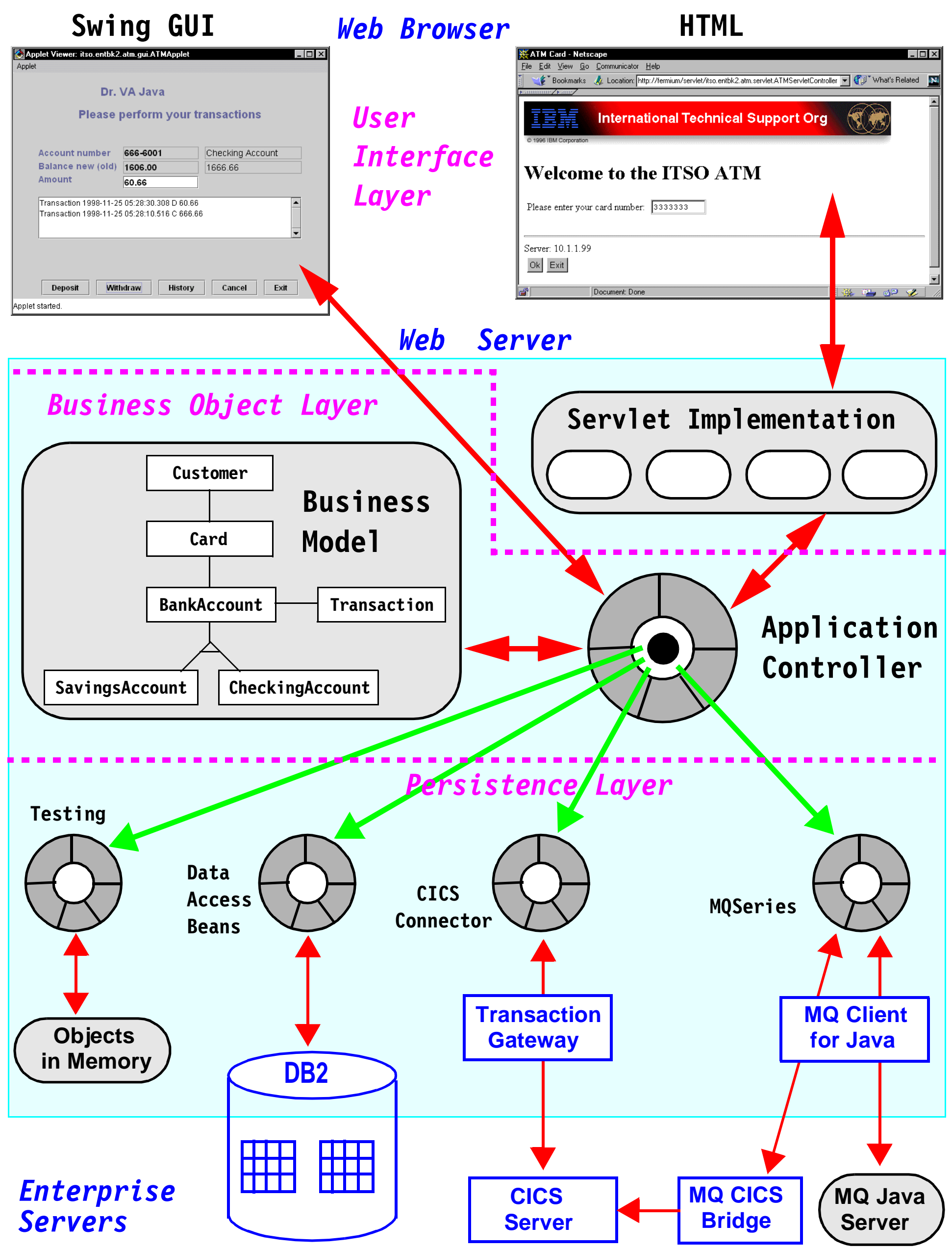


Figure 75. ATM Application Layers and Implementations

5 ATM Application Requirements and ATM Database

In Part 1, we demonstrate the power of the new features of VisualAge for Java Enterprise Version 2, such as data access beans, Servlet Builder, and the CICS Connector, and construct sample applications to explain their functionality.

In this chapter, we describe a more sophisticated application that combines several features. We define the application requirements and the underlying relational database.

In the chapters that follow, we implement the business logic together with different user interfaces and data sources.

© Copyright IBM Corp. 1998

5.1 ATM Application Requirements

Basically, the application used in this book is similar to that described in the redbook *Application Development with VisualAge for Java Enterprise*, IBM form number SG24-5081.

When we wrote this book we had to decide which sample application we would use. We concluded that the best way was to reimplement an existing sample and improve it with the extended functionality of VisualAge for Java Enterprise Version 2. This would give you the opportunity to draw on your experience using the first book and become familiar with new approaches of VisualAge for Java without spending too much time on unimportant things.

In fact, the business object model and the controller of the ATM application are almost identical. What we did change are the user interface, database access, and transaction invocation.

The ATM application handles two types of accounts, savings and checking. For both accounts, customers can perform debit and credit transactions. In addition, customers can list the transaction history belonging to an account. Customers must maintain a minimum balance in a savings account and cannot withdraw funds beyond a specified overdraft amount from a checking account.

The ATM application simulates an ATM card reader installed at real ATM machines. To start a bank transaction, the user is prompted to enter the ATM card identification number (card ID).

On receiving a valid card ID, the ATM application greets the customer with the customer's name and title in the PIN panel. The card ID is re-displayed, so that the customer can verify the card. The ATM application prompts the customer for the personal identification number (PIN).

The ATM application verifies the PIN. If the PIN is invalid, a message is displayed indicating that the number is invalid, and the customer can reenter the PIN. On successful validation, a list of accounts that belong to the card is displayed in the Account panel.

The ATM application requests the customer to select an account for further processing. When the customer selects an account, the Transaction panel showing the customer information (title, name) and the account information (account ID, account type, old balance, new balance) is displayed. At the start of a transaction, the new balance is the same as the old balance. The customer can enter an amount and proceed with a debit or a credit transaction. After every successful transaction, the new balance is displayed.

If the customer requests to see the account's transaction history, the ATM application displays the accumulated transactions in a drop-down list on the same panel.

From all panels the customer can return to the previous panel.

Figure 76 shows the basic layout of the panels and the application flow.

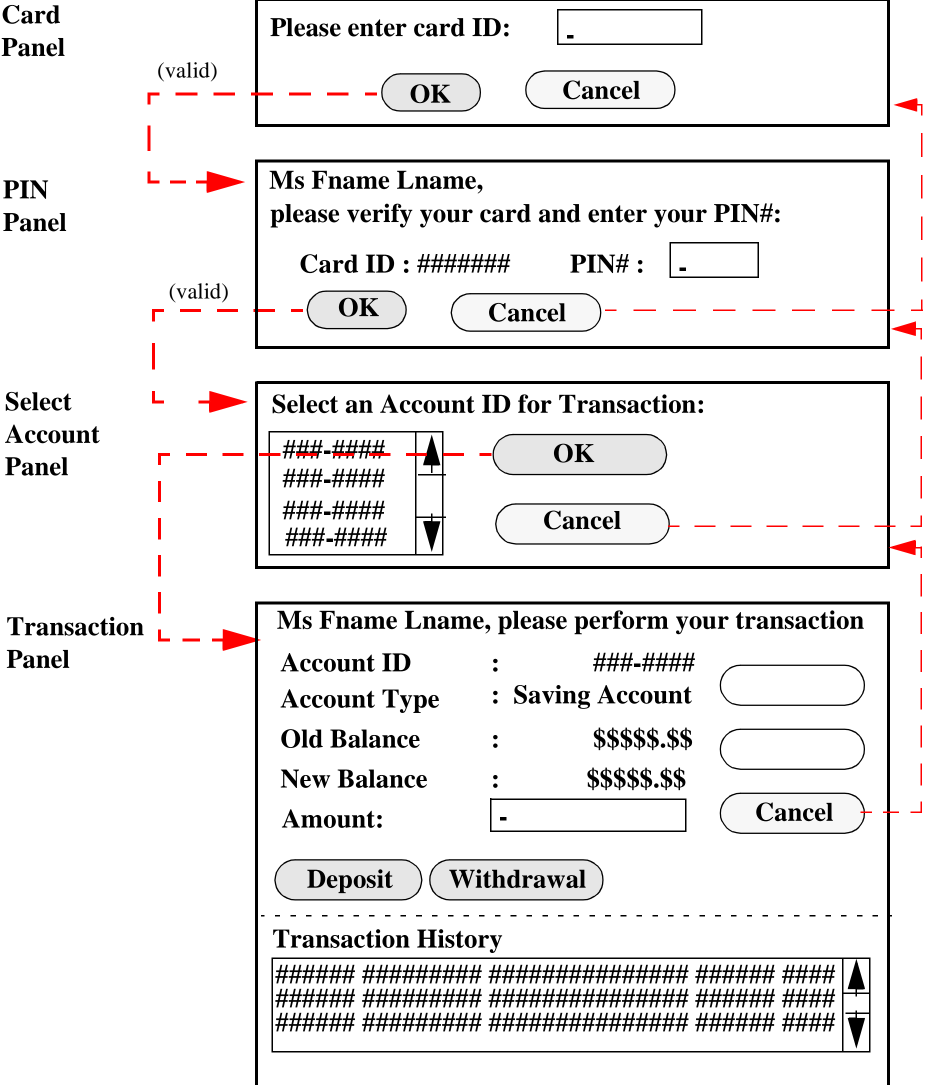


Figure 76. ATM Application Panels and Flow

5.2 ATM Database Implementation

For compatibility purposes, we use the existing database design described in the redbook *Application Development with VisualAge for Java Enterprise*, IBM form number SG24-5081.

The following description is a summary from that book. We only review the relationships among the tables (Figure 77) and the physical database design.

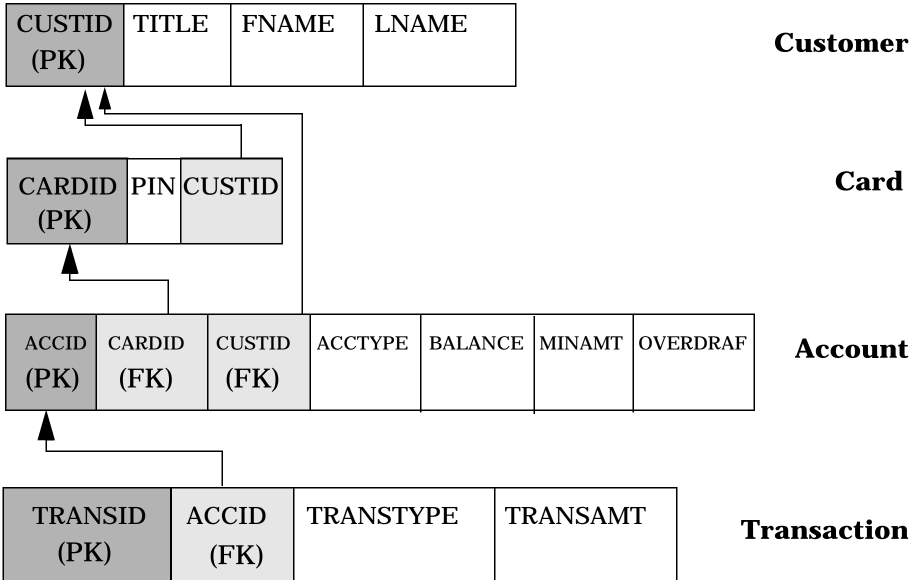


Figure 77. Relationships among the ATM Tables

Tables 10 through 13 show the physical design of the ATM tables.

Table 10. Customer Table

Column Name	Type	Length	Key	Nulls	Description
CUSTID	CHAR	4	Yes	No	Customer ID
TITLE	CHAR	3	No	No	Title
FNAME	CHAR	30	No	No	First name
LNAME	CHAR	30	No	No	Last name

Table 11. Card Table

Column Name	Type	Length	Key	Nulls	Description
CARDID	CHAR	7	Yes	No	Card ID
PIN	CHAR	4	No	No	PIN
CUSTID	CHAR	4	No	No	Customer ID

Table 12. Account Table

Column Name	Type	Length	Key	Nulls	Description
ACCID	CHAR	8	Yes	No	Account ID
CARDID	CHAR	7	No	No	Card ID
CUSTID	CHAR	4	No	No	Customer ID
ACCTYPE	CHAR	1	No	No	Account type (S = Savings C = Checking)
BALANCE	DEC	(8, 2)	No	No	Balance
MINAMT	DEC	(8, 2)	No	No	Minimum amount
OVERDRAF	DEC	(8, 2)	No	No	Overdraft amount

Table 13. Transaction Table

Column Name	Type	Length	Key	Nulls	Description
TRANSID	TIME- STAMP	26	Yes	No	Transaction ID
ACCID	CHAR	4	No	No	Account ID
TRANSTYPE	CHAR	1	No	No	Transaction type (D = Debit C = Credit T = Transfer)
TRANSAMT	DEC	(8, 2)	No	No	Transaction amount

After determining the physical database design, we use command line processor commands and SQL statements to create the database and the objects within it.

Enter the code of Figure 78 in a new text document, save it as AtmDB.ddl, and run the DB2 command line processor:

```
db2 -f AtmDB.ddl
```

```
echo --- create the ATM database ---
CREATE DATABASE ATM

echo --- connect to ATM database ---
CONNECT TO ATM

echo --- creating tables ---
CREATE TABLE ATM.CUSTOMER (                  \
  custid     CHAR( 4) NOT NULL PRIMARY KEY,  \
  title      CHAR( 3) NOT NULL,              \
  fname      CHAR(30) NOT NULL,              \
  lname      CHAR(30) NOT NULL               \
)
CREATE TABLE ATM.CARD (                                              \
   cardid      CHAR(7) NOT NULL PRIMARY KEY,                         \
   pin         CHAR(4) NOT NULL,                                     \
   custid      CHAR(4) NOT NULL,                                     \
   FOREIGN KEY (custid) REFERENCES ATM.CUSTOMER ON DELETE RESTRICT \
)
CREATE TABLE ATM.ACCOUNT (                                              \
   accid        CHAR(8)  NOT NULL PRIMARY KEY,                          \
   cardid       CHAR(7)  NOT NULL,                                      \
   custid       CHAR(4)  NOT NULL,                                      \
   acctype      CHAR(1)  NOT NULL,                                      \
   balance      DEC(8,2),                                               \
   minamt       DEC(8,2),                                               \
   overdraf     DEC(8,2),                                               \
   FOREIGN KEY (custid)   REFERENCES ATM.CUSTOMER ON DELETE RESTRICT, \
   FOREIGN KEY (cardid)   REFERENCES ATM.CARD ON DELETE RESTRICT      \
)
CREATE TABLE ATM.TRANS (                                             \
   transid    TIMESTAMP  NOT NULL PRIMARY KEY,                       \
   accid      CHAR(8)  NOT NULL,                                     \
   transtype  CHAR(1)  NOT NULL,                                     \
   transamt   DEC(8,2) NOT NULL,                                     \
   FOREIGN KEY (accid) REFERENCES ATM.ACCOUNT ON DELETE RESTRICT   \
)

echo --- execute GRANT statements ---
GRANT BINDADD ON DATABASE     TO PUBLIC
GRANT CONNECT ON DATABASE     TO PUBLIC
GRANT ALL     ON ATM.CUSTOMER TO PUBLIC
GRANT ALL     ON ATM.CARD     TO PUBLIC
GRANT ALL     ON ATM.ACCOUNT  TO PUBLIC
GRANT ALL     ON ATM.TRANS    TO PUBLIC

echo --- connect reset ---
CONNECT RESET
```

Figure 78. ATM Database Data Definition Language

Sample Data of ATM Tables

The sample data of the ATM tables shows the internal relationships among the tables:

- ❑ We have six customers, with numbers 101 to 106.
- ❑ There are seven ATM cards with numbers 1111111 to 7777777, and matching PINs 1111 to 7777.
- ❑ Account numbers are structured xxx-yyyy, where xxx is the customer number.

Tables 14 through 17 list an extract of the sample data of the ATM tables.

Table 14. Customer Table Sample Data

CUSTID	TITLE	FNAME	LNAME
101	Ms.	Avril	Kotzen
102	Mr.	Olaf	Graf
103	Mr.	Osamu	Takagiwa
104	Mr.	Frederik	Haesbrouck
105	Ms.	Unkown	Lady
106	Mr.	Ueli	Wahli

Table 15. Card Table Sample Data

CARDID	PIN	CUSTID
1111111	1111	101
2222222	2222	102
....		
5555555	5555	105
6666666	6666	106
7777777	7777	106

Table 16. Account Table Sample Data

ACCID	CARD-ID	CUST-ID	ACC-TYPE	BALANCE	MIN-AMT	OVERDRAF
101-1001	1111111	101	C	80.00	0.00	100.00
101-1002	1111111	101	C	195.22	0.00	400.00
101-1003	1111111	101	S	9375.26	100.00	0.00
102-2001	2222222	102	S	19375.26	9999.99	0.00
....						
106-6666	6666666	106	C	6.66	0.00	0.00
106-7777	7777777	106	S	111.11	11.11	0.00

Table 17. Transaction Table Sample Data

TRANSID	ACCID	TRANSTYPE	TRANSAMT
1997-10-07-14.30.26.720001	101-1001	C	80.00
CURRENT TIMESTAMP	101-1002	C	200.00
...			
CURRENT TIMESTAMP	106-7777	D	111.11

Figure 79 shows the SQL statements to load the sample data into the tables. Run this file with:

```
db2 -f AtmDB.sql
```

```
echo --- connect to ATM database ---
CONNECT TO ATM

echo --- insert into CUSTOMER table ---
INSERT INTO ATM.CUSTOMER                              \
   (custid, title, fname,      lname) VALUES  \
   ('101', 'Ms.', 'Avril',    'Kotzen'),      \
   ('102', 'Mr.', 'Olaf',     'Graf'),        \
   ('103', 'Mr.', 'Osamu',    'Takagiwa'),    \
   ('104', 'Mr.', 'Frederik', 'Haesbrouck'),  \
   ('105', 'Ms.', 'Unkown',   'Lady'),        \
   ('106', 'Mr.', 'Ueli',     'Wahli')

echo --- insert into CARD table ---
INSERT INTO ATM.CARD                          \
   (cardid,      pin,      custid) VALUES  \
   ('1111111',   '1111',   '101' ),        \
   ('2222222',   '2222',   '102' ),        \
   ('3333333',   '3333',   '103' ),        \
   ('4444444',   '4444',   '104' ),        \
   ('5555555',   '5555',   '105' ),        \
   ('6666666',   '6666',   '106' ),        \
   ('7777777',   '7777',   '106' )

echo --- insert into ACCOUNT table ---
INSERT INTO ATM.ACCOUNT                                                             \
   (accid,      cardid,   custid, acctype, balance,  minamt, overdraf) VALUES  \
   ('101-1001', '1111111', '101', 'C',       80.00,    0.00,  100.00),         \
   ('101-1002', '1111111', '101', 'C',      195.22,    0.00,  400.00),         \
   ('101-1003', '1111111', '101', 'S',     9375.26,  100.00,    0.00),         \
   ('102-2001', '2222222', '102', 'S',    19375.26, 9999.99,    0.00),         \
   ('102-2002', '2222222', '102', 'C',       75.50,    0.00, 3000.00),         \
   ('103-3001', '3333333', '103', 'S',      100.00,  100.00,    0.00),         \
   ('104-4001', '4444444', '104', 'C',      362.00,    0.00,  100.00),         \
   ('105-5001', '5555555', '105', 'C',        0.00,    0.00,    0.00),         \
   ('106-6001', '6666666', '106', 'C',     1000.00,    0.00,  100.00),         \
   ('106-6002', '6666666', '106', 'S',     2000.00,  200.00,    0.00),         \
   ('106-6003', '6666666', '106', 'S',     3000.00,  300.00,    0.00),         \
   ('106-6004', '6666666', '106', 'C',     4000.00,    0.00,  250.00),         \
   ('106-6666', '6666666', '106', 'C',        6.66,    0.00,    0.00),         \
   ('106-7777', '7777777', '106', 'S',      111.11,   11.11,    0.00)

echo --- insert into TRANS table ---
INSERT INTO ATM.TRANS                                          \
   (transid,           accid,      transtype, transamt) VALUES  \
   (CURRENT TIMESTAMP, '101-1001', 'C',         80.00 )
INSERT INTO ATM.TRANS                                          \
   (transid,           accid,      transtype, transamt) VALUES  \
   (CURRENT TIMESTAMP, '101-1002', 'C',        200.00 )
INSERT INTO ATM.TRANS                                          \
   (transid,           accid,      transtype, transamt) VALUES  \
   (CURRENT TIMESTAMP, '101-1003', 'C',      10000.00 )
...............
echo --- connect reset ---
CONNECT RESET
```

Figure 79. ATM Database Sample Data Load

6 ATM Application Business Model

In this chapter we design an implementation of the ATM application, using a layered approach with a user interface, business logic, and persistence layer.

We also implement a controller that interacts with the different layers to minimize direct interactions between the layers.

© Copyright IBM Corp. 1998

6.1 Application Design

The first step in developing an application is to create an application design based on the requirements the application should satisfy. The power of this design determines how much effort you must spend on implementing it, as well as how difficult it is to modify some features when the requirements change.

Application Layers

One prerequisite for maintainable applications is a layered architecture that assigns responsibility for certain services to an appropriate application layer. These responsibilities are similar to an object model, where we assign responsibilities to each object.

First, all business logic and application knowledge should be modeled in business objects, located in the business object layer. These objects represent core entities of the ATM application and have the responsibility for its correct behavior. To satisfy the needs of the underlying entities, the business objects implement both properties and methods in a standardized way.

Next, we define a user interface layer to separate the GUI part of the application from the core business objects. The GUI objects are responsible for handling all user interactions and for presenting business objects in a nice format to users. Whenever information from the business objects is required, or an action is triggered by a user, the respective service from the business objects is called.

We also want to separate the data access from the rest of the application. The ATM application has to remember the customer, the card, the account, and the transaction information. Neither the GUI nor the business objects should be aware of the details about where the data is stored. This separation makes it possible to have the core of the application unchanged, even if we change database access, or use other services such as a transaction system to access the enterprise data.

Application Layer Architecture

We end up with three layers for the ATM application as shown in Figure 80.

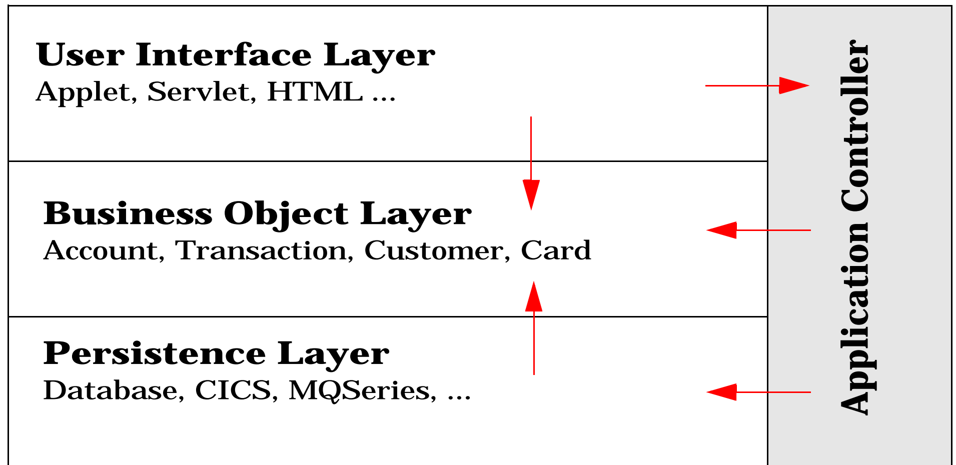


Figure 80. Layers of the ATM Application

Figure 80 also shows a cross-layer called the *Application Controller*. The application controller is necessary to connect the layers. Basically there are three ways of connecting the layers. The important principle is that there be a crisp interface among the layers.

Here are the three basic ways of connecting the layers:

- Each object has all the knowledge required to access other objects across the borders. This approach looks very appealing at first but can become quite cumbersome because changes at one point might affect multiple classes. Therefore, this approach is only feasible for small applications.
- A *framework* provides the necessary interfaces and underlying services. The application objects just connect to the framework, by either inheritance or delegation. This is a great approach if such a framework is available. However, the creation of a framework is difficult and requires another approach and different skills from those required for an application development project.
- The interfaces are modeled in objects that have some knowledge of both sides. Such objects are often called *mediators*. This approach has the advantage that we have only one place to update, if the interface of a layer or subsystem changes. The downside, of course, is a somewhat longer path for the messages.

We decided to use mediator objects in the ATM application; these objects are part of the Controller. We explain their responsibilities when we describe the objects in more detail.

The most encapsulated layer is the business object layer. Business objects are not aware of the existence of both the user interface layer and the persistence layer. The user interface layer classes use the business objects but have no connection to the persistence layer. The persistence layer knows about the business objects, but not about the user interface layer.

6.2 Business Object Layer

Let's review "ATM Application Requirements" on page 128 to isolate the classes we need (see also Figure 81 on page 141).

We deal with two type of accounts, savings and checking. The main business logic functions are customer identification and account transactions. In that respect, our implementation requires the following classes:

- *SavingsAccount* and *CheckingAccount*. Because the main functionality is identical, we introduce a third, abstract class, *BankAccount,* from which the real account classes inherit most of their behavior.
- *Card* and *Customer*. This allows customer identification.
- *Transaction*. Every bank transaction should be logged.

The following types of transactions have been identified: PIN validation, debit transaction, and credit transaction. In addition, customer information, card information, and transaction history have to be maintained.

In our implementation, Card is the class that knows the card number entered by the user and the PIN related to that card number. Starting from this information, it must perform the PIN validation when the user enters the PIN. After validation, the Card sends a successful or unsuccessful message.

Card also holds a property of type Customer. We need the information stored in a Customer object to welcome the card holder with title and name.

BankAccount, SavingsAccount, and CheckingAccount are closely related to each other. In fact, the BankAccount class represents the generic bank account, with all of the features of a bank account, such as account ID, balance, and customer ID. SavingsAccount and CheckingAccount inherit from BankAccount, but each of them has additional information and behavior. The SavingsAccount class contains the information related to the minimum amount that it can reach; the CheckingAccount, instead, contains its overdraft value. In other words, the instances of the SavingsAccount and CheckingAccount classes represent the real accounts of the customer.

When a customer uses the ATM application to perform a withdrawal transaction, different answers can come from the system, depending on the kind of account. The SavingsAccount class performs the withdrawal transaction only if the balance, after the transaction, is still greater than the minimum amount; the CheckingAccount class checks that a resulting negative balance is higher than the overdraft amount.

When a customer requires a deposit transaction, both the SavingsAccount and CheckingAccount classes have the same behavior, that is, they increase the balance by the deposit amount.

In addition, the account classes implement a property of type Transaction to log all successful transactions made by the customer in a transaction history log in the underlying database.

Figure 81 shows the complete object model of the business object layer.

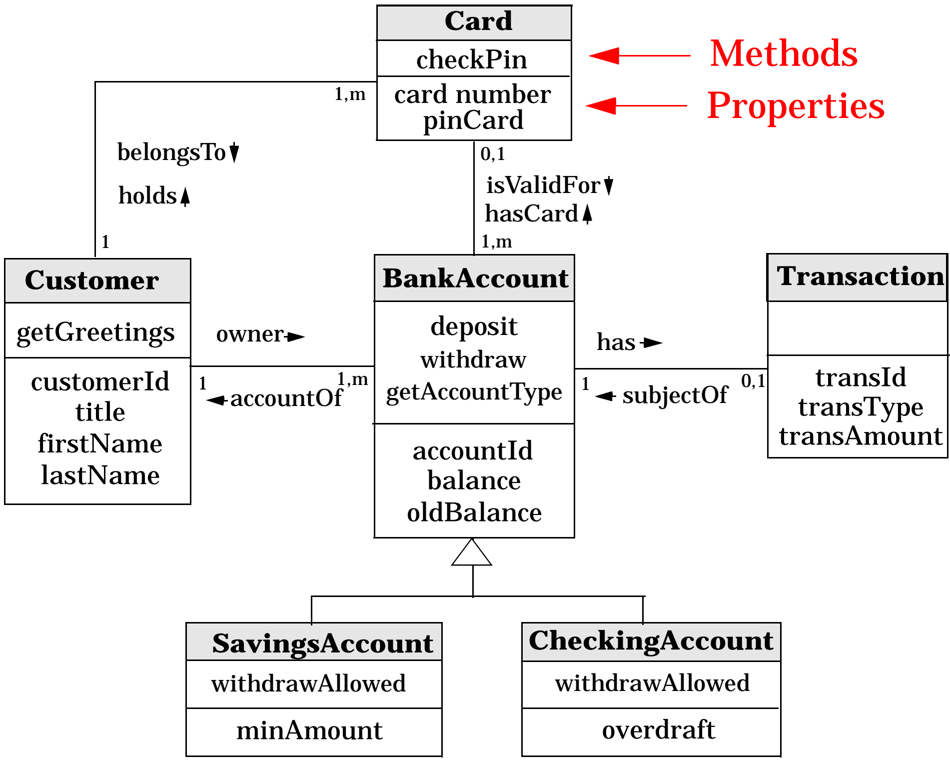


Figure 81. Object Model of the ATM Business Object Layer

Business Logic Classes

Now let us see how to implement the business logic classes. We create all classes as nonvisual beans in the **itso.entbk2.atm.model** package. Therefore the features (properties, methods, and events) are defined on the BeanInfo page of the class. Create the properties as read and write with get and set methods and define that they are *bound*, that is, they fire the PropertyChange event whenever the value changes.

Customer Class

The Customer class holds information about a customer, that is, a customer ID, a title, a first name, and a last name.

We implement the Customer class as nonvisual bean.[6] Create a new class, Customer, and switch to the BeanInfo page of the Class Details View window. Add the following features, using *New Property Feature...* in the *Features* menu:

- customerId, type java.lang.String
- title, type java.lang.String
- firstName, type java.lang.String
- lastName, type java.lang.String
- accounts, type java.util.Vector

The customer has bank accounts, but he or she also holds a card that is associated with the same accounts. To avoid storing the same information twice, the accounts property of the Customer class is never used. Instead, the ATM application uses the Card class to hold the account information.

Reviewing the created code you can see that the internal attribute name for a property has the prefix *field*, followed by the property name with the first letter in uppercase. For example, the customerId property is represented by the fieldCustomerId attribute, and the get and set methods are called getCustomerId and setCustomerId.

Next, you implement two methods, a constructor, and a method to return the greeting text:

[6] An alternative way would be to implement the Customer class as an inner class of the Card class. In fact, we do not require direct access to Customer because all functionality is encapsulated in the Card bean.

- ❑ A user-defined constructor to create a new Customer object:

```
public Customer(String customerId, String title, String firstName,
                String lastName) {
   setCustomerId(customerId);
   setTitle(title);
   setFirstName(firstName);
   setLastName(lastName);
}
```

- ❑ A getGreetings method feature (BeanInfo page) that returns a formatted string with title, first name, and last name:

```
public String getGreetings() {
   return getTitle().trim() + " " + getFirstName().trim() + " " +
          getLastName().trim();
}
```

Card Class

The Card class can validate a PIN and knows the holder (customer) and the accounts associated with the card.

Add the following properties on the BeanInfo page:

- ❑ accounts, type java.util.Vector
- ❑ cardNumber, type java.lang.String
- ❑ pinCard, type java.lang.String
- ❑ customer, type itso.entbk2.atm.model.Customer

The accounts property is a vector to store objects of class BankAccount, cardNumber is used to specify one unique card, pinCard stores the PIN to verify this card, and customer points to the customer information.

The Card class has to send messages to other objects about the result of the PIN validation. We implement this behavior as two events, pinCheckedOk, thrown when the correct PIN is entered, and pinCheckedNotOk, thrown when a bad PIN is entered.

Select *New Listener Interface* in the *Features* pull-down menu of the BeanInfo page, and the New Event Listener SmartGuide opens:

- ❑ On the first page of the dialog, enter the name of the event, pinCheckedOk (or pinCheckedNotOk), and click on *Next* (Figure 82).

SmartGuide

New Event Listener

Enter an event set name, event listener and event object for the new feature:

Event name: pinCheckedOk

Event listener: PinCheckedOkListener

Event object: PinCheckedOkEvent

Event Multicaster: PinCheckedOkEventMulticaster

< Back | Next > | Finish | Cancel

Figure 82. Defining an Event with an Event Listener (First Page)

- ❑ On the second page, enter the name of the method that the listener class has to implement, handlePinCheckedOk (or handlePinCheckedNotOk), and click on *Add*, then on *Finish* (Figure 83).

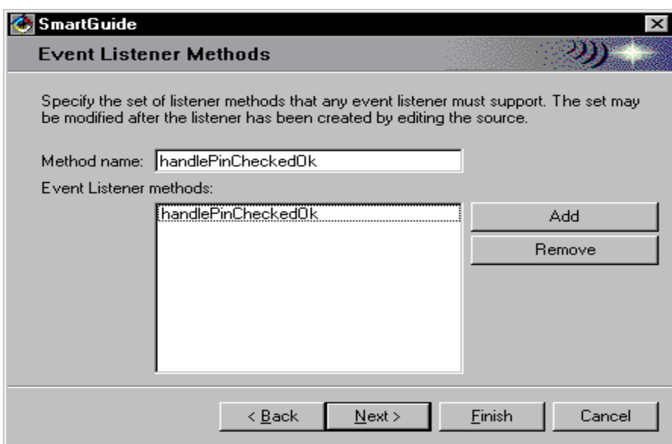


Figure 83. Defining an Event with an Event Listener (Second Page)

The system automatically generates the event and interface classes and the supporting methods.

- ❑ PinCheckedOkEvent class
- ❑ PinCheckedOkListener interface
- ❑ fireHandlePinCheckedOk method
- ❑ addPinCheckedOkListener method
- ❑ removePinCheckedOkListener method

- PinCheckedNotOkEvent class
- PinCheckedNotOkListener interface
- fireHandlePinCheckedNotOk method
- addPinCheckedNotOkListener method
- removePinCheckedNotOkListener method

To implement the logic to fire the events, add a new method feature to the Card class and call it *checkPin*. The checkPin method compares the pinEntered parameter with the pinCard property and notifies the caller by firing the pinCheckedOkEvent or the pinCheckedNotOkEvent:

```
public void checkPin(String pinEntered) {
   if ( getPinCard().trim().equals(pinEntered.trim()) )
      fireHandlePinCheckedOk(new PinCheckedOkEvent(this));
   else
      fireHandlePinCheckedNotOk(new PinCheckedNotOkEvent(this));
}
```

Additionally, implement the following methods:

- A user-defined constructor to create a new Card object:

```
public Card (String cardNumber, String pinCard, Customer customer) {
   setCardNumber(cardNumber);
   setPinCard(pinCard);
   setCustomer(customer);
   clearAccounts();          // <=== will be defined below
}
```

- A method feature to add an account to the accounts vector. To avoid errors you must also define the BankAccount class. Leave the BankAccount class empty for now; you will complete it shortly.

```
public void addAccount (BankAccount account) {
   getAccounts().addElement(account);
}
```

- A method feature that returns the account with a given *accountId.* (Note that getAccountId() will be defined later for the BankAccount class.)

```
public BankAccount getAccount(String accountId) {
   BankAccount bankaccount = null;
   for (int i = 0; i < getAccounts().size(); i++) {
      if (((BankAccount)
            getAccounts().elementAt(i)).getAccountId().equals(accountId)) {
         bankaccount = (BankAccount)getAccounts().elementAt(i);
         break;
      }
   }
   return bankaccount;
}
```

- A method feature to clear the accounts vector:

```
public void clearAccounts () {
   setAccounts(new java.util.Vector());
}
```

- A method feature that calls the customer's getGreetings method:

```
public String getGreetings() {
   return getCustomer().getGreetings();
}
```

Transaction Class

In the ATM application the transaction objects are no more than a container of transaction data. Once they are created they are just there for logging.

Add the following properties on the BeanInfo page:

- accountId, type java.lang.String
- transAmount, type java.math.BigDecimal
- transId, type java.sql.Timestamp
- transType, type java.lang.String

The accountID identifies the account and transAmount the amount of the transaction. The transId sets a unique time stamp used as a primary key in the database table. The transType property indicates whether the transaction is a debit or credit transaction.

You have to implement two user-defined constructors to create new Transaction objects:

```
public Transaction (String accountid, String transtype,
                    java.math.BigDecimal amount) {
   setAccountId(accountid);
   setTransType(transtype);
   setTransAmount(amount);
   setTransId( new java.sql.Timestamp( System.currentTimeMillis() ) );
}

public Transaction (String accountid, String transtype,
                    java.math.BigDecimal amount,java.sql.Timestamp transId){
   setAccountId(accountid);
   setTransType(transtype);
   setTransAmount(amount);
   setTransId(transId);
}
```

BankAccount Class

We implement the BankAccount class as an abstract class. In fact, we want to reuse the attributes and methods of the BankAccount class in the SavingsAccount and CheckingAccount classes. In the ATM application we create only instances of SavingsAccount or CheckingAccount, that is, the real accounts that belong to a customer.

Add the following properties on the BeanInfo page:

- accountId, type java.lang.String
- balance, type java.math.BigDecimal
- oldBalance, type java.math.BigDecimal
- tempBalance, type java.math.BigDecimal
- transactions, type java.util.Vector

The accountId identifies the bank account, the balance and oldBalance store information about the account balance, and the transactions property stores a vector of objects of type Transaction. The property tempBalance is used to simulate a commit or rollback behavior.

Additionally, we add a number of methods:

- A user-defined constructor to initialize the attributes of a bank account:

```
public BankAccount(String accId, java.math.BigDecimal balance) {
   setAccountId(accId);
   setBalance(balance);
   setOldBalance(new java.math.BigDecimal(0));
   clearTransactions();        //<=== to be defined below
}
```

- Three method features related to the transactions vector—add a transaction, clear the vector, and get the last transaction:

```
public void addTransaction(Transaction transaction) {
   getTransactions().addElement(transaction);
}

public void clearTransactions() {
   setTransactions(new java.util.Vector());
}

public Transaction getLastTransaction() {
   return (Transaction) getTransactions().lastElement();
}
```

❑ The account type should be displayed as a word, Checking Account or Savings Account, not just as code C or S. For this purpose you have to define an abstract method that you will implement later in the SavingsAccount and CheckingAccount subclasses:

```
public abstract String getAccountType();
```

Create a new public abstract method on the Methods page of the Class Details View window and add this method on the BeanInfo page, using *Features* -> *Add Available Features*.

❑ To simulate a commit or rollback behavior you implement two method features:

```
public void commit() { }

public void rollback() {
   setOldBalance(getTempBalance());
   setBalance(getOldBalance());
   getTransactions().removeElementAt(getTransactions().size()-1);
}
```

In fact, this is only a simulation to keep the code as simple as possible.

❑ The deposit transaction is independent on the account type, so you can implement the deposit method feature in the BankAccount class and use the same implementation for SavingsAccount and CheckingAccount. This methods stores the current balance in the oldBalance property and then adds the amount entered by the user:

```
public void deposit (String amount) {
   java.math.BigDecimal amt = new java.math.BigDecimal(amount);
   setTempBalance(getOldBalance());
   setOldBalance(getBalance());
   setBalance(getBalance().add(amt));
   getTransactions().addElement( new Transaction(getAccountId(),"C",amt) );
}
```

❑ The withdrawal transaction is dependent on the account type, because it has to check whether the account balance can be updated on the basis of the balance and overdraft properties. Therefore, you implement two methods:

- One abstract method that checks whether updating is allowed. This method is implemented in the SavingsAccount and CheckingAccount subclasses:

  ```
  public abstract boolean withdrawAllowed(java.math.BigDecimal amount);
  ```

- One method to perform the withdrawal transaction (the method is not transaction dependent):

```
public boolean withdraw (String amount) {
   java.math.BigDecimal amt = new java.math.BigDecimal(amount);
   if (withdrawAllowed(amt)) {
      setTempBalance(getOldBalance());
      setOldBalance(getBalance());
      setBalance(getBalance().subtract(amt));
      getTransactions().
         addElement(new Transaction(getAccountId(), "D", amt));
      return true;
   }
   return false;
}
```

BankAccount fires an event to inform the caller when there are not enough funds to allow the withdrawal transaction. The SavingsAccount and CheckingAccount classes use these methods in their own withdrawAllowed methods. If a withdrawal is not allowed, the ATM application can fire an event.

We implement this behavior in the same way we implemented it for the Card class. Select *New Listener Interface* in the *Features* pull-down menu of the BeanInfo page to open the New Event Listener SmartGuide:

- On the first page of the dialog, enter the name of the event, limitExceeded.
- On the second page, enter the name of the method that the listener class has to implement, handleLimitExeeded.

The system automatically generates the event and interface classes and the supporting methods:

- LimitExeededEvent class
- LimitExeededListener interface
- fireHandleLimitExeeded method
- addLimitExeededListener method
- removeLimitExeededListener method

To pass an event message, modify the LimitExceededEvent class. Add a public property errorMessage (of type String) to this class and a public constructor:

```
public LimitExceededEvent (java.lang.Object source,
                           java.lang.String errorMessage) {
   super(source);
   setErrorMessage(errorMessage);
}
```

CheckingAccount Class

CheckingAccount is a subclass of BankAccount and therefore inherits all of its properties. In addition it has an overdraft property, which is the maximum amount by which the account is allowed to be overdrawn. If the customer attempts to withdraw an amount that exceeds this limit, a LimitExceededEvent gets fired to provide overdraft protection.

Create the CheckingAccount class as a subclass of BankAccount, then add one additional property on the BeanInfo page:

- ❑ overdraft, type java.math.BigDecimal

Add a constructor and implement two methods:

- ❑ A user-defined constructor to create a new CheckingAccount:

```
public CheckingAccount (String accId, java.math.BigDecimal balance,
                        java.math.BigDecimal overdraft) {
   super(accId, balance);
   setOverdraft(overdraft);
}
```

- ❑ The inherited method, getAccountType:

```
public String getAccountType() {
   return "Checking Account";
}
```

- ❑ The CheckingAccount class has to implement its own withdrawAllowed method. This method checks whether the account balance can be updated on the basis of the balance and overdraft properties. The method notifies other objects of the result of the withdrawal transaction, using the *limitExceeded* event defined in the BankAccount class. The method returns true, if withdrawal of the amount is allowed.

```
public boolean withdrawAllowed(java.math.BigDecimal amount) {
   if (getBalance().add(getOverdraft()).compareTo(amount) < 0) {
      fireHandleLimitExceeded
         ( new LimitExceededEvent(this, "Sorry - your overdraft limit is "
            + getOverdraft().toString()));
      return false;
   }
   return true;
}
```

SavingsAccount Class

SavingsAccount is a subclass of BankAccount. It has an additional minAmount property, which is the minimum amount of the account's balance. If the customer attempts to withdraw an amount that would leave a balance below the limit, a LimitExceededEvent gets fired.

Create the SavingsAccount class as a subclass of BankAccount and add this property:

- ❑ minAmount, type java.math.BigDecimal

Add a constructor and implement two methods:

- ❑ In the constructor of the SavingAccount class, initialize the minAmount property with the value from the ATM account table:

```
public SavingsAccount (String accId, java.math.BigDecimal balance,
                       java.math.BigDecimal minAmount) {
   super(accId, balance);
   setMinAmount(minAmount);
}
```

- ❑ The inherited method, getAccountType:

```
public String getAccountType() {
   return "Savings Account";
}
```

- ❑ The SavingsAccount implements the withdrawAllowed method, considering the minAmount property value, and fires a *LimitExceeded* event defined in the BankAccount class if withdrawal is not allowed. The method returns true if withdrawal of the amount is allowed.

```
public boolean withdrawAllowed(java.math.BigDecimal amount) {
   if (getBalance().subtract(amount).compareTo( getMinAmount() ) < 0) {
      fireHandleLimitExceeded
         (new LimitExceededEvent(this, "Sorry - your minimum balance is "
           +getMinAmount().toString()));
      return false;
   }
   return true;
}
```

Testing the Business Objects

After the beans (classes) for the business objects are created, they are ready for testing. If you ensure now that every bean performs the task for which it is responsible, and the beans interact properly, it is easier to locate problems—if there are problems—as you add the other layers.

To test the business objects, use the Scrapbook window, the Console window, and inspectors. You can support the testing task somewhat if you save test scripts for the various test cases as either files or methods. This approach can be helpful when you have to do regression testing.

Start by implementing a toString method in each bean; it displays some of the attributes and is very handy for testing with the Scrapbook window:

❑ Customer:

```
return "Customer " + getCustomerId() + " " + getGreetings();
```

❑ Card:

```
return "Card " + getCardNumber() + " PIN " + getPinCard();
```

❑ Checking account:

```
return getAccountType() + " " + getAccountId() + " Balance " + getBalance()
       + " Overdraft " + getOverdraft();
```

❑ Savings account:

```
return getAccountType() + " " + getAccountId() + " Balance " + getBalance()
       + " MinAmount " + getMinAmount();
```

❑ Transaction:

```
return "Transaction " + getTransId() + " " + getTransType() + " "
       + getTransAmount();
```

Testing with the Scrapbook Window

To run a sample page of the Scrapbook window, add the code listed in Figure 84 to a Scrapbook page and select one of the business model classes in the *Page->Run in* menu to allow short class names without the package prefix.

```
// set Page->Run in to Customer class in itso.entbk2.atm.model

java.util.Enumeration enum;
java.math.BigDecimal amt = new java.math.BigDecimal(100);

Customer cust1 = new Customer("102", "Mr.", "Olaf", "Graf");
System.out.println(cust1);

Card card1 = new Card("2222222", "2222", cust1);
System.out.println(card1);

SavingsAccount sav1 = new SavingsAccount("102-2001", amt, amt);
CheckingAccount check1 = new CheckingAccount("102-2002", amt, amt);
card1.addAccount(check1);
card1.addAccount(sav1);
enum = card1.getAccounts().elements();
while (enum.hasMoreElements())
  { System.out.println( (BankAccount)enum.nextElement() ); }

Transaction t1 = new Transaction("102-2002", "C", amt);
check1.addTransaction(t1);
System.out.println(t1);

check1.deposit("250");
check1.withdraw("100");
check1.withdraw("351"); // not allowed
check1.withdraw("350");

sav1.deposit("2000");
sav1.withdraw("1000");
sav1.withdraw("1001"); // not allowed
sav1.withdraw("1000");

System.out.println(check1);
enum = check1.getTransactions().elements();
while (enum.hasMoreElements())
  { System.out.println( (Transaction)enum.nextElement() ); }

System.out.println(sav1);
enum = sav1.getTransactions().elements();
while (enum.hasMoreElements())
  { System.out.println( (Transaction)enum.nextElement() ); }
```

Figure 84. Scrapbook Script for Testing the Business Model

6.3 Application Controller

The application controller, or controller for short, works as a manager between the different application layers. Its task is to delegate the work and to control what is happening inside the application. We can describe the flow of information in the controller thus:

- Whenever an event occurs, for example, a user clicks on a button, the user interface layer invokes a method in the controller, and this method invokes another method in the persistence layer.
- The controller also checks the result of methods in the persistence layer where necessary and takes appropriate action, that is, the controller fires an event to notify the user interface layer.

As you can see, you have to:

- Guarantee that the persistence layer implements the methods invoked by the controller. For this purpose, you define a Java Interface that describes the communication between the controller and the persistence layer in a well-defined way. Later, any class implementing this interface can be used as the implementation for persistence services.
- Specify which methods of the controller can be invoked from the user interface layer, and which events get fired from the controller to notify the user interface layer.

Persistence Layer Interface

To guarantee that the persistence layer implements the correct methods, you have to create a new interface, ATMPersistenceInterface, in the itso.entbk2.atm.model package with the methods listed in Table 18. All methods are public abstract and throw a java.lang.Exception to signal error conditions. For clarity, we use the prefix *ext* for these methods.

Table 18. ATM Persistence Interface Methods

Method	**Return Type**	**Parameters**	**Remarks**
extConnect	void	-	Connect to the data source
extDisconnect	void	-	Disconnect from the data source
extGetCard	Card	String cardId	Retrieve the data and construct new Card object
extGetPin	Card	Card	Retrieve the PIN
extGetAccounts	void	Card	Retrieve all accounts of a card and store it in card object
extUpdateBalance	void	BankAccount	Update balance and log changes in transaction history
extGetTransactions	void	BankAccount	Retrieve all transactions and store in account object

Controller Interface

The user interface layer assumes that it can invoke the methods shown in Table 19, and that it is notified by the events listed in Table 20. In other words, our ATM application controller must provide an implementation of these methods and events.

With such a controller, we have an ATM application fully written in Java. It is also conceivable to replace the persistence layers of the application, for example, using CICS, and keep only the user interface. For such an implementation, you have to write another controller that encapsulates the CICS access but presents the user interface layer with the same programming interface as our controller.

Table 19. ATM Application Controller Methods

Method	Return Type	Parameters	Remarks
connect	void	-	Connects to the data source
disconnect	void	-	Disconnects from the data source
getCard	Card	String cardId	Construct new Card object, fire cardFound or cardNotFound event
checkPin	void	Card, String pin	Check the PIN, fire PinCheckedOk or PinCheckedNotOk event
getAccounts	Card	Card	Retrieve all accounts of a card and store it in the card object
deposit	BankAccount	BankAccount, String amount	Deposit amount, update balance, log changes in transaction history, fire newTransaction or DBOutOfSynch event
withdraw	BankAccount	BankAccount, String amount	Withdraw amount if allowed, update balance, log changes in transaction history, fire newTransaction or DBOutOfSynch event
getTransactions	BankAccount	BankAccount	Retrieve all transactions and store in account object

Table 20. ATM Application Controller Events

Event	Event Listener Method	Remarks
cardFound	handleCardFound	Valid card number entered
cardNotFound	handleCardNotFound	Invalid card number entered
pinCheckedOk	handlePinCheckedOk	Valid PIN entered
pinCheckedNotOk	handlePinCheckedNotOk	Invalid PIN entered
newTransaction	handleNewTransaction	Deposit or withdraw succeeds
DBOutOfSynch	handleDBOutOfSynch	Deposit or withdraw fails
limitExceeded	handleLimitExceeded	Withdraw not allowed because limit exceeded

Controller and Persistence Interfaces

Figure 85 shows the interaction between the controller methods and the persistence interface, and the events that may occur.

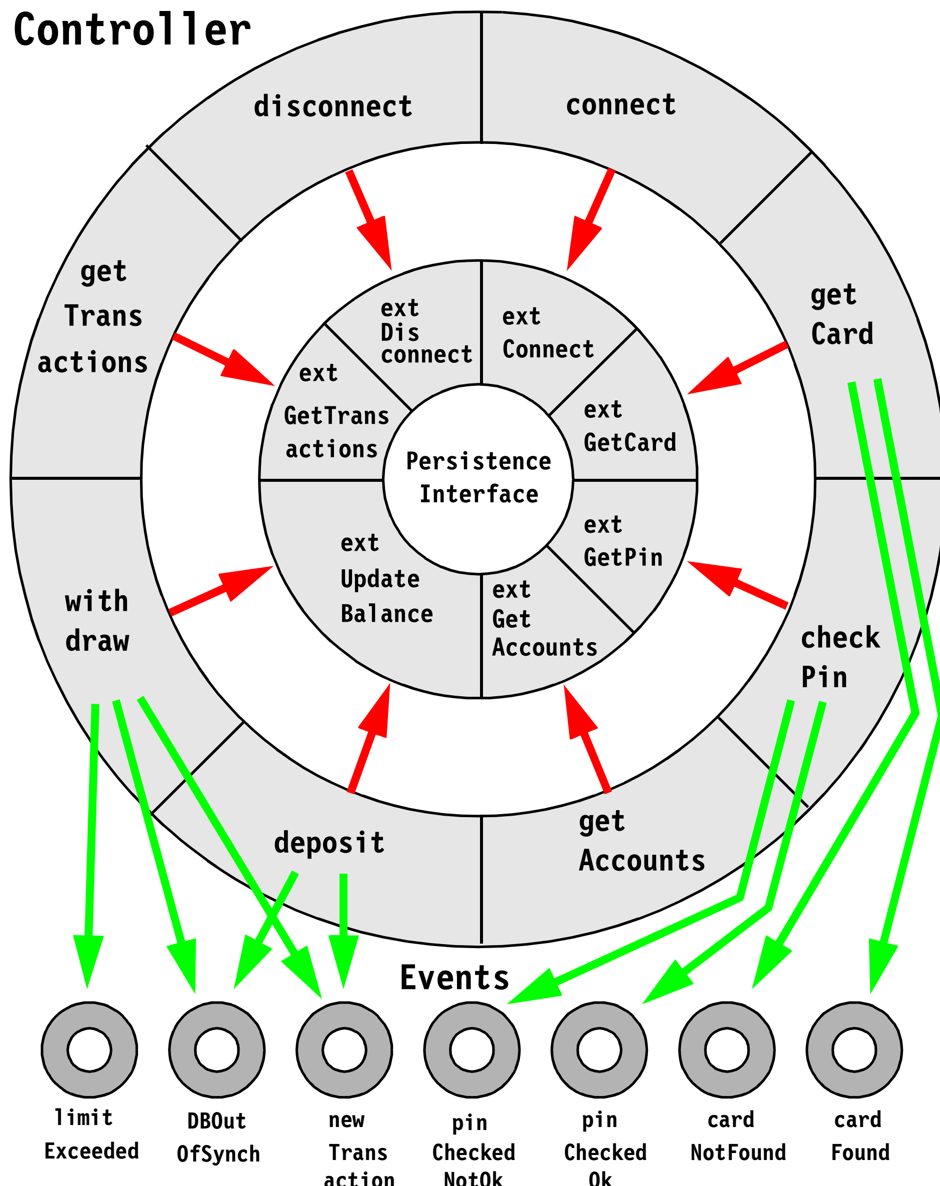


Figure 85. Controller and Persistence Interfaces

Implementing the Controller

To start your work, create a new class, ATMApplicationController, as a subclass of Object in the itso.entbk2.atm.model package.

On the BeanInfo page, add a nonbound property, ATMPersistenceLayer, of type ATMPersistenceInterface. This enables the controller to use every class that implements the interface. Simply replace the initialization of this property in the constructor to plug in another implementation of the persistence layer. In Chapter 7, “ATM Application Persistence Using Data Access Beans” we implement persistence for a relational database and in Chapter 10, “ATM Application with the CICS Connector” we use CICS transactions.

Next, add all the events listed in Table 20. Select *New Listener Interface* in the *Features* menu to create the events. On the first page of the New Event Listener SmartGuide specify the event name and on the second page add the event listener method (prefix the event name with *handle*, for example, handleCardFound).

The pinCheckedOk, pinCheckedNotOk, and limitExceeded events are originally fired by objects of the business layer (Card and BankAccount). The ATMApplicationController has to listen to these events, that is, you have to change the class definition:

```
public class ATMApplicationController implements
      itso.entbk2.atm.model.PinCheckedOkListener,
      itso.entbk2.atm.model.PinCheckedNotOkListener,
      itso.entbk2.atm.model.LimitExceededListener { ...
```

Later, you also have to add code in the checkPin and withdraw method of the ATMApplicationController to add and remove a PinCheckOkListener.

Every event that occurs in the business object must be propagated to the user interface layer. When an event gets fired, it invokes a method to handle this event. Use these methods to propagate the events:

```
public void handlePinCheckedOk
   (itso.entbk2.atm.model.PinCheckedOkEvent event) {
      fireHandlePinCheckedOk ( new PinCheckedOkEvent(this) );
}

public void handlePinCheckedNotOk
   (itso.entbk2.atm.model.PinCheckedNotOkEvent event) {
      fireHandlePinCheckedNotOk ( new PinCheckedNotOkEvent(this) );
}
```

```
public void handleLimitExceeded
   (itso.entbk2.atm.model.LimitExceededEvent event) {
      fireHandleLimitExceeded
         ( new LimitExceededEvent(this, event.getErrorMessage()));
}
```

Note that you have to create a new event in each method, because you have to set the Controller as the source of the event.[7]

Now you can review the scenario of the ATM application and implement the methods from Table 19 on page 156 step by step. Add all the methods on the BeanInfo page as new method features.

- getCard

  The customer enters the card number. If the number is valid, the system prompts for the PIN; otherwise an error message appears. Therefore, you ask the ATMPersistenceLayer to retrieve the card, and, depending on the result, fire an event.

```
public synchronized Card getCard(String cardId) {
   Card newCard = null;
   try {
      newCard = getATMPersistenceLayer().extGetCard(cardId);
      fireHandleCardFound( new CardFoundEvent(this) );
   } catch (Exception e) {
      fireHandleCardNotFound( new CardNotFoundEvent(this) );
      e.printStackTrace(System.out);
   }
   return newCard;
}
```

- checkPin

  The customer enters the PIN. This PIN has to be validated by the system. The PinCheckOk and PinCheckNotOk events are propagated.

```
public synchronized void checkPin(Card card, String pinEntered) {
   card.addPinCheckedOkListener(this);
   card.addPinCheckedNotOkListener(this);
   try {
      getATMPersistenceLayer().extGetPin(card);
      card.checkPin(pinEntered);
   } catch (Exception e) {
      e.printStackTrace(System.out);
   }
   card.removePinCheckedOkListener(this);
   card.removePinCheckedNotOkListener(this);
}
```

[7] For the LimitExceeded event, review “BankAccount Class” on page 147 to learn how to modify the LimitExceededEvent class to add the error message field.

- getAccounts

  If the PIN is valid, the system shows a list of the accounts the card is authorized to access. Later, the customer can select one account to see the details.

```
public synchronized Card getAccounts(Card card) {
   try {
      getATMPersistenceLayer().extGetAccounts(card);
   } catch (Exception e) {
      e.printStackTrace(System.out);
   }
   return card;
}
```

- deposit

  The customer performs a deposit transaction for the selected account. The account is asked to update itself, and the ATMPersistenceLayer is asked to update the data source. Depending on the result, an event gets fired.

```
public synchronized BankAccount deposit(BankAccount account, String amount,
                    Card dummy) {
   try {
      account.deposit(amount);
      getATMPersistenceLayer().extUpdateBalance(account);
      account.commit();
      fireHandleNewTransaction(new NewTransactionEvent(this));
   } catch (Exception e) {
      account.rollback();
      fireHandleDBOutOfSynch(new DBOutOfSynchEvent(this));
      e.printStackTrace(System.out);
   }
   return account;
}
```

- withdraw

  Withdraw is handled in a way similar to deposit. The system performs a check. If the overdraft or minimum balance limit of the account would be reached because of the withdrawal, the request is rejected; otherwise the balance is updated and a new transaction is added to the account history. If the withdrawal was not allowed (the account fires a limitExceeded event and returns false), the controller propagates the event and returns the account unchanged.

```
public synchronized BankAccount withdraw(BankAccount account, String amount,
                                         Card dummy) {
   account.addLimitExceededListener(this);
   try {
      if (account.withdraw(amount) == false)
         return account;
      getATMPersistenceLayer().extUpdateBalance(account);
      account.commit();
      fireHandleNewTransaction(new NewTransactionEvent(this));
   } catch (Exception e) {
      account.rollback();
      fireHandleDBOutOfSynch(new DBOutOfSynchEvent(this));
      e.printStackTrace(System.out);
   }
   account.removeLimitExceededListener(this);
   return account;
}
```

- getTransactions

  The customer might want to look at the transaction history.

```
public synchronized BankAccount getTransactions(BankAccount account,
                                                Card card) {
   try {
      getATMPersistenceLayer().extGetTransactions(account);
   } catch (Exception e) {
      e.printStackTrace(System.out);
   }
   return account;
}
```

- connect and disconnect

  All functions require a connection to the ATM database. The controller implements connect and disconnect methods that pass the request to the persistence layer.

```
public synchronized void connect ( ) {
   try {
      getATMPersistenceLayer().extConnect();
   } catch (Exception e) {
      e.printStackTrace(System.out);
   }
}
```

```
public synchronized void disconnect() {
   try {
      getATMPersistenceLayer().extDisconnect();
   } catch (Exception e) {
      e.printStackTrace(System.out);
   }
}
```

The AtmApplicationController bean is ready. Note that all methods are synchronized to work properly in a multithreaded environment and that exceptions are caught and logged on the console standard output. Later, you may change the controller to have an improved logging mechanism.

6.4 Persistence Layer

The persistence layer is implemented by using new features of VisualAge for Java Enterprise Version 2. In subsequent chapters we concentrate on the implementation of the user interface and the persistence layer, now that the business model and the controller are ready.

Persistence Layer for Testing

For testing you can create a simple memory implementation of the persistence layer. Create a new class named *ATMPersistenceDefault* that implements the ATMPersistenceInterface.

Let VisualAge for Java generate all required methods:

```
public class ATMPersistenceDefault implements ATMPersistenceInterface {...}
```

The only method you must implement is *extGetCard*, where you create a few objects in memory (the method accepts any card ID that is 7 characters long):

```
public Card extGetCard(String cardId) throws Exception {
   if (cardId.length() != 7)
         throw new java.lang.Exception("Card "+cardId+" not found");
   java.math.BigDecimal amt1 = new java.math.BigDecimal(1000);
   java.math.BigDecimal amt2 = new java.math.BigDecimal(100);
   String id = cardId.substring(0,3);
   Customer cust = new Customer(id,"Dr.","VA","Java");
   CheckingAccount acct1 = new CheckingAccount(id+"-6001", amt1, amt2);
   SavingsAccount  acct2 = new  SavingsAccount(id+"-6002", amt1, amt2);
   Card card = new Card(cardId, cardId.substring(0,4), cust);
   card.addAccount(acct1);
   card.addAccount(acct2);
   return card;
}
```

7 ATM Application Persistence Using Data Access Beans

In Chapter 6, "ATM Application Business Model" we explain how the business objects and the application controller work. Now we can move on and create beans to make our data persistent.

One way of handling database access in VisualAge for Java is through data access beans. You should choose this solution if you need access to an existing database. You can reuse the database design and only implement SQL queries to map relational data into Java objects.

© Copyright IBM Corp. 1998

7.1 Persistence Layer Design

Before we start, some preliminary remarks. We use the full power of the Select bean and the SQL Assist SmartGuide to create sophisticated SQL statements. This moves most of the work to the database management system and keeps our persistence layer simple.

From the database perspective, this approach has additional advantages. Only data required by the application is retrieved from the database. Low network traffic relieves the network connection and benefits all network users. We also allow the underlying database management system to optimize access, which speeds up the return of requested data.

We spend no time tuning the database server by creating views, stored procedures, and so forth. Tuning and optimization are very database-specific tasks and contradict our goal of developing highly portable applications.

Our implementation of the persistence layer is based on several beans (see Table 21):

- One nonvisual bean, AtmDB, implementing the ATMPersistenceInterface of the itso.entbk2.atm.model package.

  To provide an implementation of data persistence with data access beans, we assign an instance of AtmDB to the ATMPersistenceLayer property of the ATMApplicationController (see "Implementing the Controller" on page 158).

- Four visual beans, PinCustInfo, Accounts, UpdateBalance, and Transactions, to implement the database transactions needed to satisfy the controller requirements, that is, the methods described by the ATMPersistenceInterface (Table 18 on page 155).

  We provide no implementation for the extGetPin, extConnect, and extDisconnect methods. The extGetPin functionality is covered by extGetCard, and database connect and disconnect are handled automatically by the data access beans.

- One database access class, AtmDatabase, to handle JDBC access to the database. This class is automatically generated when we design the Select beans.

Table 21. ATM Database Beans

ATM Database Bean	Related Persistence Interface Methods	Remarks
AtmDB	all	Implements the persistence interface, delegates to other beans
---	extConnect() extDisconnect()	Automatically handled by the Select beans
PinCustInfo	extGetCard()	Retrieve customer data and PIN to validate the card
---	extGetPin()	Not provided, handled by extGetCard()
Accounts	extGetAccounts()	Return list of accounts for one specified card
UpdateBalance	extUpdateBalance()	Update balance for an account and add new transaction to history log
Transactions	extGetTransactions()	Retrieve transaction history log
AtmDatabase	-	JDBC connection and SQL statements

7.2 Database Access with ATM Database Beans

In this section we describe how to use Select beans as part of the ATM database beans to specify the database access. In "Business Object Creation with ATM Database Beans" on page 178 we describe how to complete the development. You will visually define the steps necessary to return requested business objects from database data or update database contents based on information in the business objects.

All work is done in the **itso.entbk2.atm.databean** package.

PIN Validation

When a user starts a business transaction, he or she is asked for the number of the ATM card. On the basis of the card identification number (card Id), the application retrieves data to create two business objects, a Customer and a Card. The Customer object (title, first name, last name) holds information to welcome the customer, and the data in the Card object (PIN) is needed to validate the card.

In fact, we have to define two business transactions. The first transaction retrieves the Customer data for the card Id entered by the user. Then the customer is asked to enter the PIN. The second transaction retrieves the card information (PIN) from the database to compare both PINs.

What can go wrong? The customer may enter the wrong PIN. In this case, the customer has the opportunity to reenter the PIN, and the application must connect to the database and retrieve the same data again.

We can simplify the process by handling both business transactions as one database transaction and storing the PIN retrieved from the database in the card object. Moreover, this gives us the opportunity to demonstrate a join between tables with data access beans.

To retrieve both the card and the customer data in one database transaction, we join the two tables CARD and CUSTOMER (Figure 86).

```
select ATM.CARD.PIN, ATM.CUSTOMER.TITLE, ATM.CUSTOMER.FNAME,
       ATM.CUSTOMER.LNAME, ATM.CUSTOMER.CUSTID
  from ATM.CARD, ATM.CUSTOMER
 where ((ATM.CUSTOMER.CUSTID = ATM.CARD.CUSTID)
   and  (ATM.CARD.CARDID = :CARDID))
```

Figure 86. Select Statement for Customer Information and PIN Validation

In this SQL statement *:CARDID* represents a host variable, that is, the card ID parameter entered by the user.

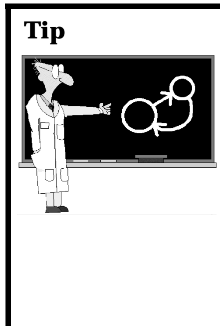
The name of the ATM database bean responsible for retrieving the customer and card data is *PinCustInfo* (in the itso.entbk2.atm.databean package). Create such a class in the Workbench and open the Visual Composition Editor for it. Drop a Select bean from the *Database* category of the *Beans Palette* on the free-form surface, name it *PinCustSelect*, and open the property editor for the *query* property. First select *Properties* in the pop-up menu or double-click on the Select bean to open the Properties window, then select the *query* property in the Properties window and click on the button on the right side of the selected field.

Connection Alias Definition

On the Connection page of the Query property editor you specify the information needed to access the database through JDBC.

We decided to have only one database access class, AtmDatabase, and one connection alias for the ATM application, to facilitate maintaining the code.

Tip



Remember that all Select beans identifying the same connection alias also share the database connection associated with that connection alias, provided that instances of these beans exist in parallel.

Sometimes it is advantageous to have only one database connection per application. If you have many users, however, they may block each other, and each connect, disconnect, commit, and rollback forced for one user affects every user. To avoid such situations, define a pool of connection aliases to assign the aliases to different users and different tasks.

To define a new connection alias, create a new database access class, AtmDatabase, in the itso.entbk2.atm.databeans package. Click on the *New...* button and fill in the required fields (Figure 87).

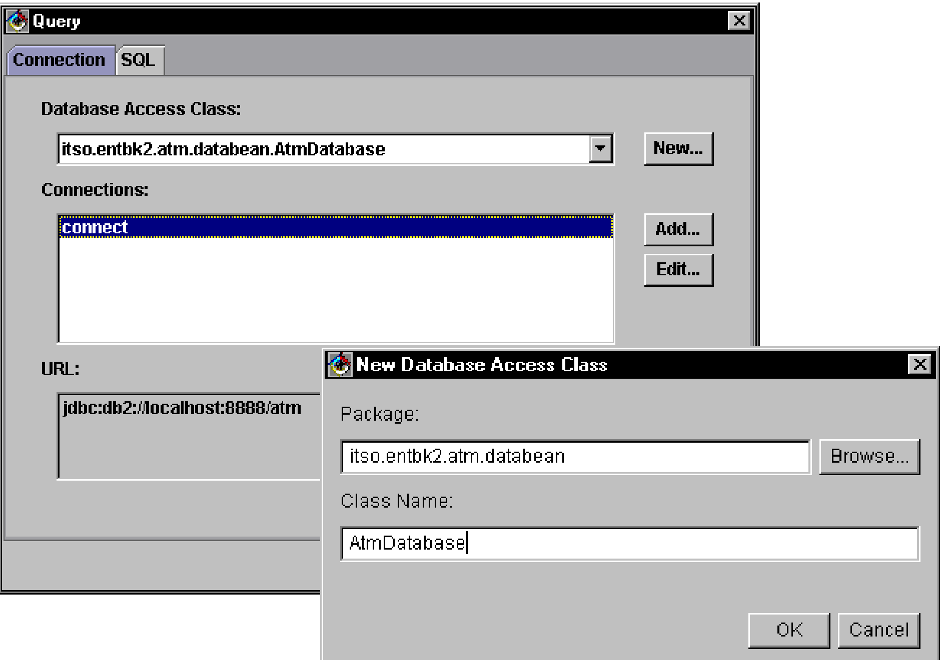


Figure 87. Specification of the Database Access Class

Now create a new connection alias by clicking on the *Add...* button and entering the connection name, JDBC driver, database URL, and user ID and password information (Figure 88).

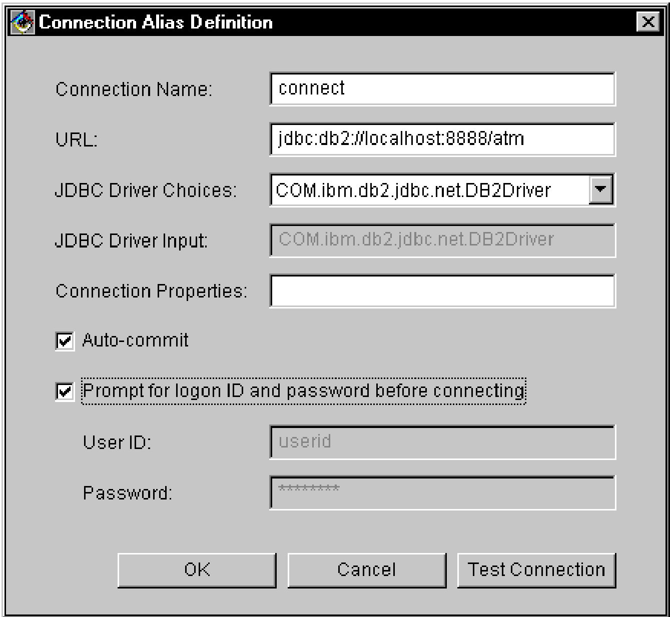


Figure 88. Connection Alias Definition for the ATM Application

We strongly recommend that you test your connection by clicking on the *Test Connection* button and check that you are successful. In this way you can fix problems as early as possible. Click on the *OK* button to finish your work.

SQL Statement Specification

The Select bean has predefined methods to add, update, and delete the current row of a result set. This reduces your work; you only have to define the SELECT statement to retrieve the result set from the database. Switch to the SQL page to specify the necessary SQL statement.

Warning	Keep in mind that when you repeat a query you use the same result set. It does not matter whether you run the query in the same Select bean or in another one. It is important to realize that changes of the result set, for example, of the position of the current row, between different calls of the same query are possible. You should encapsulate all database transactions in synchronized Java methods and test your application thoroughly to detect obscure errors. It is dangerous to trust that any parameter has not changed since the previous query execution.

To keep all methods needed to query the ATM database together in one class and to simplify maintenance tasks, we decided to reuse the database access class we defined for the connection alias.

Choose the AtmDatabase class in the itso.entbk2.atm.databean package from the list of classes in the Database Access Class field. Then click on the *Add...* button to add an SQL specification. In the New SQL Specification window enter getPinCustInfo in the SQL Name field and select *Use SQL Assist SmartGuide* (Figure 89).

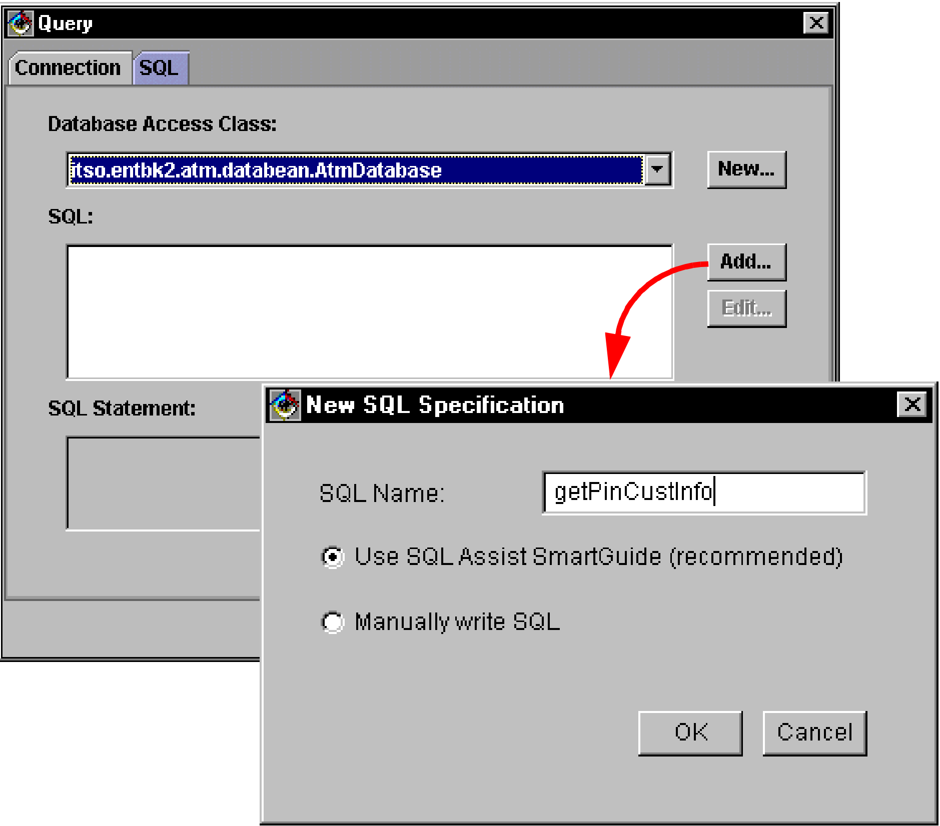


Figure 89. New SQL Specification

The SQL Assist SmartGuide window opens. On the Tables page you can see all tables that are accessible in the ATM database. Click on the *View Schema(s)...* button and make sure that only the ATM schema is displayed, then mark the ATM.CARD and ATM.CUSTOMER tables (Figure 90).

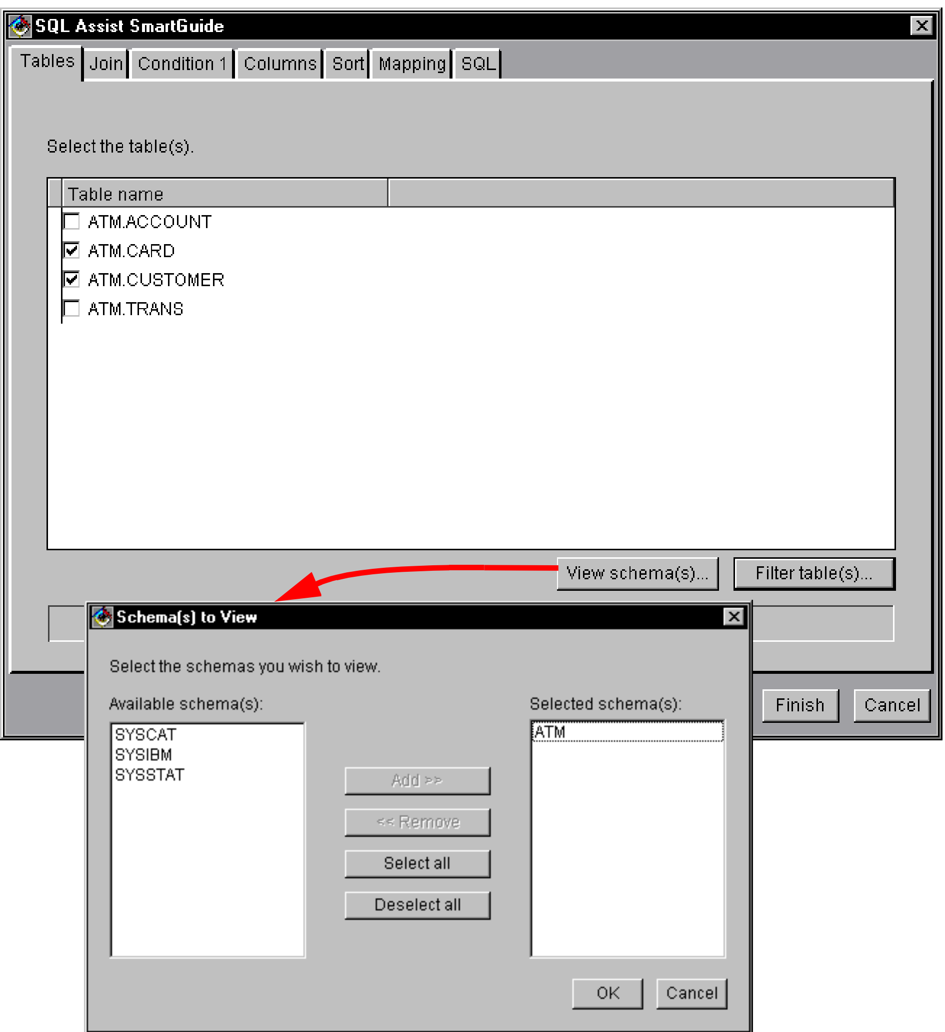


Figure 90. SQL Assist SmartGuide: ATM Table Specification

Switch to the Join page to join the CUSTID column of the ATM.CARD table with the CUSTID column of the ATM.CUSTOMER table. Select both columns to create a link and click on the *Join* button to activate the link. The line becomes red to signal that the join has been successful (Figure 91).

You need an inner join that includes only those rows where the joined fields from both tables are equal. You could also create left and right outer joins; click on the *Options...* button for more information.

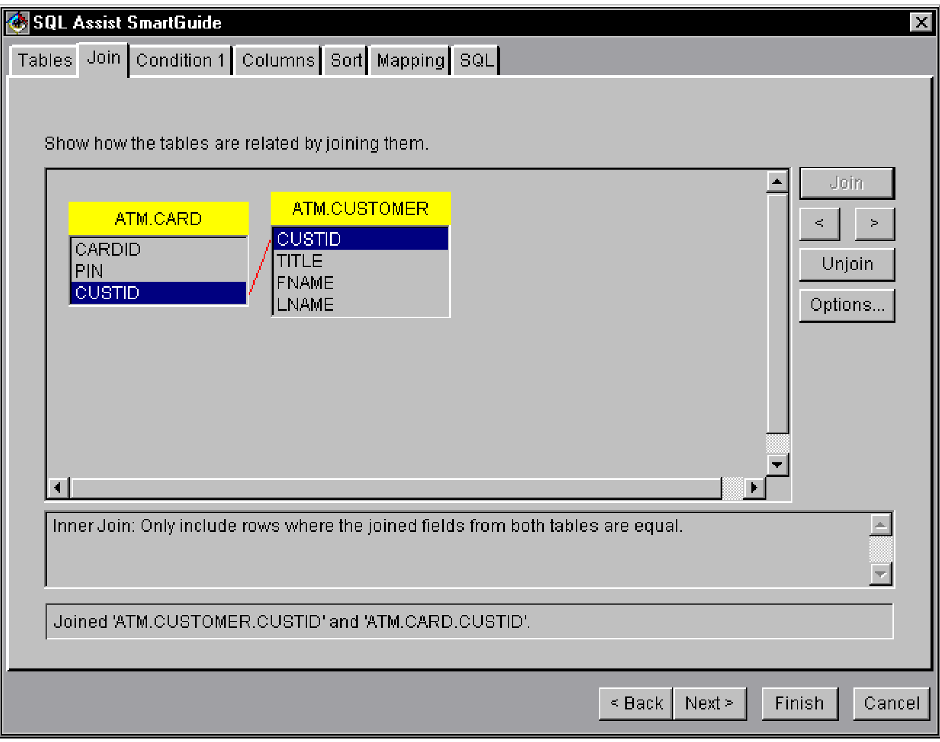


Figure 91. SQL Assist SmartGuide: ATM Join Specification

Use the Condition 1 page to specify that you need all rows of the ATM.CARD table where the CARDID column value is exactly equal to a given value. Choose *ATM.CARD* in the *Selected table(s)* list and *CARDID* in the *Columns* list. Select the *is exactly equal to* operator and enter :CARDID in the *Values* field (Figure 92).

The :CARDID specification indicates that a host variable is used. The real value will be set at run time.

You can click on the *Find...* button to see which card IDs are stored in the ATM database, and you can use the *Find on another column* button to add a Condition 2 page. The Condition 2 page would help to define a second condition, but there is no need for that in the ATM application.

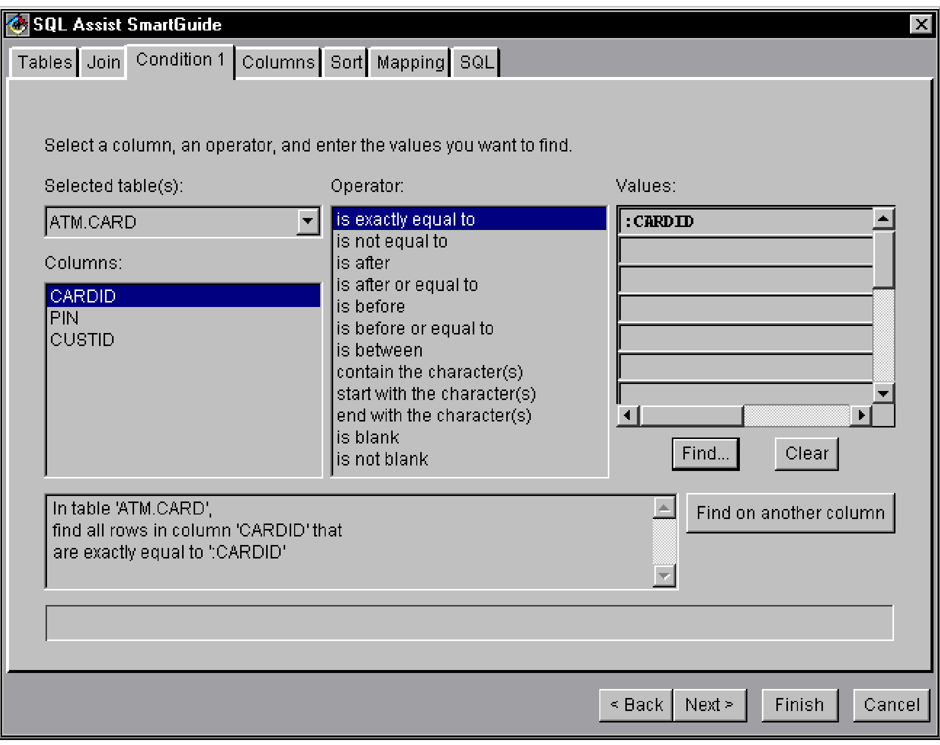


Figure 92. SQL Assist SmartGuide: Condition Specification

Switch to the Columns page and add the following columns:

- ❑ ATM.CARD.CARDID
- ❑ ATM.CARD.PIN
- ❑ ATM.CUSTOMER.TITLE
- ❑ ATM.CUSTOMER.FNAME
- ❑ ATM.CUSTOMER.LNAME
- ❑ ATM.CUSTOMER.CUSTID

Choose the tables in the *Selected table(s)* list and the columns in the *Columns* list. Use the *Add>>*, *<<Remove*, *Select all*, and *Deselect all* buttons to add or remove the columns of selected tables to or from the list in the *Columns to include* panel (Figure 93).

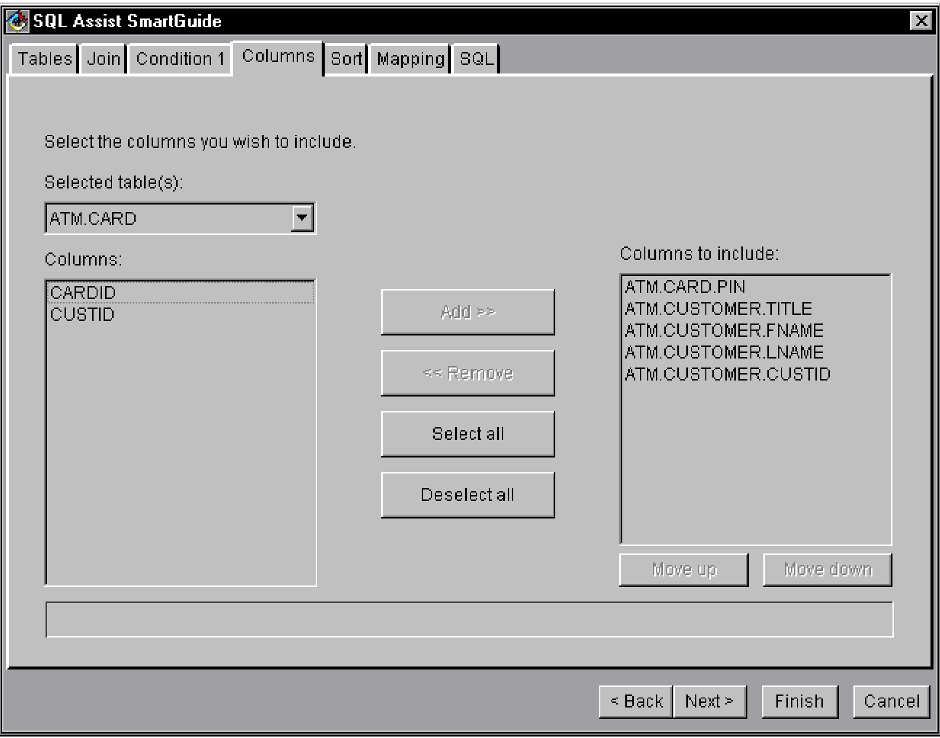


Figure 93. SQL Assist SmartGuide: Columns Specification

There is no need to sort the result tables (there is only one result row) or to change the mapping of the SQL types, that is, you can skip the Sort and the Mapping pages.

Check that the final SQL statement on the SQL page matches Figure 86 on page 166. As you can see, you implemented a sophisticated SQL query without any knowledge of SQL.

We recommend that you spend one minute to test your work by clicking on the *Run SQL....* button. (Note that the Run SQL button is grayed out in some implementations of VisualAge for Java Enterprise Version 2 if a host variable is used.) A Specify Parameter Value(s) window appears to let you specify a value for the :CARDID parameter. Click on the *Run SQL...* button and see the result (Figure 94).

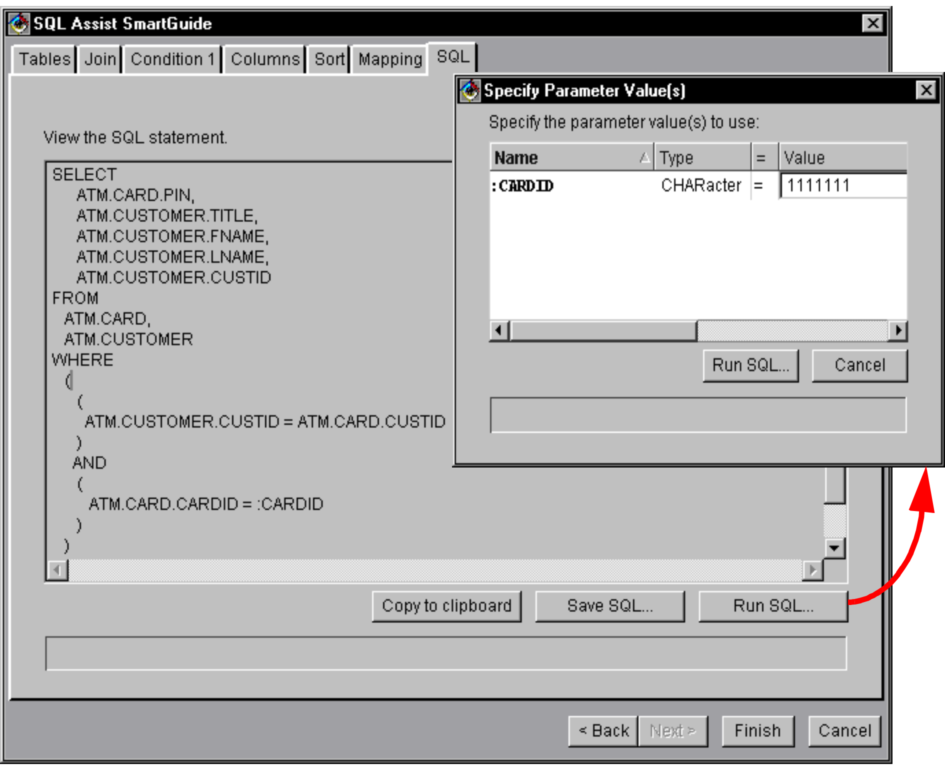


Figure 94. SQL Assist SmartGuide:View and Test the SQL Statement

As a last step, click on the *Finish* button to complete your work. This generates a new getPinCustInfo method in the AtmDatabase class and adds the name of this method to the names list in the SQL field of the SQL page.

Close the Query property editor by clicking on the *OK* button and return to the Visual Composition Editor. Save the bean and close the window.

Now you have seen how you can define both the connection alias and the SQL specification. Use this section as an outline to complete all other SQL queries yourself.

List of Accounts

When the PIN is successfully validated, the ATM application retrieves all accounts for one ATM card based on the card ID. These accounts are shown in a table. The customer can select an account to see the details (account ID, account type, and account balance).

Implement the SQL statement shown in Figure 95.

```
select * from ATM.ACCOUNT where ((ATM.ACCOUNT.CARDID = :CARDID))
```

Figure 95. Select Statement to Retrieve Accounts of a Card

In this statement *:CARDID* is the parameter passed to the SQL query at run time.

Create a class, Accounts, in the itso.entbk2.atm.databean package, open the Visual composition Editor, place a Select bean on the free-form surface, and name it AccountsSelect.

Then open the Query property editor:

❑ On the Connection page, reuse the connection alias definition. Select *itso.entbk2.atm.databeans.AtmDatabase* in the Database Access Class field and *connect* in the Connections field.

❑ On the SQL page, select *itso.entbk2.atm.databeans.AtmDatabase* as database access class, add a new SQL specification, and name it getAccounts.

❑ In the SQL Assist SmartGuide, specify the following values:

- Tables page: select *ATM.ACCOUNT* table
- Condition page: column *CARDID* is exactly equal to :CARDID

Validate that the SQL statement on the SQL page is correct and click on *OK* to finish your work. Return to the Visual Composition Editor, save the bean, and close the window.

Debit and Credit Transactions

The customer can choose one account from the account table to start the debit and credit transaction. For that, the application has to perform two steps:

❑ Update the balance for an account in the ACCOUNT table.

First, retrieve the result set. Implement the SQL statement shown in Figure 96. Host variable *:ACCID* is a parameter passed to the SQL select at run time.

```
select ATM.ACCOUNT.BALANCE from ATM.ACCOUNT
where ((ATM.ACCOUNT.ACCID = :ACCID))
```

Figure 96. Select Statement to Update the Account Balance

Second, update the current row of the result set, using the predefined updateRow method in the Select bean.

❑ Add a new transaction to the transaction history table, TRANS.

First execute an SQL statement to retrieve all transactions (Figure 97).

```
select * from ATM.TRANS
```

Figure 97. Select Statement to Retrieve All Transactions

Actually we will retrieve only a few rows and then add a new row. Select beans require a successful retrieve before an insert can be performed. We add a new row to the result set through the predefined newRow method of the Select bean. Update the values of this new row, using the setColumnValue method, and apply the changes to the database, using the updateRow method.

Create a class, UpdateBalance (in the itso.entbk2.atm.databean package), open the Visual Composition Editor, and place two Select beans on the free-form surface. Rename the beans as UpdateBalanceSelect and AddTransactionSelect.

Open the Query property editor for each bean:

❑ On the Connection page, use the same connection alias definition for both beans. Select *itso.entbk2.atm.databeans.AtmDatabase* in the Database Access Class field and *connect* in the Connections field.

❑ On the SQL page, select *itso.entbk2.atm.databeans.AtmDatabase* as the database access class:

1. UpdateBalanceSelect

   Add a new SQL specification named updateBalance and specify the following values in the SQL Assist SmartGuide:

   - Tables page: select *ATM.ACCOUNT* table
   - Condition page: column *ACCID* is exactly equal to :ACCID
   - Columns page: select *BALANCE* column

2. AddTransactionSelect

   Add a new SQL specification named addTransaction and specify the following values in the SQL Assist SmartGuide:

   - Tables page: select *ATM.TRANS* table

   To limit the number of rows retrieved, open the properties of the AddTransactionSelect bean and set some of the expert features:

- maximumPacketsInCache: 1
- packetSize: 2
- maximumRows: 0

Validate each SQL statement on the SQL page, then click on *OK* to finish your work. Return to the Visual Composition Editor, save the bean, and close the window.

Transaction History

The ATM application provides information about all transactions related to the specified account. This transaction history includes, for each transaction, a transaction ID (time stamp), an account ID, a transaction type, and a transaction amount.

To retrieve all transactions of an account from the TRANS table, you have to implement the SQL statement shown in Figure 98. In this statement *:ACCID* is the parameter account ID to specify an account.

```
select * from ATM.TRANS where ((ATM.TRANS.ACCID = :ACCID))
order by ATM.TRANS.TRANSID
```

Figure 98. Select Statement to Retrieve the Transactions of an Account

Create a class, Transactions (in the itso.entbk2.atm.databean package), open the Visual Composition Editor, place a Select bean on the free-form surface, and name it TransactionsSelect.

In the Query property editor:

❑ On the Connection page, reuse the connection alias definition. Select *itso.entbk2.atm.databeans.AtmDatabase* in the Database Access Class field and *connect* in the Connections field.

❑ On the SQL page, select *itso.entbk2.atm.databeans.AtmDatabase* as the database access class, add a new SQL specification, and name it getTransactions.

❑ In the SQL Assist SmartGuide, specify the following values:

- Tables page: select *ATM.TRANS* table
- Condition page: column *ACCID* is exactly equal to :ACCID
- Sort page: select *TRANSID* column

Validate the SQL statement on the SQL page and click on *OK* to finish your work. Return to the Visual Composition Editor, save the bean, and close the window.

7.3 Business Object Creation with ATM Database Beans

The ATM database beans you have created form the interface between the ATM database and the Java application, acting as object wrappers around result tables returned by Select beans. They are also responsible for creating and initializing business objects with the data in the result tables. The objects are passed to the AtmDB bean that performs a final check of the results.

For a good object-oriented design, we only use the constructors of the business objects to create new objects and the get methods to ask for the values of the property fields. The only exception is when we have to manage lists of accounts and transactions. The Card class has two methods, clearAccounts and addAccount, and the Account class has two methods, clearTransactions and addTransaction, to hide the internal implementation.

In this section we complete the four ATM database beans, using the Visual Composition Editor.

PIN Validation

Figure 99 shows the visual composition of the PinCustInfo bean.

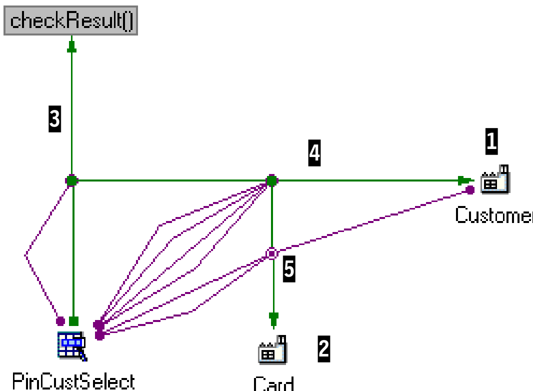


Figure 99. Visual Composition of PinCustInfo Bean

Perform the following steps in the Visual Composition Editor:

❑ The PinCustInfo bean is used to create two new business objects. Add two factories from the Others category of the Beans palette, change the types

to Customer and Card (in the itso.entbk2.atm.model package) and rename them to Customer (1) and Card (2).

- You need external access to tree features of the bean, that is, you have to promote these features:
  - *Parm_CARDID_String* property (PinCustSelect bean) as CardId
  - *execute* method (PinCustSelect bean) as execute
  - *this* property (Card factory) as Card

  To change the name of a feature, make sure that it is in the list of promoted features in the Promote Feature SmartGuide, then double-click on the name in the *Promote Name* column. The field becomes white to indicate that you can make your change.

- Make an event-to-code connection from the *executed* event of the Select bean to the free-form surface (3), and add this code:

```
public void checkResult(int numRows) throws java.lang.Exception {
   if (numRows < 1) {
      setCard(null);
      throw new java.lang.Exception("Result Set is empty");
   }
}
```

  Pass the *numRows* property of the Select bean as a parameter to avoid creating a card bean without any data.

- To construct a new customer object, connect the *normalResult* of the previous connection (3) to the constructor *Customer(java.lang.String, ...)* of the Customer factory and pass the *CUSTOMER.CUSTID_String*, *CUSTOMER.TITLE_String*, *CUSTOMER.FNAME_String* and *CUSTOMER.LNAME_String* properties of the Select bean as parameters (4).

- Connect the *normalResult* of the previous connection to the constructor *Card(java.lang.String, ...)* of the Card factory and pass the *Parm_CARDID_String* and *CUSTOMER.PIN_String* properties of the Select bean and the *this* property of the Customer factory as parameters. This constructs a new card object (5).

Save the bean to generate the code and close the Visual Composition Editor.

List of Accounts

Figure 100 shows the visual composition of the Accounts bean.

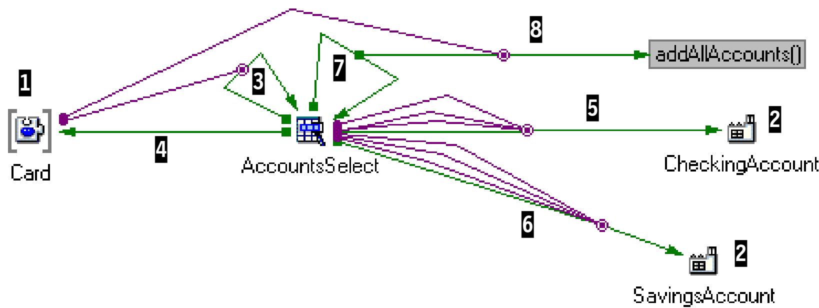


Figure 100. Visual Composition of Accounts Bean

Perform the following steps in the Visual Composition Editor:

❑ To pass a card object to the Accounts bean and access it inside the Visual Composition Editor, add a variable of type itso.entbk2.atm.model.Card to the free-from surface and name the variable Card (1).

❑ You have to create new checking and savings account objects. Add two factories, change their types to CheckingAccount and SavingsAccount (in itso.entbk2.atm.model), and name them CheckingAccount and SavingsAccount (2).

❑ You need external access to the features, therefore, promote these bean features:

- *this* property (Card variable) as Card
- *this* property (CheckingAccount factory) as CheckingAccount
- *this* property (SavingsAccount factory) as SavingsAccount
- *execute* method (Select bean) as execute

❑ Connect the *aboutToExecute* event (AccountsSelect bean) to the *Parm_CARDID* property (same bean) and pass the *cardNumber* (Card variable) as a parameter (3).

This connection sets the parameter for the Select bean from a Card property, every time, before it executes the query. The advantage of an event connection instead of a property-to-property connection is that no exception is fired during initialization of the Accounts bean, and you have control when the parameter update occurs.

❑ Connect the *aboutToExecute* event (Select bean) to the *clearAccounts* (Card variable) (4) to discard all old accounts before the new result set is retrieved.

❑ For every row retrieved create both a new checking and savings account object. You have to write code later to see which one is the correct one to be added to the card.

Connect the *ACCOUNT.ACCID* event (AccountsSelect bean) to the *CheckingAccount(String,BigDecimal,BigDecimal)* constructor of the CheckingAccount factory and pass *ACCOUNT.ACCID_String, ACCOUNT.BALANCE,* and *ACCOUNT.OVERDRAF* as parameters (5).

Connect the *ACCOUNT.ACCID* event (AccountsSelect bean) to the *SavingsAccount(String,BigDecimal,BigDecimal)* constructor of the SavingsAccount factory and pass *ACCOUNT.ACCID_String, ACCOUNT.BALANCE,* and *ACCOUNT.MINAMT* as parameters (6).

❑ It is necessary to reset the currentRow property of the Select bean. Connect the *executed* event to the *firstRow* method (AccountsSelect bean) (7).

❑ Now you have to loop through all the retrieved accounts and add either the checking or the savings account to the card variable.

Make an event-to-code connection from the *normalResult* of the previous connection (7) to the free-form surface (8) and enter the following code to add the accounts to the Card object:

```
private void addAllAccounts(itso.entbk2.atm.model.Card card)
                throws com.ibm.db.DataException {
   for (int row=0; row < getAccountsSelect().getRowCount(); row++) {
      if ( getAccountsSelect().getColumnValueToString("ACCOUNT.ACCTYPE").
             trim().equalsIgnoreCase("C") )
         card.addAccount( getCheckingAccount() );
      else
         card.addAccount( getSavingsAccount() );
      getAccountsSelect().nextRow();
   }
}
```

Pass the *this* property (Card variable) as a parameter. This code checks the account type, adds the appropriate account to the card, and gets the next row.

Save the bean to generate the code and close the Visual Composition Editor.

Debit and Credit Transactions

Figure 101 shows the visual composition of the UpdateBalance bean.

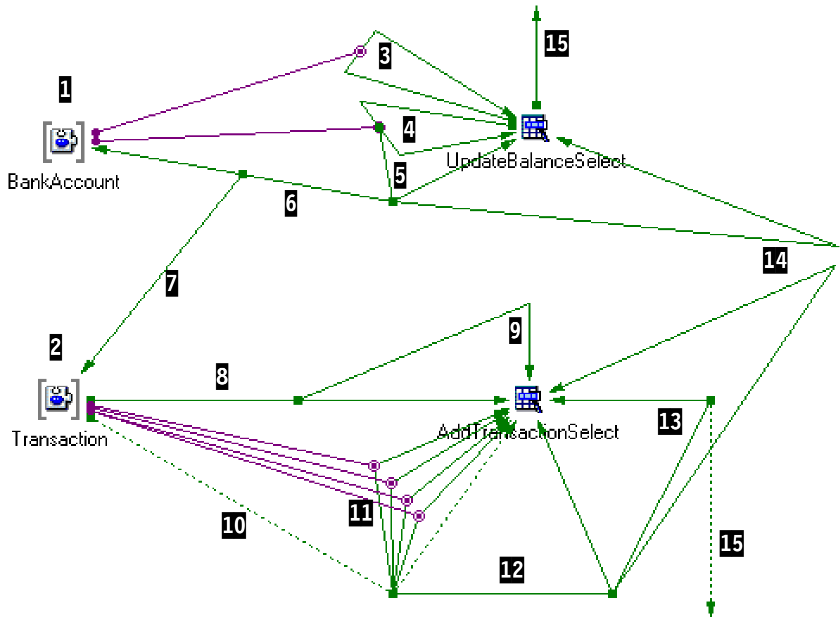


Figure 101. Visual Composition of UpdateBalance Bean

Perform the following steps in the Visual Composition Editor:

❑ Add two variables, one of type BankAccount and one of type Transaction (itso.entbk2.atm.model) to the free-from surface, name them BankAccount (1) and Transaction (2). BankAccount is necessary to pass an account to the UpdateBalance bean, and the Transaction variable refers to the last transaction of the bank account.

❑ Because you need external access, you have to promote the following bean features:

- *this* property (BankAccount variable) as BankAccount
- *this* property (Transaction variable) as Transaction
- *execute* method (UpdateBalanceSelect bean) as execute
- *commit* method (UpdateBalanceSelect bean) as commit
- *rollback* method (UpdateBalanceSelect bean) as rollback

❑ Set the accountId each time before executing the UpdateBalance query. Connect the *aboutToExecute* event to the *Parm_ACCID_String* property (UpdateBalanceSelect bean) (3) and pass the *accountId* property (BankAccount variable) as a parameter.

- Connect the *executed* event (UpdateBalanceSelect bean) to the *ACCOUNT.BALANCE* property (4) and pass the *balance* property (BankAccount variable) as a parameter. Connect the *normalResult* of this connection to the *updateRow* method (5). These steps are necessary to update the balance in the ACCOUNT table.
- Connect the *normalResult* of the updateRow connection (5) to the *getLastTransaction* method (BankAccount variable) (6). Connect the *normalResult* of this connection to the *this* property of the Transaction variable (7). After executing these steps, the Transaction variable refers to the last transaction stored in BankAccount. This transaction was created by the deposit or withdraw method.
- Connect the *this* (Transaction) to the *execute* method (AddTransactionSelect bean) (8). Connect the *normalResult* of this connection to the *firstRow* method (9). This retrieves a few rows from the TRANS table.
- Connect the *this* (Transaction) to the *newRow* method (AddTransactionSelect bean) (10) Leave the parameter as *false*. Make four connections from the normalResult of this connection to the *TRANS.TRANSID, TRANS.TRANSAMT, TRANS.TRANSTYPE_String,* and *TRANS.ACCID_String* (AddTransactionSelect bean) attributes. Pass the appropriate attributes of the Transaction variable as parameters (11). These steps prepare a new row in the TRANS table.
- Connect the *normalResult* of the newRow connection (10) to the *updateRow* method (AddTransactionSelect bean) (12). This step inserts the row into the TRANS table.
- Connect the *normalResult* of the updateRow connection to the *commit* method (AddTransactionSelect bean) (13). The updates to the balance and the new transaction are committed to the database.
- You have to roll back the updates if anything goes wrong. Connect the *exceptionResult* of the two updateRow connections (5 and 12) to the *rollback* method of the Select bean (14).

  Note that by default Auto-commit is enabled (Figure 88 on page 168). This could lead to the situation where the balance update is performed but the insert of the new transaction fails. Go back to the definition of the connection and deselect the Auto-commit checkbox.
- Add a new public nonbound int property *errorCode* to the UpdateBalance bean and set it 0 or 1, depending on whether the database update was successful or not (connections 15). Add a connection from the *aboutToExecute* event (UpdateBalanceSelect bean) to the *errorCode* property and set the value to 1 (this sets the initial value as unsuccessful). Add a connection from the *normalResult* of the commit method (13) to the

errorCode property and set the value to 0 (indicating successful). You only set a successful return code when the final commit has been executed.

Save the bean to generate the code and close the Visual Composition Editor.

Transaction History

Figure 102 shows the visual composition of the Transactions bean.

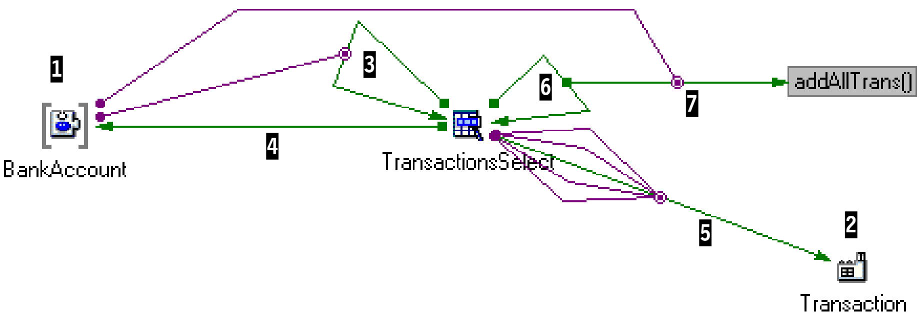


Figure 102. Visual Composition of Transactions Bean

Perform the following steps in the Visual Composition Editor. These steps are very similar to those you performed for the Accounts bean, so a short description will suffice:

❑ Add a variable of type itso.entbk2.atm.model.BankAccount to the free-from surface (1), name it BankAccount. Add a factory of type Transaction to the free-form surface (2). The factory is used to create Transaction objects to be added to the BankAccount.

❑ Promote the following bean features:

- *this* property (BankAccount variable) as BankAccount
- *this* property (Transaction factory) as Transaction
- *execute* method (Select bean) as execute

❑ Connect the *aboutToExecute* event (TransactionsSelect bean) to the *Parm_ACCID_String* property (TransactionsSelect bean) and pass the *accountId* (BankAccount variable) as a parameter (3).

❑ Connect the *aboutToExecute* event (TransactionsSelect bean) to the *clearTransactions* (BankAccount variable) (4).

❑ For every row retrieved create a Transaction object. Connect the *TRANS.ACCID* event (TransactionsSelect bean) to the *Transaction(String,..,..,..)* constructor of the Transaction factory and pass the four attributes, *TRANS.TRANSID*, *TRANS.TRANSAMT*,

TRANS.ACCID_String, and *TRANS.TRANSTYPE_String,* as parameters (5).

- ❑ Connect the *executed* event to the *firstRow* method (TransactionsSelect bean) (6). Make an event-to-code connection from the *normalResult* of this connection to the free-form surface (7) and add this code:

```
private void addAllTrans(itso.entbk2.atm.model.BankAccount account)
             throws com.ibm.db.DataException {
   for (int row=0; row < getTransactionsSelect().getRowCount(); row++) {
      account.addTransaction( getTransaction() );
      getTransactionsSelect().nextRow();
   }
}
```

Pass the *this* property (BankAccount variable) as a parameter.

Save the bean to generate the code and close the Visual Composition Editor.

7.4 Implementing the Persistence Interface

Now you create the link between the ATM database access beans and the application controller.

AtmDB Bean

The AtmDB bean works as a mediator between the application controller and the ATM database beans. It implements the methods the controller expects from the persistence interface and calls the related methods in the ATM database beans to handle the controller requests.

The AtmDB provides the implementation of the persistence layer. Add a class AtmDB derived from java.lang.Object to the itso.entbk2.atm.databean package and add to it the itso.entbk2.atm.model.ATMPersistenceInterface interface.

Make sure that the *Methods which must be implemented* checkbox is marked on the second page of the Create Class SmartGuide (the checkbox is marked by default). A new class, AtmDB, is generated with stubs for all methods to implement.

Add four read-only, nonbound properties—pinCustInfo, accounts, updateBalance, and transactions—to the AtmDB bean, one for each ATM database bean:

```
private PinCustInfo fieldPinCustInfo = new PinCustInfo();
private Accounts fieldAccounts = new Accounts();
private UpdateBalance fieldUpdateBalance = new UpdateBalance();
private Transactions fieldTransactions = new Transactions();
```

Implementing the Methods of the Interface

Now you implement the code for the methods of the ATMPersistence interface.

extGetCard

```
public itso.entbk2.atm.model.Card extGetCard(String cardId)
            throws java.lang.Exception {
   getPinCustInfo().setCardId(cardId);
   getPinCustInfo().execute();
   itso.entbk2.atm.model.Card card = getPinCustInfo().getCard();
   if (card == null)
      throw new java.lang.Exception("Card "+cardId+" not found");
   return card;
}
```

extGetAccounts

```
public void extGetAccounts(itso.entbk2.atm.model.Card card)
               throws java.lang.Exception {
   getAccounts().setCard(card);
   getAccounts().execute();
   if (card.getAccounts().size() == 0) throw new java.lang.Exception
         ("Accounts for card "+card.getCardNumber()+" empty");
}
```

extUpdateBalance

```
public void extUpdateBalance(itso.entbk2.atm.model.BankAccount account)
               throws java.lang.Exception {
   getUpdateBalance().setErrorCode(0); // done in bean as well
   getUpdateBalance().setBankAccount(account);
   getUpdateBalance().execute();
   if (getUpdateBalance().getErrorCode() != 0) {
      getUpdateBalance().rollback();   // just to be safe
      throw new java.lang.Exception
         ("Updating the account or transaction log failed");
      }
   else getUpdateBalance().commit();  // just to be safe
}
```

extGetTransactions

```
public void extGetTransactions(itso.entbk2.atm.model.BankAccount account)
               throws java.lang.Exception {
   getTransactions().setBankAccount(account);
   getTransactions().execute();
   if (account.getTransactions().size() == 0) throw new java.lang.Exception
         ("Transactions for account "+account.getAccountId()+" empty");
}
```

As you can see, each of the above methods passes a parameter (cardId, card, or account) to the related ATM database bean, calls the execute method, and gets the result. Then it checks the result of the execution.[8] Whenever something goes wrong, an exception gets fired.

The remaining methods of the interface do not require an implementation:

- Database connect and disconnect are performed automatically by the data access beans.
- extGetPin is not necessary because the PIN is stored in the card after extGetCard and can be checked by the ATMApplicationController.

[8] This check is necessary, because there is no easy way to forward exceptions from inside the ATM database bean. The beans are generated visually, that is, all exceptions are caught inside the ATM database bean and handled by the handleException method. In this method you could fire an event or set a returnCode, as shown in the UpdateBalance bean.

Testing the Implementation of the Persistence Interface

Most of the functionality of the AtmDB bean can be tested in the Scrapbook by creating a controller (ATMApplicationController) and assigning an AtmDB bean for the implementation of the interface.

By calling some of the methods of the controller you can debug the functions of the AtmDB bean. Figure 103 shows a sample Scrapbook script.

```
// set Page->Run in to ATMApplicationController in itso.entbk2.atm.model

java.util.Enumeration enum;
BankAccount acct1;

ATMApplicationController ctl = new ATMApplicationController();
itso.entbk2.atm.databean.AtmDB atmdb = new itso.entbk2.atm.databean.AtmDB();
ctl.setATMPersistenceLayer(atmdb);

Card card1 = ctl.getCard("1111111");
System.out.println(card1);
boolean pinok = ctl.checkPin(card1,"1111");
System.out.println("PIN OK " + pinok);

ctl.getAccounts(card1);
enum = card1.getAccounts().elements();
while (enum.hasMoreElements())
  { System.out.println( (BankAccount)enum.nextElement() ); }

acct1 = (BankAccount)card1.getAccount("101-1001");
//acct1 = ctl.deposit(acct1,"180");
//acct1 = ctl.withdraw(acct1,"240");
acct1 = ctl.getTransactions(acct1);
System.out.println(acct1);
enum = acct1.getTransactions().elements();
while (enum.hasMoreElements())
  { System.out.println( (Transaction)enum.nextElement() ); }

acct1 = (BankAccount)card1.getAccount("101-1003");
acct1 = ctl.deposit(acct1,"400");
acct1 = ctl.getTransactions(acct1);
System.out.println(acct1);
enum = acct1.getTransactions().elements();
while (enum.hasMoreElements())
  { System.out.println( (Transaction)enum.nextElement() ); }
```

Figure 103. Scrapbook Script for Testing the AtmDB Bean

7.5 Preparation for Servlet Usage

In Chapter 9, "ATM Application Using Servlets" we use this ATM persistence implementation. To make the DB2 connection work without being prompted for a user ID on the Web server, you must change the connection definition to use a fixed user ID and password specification.

Figure 88 on page 168 shows the initial connection specification. Open any of the ATM database beans, for example, PinCustInfo, and edit the connection to specify a valid user ID and password, and deselect the Prompt checkbox. At the same time make sure that the Auto-commit checkbox is not selected (Figure 104).

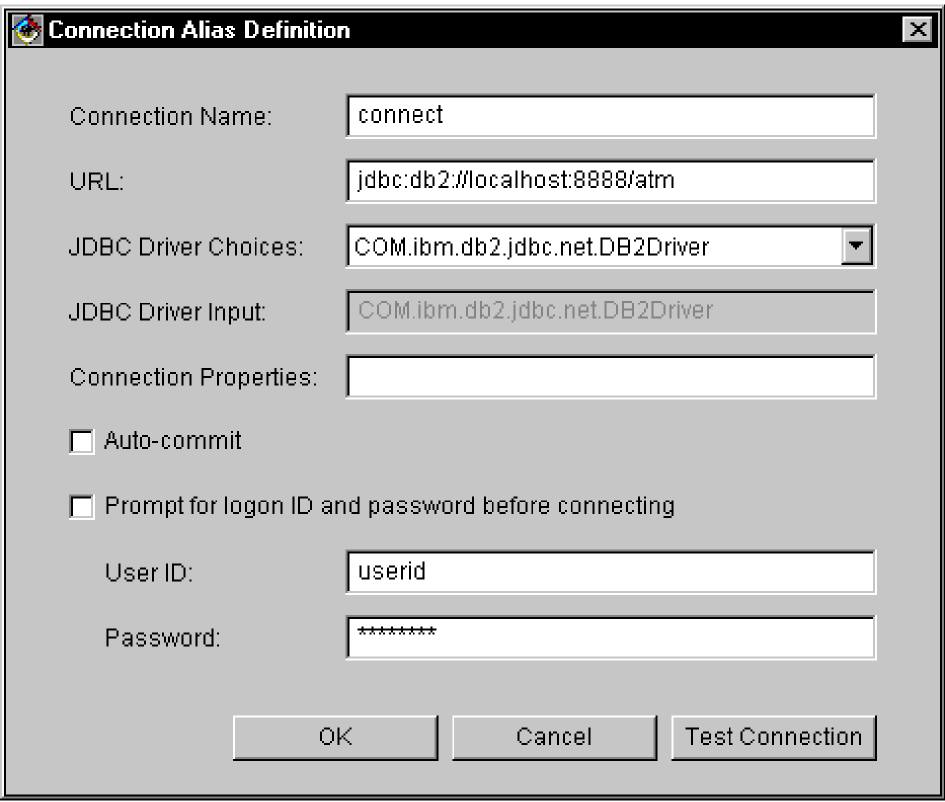


Figure 104. Updated Connection Specification for the ATM Application

8 Swing GUI for ATM Application

In this chapter we develop a GUI for the ATM application, using the Java Swing classes, also known as the Java Foundation Classes (JFC).

We basically copy the approach used for the RMI design in the redbook *Application Development with VisualAge for Java Enterprise*, SG24-5081.

We already have the business model and the database implementation of the ATM application, so all we need to develop is a GUI that can run as an applet or application.

We do not explain the Swing classes in any detail because many books about Swing are available. We use Swing to create a simple GUI that is more attractive than an AWT GUI.

© Copyright IBM Corp. 1998

8.1 Design of the GUI Application

Figure 105 shows the basic layout of the application. We use one main panel with a card layout of four panels:

- ❑ Card panel to enter the ATM card number
- ❑ PIN panel to enter the PIN for the ATM card
- ❑ Select account panel to select an account that belongs to the card
- ❑ Transaction panel for deposit and withdrawal transactions

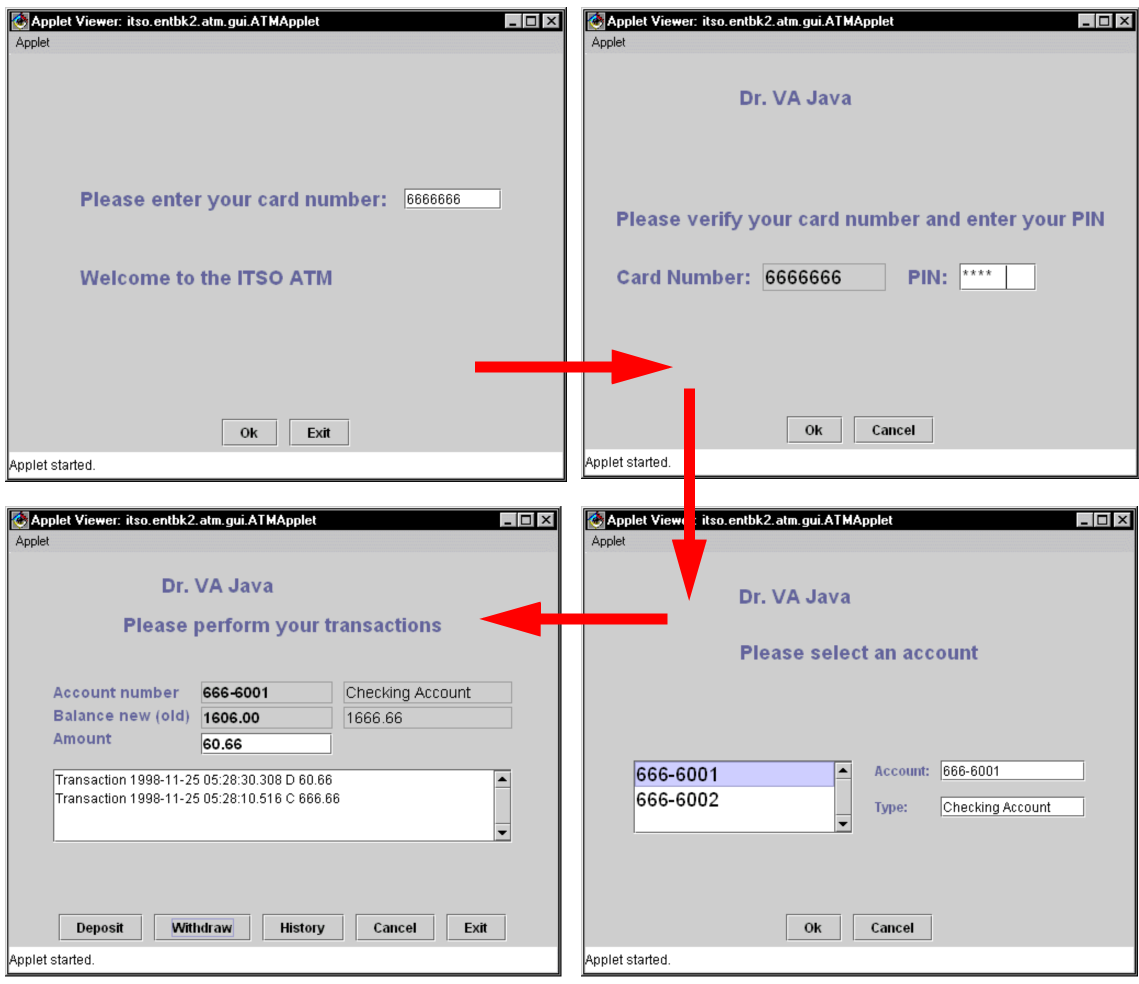


Figure 105. ATM Application Panels

Application Controller

The heart of the GUI application is the ATM application controller that performs all the processing together with an implementation of the persistence interface (Figure 106).

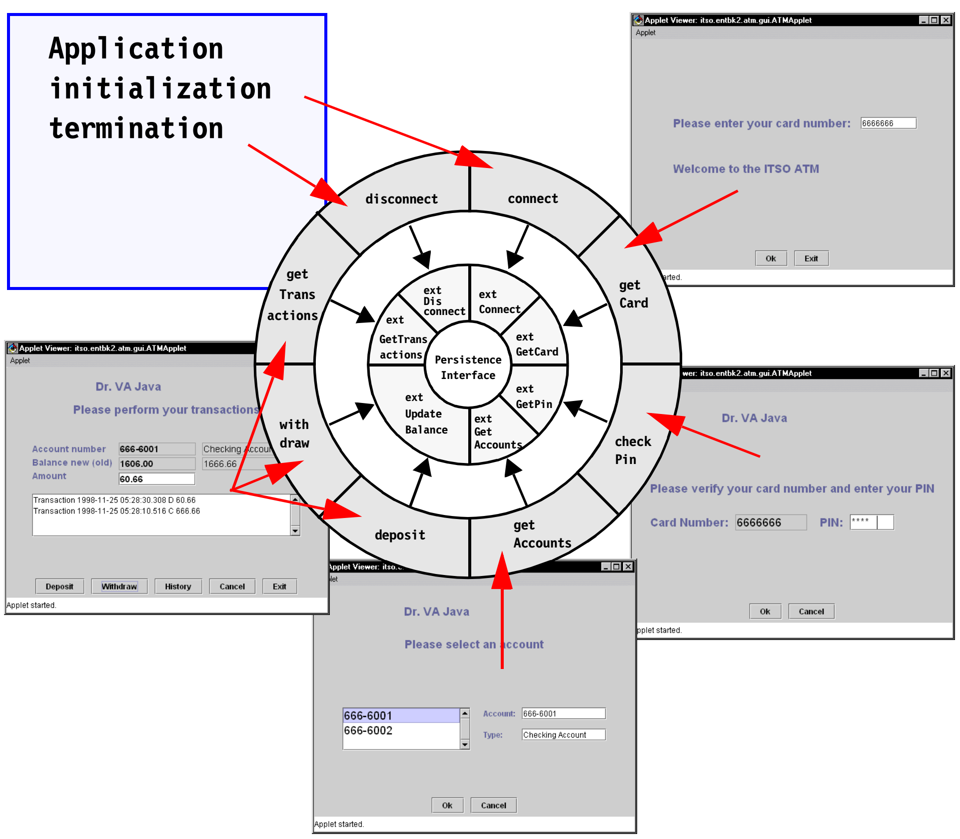


Figure 106. GUI Application with Application Controller

Panel Design

The four panels have the same basic layout:

- ❑ A container panel with a border layout
- ❑ A greeting panel in the North area (flow or grid bag layout)
- ❑ A button panel in the South area (flow layout)
- ❑ A processing panel in the center (grid bag layout)

Nonvisual Variables

All four panels have the same four nonvisual variables:

- Controller (of type ATMApplicationController), used for application processing
- Card (of type Card), used to display the card number and customer name
- Main (of type JPanel), required as parent in the calls to CardLayout
- CardLayout (of type CardLayout), used to switch to the next or previous panel

The select account panel and the transaction panel contain one additional variable that represents the current bank account.

Set Up of the Variables

The Controller, Card, and Main variable are set up at the start of the application and the CardLayout variable is prepared on each subpanel:

- The Controller is allocated in the main panel and assigned to the variables on each subpanel.
- A Card variable is set up in the main panel and connected to all subpanels. An actual card object is retrieved in the card panel.
- The Main variable represents the main panel itself and is passed on to all subpanels.
- The CardLayout is the layout manager property of the main panel and is allocated in each subpanel. To set the CardLayout variable, connect the *this* event of the Main variable to the *this* property of the CardLayout variable and pass the *layout* property as a parameter.

Promotion of the Variables

The *this* property of the Controller, Card, and Main variables on all subpanels is promoted so that the variables are accessible from the main panel. This creates the properties:

- controllerThis
- cardThis
- mainThis

8.2 Implementation of the Application Panels

We implement the four panels independently. We assemble the panels in the main applet panel in a later step.

We create a new **itso.entbk2.atm.gui** package for the GUI application and create the four panels as subclasses of the Swing JPanel class.

Card Panel

Figure 107 shows the visual composition of the card panel (CardPanel class).

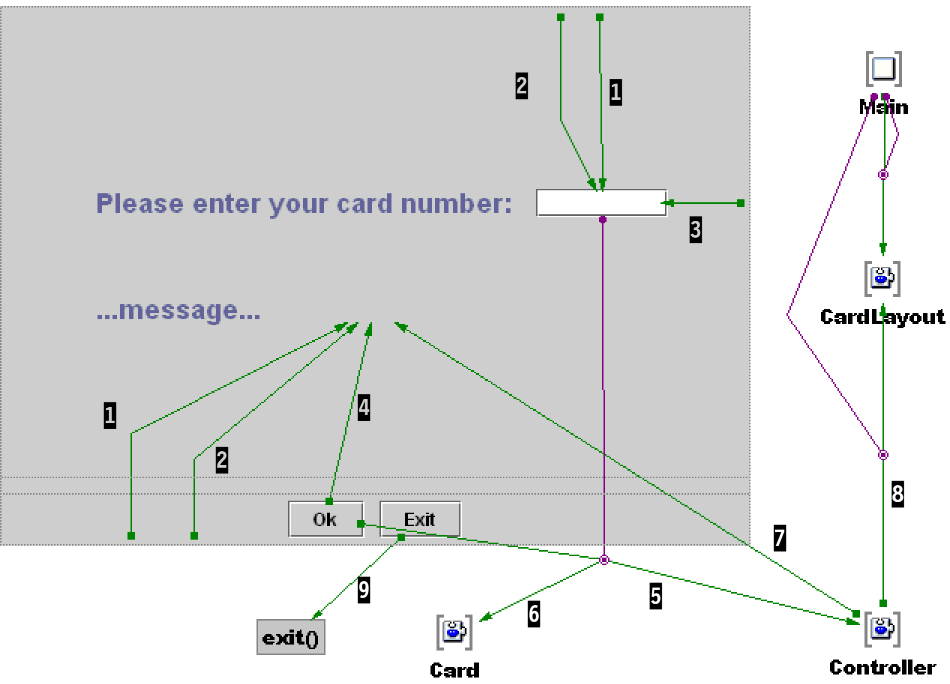


Figure 107. Visual Composition of the Card Panel

Initialization

- Connect the *initialize* event of the panel to the message and display the welcome message. Connect the same event to the *requestFocus* method (expert) of the entry field (1). Draw the same connections from the *componentShown* event of the panel (2).
- Use the *componentShown* event to set the entry field to null (3).

Processing

❑ Connect the *Ok* button to the message and display *Please wait* (4).

❑ Connect the *Ok* button to the *getCard* method of the controller with the card number entry field as a parameter (5). Connect the *normalResult* to the *this* property of the card variable (6).

❑ Connect the *cardNotFound* event of the controller to the message and display *Invalid card* (the same panel is displayed) (7).

Next Panel

❑ Connect the *cardFound* event of the controller to the *next* method of the card layout with the main panel as *parent* parameter (8).

❑ Connect the *Exit* button to a new *exit* method (9) that contains this code: `System.exit(0)`.

PIN Panel

Figure 108 shows the visual composition of the PIN panel (PinPanel class).

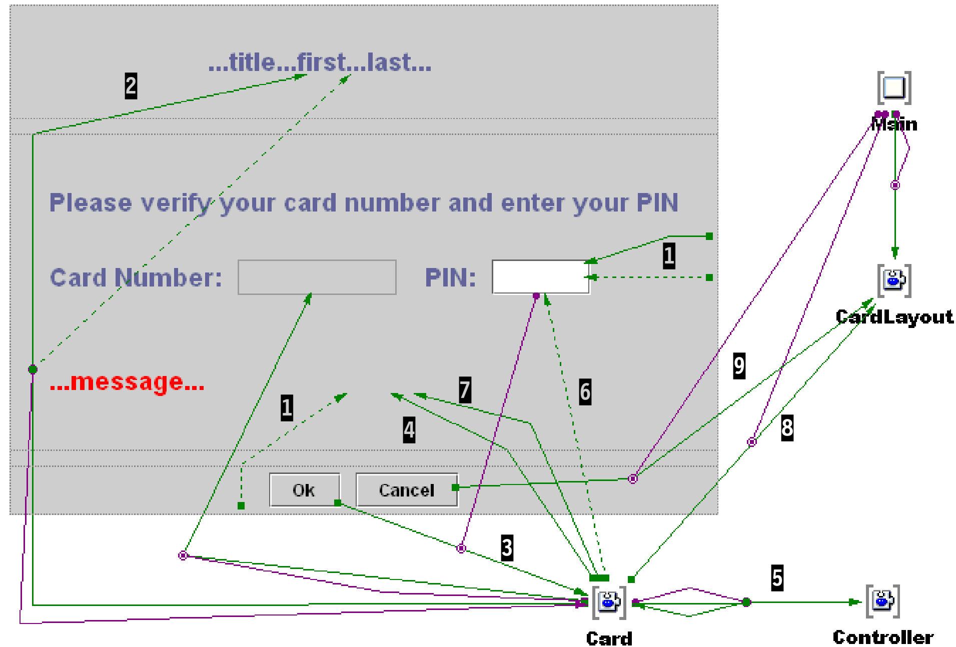


Figure 108. Visual Composition of the PIN Panel

Initialization

- ❑ Use the *componentShown* event and set the message and the PIN entry field to null. Connect the same event to the *requestFocus* method of the entry field (1).
- ❑ Use the *this* event of the card to set the customer text (...title...first...last...), using the *getGreetings* method as a parameter value. Use the *exceptionOccurred* event to set the customer text to null (2).

Processing

- ❑ Connect the *Ok* button to the *checkPin* method of the card with the PIN entry field as parameter (3).
- ❑ Use the *checkPinOk* event of the card to set the message to null (4).
- ❑ Connect the *checkPinOk* event to the *getAccounts* method of the controller and pass the card as parameter. Assign the *normalResult* back to the card. The card now contains a vector of associated accounts (5).
- ❑ Use the *checkPinNotOk* event to set the PIN entry field to null (6).
- ❑ Use the *checkPinNotOk* event to set the error message as *PIN invalid, please reenter* (the same panel is displayed) (7).

Next Panel

- ❑ Connect the *checkPinOk* event to the *next* method of the card layout (with main as parent) (8).
- ❑ Connect the *Cancel* button to the *previous* method of the card layout (with main as parent) (9).

Select Account Panel

Figure 109 shows the visual composition of the select account panel (SelectAccountPanel class).

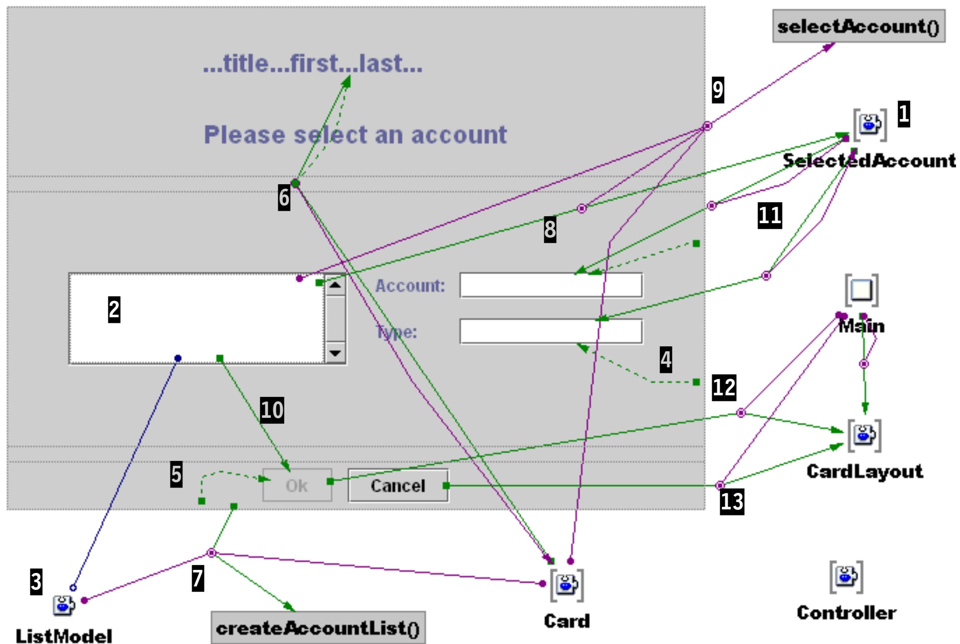


Figure 109. Visual Composition of the Select Account Panel

Additional Beans

- ❑ Drop a BankAccount variable and name it SelectedAccount (1). Promote its *this* property as *selectAccountThis*.
- ❑ The list of accounts is displayed in a list box (JList) that is inside a scroll pane (JScrollPane) (2).
- ❑ Drop a DefaultListModel bean and name it ListModel. Connect its *this* property to the *model* property of the list box (3).

Initialization

- ❑ Use the *componentShown* event to set the two fields for the account and type to null (4). (These fields are entry fields set as noneditable.)
- ❑ Open the *Ok* button and set it to be disabled. Connect the *componentShown* event to the *setEnabled* method with *false* as parameter (5).

❑ Use the *this* event of the card to set the customer text (same as in PIN panel) (6).

❑ Connect the *componentShown* event to a new *createAccountList* method and pass the list model and the card as parameters (7). This method fills the list model with the accounts stored in the card:

```
private void createAccountList(com.sun.java.swing.DefaultListModel list,
                    itso.entbk2.atm.model.Card card) {
   list.clear();
   java.util.Vector accounts = card.getAccounts();
   if (accounts != null) {
      java.util.Enumeration enum = accounts.elements();
      while (enum.hasMoreElements())
         list.addElement( ((itso.entbk2.atm.model.BankAccount)
                           enum.nextElement()).getAccountId() );
   }
}
```

Processing

❑ Connect the *listSelectionEvents* event of the list box to the *this* property of the selected account variable (8). Connect the parameter to a new *selectAccount* method and pass the *selectedIndex* of the list box and the card as parameters (9). This method extracts the account selected in the list box from the card:

```
private itso.entbk2.atm.model.BankAccount selectAccount(int index,
                          itso.entbk2.atm.model.Card card) {
   return (itso.entbk2.atm.model.BankAccount)
                        card.getAccounts().elementAt(index);
}
```

❑ Connect the *listSelectionEvents* event of the list box to the *setEnabled* method of the Ok button with *true* as a parameter (10).

❑ Connect the *this* event of the selected account variable to the account text field with the *accountID* property as a parameter (11). Connect the same event to the type text field with the *getAccountType* method as a parameter. The selected account information is displayed in the two text fields.

Next Panel

❑ Connect the *Ok* button to the *next* method of the card layout (with main as parent) (12).

❑ Connect the *Cancel* button to the *previous* method of the card layout (with main as parent) (13).

Transaction Panel

Figure 110 shows the visual composition of the transaction panel (TransactionPanel class).

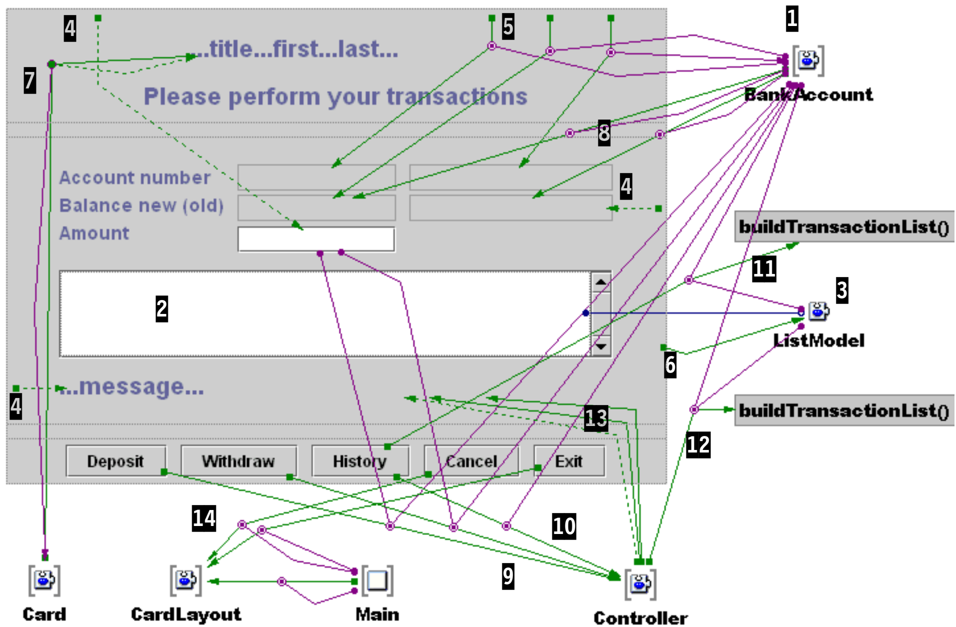


Figure 110. Visual Composition of the Transaction Panel

Additional Beans

- Drop a *BankAccount* variable and name it BankAccount (1). Promote its *this* property as *bankAccountThis*.
- The list of transactions is displayed in a list box (JList) that is inside a scroll pane (JScrollPane) (2).
- Drop a *DefaultListModel* bean and name it *ListModel* (3). Connect its *this* property to the *model* property of the list box.

Initialization

- Use the *componentShown* event to set the message, amount entry field, and old balance text field (noneditable) to null (4).
- Use the *componentShown* event to set the three fields, account number, account type, and new balance, to the values of the matching properties of the bank account (5). (Use the *getAccountType* method for the type field.)

- ❑ Connect the *componentShown* event to the *clear* method of the list model to empty the list box (6).
- ❑ Use the *this* event of the card to set the customer text (same as in PIN panel) (7).

Processing

- ❑ Connect the *balance* event of the bank account to the balance text field and pass the *balance* property as a parameter. Do the same for the *oldBalance* event of the bank account. These connections display the current balance values (8).
- ❑ Connect the *Deposit* button to the *deposit* method of the controller and pass the amount and the bank account as parameters (9).
- ❑ Connect the *Withdraw* button to the *withdraw* method of the controller and pass the amount and the bank account as parameters.
- ❑ Connect the *History* button to the *getTransactions* method of the controller and pass the amount and the bank account as parameters (10).
- ❑ Connect the History button to a new *buildTransactonList* method and pass the list model and the bank account as parameters (11). This method fills the list box with the transactions:

```
private void buildTransactionList(com.sun.java.swing.DefaultListModel list,
                                  itso.entbk2.atm.model.BankAccount account) {
   list.clear();
   for (int i=account.getTransactions().size()-1; i>=0; i--)
      list.addElement( account.getTransactions().elementAt(i).toString() );
   return;
}
```

- ❑ Connect the *newTransaction* event of the controller with the *buildTransactionList* method and pass the list model and the bank account as parameters (12).
- ❑ Use the *newTransaction* event to set the message to *null* (13).
- ❑ Use the *limitExceeded* event of the controller to display the *Limit exceeded - not enough funds available* message.
- ❑ Use the *DBOutOfSynch* event of the controller to display the *Transaction failed - please reenter* message.

Next Panel

- ❑ Connect the *Cancel* button to the *previous* method of the card layout (with main as parent) (14).
- ❑ Connect the *Exit* button to the *first* method of the card layout (with main as parent).

ATM Applet

You create the ATM applet (ATMApplet class) as a subclass of the Swing JApplet class. You set the layout manager to CardLayout and drop five panels on the main applet's panel:

- A JPanel from the palette. This is the starting panel where the user selects what persistence implementation to use for the applet.
- Four beans of type CardPanel, PinPanel, SelectAccountPanel, and TransactionPanel. This is the sequence of the application panels.

Use the Beans List to make individual panels of the card layout visible in the applet. This is necessary for some of the connections.

Figure 111 shows the visual composition of the ATM applet (ATMApplet class), with the additional persistence panel displayed.

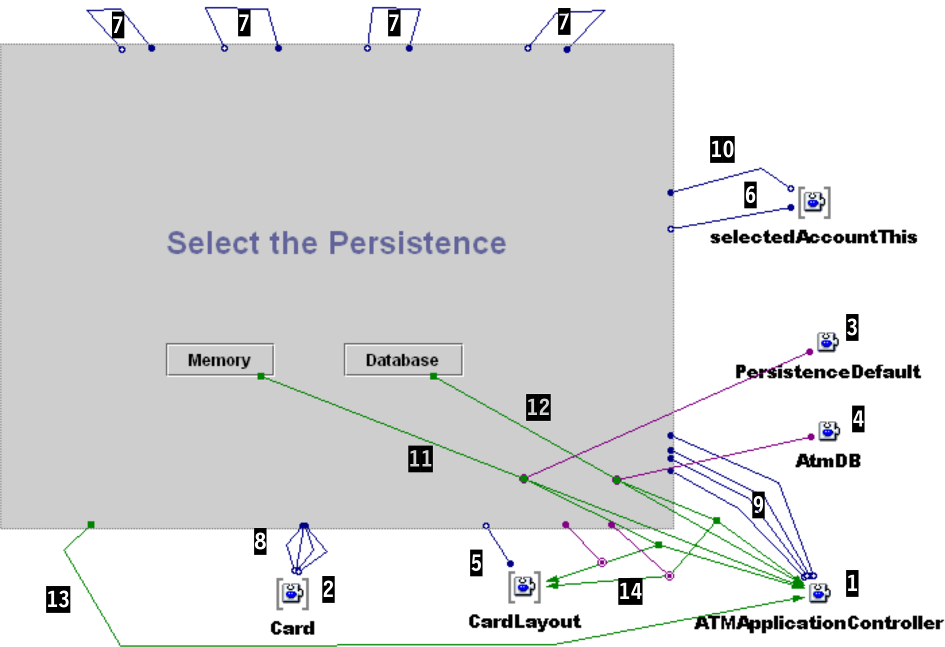


Figure 111. Visual Composition of the ATM Applet

Additional Beans

- ❑ Drop an ATMApplicationController bean (1) and a Card variable (2).
- ❑ Drop an ATMPersistenceDefault bean (3) (memory implementation) and an AtmDB bean (4) (relational database implementation).
- ❑ Tear off the *layout* property of the applet's main panel (5). Change its type to CardLayout and name it CardLayout.
- ❑ Tear off the *selectedAccountThis* property from the SelectAccountPanel (6).

Initialization

- ❑ Connect the *this* of the applet panel to the *mainThis* property of each of the four subpanels (7). This sets the Main variable on all panels.
- ❑ Connect the *this* of the Card variable to the *cardThis* property of each of the four subpanels (8). This sets the Card variable on all panels.
- ❑ Connect the *this* of the ATM application controller bean to the *controllerThis* property of each of the four subpanels (9). This sets the Controller variable on all panels.
- ❑ Connect the *this* of the selectedAccountThis to the *bankAccountThis* property of the TransactionPanel (10). This passes the selected bank account from the SelectAccountPanel to the TransactionPanel.

Processing

- ❑ Connect the *Memory* button to the *setPersistenceLayer* method of the controller and pass the PersistenceDefault bean as a parameter (11). Connect the *normalResult* to the *connect* method of the controller.
- ❑ Connect the *Database* button to the *setPersistenceLayer* method of the controller and pass the AtmDB bean as a a parameter (12). Connect the *normalResult* to the *connect* method of the controller.
- ❑ Connect the *destroy* event of the applet to the *disconnect* method of the controller (13).

Next Panel

- ❑ Connect the *normalResult* of the two connect method calls to the *next* method of the card layout and pass the applet's main panel as parent parameter (14).

8.3 Running the ATM GUI Applet

Before running the ATM GUI applet, make sure to check the class path for the ATMApplet class through the *Run -> Check class path* pop-up. The JFC and data access beans must be included in the class path.

Also check that DB2 is started, including the Java daemon process:

```
db2jstrt 8888
```

9 ATM Application Using Servlets

In this chapter we implement all of the user interfaces of the ATM application with servlets. The servlets interact with the ATM application controller designed in Chapter 6, “ATM Application Business Model” and will work with any implementation of the persistence interface that is attached to the application controller.

We implement the user interface with five visual servlets and one nonvisual controller servlet that controls the flow of the application. The six servlet classes are:

- ❑ Card servlet (CardView class)
- ❑ PIN servlet (PinView class)
- ❑ Account servlet (AccountView class)
- ❑ Transaction servlet (TransactionView class)
- ❑ Thank you servlet (ThankYouView class)
- ❑ ATM controller servlet (ATMServletController class)

All six classes inherit from the VisualServlet class. The controller servlet does not have a user interface, however.

© Copyright IBM Corp. 1998

9.1 Create a Skeleton Controller Servlet

The application flow is handled by the controller servlet, that is, all other servlets invoke the controller through the action specification of the form. The controller contains most of the business logic and decides which servlet to invoke next.

To facilitate development of the view servlets, it is good to have a skeleton controller from the beginning. Let's develop the skeleton controller first.

Create a new package named **itso.entbk2.atm.servlet**. Create a visual servlet named *ATMServletController*. Delete the HTML page from the visual composition and save the servlet. Your skeleton controller is in place.

9.2 Servlet Views

Now let's design the five views that represent the ATM application.

Card Servlet

When a user requests to start the ATM application, the card servlet displays a form for the customer to enter the ATM card number. This simulates the action of sliding the ATM card through a reader and is similar to today's home banking systems (Figure 112).

Figure 112. Card Servlet View

The card servlet has an image, a welcome text, and a form with an entry field, push buttons, and a message field.

An an image we use the ITSO banner that is displayed on the home page of the International Technical Support Organization (point your browser to http://www.redbooks.ibm.com or http://w3.itso.ibm.com).

Let's construct the card servlet named *CardView*. We do not describe all of the steps to visually create a servlet. Refer to Chapter 3, "Enterprise Application Development with Servlets" for detailed instructions.

The purpose of the card servlet is to get the card number. We minimize the logic in this servlet to keep it simple. The development steps are:

- GUI layout
- Auxiliary properties
- Auxiliary method
- Connections

GUI

Figure 113 shows the design of the card servlet. We use a form and set its *action* property to the ATMApplicationController. The HTML image bean has a *source* property that can point to a URL (use your own favorite URL) or a local file. Only with a local file can you see the image at design time.

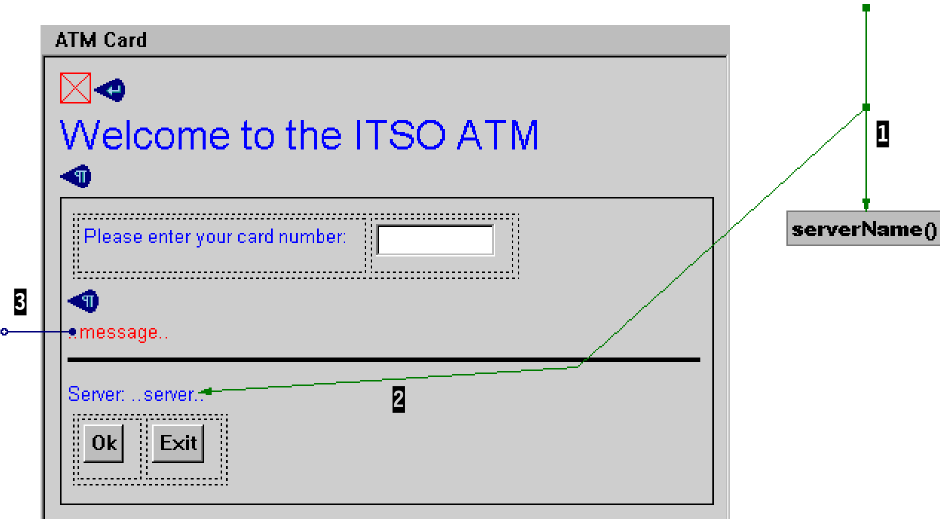


Figure 113. Card Servlet Design

Table 22 shows the GUI beans. This table does not represent all of the GUI elements; only a subset is listed with the important properties. We also use a table, paragraphs, and a rule. You can arrange the beans yourself.

Table 22. GUI Beans in Card Servlet

Type	Bean Name	Property	Property Value
HTMLImage	logo	source	/itso/image/itso.gif
HTMLForm	form	action	ATMServletController
HTMLEntryField	cardId		
HTMLText	message	string	..message..
HTMLText	server	string	..server..
HTMLTable			
HTMLPushButton	okButton	string	Ok
HTMLPushButton	exitButton	string	Exit

Auxiliary Properties

Define a *messageText* property on the BeanInfo page. This property will be connected to the message field in the GUI to set an error message.

The logo image is hard corded as *source* property of HTMLImge. Somebody might not like such a hardcorded reference. But think again, when you create an HTML file, you write a source URL directly, don't you? However, it is possible to set this URL from the controller. We promote the *source* property of the logo image as *logoImageURL*.

Auxiliary Method

We define a *serverName* method that returns the HTTP server's IP address:

```
public String serverName() {
    try {return java.net.InetAddress.getLocalHost().getHostAddress(); }
    catch (java.net.UnknownHostException e) { return null; }
}
```

Connections

We use connections to set up the server name.

- ❑ Connect the *aboutToGenerateOrTransfer* event (of the servlet) to the *serverName* method, using an event-to-code connection (1).
- ❑ Connect the *normalResult* to the *string* of the server field (2).
- ❑ Connect the *messageText* property to the *string* of the message field (3).

PIN Servlet

The PIN servlet (PinView class) is invoked by the controller servlet and asks the user to enter the PIN associated with the ATM card (Figure 114).

Figure 114. PIN Servlet View

The user name consists of title, first name, and last name. This user-unique data is defined in the customer class of the ATM model, and the customer class is accessed through the card class (see “Business Object Layer” on page 140). We keep the card object in a SessionDataWrapper Bean to have it available in all subsequent servlets.

This view uses a number of GUI beans and connections to extract data for the GUI, but it has no business logic. The development steps are:

- ❑ GUI layout
- ❑ Session data for user-unique information
- ❑ Auxiliary property
- ❑ Connections

GUI

Figure 115 shows the design of the PIN servlet, and Table 23 shows the major GUI beans and their properties.

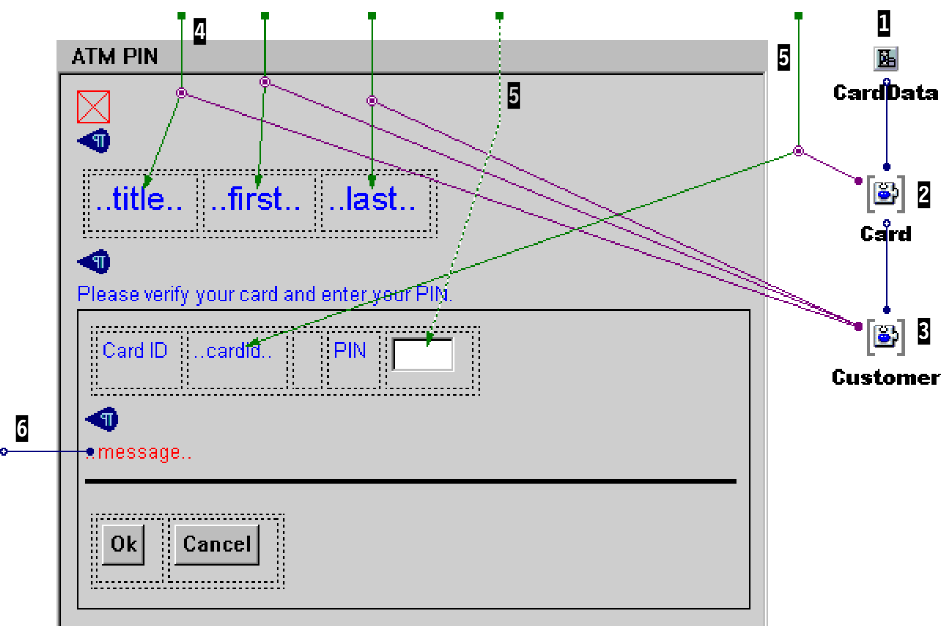


Figure 115. PIN Servlet Design

Table 23. GUI Beans in PIN Servlet

Type	Bean Name	Property	Property Value
HTMLImage	logo	source	/itso/image/itso.gif
HTMLText	custTitle		
HTMLText	custFirst		
HTMLText	custLast		
HTMLText	message	string	..message..
HTMLForm	form	action	ATMServletController
HTMLText	cardId	string	..cardid..
HTMLEntryField	pin	password-Protected	true
HTMLPushButton	okButton	string	Ok
HTMLPushButton	cancelButton	string	Cancel

Notice that we protect the PIN code of the user through the *passwordProtected* property of the entry field.

Session Data

We use session data to keep user-unique data. Every servlet session is separate for each user's HTTP session. The servlet handles multiple users, so we must keep the data of multiple users for the next servlet interaction, and we have to control the lifetime of the data. To keep it simple, we decide to keep the data as session data, which, as we learned, is pointed to by a cookie that is generated automatically.

If do not want to use a cookie, you should create a thread that controls all user-unique data, return the data by request from the servlet, and purge the data at some point in time. Our servlet lifetime is quite short, only as long as one HTTP session.

The SessionDataWrapper contains the card object. We will implement this later in the construction of the controller servlet. For now we can just extract a card object from the SessionDataWrapper to get the card ID and the customer object.

❑ Drop a SessionDataWrapper bean on the free-form surface, name it CardData, and name its propertyName property card (❶). The propertyName is important to access the same data.

❑ Tear off the *propertyValue* to extract a card object. Change the type of the variable to the Card class of the itso.entbk.atm.model package (ignore the warning message) and name the variable Card (❷).

❑ Tear off the customer property from the Card variable to extract a customer object. Name the variable Customer (❸).

Auxiliary Property

Define a *messageText* property on the BeanInfo page. This property will be connected to the message field in the GUI to set an error message.

Promote the *source* property of the logo image as *logoImageURL*.

Connections

We set up the title, first name, last name, and card ID before generating the HTML. We also clear the PIN entry field.

❑ Use the *aboutToGenerateOrTransfer* event to set a title text string and pass the *title* property of the Customer object as a parameter (❹). Set up the first name and last name in the same way.

- ❑ Set the up the card ID from the *cardNumber* property of the Card variable, and clear the PIN entry field, using similar connections from the *aboutToGenerateOrTransfer* event (5).
- ❑ Connect the *messageText* property to the *string* of the message field (6).

Account Servlet

After PIN validation, the user can choose one of the accounts. The account servlet (AccountView class) displays the customer's title and name, and a list of the accounts. The customer's name comes from the session data bean (Figure 116).

When an account is selected, the account ID and the account type (savings or checking) are displayed. We implement this with a JavaScript that runs on the client. The account ID and type are displayed in a drop-down list instead of a text field because the selected list item can be changed by the JavaScript without regenerating the HTML.

Figure 116. Account Servlet View

This view is composed of beans, connections, and some logic to extract the account data into two arrays of strings to fit into an HTML list bean. The steps to complete this view are:

- GUI layout
- Session data for user-unique information
- Auxiliary property and methods to extract account data
- JavaScript for dynamic account selection
- Connections

GUI

Figure 117 shows the design of the account servlet, and Table 24 shows the major GUI beans and their properties.

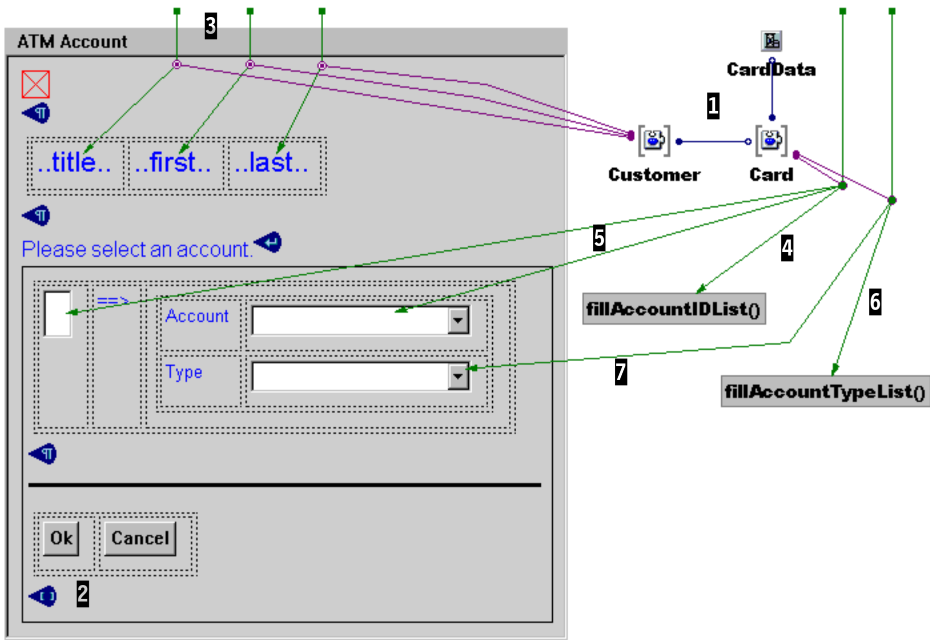


Figure 117. Account Servlet Design

Table 24. GUI Beans in Account Servlet

Type	Bean Name	Property	Property Value
HTMLImage	logo	source	/itso/....../itso.gif
HTMLText	custTitle		
HTMLText	custFirst		
HTMLText	custLast		
HTMLForm	form	action	ATMController Servlet

Type	Bean Name	Property	Property Value
HTMLList	AccountList	visibleItemCount	4
HTMLDropDownList	AccountID	visibleItemCount	1
HTMLDropDownList	AccountType	visibleItemCount	1
HTMLPushButton	okButton	string	Ok
HTMLPushButton	cancelButton	string	Cancel
HTMLScript	SelectionScript	string	<see below>

Session Data

We use the session data in the same way as for the PIN view. Place a SessionDataWrapper (with propertyName card), tear off the propertyValue (as type Card), and tear off the customer property from the Card (1).

Auxiliary Property and Methods for List of Accounts

We promote the *source* property of the logo image as *logoImageURL*.

The card object contains the list of accounts. The card object has a vector of account objects and each account object contains the account ID and the account type. We write two methods to extract the account IDs and account types into an array of strings:

```
private String[] fillAccountIDList(java.util.Vector accounts) {
    String[] result = new String[accounts.size()];
    for (int i=0; i<accounts.size(); i++)
        result[i] = new String (
            ((itso.entbk2.atm.model.BankAccount)accounts.elementAt(i)).getAccountId() );
    return result;
}

private String[] fillAccountTypeList(java.util.Vector accounts) {
    String[] result = new String[accounts.size()];
    for (int i=0; i<accounts.size(); i++)
        result[i] = new String (
            ((itso.entbk2.atm.model.BankAccount)accounts.elementAt(i)).getAccountType() );
    return result;
}
```

JavaScript

We implement a JavaScript (2) to display the selected account ID and type when the user selects an account from the list of accounts. We allow selection of the account in either of the three lists and adjust the other two lists. You write the JavaScript code in the *string* property of the HTMLScript bean that we named SelectionScript (below the push buttons). Define a function for

each of the three lists. Each function accepts a parameter, the name of the form that contains the list.

```
function selectAccountListFunc(s){
   s.AccountID.options[s.AccountID.selectedIndex].selected =true;
   s.AccountType.options[s.AccountID.selectedIndex].selected =true;
}
function selectAccountIDFunc(s){
   s.AccountList.options[s.AccountList.selectedIndex].selected =true;
   s.AccountType.options[s.AccountList.selectedIndex].selected =true;
}
function selectAccountTypeFunc(s){
   s.AccountList.options[s.AccountType.selectedIndex].selected =true;
   s.AccountID.options[s.AccountType.selectedIndex].selected =true;
}
```

Each function changes the selection in the two other lists to the selected index of the originating list box. To invoke this function, add an event handler to each of the lists. Edit the *extraAttributes* property of each list to one line of code:

```
onChange=selectAccountListFunc(form)        <== in AccountList
onChange=selectAccountIDFunc(form)          <== in AccountID
onChange=selectAccountTypeFunc(form)        <== in AccountType
```

This code will be written to the list tag of the generated HTML:

```
<LIST ...... onChange=selectAccountXxxxFunc(form) ...>
```

The JavaScript now contains three functions, one for each of the three lists. A selection in either list triggers the other two to be positioned on the matching item. Note that the names in the script must match the names of the GUI beans defined in Table 24.

Connections

We set up the customer and account information before generating the HTML:

- ❑ Connect the *aboutToGenerateOrTransfer* event to extract the customer information (same as in PIN servlet) (3).
- ❑ Connect the *aboutToGenerateOrTransfer* event to the *fillAccountIDList* method (event-to-code) and pass the *accounts* property of the Card variable as a parameter (4). Connect the *normalResult* to the *items* property of both the accountList and accountID bean (5).
- ❑ Connect the *aboutToGenerateOrTransfer* event to the *fillAccountTypeList* method (event-to-code) and pass the *accounts* property of the Card variable as parameter (6). Connect the *normalResult* to the *items* property of the accountType bean (7).

Transaction Servlet

The transaction servlet (TransactionView class) provides the main function of the ATM application. The transaction servlet handles deposit, withdraw, and query operations A bank account object created by the model contains the account ID, account type, current balance, and transaction history.

The bank account object is not saved as session data but accessed as a property of the servlet that is set by the controller servlet before invoking the transaction servlet. Each deposit, withdraw, and query operation invokes the controller servlet for processing, and the updated bank account object is assigned to the transaction servlet.

The *Cancel* button returns to the account selection view and the *Exit* button terminates the dialog with the thank you view (Figure 118).

Figure 118. Transaction Servlet View

This view is composed of beans, connections, and some logic to extract the transaction history data into an array of strings to fit into an HTML list bean. The steps to complete this view are:

- ❑ GUI layout
- ❑ Session data for user-unique information
- ❑ Auxiliary method to extract transaction history data
- ❑ Connections

GUI

Figure 119 shows the design of the transaction servlet, and Table 25 shows the major GUI beans and their properties.

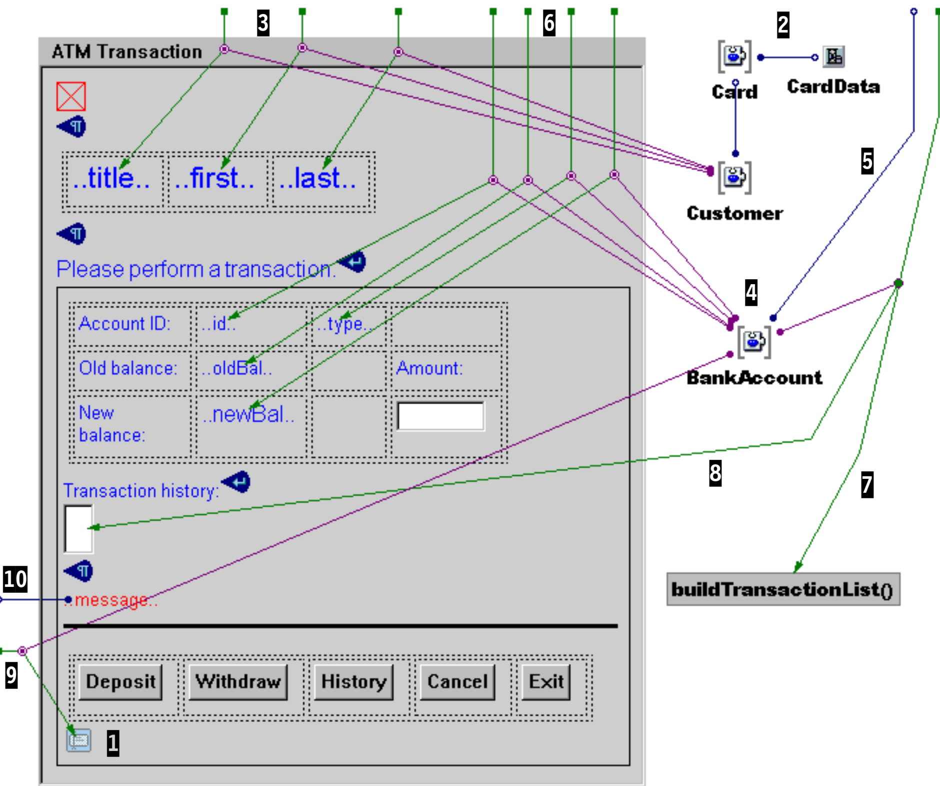


Figure 119. Transaction Servlet Design

Table 25. GUI Beans in Transaction Servlet

Type	Bean Name	Property	Property Value
HtmlImage	logo	source	/itso/image/itso.gif
HtmlText	custTitle	string	..title..
HtmlText	custFirst	string	..first..
HtmlText	custLast	string	..last..
HtmlForm	form	action	ATMServletController
HtmlText	accountID	string	..id..
HtmlText	accountType	string	..type..
HtmlText	oldBalance	string	..oldBal..
HtmlText	newBalance	string	..newBal..
HtmlEntryField	amount	size	8
HtmlList	TransactionList	visible...	5
HtmlText	message	string	..message..
HtmlPushButton	depositButton	string	Deposit
HtmlPushButton	withdrawButton	string	Withdraw
HtmlPushButton	historyButton	string	History
HtmlPushButton	cancelButton	string	Cancel
HtmlPushButton	exitButton	string	Exit
HtmlHiddenInput	accountIdHidden		

The account ID is not stored in a live object. The account ID is used only in the transaction servlet and for processing in the controller servlet. As you know, the card object contains all the accounts, but it does not record which account the user selected.

We use a hidden field to keep the account ID (**1**). The controller servlet gets the (hidden) account ID from the form.

Of course we could use a cookie or add a property in the Card class, but we do not want to modify the business model.

Session Data

We use the session data in the same way as for the PIN view. Place a SessionDataWrapper (with propertyName card), tear off the propertyValue (as type Card), and tear off the customer property from the Card (2).

Auxiliary Properties and Method

Define a *messageText* property on the BeanInfo page. This property will be connected to the message field in the GUI to set an error message.

Promote the *source* property of the logo image as *logoImageURL*.

We define a bankAccount property of type BankAccount. (The BankAccount class is in the itso.entbk2.atm.model package.)

We define a method to extract the transaction history from the bank account. Input is a Vector object that contains itso.entbk2.atm.model.Transaction objects, and output is an array of formatted Strings for the list.

```
private String[] buildTransactionList(java.util.Vector trans) {
    String[] result = new String[trans.size()];
    for (int i=0; i<trans.size(); i++) {
        result[i] = new String(
          ((itso.entbk2.atm.model.Transaction)trans.elementAt(i)).toString() );
    }
    return ret;
}
```

Connections

- Connect the *aboutToGenerateOrTransfer* event to extract the customer information (same as in PIN and account servlets) (3).
- Place a variable on the free-form surface and change the type to itso.entbk2.atm.model.BankAccount (4). This is the current account. Connect the *bankAccount* property of the servlet to the *this* of the variable (5).
- Extract account ID, account type, old balance, and new balance from the bank account variable using the *aboutToGenerateOrTransfer* event (6).
- For the transaction history, connect the *aboutToGenerateOrTransfer* event to the *buildTransactionList method (*event-to-code). Pass the *transactions* property of the bank account as a parameter (7). Connect the *normalResult* to the *items* property of the list box (8).
- To store the account ID for the next transaction process, connect the *aboutToGenerateOrTransfer* event to the hidden field with the *accountid* of the bank account as a parameter (9).
- Connect the *messageText* property to the *string* of the message field (10).

Thank You Servlet

The thank you servlet (ThankYouView class) is displayed at the end of an ATM session (Figure 120). This servlet runs when the user clicks on the *Exit* button in the card or transaction servlet. The *Restart Transaction* button tells the controller servlet to start a new session with the card servlet.

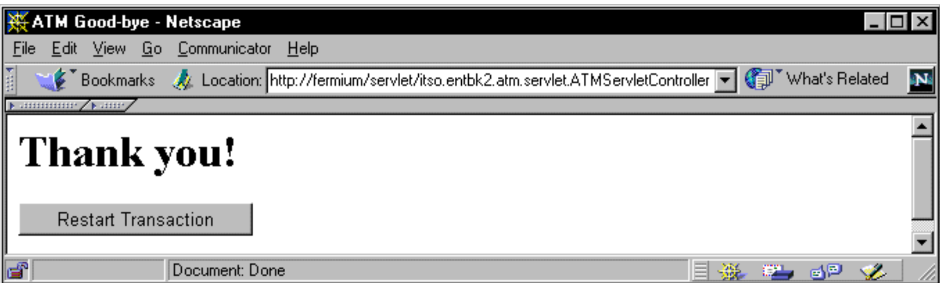


Figure 120. Thank You Servlet View

The design of this servlet is so simple that we leave it to you to complete. Do not forget to the set the *action* property in the form!

9.3 Application Flow Design

In this section we describe the general flow of the ATM application. In “Implementing the Controller Servlet” on page 223 we construct the controller servlet based on this design.

The controller servlet has two functions: to control the sequence of the servlet views and to act as the gateway to the application controller layer (see “Application Controller” on page 154) that interfaces with the persistent storage (Figure 121).

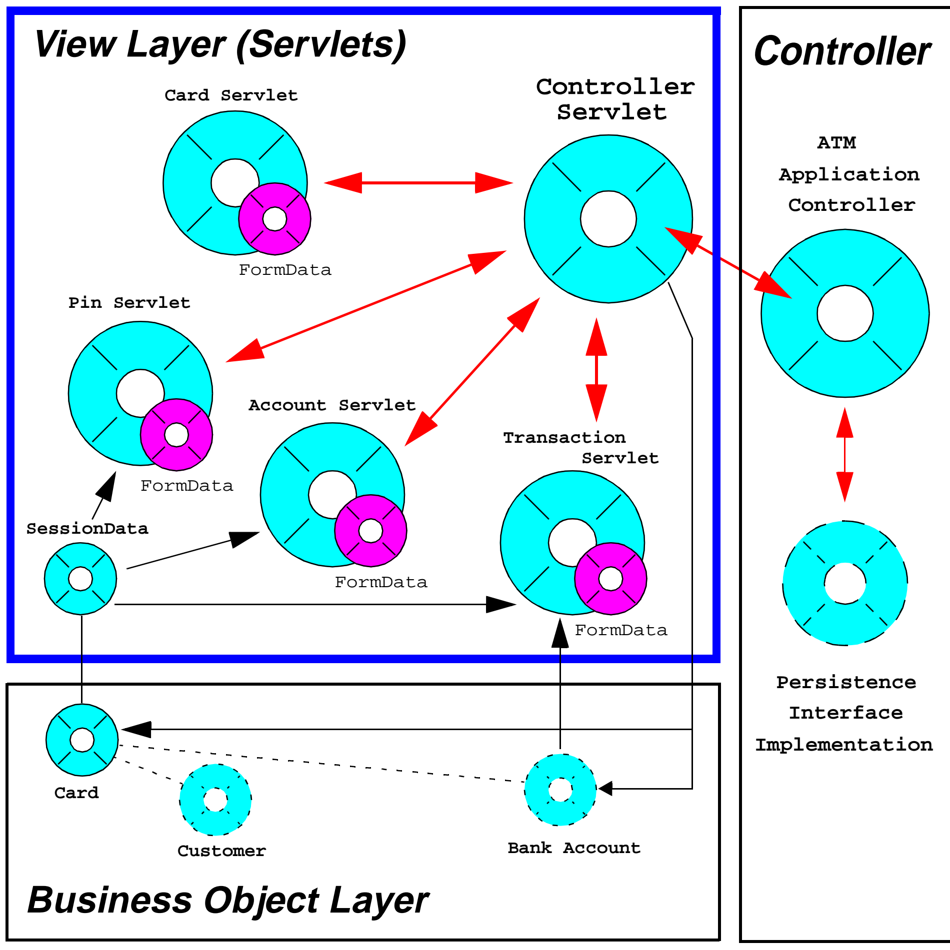


Figure 121. Servlet Application Flow

Keep the ATM Application Controller Alive

Servlets are instantiated when invoked and dismissed after processing However, we want to keep the ATM application controller object alive because it has database or enterprise connections. Therefore we instantiate the application controller when the first user invokes the controller servlet. We store the ATMApplicationController object in a static variable.

Flow Control

Each servlet passes control to the controller servlet through the action in the form. All processing is done in the controller servlet that decides to which servlet to pass control to display a user view.

Customer Verification

The entry point of the application is the card servlet. The controller servlet requests a card object from the ATM application controller, which generates a cardFound or cardNotFound event. A valid card is stored in session data, and the PIN servlet is invoked, for an invalid card the card servlet displays an error message.

PIN Verification

The PIN servlet passes the PIN to the controller servlet. The PIN is validated against the card object (no database access required). For a correct PIN the account servlet is invoked; for an invalid PIN the PIN servlet displays an error message.

Selecting an Account

The account servlet passes the selected account to the controller servlet, which retrieves the bank account object and invokes the transaction servlet.

Processing a Transaction

The account information is available in the bank account object. Each deposit, withdraw, or history transaction is processed by the controller servlet and an updated bank account object is sent back to the transaction servlet. The *Cancel* button invokes the account servlet to select another account, and the *Exit* button invokes the thank you servlet.

Exiting the ATM Application

The controller servlet handles the exit request from the card and transaction servlets and invokes the thank you servlet. The *Restart Transaction* button from the thank you servlet invokes the card servlet.

9.4 Implementing the Controller Servlet

It is time to implement the controller servlet (ATMServletController class) with the processing and flow logic. The controller servlet is a kind of connector with visual composition and business logic. We implement the function in small and understandable pieces:

- ❑ Preparation for testing
- ❑ Initialization
- ❑ Customer verification
- ❑ PIN verification
- ❑ Account selection
- ❑ Deposit transaction
- ❑ Withdraw transaction
- ❑ Query transaction history
- ❑ Termination and restart
- ❑ Disable caching of the output HTML

Preparation for Testing

The servlets use the ATM application controller for processing. The ATM application controller must have an implementation of the ATMPersistenceInterface assigned.

For initial tests you can use the ATMPersistenceDefault memory implementation (see “Persistence Layer” on page 162), and later you can switch to the AtmDB database implementation (see Chapter 7, “ATM Application Persistence Using Data Access Beans”). For the database test, make sure that:

- ❑ DB2 is started
- ❑ The DB2 Java daemon is started (db2jstrt 8888)

Initialization

To keep the ATMApplicationController alive for a whole session, we use a static field and assign it to a variable using visual composition.

- ❑ Create a static field named applicationController of type itso.entbk2.atm.model.ATMApplicationController:

```
private static .......ATMApplicationController applicationController = null;
```

- ❑ Create a getApplicationController method to initialize the application controller including setting a suitable persistence implementation:

```
public .......ATMApplicationController getApplicationController() {
   if (applicationController == null) {
      applicationController = new itso.entbk2.atm.model.ATMApplicationController();
      applicationController.setATMPersistenceLayer
      // ( new itso.entbk2.atm.databean.AtmDB() );              <=== database
         ( new itso.entbk2.atm.model.ATMPersistenceDefault() );    <=== memory
      setTransferToServiceHandler(getCardView());
   }
   return applicationController;
}
```

Initially we use the memory implementation of the persistence interface.

The *setTransferToServiceHandler* method specifies the card servlet as the default target. We can start the application by invoking the controller servlet.

- ❑ Go to the Visual Composition page and add a variable of type ATMApplicationController (1).
- ❑ Connect the *initialize* event (of the servlet) to the *this* property of the controller variable and get the parameter from the *getApplicationController* method (parameter-from-code) (2).
- ❑ To indicate that the controller servlet does not generate HTML, connect the *initialize* event (of the servlet) to the *isTransferring* property with a parameter value of *true* (3).

Figure 122 shows the visual composition of the initialization phase.

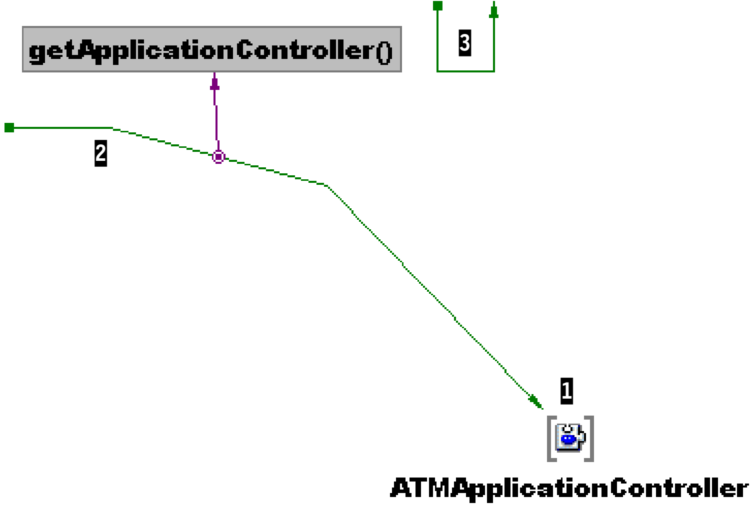


Figure 122. Initializing the Controller Servlet

Customer Verification

Figure 123 shows the customer verification implementation.

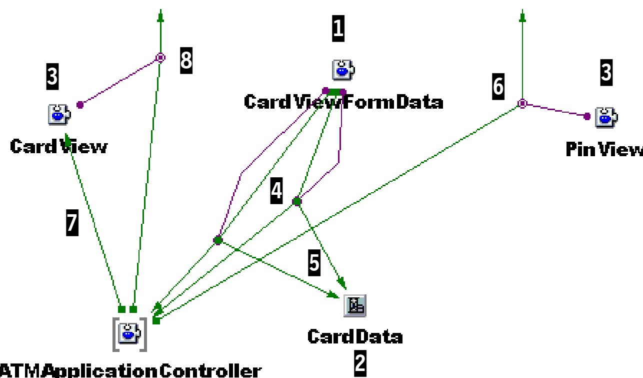


Figure 123. Customer Verification

The steps to implement the customer verification are:

- ❑ Add a CardViewFormData bean to receive the data and event from the card servlet (1).
- ❑ Add a SessionDataWrapper bean, name it CardData, with card as the property name (2). We will save the card object in session data for subsequent servlets.
- ❑ Add the two target servlets, CardView and PinView, as beans (3).
- ❑ The starting point for processing is the *Ok* button in the card servlet. Connect the *okButtonPressed* event (in CardViewFormData) to the *getCard* method of the ATMApplicationController. Pass the *cardIdString* (of CardViewFormData) as a parameter (4). Do the same for the *enterKeyPressed* event.
- ❑ The getCard method retrieves the card from the database. Connect the *normalResult* to the *propertyValue* in the CardData session data bean (5).
- ❑ The ATMApplicationController fires a *cardFound* or *cardNotFound* event after retrieving the card. We use these events to invoke the next servlet.
  - For a valid card we invoke the PIN servlet. Connect the *cardFound* event to the *transferToServiceHandler* property of the controller servlet and pass the *this* of PinView as a parameter (6).
  - For an invalid card we prepare an error message and invoke the card servlet. Connect the *cardNotFound* event to the *messageString* property of CardView and set the parameter to *The card number is invalid* (7). Connect the *cardNotFound* event to the

transferToServiceHandler property of the controller servlet and pass the *this* of CardView as a parameter (8).

❑ We will handle the Exit button event later.

Save the controller servlet and run it. When the card servlet displays the greeting, enter a valid card number and click on *Ok*. The PIN servlet should get control and display the next form.

PIN Verification

Figure 124 shows the PIN verification connections.

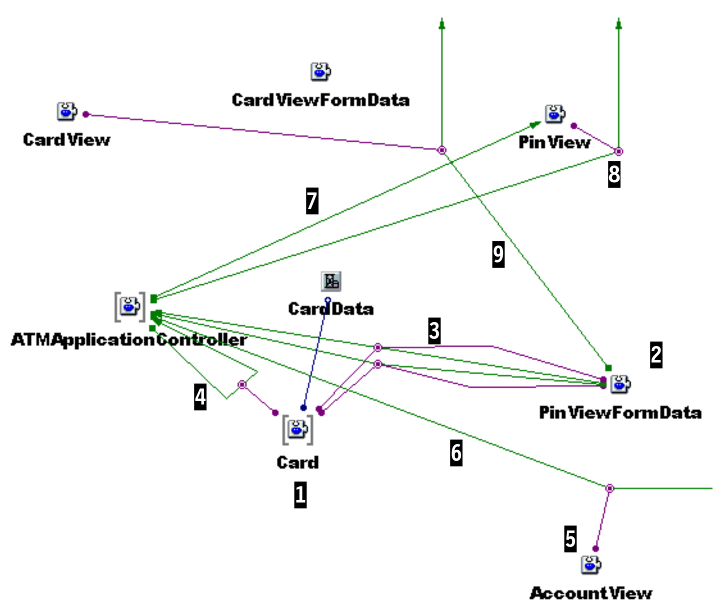


Figure 124. PIN Verification

The steps to implement the PIN verification are:

❑ Tear off the *propertyValue* of CardData (session data) and name it Card (1). Change the type to itso.entbk2.atm.model.Card (ignore the warning).

❑ Add a PinViewFormData bean to receive the PIN entered by the user (2).

❑ The PIN is checked when the user clicks on the *Ok* button or presses the Enter key. Connect the *okButtonPressed* event (of PinViewFormData) to the *checkPin* method of the application controller (3). Two parameters are required, the *this* of the Card, and the *pinString* of the PinViewFormData.

Create the same connections for the enterKeyPressed event.

- ❑ After checking a PIN, the application controller fires an event, either pinCheckedOk and pinCheckedNotOk.

  If the PIN is correct, we retrieve the accounts of the card and pass control to the account servlet.

  - Connect the *pinCheckedOk* event to the *getAccount* method of ATMApplicationController to add the account data to the card object. (This data is required for the account servlet.) Pass the *this* of the Card as a parameter (4).
  - Add a bean of type AccountView (5). Connect the *pinCheckedOk* event to the *transferToServiceHandler* method and pass the *this* of AccountView as a parameter (6).

- ❑ If the PIN is incorrect, we set an error message and return to the pin servlet. Connect the *pinCheckedNotOk* event (of the application controller) to the *messageText* property of PinView with the text *PIN invalid, please reenter!* (7) Connect the *pinCheckedNotOk* to the *transferToServiceHandler* method with the *this* of PinView as a parameter (8).
- ❑ When the *Cancel* button is clicked (in PinView) we return to the card servlet. Connect the *cancelButtonPressed* event to the *transferToServiceHandler* method with the *this* of CardView as a parameter (9).

Account Selection

Figure 125 shows the account selection connections. The steps to implement account selection are:

- ❑ Add an AccountViewFormData and a TransactionView bean to the free-form surface (1).
- ❑ The TransactionView requires a bank account object that contains the balance and the transactions. The getAccount method of the application controller returns a bank account based on the account ID.

  - Connect the *okButtonPressed* (and same for *enterKeyPressed*) event of the AccountViewFormData to the *getAccount* method of the application controller. Pass the *accountIDSelectedItemString* property of AccountViewFormData as a parameter; this is the selected item in the account ID list (2).
  - The result of the connection is the bank account object. Connect the *normalResult* of the connections to the *bankAccount* property of the TransactionView (3).

- Connect the *okButtonPressed* and *enterKeyPressed* events to the *transferToServiceHandler* method with the *this* of TransactionView as a parameter (4).

❑ Connect the *cancelButtonPressed* event to the *transferToServiceHandler* method with the *this* of PinView as a parameter (5).

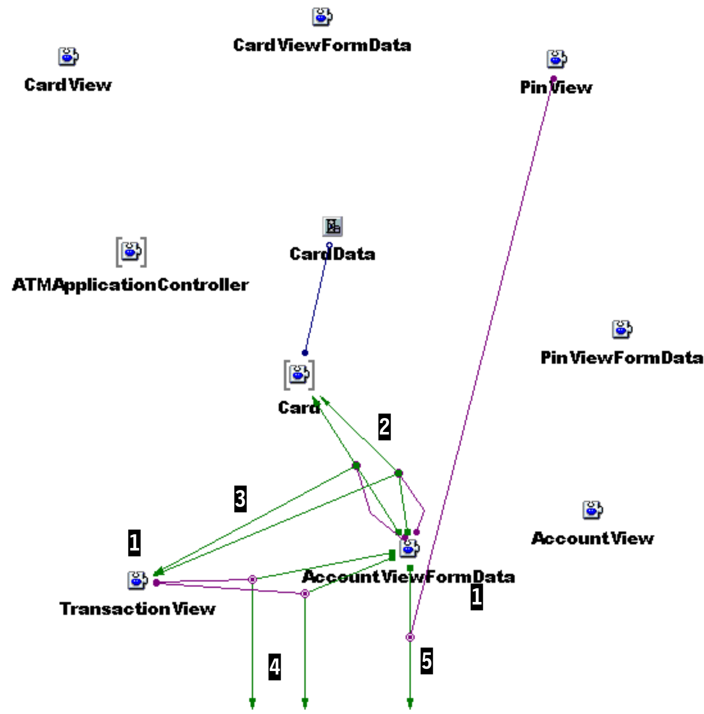


Figure 125. Account Selection

Deposit Transaction

Figure 126 shows the connections for the deposit transaction. The transaction is applied to the bank account, and the account servlet is invoked again with the updated account object.

❑ Add a TransactionViewFromData bean to the free-form surface (1).

❑ The *Deposit* button invokes the deposit transaction. Connect the *depositButtonPressed* event (of TransactionViewFormData) with the *deposit* method of the application controller to update the bank account (2). The deposit method requires two parameters: a bank account and an amount:

- The bank account we get from the card. Connect the *account* parameter to the *getAccount* method of the card and with the *accountIDhiddenString* property of TransactionViewFormdata as a parameter (3). (We saved the account ID as hidden data in the form.)
- We get the *amount* from the *amountString* property of the form (4).

❑ The result of the deposit method is the updated bank account. Connect the *normalResult* to the *bankAccount* property of the TransactionView (5).

❑ The application controller fires a newTransaction event when the deposit is complete. Connect the *newTransaction* event to the *messageText* property of the TransactionView with the text *Transaction complete, account updated* (6). Then connect the *newTransaction* event to the *transferToServiceHandler* method with the *this* of TransactionView as a parameter (7).

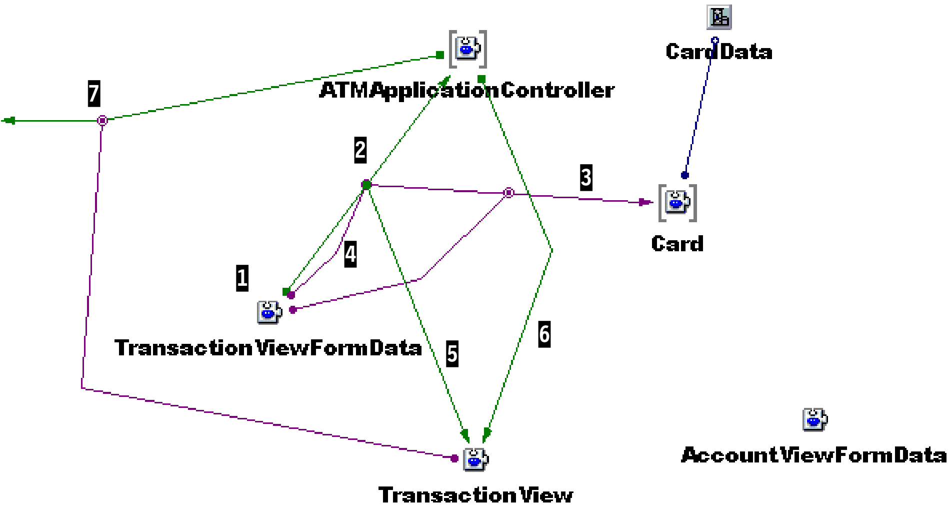


Figure 126. Deposit Transaction

Withdraw Transaction

Figure 127 shows the connections for the withdraw transaction. Withdraw is similar to deposit but may fire a limit exceeded event if not enough funds are available.

❑ The withdraw transaction is implemented with connections that match the deposit transaction; connect the *withdrawButtonPressed* event to the *withdraw* method of the application controller and use the same two parameters (*account* as *getAccount* of Card and *amount* as *amountString*

of the form). Connect the *normalResult* to the *bankAccount* property of the TransactionView (1).

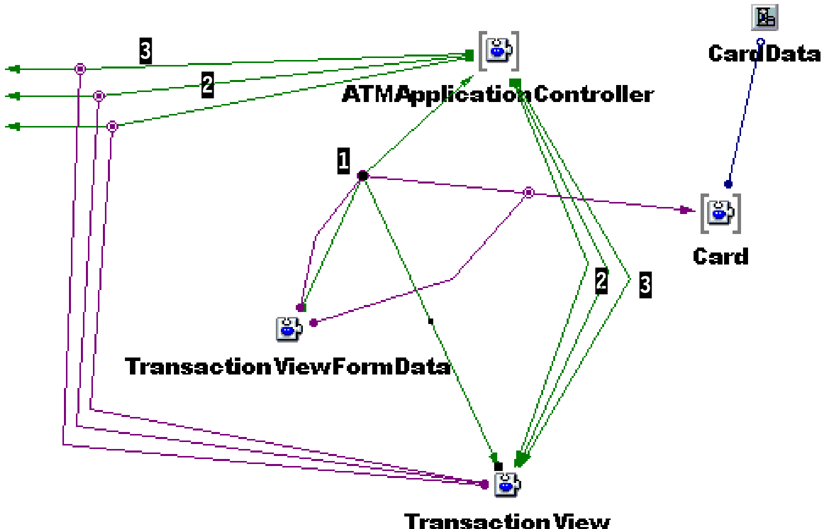


Figure 127. Withdraw Transaction

❑ A complete withdrawal transaction fires the new transaction event. We have already handled that in the deposit transaction.

❑ To handle the case of not enough funds, connect the *limitExceeded* event (of the application controller) with the *messageText* property of the TransactionView and set the text to *Withdraw failed, not enough funds*. Then connect the *limitExceeded* event to the *transferToServiceHandler* method with the *this* of TransactionView as a parameter (2).

❑ The application controller may fire a *DBOutOfSynch* event if the account in the database does not match the account object. Deposit or withdraw transactions are not performed. Connect the *DBOutOfSynch* event to a suitable *messageText* in the TransactionView and to the *transferToServiceHandler* method with the *this* of TransactionView as a parameter (3).

Query Transaction History

Figure 128 shows the connections for the transaction history that retrieves all transaction objects of an account for display. The initial list of transactions displays current transactions only.

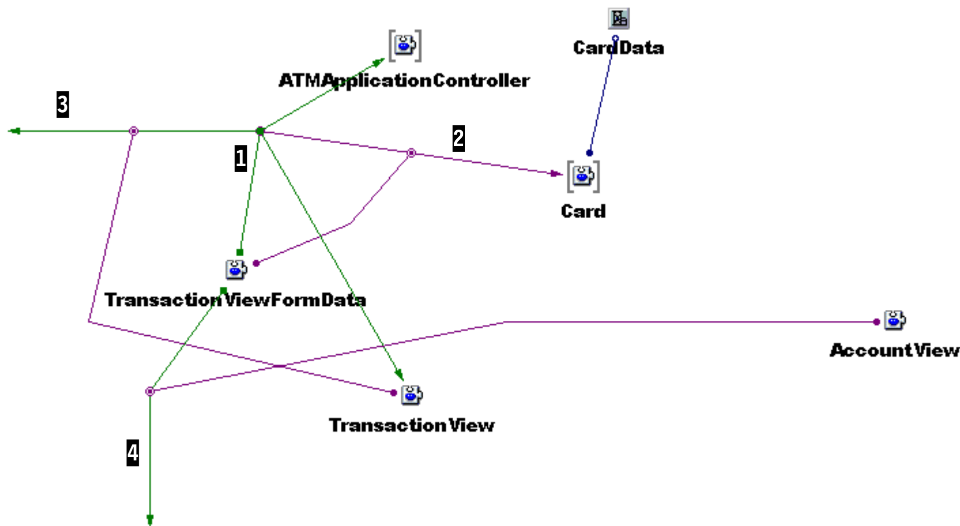


Figure 128. Query Transaction History

❑ Connect the *historyButtonPressed* event to the *getTransactions* method of the application controller (1).

❑ The parameter of this connection is the bank account that we get in the same way as for deposit and withdraw (use *getAccount* method in card) (2). Connect the *normalResult* to the *bankAccount* property of the TransactionView.

❑ No event is fired by the application controller. Connect the *normalResult* of the *getTransactions* method to the *transferToServiceHandler* method with the *this* of TransactionView as a parameter (3).

Cancel

❑ Connect the *cancelButtonPressed* event of the TransactionViewDataForm to the *transferToServiceHandler* method with the *this* of AccountView as a parameter (4).

Termination and Restart

Figure 129 shows the exit and restart connections. Here we handle the *Exit* button in the card and transaction servlet to invoke the thank you servlet, and we schedule the card servlet for the *Restart* button in the thank you servlet.

We also have to make sure that the event listeners of the servlet are removed from the application controller because each user interaction creates a new instance of the servlet.

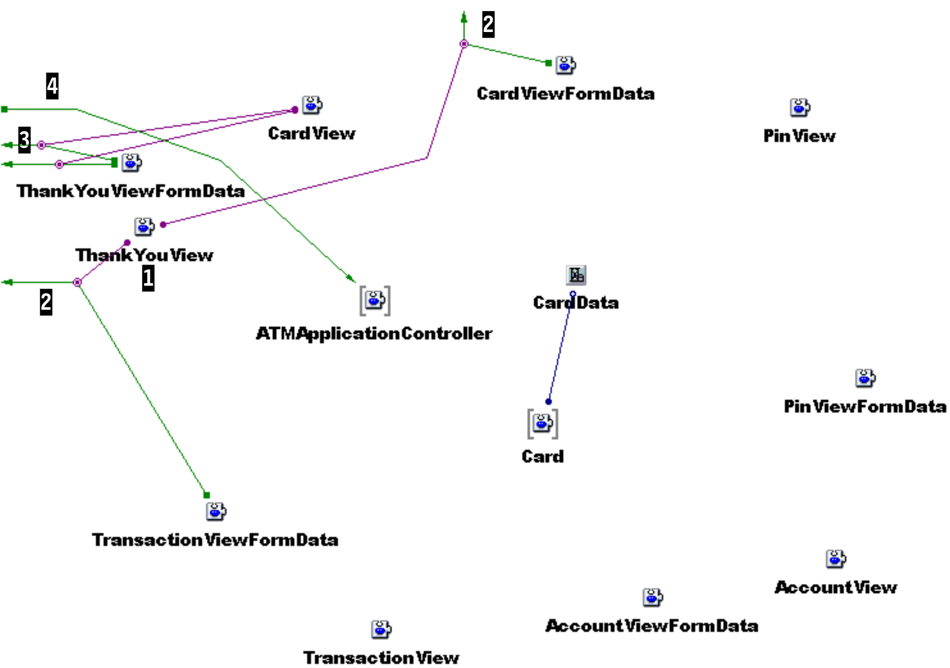


Figure 129. Termination and Restart

❑ Add a ThankYouView bean to the free-form surface (1).

❑ Connect the *exitButtonPressed* event of the CardViewFormData and of the TransactionViewFormData to the *transferToServiceHandler* method with the *this* of ThankyouView as a parameter (2).

❑ Add the ThankYouViewFormData bean to receive the *Restart* button event and connect the *restartButtonPressed* and *enterkeyPressed* events to the *transferToServiceHandler* method with the *this* of CardView as a parameter to restart the ATM application (3).

❑ Add a connection from the *aboutToGenerateOrTransfer* event of the servlet to the *this* property of the ATMApplicationController and set the value to *null* (4). This connection removes all event listeners that the controller servlet has set up with the application controller. (We found that without this connection all previous instances of the controller servlet received the events of the current interaction.)

Disable Caching of the Output HTML

When testing the ATM servlets we found that sometimes wrong accounts were displayed for an ATM card, or that entering a PIN number of a previously displayed card worked for another card ID.

Analysis proved that bad HTML pages were displayed to the end user because they were cached from previous requests. The solution is to disable the caching of the output HTML pages as discussed in “Disable Caching of Generated HTML” on page 87.

Instead of disabling caching in each of the servlets, you can add the required code to the initialize method of the controller servlet.

We also found that when invoking the ATMServletController in the middle of a session without any parameters, a null pointer exception occurred because no target servlet was set up with the setTransferToServiceHandler method. The easiest solution for this problem is to set up the card servlet as the default transfer servlet.

Figure 130 shows the tailored initialize method of the controller servlet. Note that the connection method numbers (connEtoMx) may be different in your implementation.

```
private void initialize() {
   // user code begin {1}
   // user code end
   initConnections();
   connEtoM1();          // setIsTransferring(true)
   connEtoM3();          // setATMApplicationController(...)
   // user code begin {2}
   setTransferToServiceHandler(getCardView());     // default target servlet
                                                   // get response object
   javax.servlet.http.HttpServletResponse resp = getResponse();
   resp.setContentType("text/html");
   resp.setHeader("Pragma","no-cache");            // no caching
   resp.setHeader("Cache-Control","no-cache");
   resp.setDateHeader("Expires",0);                // cache expires
   // user code end
}
```

Figure 130. Disabling Caching for the ATM Servlets

Controller Servlet Total Design

Figure 131 shows the total design of the controller servlet in the Visual Composition Editor.

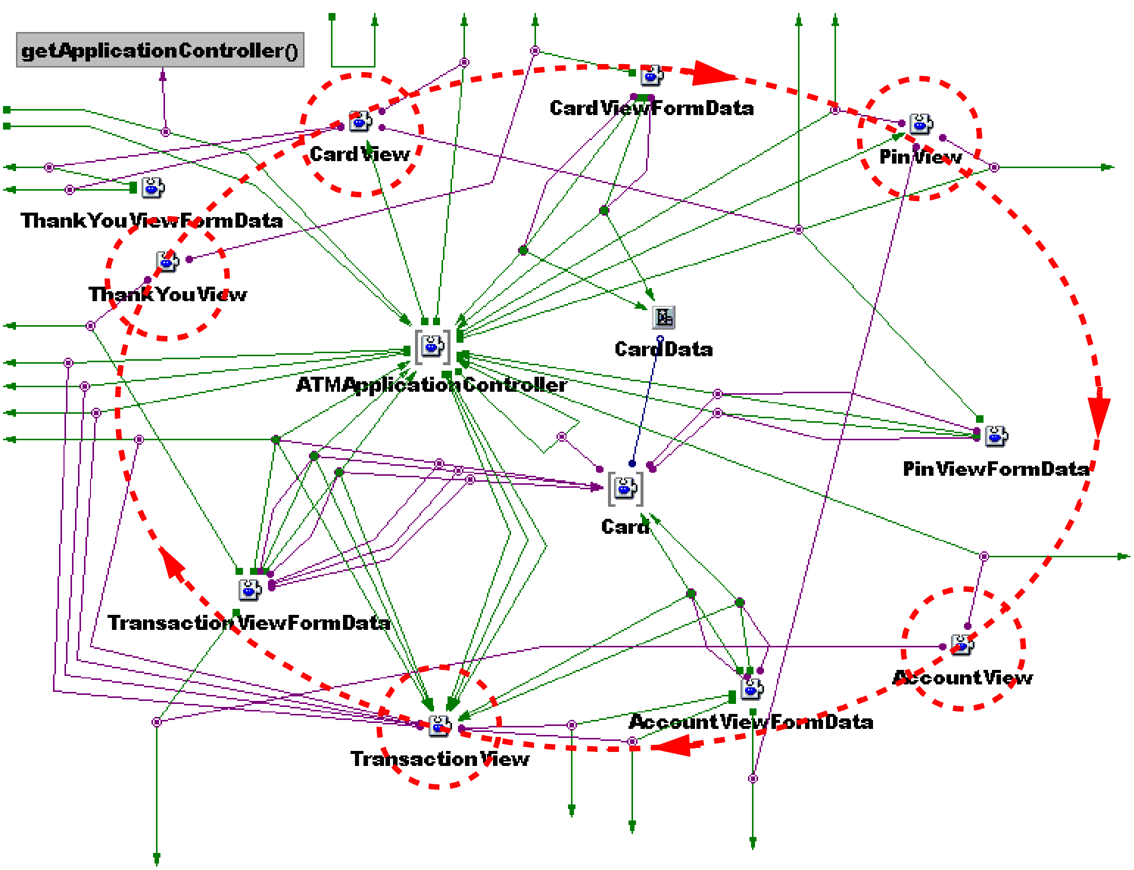


Figure 131. Controller Servlet Total Design

The dashed lines highlight the five servlets as they are invoked in sequence by the ATM application controller.

9.5 Testing the ATM Servlet Application

Confirm that the functions of the system service layer, such as DB2, CICS, or MQSeries, are properly configured.

Select the controller servlet (ATMServletController) and click on the *Run* button in the tool bar. VisualAge for Java starts the internal HTTP server, and a Web browser (defined in the *Window->Options->Help* menu) that invokes the selected servlet. The card servlet should display the first user interface. You can also select the card servlet (CardView) as the starting point. The browser points to the URL:

```
http://127.0.0.1:8080/servlet/itso.entbk2.atm.servlet.ATMServletController
```

Built-in HTTP Server

com.ibm.ivj.servlet.runner.HttpServerStarter is the class that starts the HTTP server to test a servlet. You can see the server running status in the Console window. While the server is running, you can access the ATM servlet from any browser that is in the network of your machine. The port address to access the servlet is defined in the server configuration file (IBMVJava\ide\project_resource\IBM Servlet Builder class libraries\com\ibm\ivj\servlet\runner\configuration.properties) and is displayed in the Console window. If you access a servlet from another machine, you must specify the port; the default is 8080.

The default HTTP server comes from the Sun JSDK and is known as the Servlet Runner. The Servlet Runner works as a servlet run-time engine and not really as a Web server.

Using the WebSphere Application Server

You can use the WebSphere Application Server inside VisualAge for Java to test your servlets.

Detailed instructions on how to make the WebSphere Application Server work within VisualAge for Java depend on the version of WebSphere. The instructions are available from these IBM Web sites:

```
http://www.software.ibm.com/ad/vajava
http://www.software.ibm.com/webservers
```

Using the ATM Servlet Application with DB2

After initial tests with the memory implementation of the persistence interface we can now switch to the DB2 implementation.

Edit the getApplicationController method of the controller servlet to specify the AtmDB bean as the persistence interface:

```
public .......ATMApplicationController getApplicationController() {
   if (applicationController == null) {
      applicationController = new itso.entbk2.atm.model.ATMApplicationController();
      applicationController.setATMPersistenceLayer
         ( new itso.entbk2.atm.databean.AtmDB() );     // <=== database
      setTransferToServiceHandler(getCardView());
   }
   return applicationController;
}
```

Attention	Verify the connection information in the AtmDatabase class. A user ID and password must be specified, and prompting and auto-commit must not be checked. You can check this in the code of the connect method
	`connection.setPromptUID(false);`
	`connection.setAutoCommit(false);`
	or you can open one of the select beans and verify it in the Visual Composition Editor (see Figure 104 on page 189). If user ID prompting is active, the prompt window appears but may be hidden behind the Web browser!

Before starting the servlet test, make sure that DB2 is active and that the DB2 Java daemon is started:

```
db2start
db2jstrt 8888
```

9.6 Deploying Servlets

The target for deploying servlets is a Web server. You can use Lotus Domino Go Webserver, Netscape Application Server, Apache, and many others; they all support servlets.

We used the WebSphere Application Server plugin with Lotus Domino Go Webserver on Windows NT.

Install Lotus Domino Go Webserver

To use the Lotus Domino Go Webserver with WebSphere, do not install its servlet function, because WebSphere replaces that function with its own servlet support.

Install WebSphere and Customize

WebSphere is installed on top of an existing Web server and adds a servlet run-time facility. WebSphere comes with an administration dialog that enables you to tailor the WebSphere run time. Invoke the administration dialog with:

```
http://host.machine.com:9090
```

One of the important tasks is to set up the class path for servlets. If you use any of the VisualAge for Java Enterprise builders, you must make sure that enterprise builder classes or jar files are accessible to WebSphere through the WebSphere class path specification.

For more information see “Deployment of Servlets” on page 338.

10 ATM Application with the CICS Connector

In Chapter 4, “CICS Access with the CICS Connector”, we discuss the CICS Connector. We cover the Common Connector Framework, the Universal CICS Client, and the CICS Transaction Gateway, as well as record beans, commands, navigators, and mappers.

In this chapter we examine the way in which the persistence layer of the ATM application can be extended to make use of the CICS Connector to access enterprise data through CICS programs.

© Copyright IBM Corp. 1998

10.1 A Review of the ATM Application Design

The ATM application design is based on a layered approach, as discussed in Chapter 5, “ATM Application Requirements and ATM Database.”

The three layers of the application design are:

- User interface layer

  The user interface layer deals with the GUI part of the application.

- Business object layer

  The business object layer is responsible for the business logic. Core entities of the ATM application are represented here.

- Persistence layer

  The persistence layer is used to separate data access from the rest of the application. This separation allows for different services to be used to access the data without affecting the rest of the application.

Because this chapter is about accessing enterprise data with the CICS Connector, the persistence layer is of special interest. The persistence layer knows about the business objects, but not about the user interface layer. Therefore the user interface layer is not addressed at all in this chapter.

Managing these layers is an application controller. Mediator objects are used to connect the different layers. These objects are part of the application controller. A Java interface, ATMPersistenceInterface, specifies the interface to the persistence layer.

The Persistence Interface

The ATMPersistenceInterface consists of the methods listed in Table 26.

Table 26. ATM Persistence Interface Methods

Method	Return Type	Parameters	Remarks
extConnect	void	-	Connects to the data source
extDisconnect	void	-	Disconnects from the data source
extGetCard	Card	String cardId	Retrieves the data and constructs a new Card object
extGetPin	Card	Card	Retrieves the PIN
extGetAccounts	void	Card	Retrieves all accounts (of a card) and stores them in the card object
extUpdateBalance	void	BankAccount	Updates balance and logs changes in transaction history
extGetTransactions	void	BankAccount	Retrieves all transactions (of an account) and stores them in the account object

The application controller itself, represented by the ATMApplicationController class, instantiates an object of a class that implements the ATMPersistenceInterface. Each access method in the persistence layer can have its own implementation of the ATMPersistenceInterface.

We have already shown one such implementation when accessing enterprise data with data beans (see Chapter 7, “ATM Application Persistence Using Data Access Beans”). In this chapter we show another implementation for accessing enterprise data through the CICS Connector. To use the different enterprise access mechanisms, we only have to replace the persistence class being instantiated.

Whenever a user-driven event occurs, such as a button being clicked, the user interface layer calls a function in the application controller, which calls a method of a class in the persistence layer that implements the ATMPersistenceInterface. The behavior of the called method differs according to the enterprise access mechanism. In this chapter we create a class called AtmCICS that implements the ATMPersistenceInterface. AtmCICS uses the CICS Connector.

10.2 Task Overview

In this section we briefly outline what parts of the persistence interface we implemented to show how the ATM application can access enterprise data through the CICS Connector.

Conventions

All classes created in this chapter are in the **itso.entbk2.atm.cics** package.

Only a Subset of the Interface Methods

The ATMPersistenceInterface consists of seven methods. For each type of enterprise access mechanism, a nonempty method body should be developed for each of these methods. Because of time constraints we implemented only two of the interface methods, namely, extGetCard and extGetAccounts.

The main purpose of this chapter is to show an approach to implementing ATMPersistenceInterface methods. The approach involves the use of the CICS Connector. We hope that with such an approach other interface methods can be implemented in a similar or even modified way.

CICS Infrastructure Assumptions

We assume that a CICS Transaction Gateway (with a CICS Gateway for Java and a Universal CICS Client) and a CICS server have been installed and are operational. We do not discuss the installation and customization of these in detail.

CICS Programs

The CICS programs are the same as those used in Chapter 11, “ATM Application Using MQSeries.” These programs are called through the ECI interface. For more information about the programs, see “COBOL Sample Programs” on page 366.

Unit of Work Considerations

The ECI call is associated with a start code of DS. This is equivalent to a distributed program link (DPL) with the SYNCONRETURN option. Thus in the absence of any explicit unit of work requests, the unit of work control is dealt with implicitly.

For enquiry type transactions, whether successful or unsuccessful, all unit of work is controlled implicitly when the CICS program issues an EXEC CICS RETURN.

For update transactions that are successful, all unit of work is controlled implicitly when the CICS program issues an EXEC CICS RETURN.

For update transactions that are unsuccessful, the unit of work is explicitly rolled back before an EXEC CICS RETURN is issued.

Simulated DB2 Calls

DB2 calls have not been coded in the CICS COBOL programs. The data is simulated in memory. Although it certainly would have been preferable to have coded the DB2 calls, the simulation does not in any way affect the discussion on the use of the CICS Connector.

Tasks Implemented

In the rest of this chapter we deal with the creation of classes and beans that are needed to implement the extGetCard and extGetAccounts methods, in the context of using the CICS Connector. We briefly describe the CICS infrastructure needed to access CICS programs through the CICS Connector. In our implementation we used TXSeries Version 4.2 for Windows NT.

10.3 CICS Infrastructure Requirements

In this section we cover the CICS infrastructure requirements for running the CICS Connector.

CICS Server Resources

The CICS server definitions include a client listener and programs.

Client Listener

Define a client listener. In our example it is called ATMTCP and uses TCP/IP. Figure 132 shows the definition using TXSeries Version 4.2 for Windows NT.

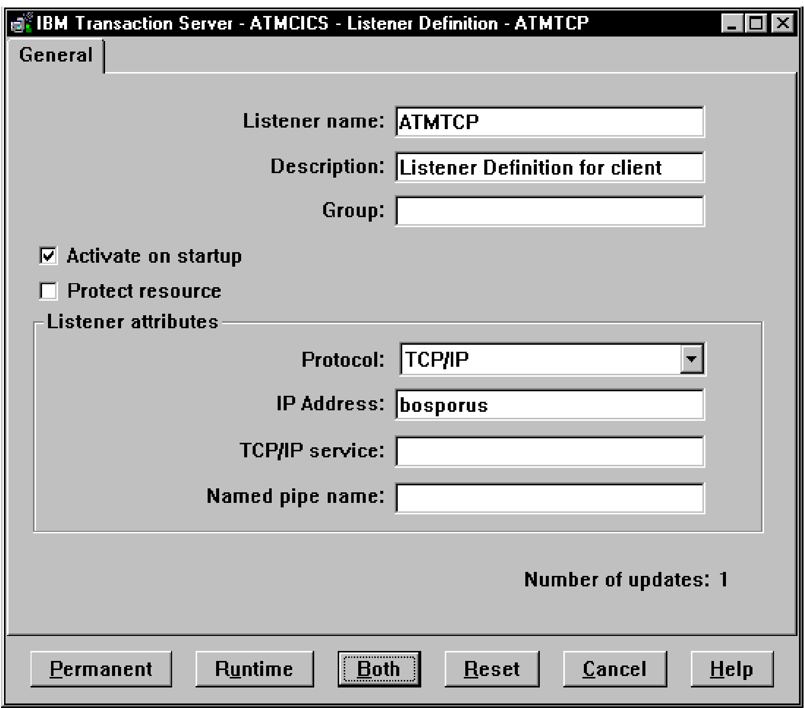


Figure 132. Client Listener Definition

Program Definitions

Assuming that program autoinstall is not active, create a program definition for every program that will be called.

CICS Client Configuration and Startup

On the client machine you have to configure and start the CICS Client.

The CICSCLI.INI file needs to be customized to point to the CICS Server.

- ❑ Make a copy of the CICSCLI.INI file and rename it.
- ❑ Update environment variable CICSCLI to point to this file.
- ❑ Update the server stanza to point to the relevant server, host name, and protocol, in our example, Server=ATMTCP. The customized stanza is shown in Figure 60 on page 98.

Click on the appropriate icon to start the CICS Client (*Programs -> IBM Connectors -> IBM CICS Transaction Gateway -> Start Client*).

Starting the CICS Transaction Gateway

Click on the appropriate icon to start the CICS Transaction Gateway (*Programs -> IBM Connectors -> IBM CICS Transaction Gateway -> CICS Transaction Gateway*).

10.4 Initial Creation of AtmCICS Class

The AtmCICS class implements the ATMPersistenceInterface. We create the AtmCICS class with empty methods. As we develop the CICS Connector for the ATM application, we will modify this class.

Define the AtmCICS class in the itso.entbk2.atm.cics package. Add the ATMPersistenceInterface to the interface implementation list. On the second page of the Create Class SmartGuide, check the box labeled *Methods which must be implemented* so that skeletons for the seven methods of the ATMPersistenceInterface are created.

Save the class for now. We will implement the extGetCard and extGetAccounts methods later.

10.5 ATM Header for the COMMAREA

A design decision was taken to prefix all requests and responses with a header. Every COMMAREA contains a header known as the ATM header.

The ATM header consists of four fields, namely id, date, output length, and return code. The id and date fields should be primed for input requests. The return code should be checked on getting a response back. A nonzero return code indicates that an error has occurred.

To implement the ATM header create an AtmHeader class as a subclass of Object with four properties:

- atmHdrId, of type java.lang.String, read/write, bound
- atmHdrDate, of type java.lang.String, read-only
- atmHdrReturnCode, of type short, read/write, bound
- atmHdrOutputLength, of type int, read/write, bound

All fields will be mapped into a COBOL COMMAREA.

Update the default constructor to initialize the properties:

```
public AtmHeader() {
   super();
   setAtmHdrId("ATM");
   setAtmHdrReturnCode( (short) 9999 );
   setAtmHdrOutputLength(44);
}
```

The header date is set to the current date whenever it is accessed. Update the getAtmHdrDate method:

```
public String getAtmHdrDate() {
   fieldAtmHdrDate = java.util.Calendar.getInstance().getTime().toString();
   fieldAtmHdrDate = (fieldAtmHdrDate + "        ").substring(0,28);
   return fieldAtmHdrDate;
}
```

Auxiliary Method

Add a checkReturnCode method feature that throws an exception if the return code in the header is not zero. The return code is set to a zero string by the CICS COBOL program only if its execution was successful. This method must be called to check the results of the CICS transaction (Figure 133).

```
public void checkReturnCode(short returnCode) throws Exception {
   /* Perform the checkReturnCode method. */
   if (returnCode != 0) {
      System.out.println("ATM Header exception: return code " + returnCode);
      throw (new Exception("Transaction failed: return code " + returnCode));
   }
   return;
}
```

Figure 133. Checking the Return Code of the CICS Transaction

10.6 CICS Transaction to Retrieve an ATM Card

The first interface method we look at is the extGetCard method. This method is invoked when a user enters a card ID.

To implement the extGetCard method, using the CICS Connector, we require:

- ❑ A CICS COBOL program that accesses the card and customer enterprise data. We assume that this program exists and is called ATMCARDI.
- ❑ A record bean representing the COMMAREA associated with the ATMCARDI program.

- A command representing the input-interaction-output flow.
- A navigator that executes the command.
- An implementation of the extGetCard method.

CICS COBOL Program ATMCARDI

When the user enters his or her card ID, enterprise data must be accessed to create a card and customer object as defined in the business object layer. The ATMCARDI program is a CICS COBOL program that performs this function.

Figure 134 shows the COMMAREA of the ATMCARDI program.

```
01 DFHCOMMAREA.
   03 IO-COMMAREA.
      05 ATM-HEADER.
         07 ATMH-ID                  PIC X(4).
         07 ATMH-DATE                PIC X(28).
         07 ATMH-RETURN-CODE.
            09  ATMH-RETURN-CODE-N   PIC 9(4).
         07 ATMH-OUTPUT-LENGTH       PIC 9(8).

      05 ATM-TRANSACTION-SPECIFIC.
         07 ATM-INFO-MESSAGE         PIC X(50).
         07 FILLER                   PIC X(1950).

      05 ATM-CARD-REQ-RESP  REDEFINES ATM-TRANSACTION-SPECIFIC.
         07 ATM-CARD-INFO.
            09  ATM-CARDID          PIC X(7).
            09  ATM-PIN             PIC X(4).
            09  ATM-CUSTID          PIC X(4).
            09  ATM-CUST-TITLE      PIC X(3).
            09  ATM-CUST-FNAME      PIC X(30).
            09  ATM-CUST-LNAME      PIC X(30).
```

Figure 134. COMMAREA of the ATMCARDI Program

The ATM-HEADER is not pertinent to the actual enterprise data. However, it is used for internal processing and for checking the success or failure of the transaction. Originally the COMMAREA was padded to cater for MQSeries messages of varying sizes. To cater for variable record lengths, we use the dynamic record structure provided by the Java record framework. In these examples we have used custom records. Thus we are assuming a fixed length COMMAREA knowing full well that for all the transactions discussed here most of the COMMAREA is not referenced.

The COMMAREA is input for the COBOL dynamic record type generation.

Card Record Bean

The command needs to reference a record bean as input and output. This is a two-step process (see Figure 62 on page 107). The first step creates a COBOL record type; the second step generates the record bean from the record type.

Create the COBOL Card Record Type

To generate a record type from the COBOL source, complete these steps:

❑ Download a local representation of the source code of ATMCARDI into the codepage of your current locale.

❑ In the Workbench, select the *itso.entbk2.atm.cics* package to contain the generated record type.

❑ From the Selected menu, select *Tools -> Records -> Create COBOL RecordType.* The Create a COBOL RecordType SmartGuide opens.

❑ In the Class Name field specify CardRecordType.

❑ In the COBOL file field specify the path and name of the COBOL program you downloaded (Figure 135).

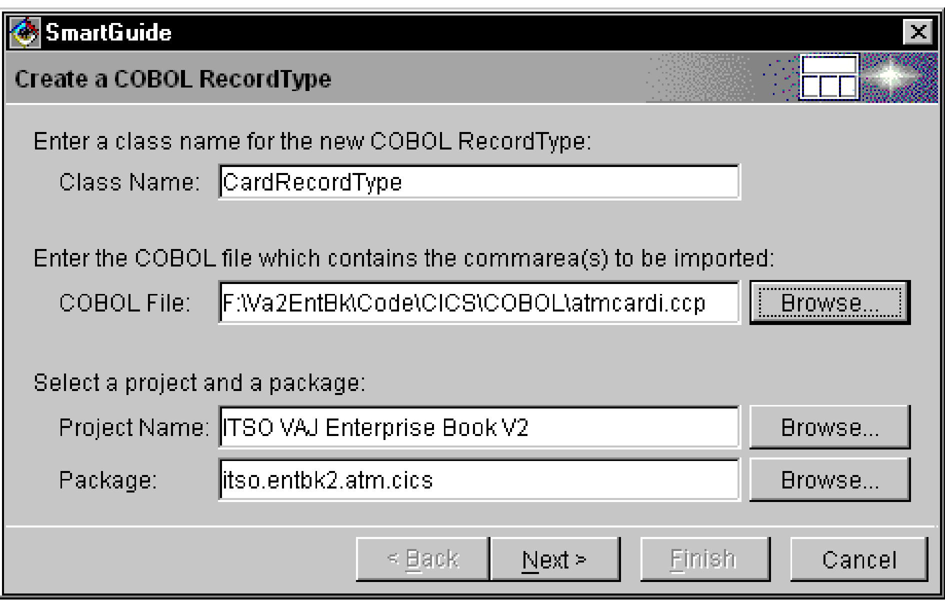


Figure 135. Card Record Type Creation: Class and COBOL File

❑ Click on *Next*. On the second page of the SmartGuide select a level 01 COMMAREA from the COBOL source file and click on > to add the COMMAREA to the Selected commareas list (Figure 136).

❑ Click on *Finish*. A record type is generated into the package you selected.

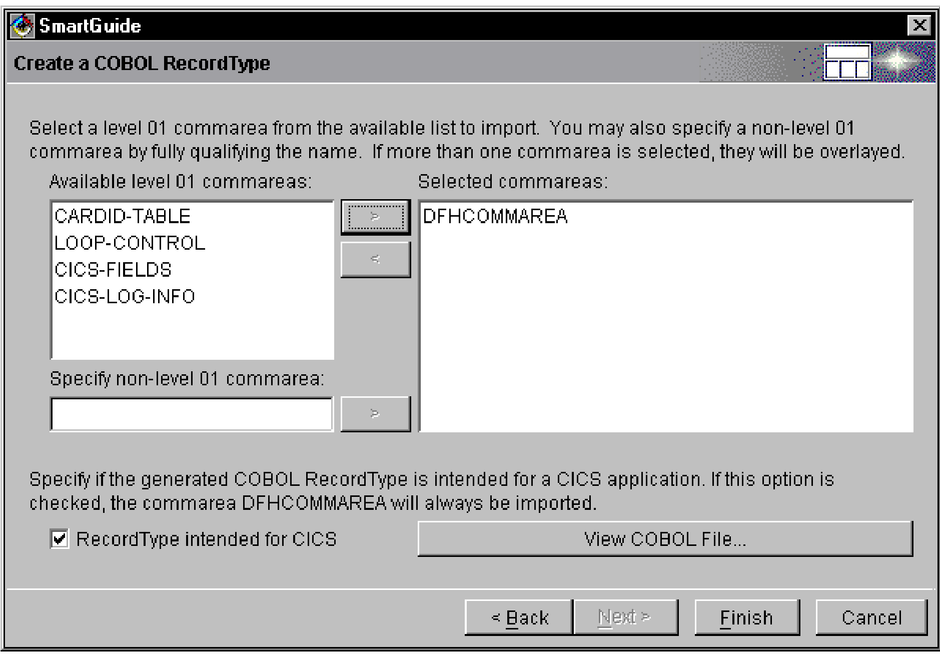


Figure 136. Card Record Type Creation: COMMAREA Selection

Generate the Card Record Bean

Now you can generate the record bean from the COBOL record type:

❑ Select CardRecordType and right-click.

❑ Select *Tools -> Record -> Generate Record*.

❑ Enter a class name of CardRecord.

❑ Check Beans, Direct, and Custom Records.

❑ Click on *Next*. The properties of the Record Attributes Bean are displayed.

❑ If necessary alter the properties according to the platform of the target CICS Server.

❑ Click on *Finish*.

Two classes are generated, namely, CardRecord and the associated CardRecordBeanInfo. (CardRecord is a real JavaBean.)

Figure 137 shows the setting of the properties for TXSeries Version 4.2 on Windows NT.

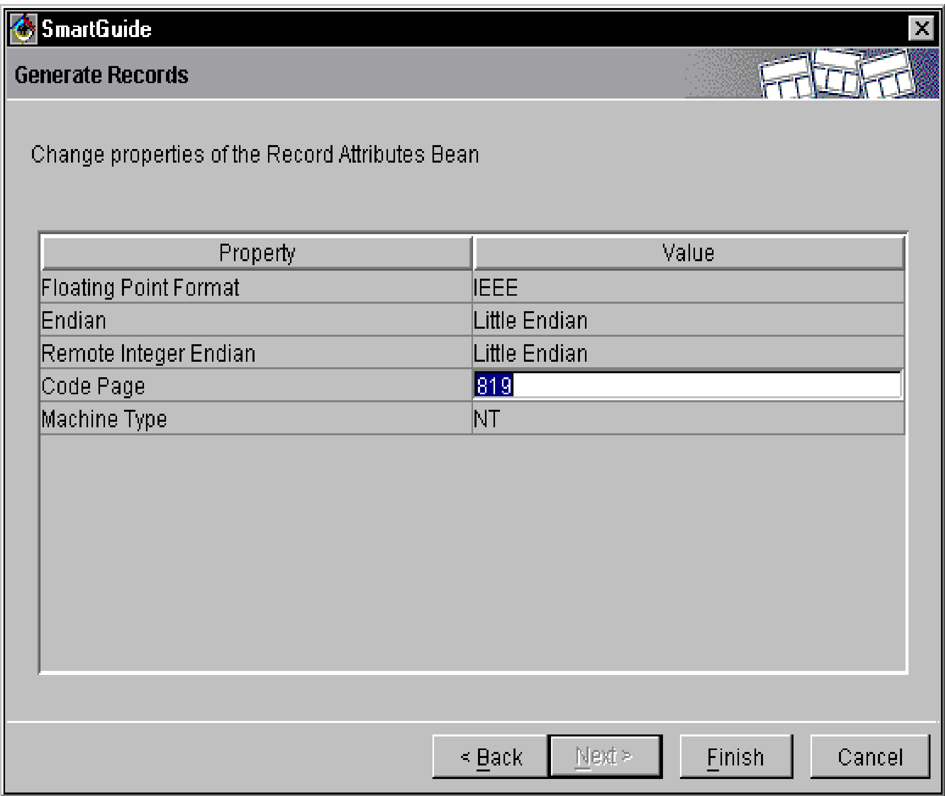


Figure 137. Card Record Bean Generation

Card Command

To access the enterprise data with the CICS Connector it is necessary to create a command. For this transaction we create a command that does not use a mapper bean. The command is constructed with the Command Editor. Note that it is also possible to construct a command with the Visual Composition Editor.

CardCommand Class

Create the CardCommand class as a subclass of com.ibm.ivj.eab.command.CommunicationCommand. See “Constructing a Command” on page 112 for detailed instructions.

Command Editor

Once the class is created we construct the command, using the Command Editor:

❑ Select the CardCommand class and open the Command Editor (*Tools -> Command Editor*).

❑ Add a CICSConnectionSpec and an ECIInteractionSpec.

❑ Open the ceConnectionSpec and change its properties. In our case the CICSServer is ATMTCP and the URL is bosporus (Figure 138). Our CICS Transaction Gateway is installed on a machine with the host name bosporus.

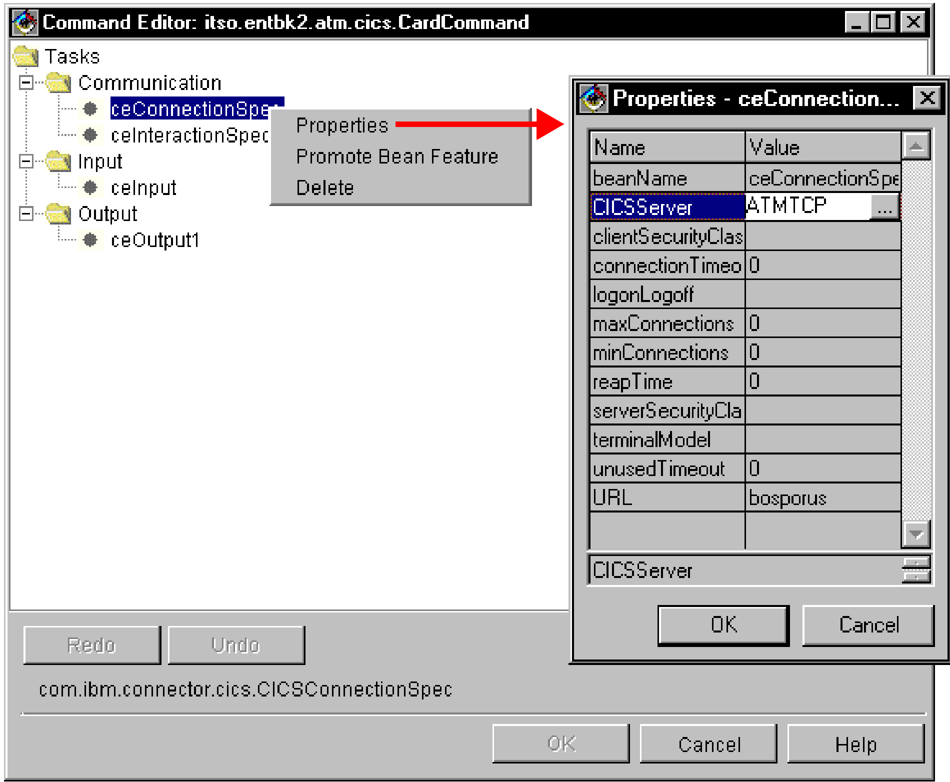


Figure 138. CardCommand with CICSConnectionSpec Properties

❑ Open the ceInteractionSpec and change the CICS program property to ATMCARDI (our CICS program).

❑ Associate the Input with the CardRecord input bean.

- ❑ On the input record select *Promote Bean Features* and promote the ATM_CARDID, ATMH_ID, ATMH_DATE, ATMH_OUTPUT_LENGTH, and ATMH_RETURN_CODE_N properties.
- ❑ Associate the Output with the CardRecord output bean.
- ❑ On the output record select *Promote Bean Features* and promote the ATM_CARDID, ATM_CUST_FNAME, ATM_CUST_LNAME, ATM_CUST_TITLE, ATM_CUSTID, ATM_PIN, and ATMH_RETURN_CODE_N properties.
- ❑ Click on *OK* to generate the CardCommand.

Building a Navigator to Execute the CICS Transaction

It is necessary to create a class that executes the CardCommand command. We could just create a class that does the job, but the CICS Connector provides a suitable class for us, the navigator.

What Is a Navigator?

A navigator is a wrapper that can execute one command or multiple commands in sequence. Commands can be chained together depending on the successful or unsuccessful event of previous commands, or depending on the results of the command execution.

A navigator provides an execute method to start processing. It also provides methods to set a successful or unsuccessful completion event. This allows you to build higher level beans by nesting navigators and commands.

We describe an advanced navigator in “Using an Advanced Navigator” on page 273. For now we build a simple navigator that executes one command.

Navigator for the Card Command

We create the CICSCardNavigator class as a subclass of the navigator class, com.ibm.ivj.eab.command.CommunicationNavigator, and open the Visual Composition Editor (Figure 139):

- ❑ Add a CardCommand and an AtmHeader bean to the free-form surface.
- ❑ Add two factories of types Card and Customer to the free-form surface.
- ❑ Add a variable of type String to the free-form surface and name it CardId.

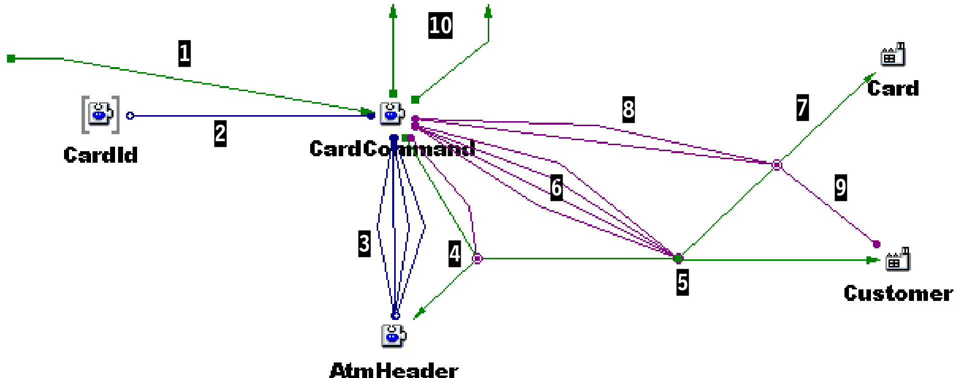


Figure 139. Visual Composition of CICSCardNavigator Class

Connections

In the Visual Composition Editor add the following connections:

- ❑ Connect the *internalExecutionStarting* event of the navigator class to the *execute* method of CardCommand (1).
- ❑ Connect the *this* property of CardId to the *ceInputATM_CARDID* property of CardCommand. Make sure that the target event is set to *none* (2).
- ❑ Connect the *atmHdrId*, *atmHdrDate*, *atmHdrOutputLength*, and *atmHdrReturnCode* properties of AtmHeader to the corresponding CardCommand properties (*ceInputATMH__ID*, *ceInputATMH__DATE*, *ceInputATMH__OUTPUT__LENGTH*, *ceInputATMH__RETURN__CODE__N*) (3).
- ❑ Connect the *executionSuccessful* event of the CardCommand to the *checkReturnCode* method of AtmHeader and pass the *ceOutput1ATMH__RETURN__CODE__N* property (of CardCommand) as a parameter (4).
- ❑ Connect the *normalResult* of the connection to the constructor (with parameters) of Customer (5).
- ❑ Connect each constructor parameter to the corresponding *ceOutput1xxxx* properties of CardCommand (6).
- ❑ Connect the *normalResult* of the instantiation of Customer to the constructor (with parameters) of Card (7).
- ❑ Connect the *cardNumber* and *pinCard* parameters to the corresponding *ceOutput1xxxx* properties of CardCommand (8).
- ❑ Connect the *customer* parameter or the Card constructor to the *this* property of Customer (9).

- ❑ Connect the *executionSuccessful* event of the CardCommand to the *returnExecutionSuccessful* method of the navigator, and connect the *executionUnsuccessful* event of the CardCommand to the *returnExecutionUnsuccessful* method (**10**).
- ❑ Save the bean.

Promote External Features

The *this* properties of CardId and Card must be promoted for external access. This generates public methods *setCardIdThis* and *getCard,* which will be used in the extGetCard method of the AtmCICS bean.

Implement the extGetCard Method

Now that all the components have been developed, we can implement the extGetCard method in the AtmCICS class.

Add CICSCardNavigator Bean to AtmCICS

Open the Visual Composition Editor for the AtmCICS class. Add a CICSCardNavigator bean to the free-form surface and label it CICSCardNavigator. Close the window and save.

Code the extGetCard Method

Figure 140 shows how the extGetCard is implemented to instantiate a CICSCardNavigator object, set the card ID, and call its execute method. The resulting card object is retrieved after executing the navigator.

```
public itso.entbk2.atm.model.Card extGetCard(String cardId) throws Exception {
   itso.entbk2.atm.model.Card card = null;
   getCICSCardNavigator().setCardIdThis(cardId);
   getCICSCardNavigator().execute();
   card = getCICSCardNavigator().getCard();
   return card;
}
```

Figure 140. Implementation of extGetCard

At this stage the CICS function to retrieve card and customer information can be invoked.

10.7 Using Mappers

Instead of using the Visual Composition Editor to connect the properties of one or more business objects to the properties of a command, it is possible to achieve this mapping through mapper beans. These mapper beans map the properties of one or more business objects or classes to a record bean. Mappers are added to a command.

In this section we develop an alternative command with input and output mappers for the extGetCard function. The new command requires a change in the class that executes the command.

Input Mapper for Card

The input request consists of the AtmHeader and a field representing a card ID. We map the AtmHeader to the input record but pass the card ID directly.

Use Mapper Editor to Create CardInputMapper

We create a mapper named CardInputMapper to set up the COMMAREA for input.

- ❑ Select the CardRecord and *Tools -> Mapper Editor* from the context menu. The Mapper Editor is presented with the input bean primed with the properties of the CardRecord bean.
- ❑ Click on *Add* and add the AtmHeader class to the output beans.
- ❑ Map the four AtmHeader properties to the CardRecord properties:
  - Select the AtmHdrDate property in the AtmHeader (output bean).
  - Select the ATMH_ID property in the CardRecord (input bean).
  - Select the connection direction from the AtmHeader to the CardRecord (arrow left to right).

  Perform the same steps for the other three properties (Figure 141).

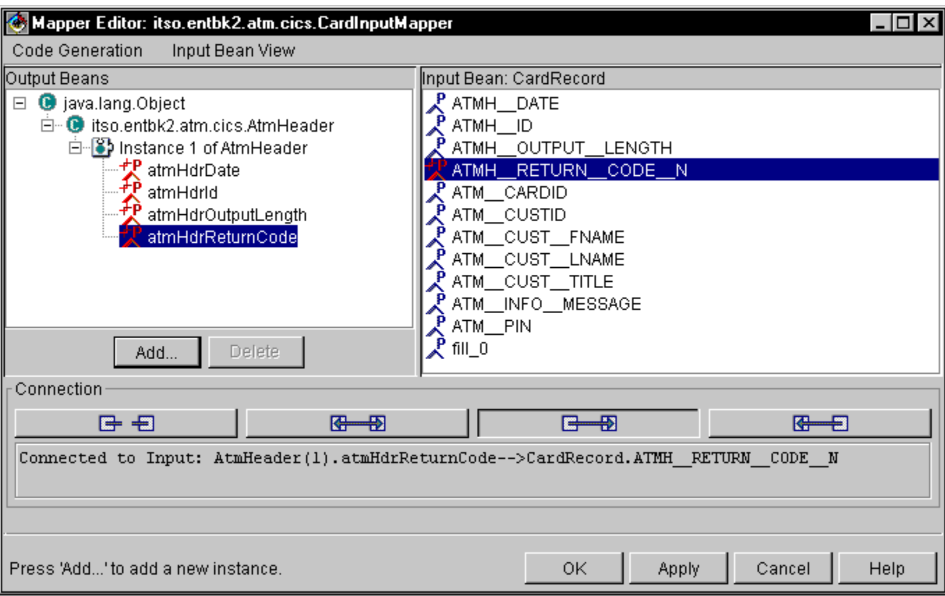


Figure 141. Mapper Editor for Input Record Mapping

Click on *OK* and enter CardInputMapper when you are prompted for a class name. The CardInputMapper class is generated.

Output Mapper for Card

The output response consists of the AtmHeader, customer, and card information. To map the COMMAREA to properties of the Card and Customer beans create the CardOutputMapper:

- ❑ Open a new Mapper Editor on the CardRecord. The input bean is primed with the properties.
- ❑ Add three classes to the output bean: Customer, Card, and AtmHeader.
- ❑ Map the customer ID, first name, last name, and title of the customer to matching properties in the CardRecord. Be sure to click on the right to left arrow (input bean to output bean).
- ❑ Map the card ID and PIN of the card to matching properties. You cannot set the customer property through mapping; this must be done by coding.
- ❑ Map the return code of the AtmHeader to the return code in the input.
- ❑ Click on *OK* and enter CardOutputMapper as the class to be generated.

Create a Command with Mappers

To use the input and output mappers, we create a new CardMapperCommand class, similar to the CardCommand class. Follow the steps outlined in “Command Editor” on page 251, replacing references to CardCommand with CardMapperCommand.

Now we add the mappers. Open the Command Editor on the CardMapperCommand class.

Add the Mappers

To add the mapper to the input record select *Add Mapper* in the context menu and specify CardInputMapper as the class. In the same way add the CardOutputMapper class to the output record (Figure 142).

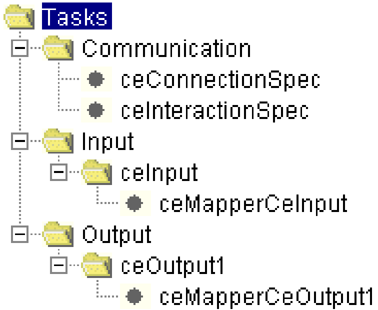


Figure 142. Command Editor with Mappers

Execute the CICS Transaction with Mappers

In “Building a Navigator to Execute the CICS Transaction” on page 252 we create a CICSCardNavigator class that calls the execute method of the CardCommand bean.

In this section we create a CICSCardMapperNavigator class that calls the execute method of the CardMapperCommand. CICSCardMapperNavigator is similar to CICSCardNavigator except that the creation of the card and customer objects occurs within the command itself.

Add a new CICSCardMapperNavigator class to the itso.entbk2.atm.cics package. In the Visual Composition Editor:

❑ Add a CardMapperCommand and an AtmHeader bean to the free-form surface.

❑ Add a variable of type Card to the free-form surface.

❑ Add a variable of type String to the free-form surface and name it CardId (Figure 143).

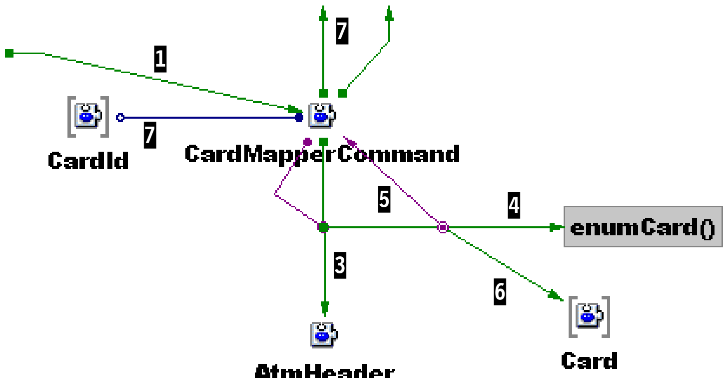


Figure 143. Visual Composition of the CICSCardMapperAccess Class

Extracting the Card and Customer

Once the execute method of the CardMapperCommand has run successfully, an instance of the card and customer classes is created. The card object is associated with the customer object. This association has not been mapped in the mapper bean and must be created now.

The command has a *mappedObjects* property (an enumeration) that contains all the objects that were created. We extract the card and customer objects and associate them with a new method called enumCard (Figure 144).

```
private itso.entbk2.atm.model.Card  enumCard(java.util.Enumeration enum ) {
   Object nextObj = null;
   itso.entbk2.atm.model.Card card = null;
   itso.entbk2.atm.model.Customer customer = null;
   if (enum == null) return null;
   while (enum.hasMoreElements()) {
      nextObj = enum.nextElement();
      if (nextObj instanceof itso.entbk2.atm.model.Card)
         card = (itso.entbk2.atm.model.Card)nextObj;
      if (nextObj instanceof itso.entbk2.atm.model.Customer)
         customer  = (itso.entbk2.atm.model.Customer)nextObj;
   }
   card.setCustomer(customer);
   return card;
}
```

Figure 144. Extracting Objects from a Command

Connections

We create the logic in the Visual Composition Editor:

- ❑ Connect the *internalExecutionStarting* event of the navigator class to the *execute* method of CardMapperCommand (1).
- ❑ Connect the *this* property of CardId to the *ceInputATM_CARDID* property of CardMapperCommand. Make sure that the target event is set to *none* (2).
- ❑ Connect the *executionSuccessful* event of the CardMapperCommand to the *checkReturnCode* method of AtmHeader and pass the *ceOutput1ATMH_RETURN_CODE_N* property as a parameter (3).
- ❑ Connect the *normalResult* of the connection to the *enumCard* method (event-to-code) (4).
- ❑ Connect the *enum* parameter of the connection to the *connectionMappedObjects* method of the CardMapperCommand (5).
- ❑ Connect the *normalResult* of the enumCard method to the *this* of Card (6).
- ❑ Connect the *executionSuccessful* (and *executionUnsuccessful*) event of the CardMapperCommand to the *returnExecutionSuccessful (and returnExecutionUnsuccessful)* method of the navigator (7).
- ❑ Save the bean.

Promote External Features

The *this* properties of CardId and Card must be promoted for external access. This generates public methods *setCardIdThis* and *getCardThis,* which will be used in the extGetCard method of the AtmCICS bean.

Note that for a variable the method name is *getCardThis*; for the factory it was getCard.

Change the AtmCICS Class to Use the Mappers

CICSCardMapperNavigator is an alternative to CICSCardNavigator. The extGetCard instantiates an object of class CICSCardNavigator (Figure 140 on page 254).

We enhance the AtmCICS class to work with either the CICSCardNavigator or CICSCardMapperNavigator class:

- ❑ Open the AtmCICS class and add a *cardMapperSwitch* property of type boolean in the BeanInfo page. With this switch you can set which class is used in the extGetCard method.

- ❑ In the Visual Composition Editor add a bean of type CICSCardMapperNavigator to the free-form surface.
- ❑ Change the extGetCard method to initialize either CICSCardNavigator or CICSCardMapperNavigator (Figure 145).

```
public itso.entbk2.atm.model.Card extGetCard(String cardId) throws Exception {
   itso.entbk2.atm.model.Card card = null;
   if (getCardMapperSwitch() == false) {
      getCICSCardNavigator().setCardIdThis(cardId);
      getCICSCardNavigator().execute();
      card = getCICSCardNavigator().getCard();
   }
   else {
      getCICSCardMapperNavigator().setCardIdThis(cardId);
      getCICSCardMapperNavigator().execute();
      card = getCICSCardMapperNavigator().getCardThis();
   }
   if (card == null) throw new java.lang.Exception("Card not found");
   return card;
}
```

Figure 145. Implementation of Enhanced extGetCard

At this stage the CICS function to retrieve card and customer information can be invoked with either the CICSCardNavigator or the CICSCardMapperNavigator.

10.8 Test the CICS Card Transaction

You can now go ahead and test the CICS transaction that retrieves a card. With systems like CICS it is a good idea, however, to prepare for testing.

Prepare Test Output for Card Transaction

To display some useful information when testing the CICS transaction we modify the two command classes, CardCommand and CardMapperCommand:

- ❑ Change the handleException method according to the help documentation by adding one line of code:

```
private void handleException(Throwable exception) {
   this.internalExceptionHandler(exception);
}
```

- ❑ Overwrite the checkInputState method that is inherited from the CommunicationCommand class to display input record data:

```
public void checkInputState()
                throws com.ibm.ivj.eab.command.InvalidInputStateException {
   System.out.println("Card Command Input State:");
   System.out.println("cardid="+getCeInputATM__CARDID());
   System.out.println("headid="+getCeInputATMH__ID());
   System.out.println("date  ="+getCeInputATMH__DATE());
   System.out.println("outlg ="+getCeInputATMH__OUTPUT__LENGTH());
   System.out.println("retcod="+getCeInputATMH__RETURN__CODE__N());
}
```

- ❑ Overwrite the afterInternalExecution method to display the output record data after the execution of the command:

```
public void afterInternalExecution
                  (com.ibm.ivj.eab.command.CommandEvent param1) {
   System.out.println("Command execute: ");
   System.out.println("cardid="+getCeOutput1ATM__CARDID());
   System.out.println("pin   ="+getCeOutput1ATM__PIN());
   System.out.println("cust  ="+getCeOutput1ATM__CUST__FNAME()+
      " "+getCeOutput1ATM__CUST__LNAME()+
      " "+getCeOutput1ATM__CUST__TITLE());
   System.out.println("retcod="+getCeOutput1ATMH__RETURN__CODE__N());
}
```

- ❑ For testing we directly invoke the extGetCard method of the AtmCICS class.

Testing Card Transaction with a Scrapbook Script

The functionality of the card transaction can be tested with a small Scrapbook script (Figure 146).

```
// Test the CICS card transaction

itso.entbk2.atm.cics.AtmCICS atmcics = new itso.entbk2.atm.cics.AtmCICS();

atmcics.setCardMapperSwitch(false);   // set to true for CardMapperCommand

itso.entbk2.atm.model.Card card1 = atmcics.extGetCard("1111111");
System.out.println(card1);
```

Figure 146. Scrapbook for CICS Card Transaction Testing

Testing without CICS

When you run the Scrapbook test without a CICS system, the Console window should display output similar to this:

```
Card Command Input State:
cardid=1111111
headid=ATM
date  =Thu Nov 19 20:45:37 PST 1998
outlg =44
retcod=9999
*** CICSManagedConnection.getCICSGatewayObject() exception:
   java.io.IOException: CCL6651E: Unable to connect to the Gateway.
   [address = bosporus, port = 2006] [java.net.SocketException: No route to host].
*** ECIHelper.eciRequest() exception: java.io.IOException: CCL6651E:
   Unable to connect to the Gateway. [address = bosporus, port = 2006]
   [java.net.SocketException: No route to host].
Command execute:
cardid=
pin   =
cust  =
retcod=0
java.lang.Exception: Card not found
```

10.9 Discussion Review

Thus far we have discussed how to use the CICS Connector to create classes and beans necessary to implement the extGetCard method. We have shown the construction of a record bean, a command, a mapper, and a navigator that invokes the execute method of the command. On completion of the extGetCard method, a card and customer object should have been instantiated.

The next requirement is to populate the card object with the associated account details. This is implemented in the extGetAccounts method of the persistence layer.

10.10 CICS Transaction to Retrieve Accounts

Once a card object has been instantiated, it is necessary to get the details of all accounts associated with that card. In the ATM application, there are two types of accounts, checking and savings. These two types are represented, respectively, by the CheckingAccount and SavingsAccount classes in the itso.entbk2.atm.model package. A card object is associated with a vector of BankAccounts. BankAccount is a superclass of SavingsAccount and CheckingAccount.

In this section, the enterprise account data is retrieved by a CICS COBOL program called ATMACCNT. Given a card, it retrieves card account data. This is returned in the COMMAREA. It is the function of the extGetAccounts method to call this CICS COBOL program. On getting a response, the accounts must be set up in a vector that consists of both SavingsAccount and CheckingAccount instances.

The implementation of the extGetAccounts method with the CICS Connector requires:

- A CICS COBOL program that accesses the accounts associated with a customer card. This program exists and is called ATMACCNT.
- A record bean representing the COMMAREA associated with the ATMACCNT program.
- A command representing the input-interaction-output flow.
- A mapper for the output COMMAREA to move the return code to the return code associated with the AtmHeader.
- A navigator that executes the command.
- An implementation of the extGetAccounts method.

CICS COBOL Program ATMACCNT

Once a customer and card object have been instantiated, the account details associated with that card must be retrieved from the enterprise data store. The ATMACCNT program performs this function.

Figure 147 shows the COMMAREA of the ATMACCNT program.

```
01 DFHCOMMAREA.
   03 IO-COMMAREA.
      05 ATM-HEADER.
         07 ATMH-ID                  PIC X(4).
         07 ATMH-DATE                PIC X(28).
         07 ATMH-RETURN-CODE.
            09  ATMH-RETURN-CODE-N   PIC 9(4).
         07 ATMH-OUTPUT-LENGTH       PIC 9(8).

      05 ATM-TRANSACTION-SPECIFIC.
         07 ATM-INFO-MESSAGE         PIC X(50).
         07 FILLER                   PIC X(1950).

      05 ATM-ACCOUNTS-REQRESP REDEFINES ATM-TRANSACTION-SPECIFIC.

         07  ATM-ACCOUNTS-FIXED.
             09  ATM-CARDID          PIC X(7).
             09  ATM-NO-ACCOUNTS     PIC 9(2).

         07  ATM-ACCOUNTS.
             09  ATM-ACCOUNT-DETAILS OCCURS 10.
                 11  ATM-ACCID       PIC X(8).
                 11  ATM-ACCTYPE     PIC X(1).
                 11  ATM-BALANCE     PIC 9(6).99.
                 11  ATM-MINAMT      PIC 9(6).99.
                 11  ATM-OVERDRAF    PIC 9(6).99.
```

Figure 147. COMMAREA of the ATMACCNT Program

The COMMAREA consists of an ATM header, a card id (ATM-CARDID), a field specifying the number of accounts (ATM-NO-ACCOUNTS), and an array of account details (ATM-ACCOUNTS). An artificial limit of 10 has been imposed on the number of accounts associated with a card. In reality there would be no limit, and the dynamic record access mechanism would be used for record bean generation. We have instead chosen to implement the custom record option where we have imposed a fixed length record layout.

Accounts Record Bean

A command needs to reference a record bean as input and output. It is therefore necessary to generate a record bean from a COBOL record type. This record bean is called AccountsRecord.

Create the COBOL Accounts Record Type

The steps are identical to "Create the COBOL Card Record Type" on page 248 and are not repeated here. The output record type is called AccountsRecordType.

Edit the Accounts Record Type

We are not actually going to change the record type but want to show you the Java Record Editor. Select *Tools -> Records -> Edit RecordType* on the AccountsRecord (Figure 148).

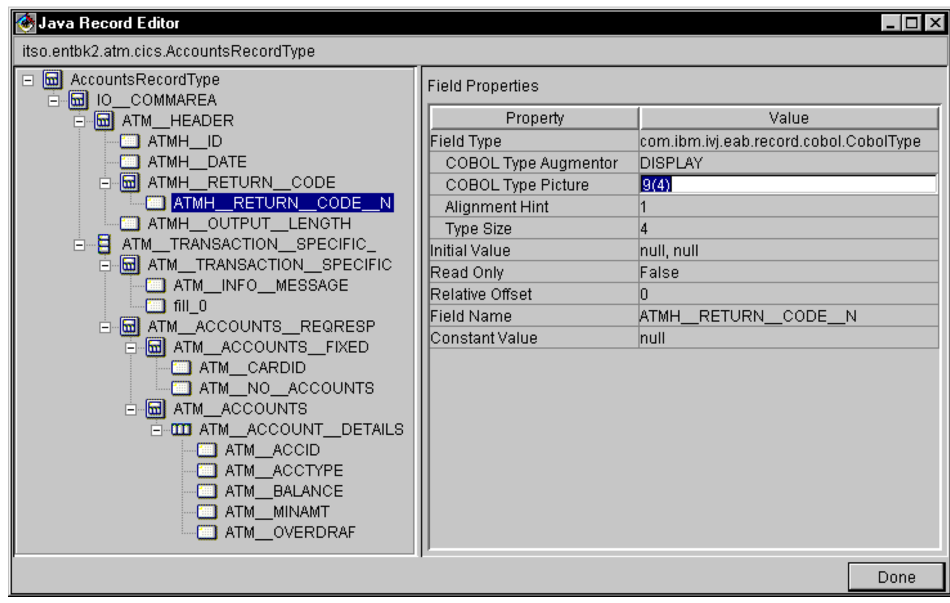


Figure 148. Edit of Accounts Record Type

Should the return code contain alphanumeric values, we would change the COBOL Type Picture field to X(4).

Generate the Accounts Record Bean

Now we generate the AccountsRecord bean from the record type in the same way as in “Generate the Card Record Bean” on page 249. Figure 149 shows the classes that are generated.

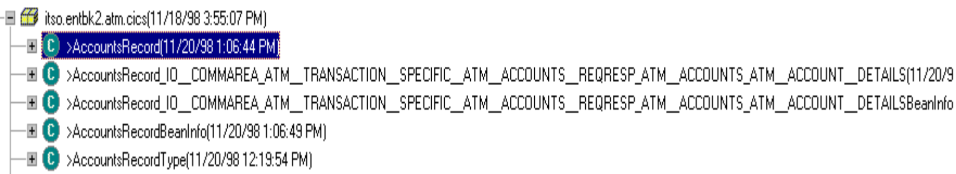


Figure 149. Classes Generated from Accounts Record Type

Classes for Arrays

The class that is generated with the very long name

```
AccountsRecord_IO__COMMAREA_ATM__TRANSACTION__SPECIFIC__ATM__ACCOUNTS__REQRESP_
ATM__ACCOUNTS_ATM__ACCOUNT__DETAILS
```

is used to access an account array occurrence. If you look inside the AccountsRecord class, you find two methods to access the account occurrences:

- getATM__ACCOUNT__DETAILS() retrieves the whole array of accounts.
- getATM__ACCOUNT__DETAILS(int) retrieves one account instance.

It looks like the COBOL REDEFINES and OCCURS clauses cause the generation of very long names!

Accounts Input Mapper

We create a mapper named AccountsInputMapper to set up the COMMAREA with the AtmHeader information.

Start the Mapper Editor for the AccountsRecord (see "Use Mapper Editor to Create CardInputMapper" on page 255).

Add an AtmHeader instance and map the four properties to the matching properties of the AccountsRecord. Generate the mapper class as AccountsInputMapper.

Accounts Command

To access the enterprise data with the CICS Connector we create a command with the Command Editor.

AccountsCommand Class

Create the AccountsCommand class as a subclass of CommunicationCommand.

Command Editor

Start the Command Editor on the AccountsCommand class:

- Add a CICSConnectionSpec and an ECIInteractionSpec and set the properties to ATMTCP (CICSServer), bosporus (URL), and ATMACCNT (Program).
- Associate the input and the output with the AccountsRecord bean.

- ❑ Promote the ATM__CARDID property of the input bean.
- ❑ Add the AccountsInputMapper to the input bean.
- ❑ Promote the ATMH_RETURN_CODE_N, ATM_ACCOUNT_DETAILS, and ATM_NO_ACCOUNTS properties of the output bean.
- ❑ Click on *OK* to generate the AccountsCommand.

Navigator to Execute the CICS Accounts Transaction

To execute the AccountsCommand we create a CICSAccountsNavigator class (subclass of CommunicationNavigator), using the Visual Composition Editor (Figure 150):

- ❑ Add an AccountsCommand and an AtmHeader bean to the free-form surface.
- ❑ Add a variable of type String and name it CardId.
- ❑ Add a variable of type Vector and name it AccountsVector (this is the result of the accounts transaction).

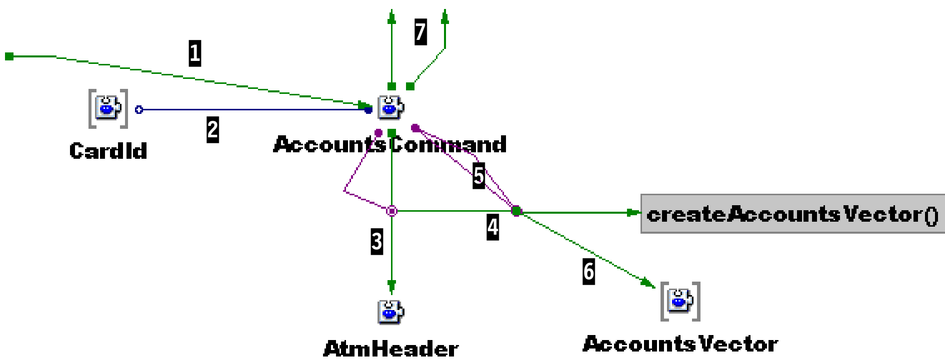


Figure 150. Visual Composition of CICSAccountsNavigator

Connections

In the Visual Composition Editor add the following connections:

- ❑ Connect the *internalExecutionStarting* event of the navigator class to the *execute* method of AccountsCommand (1).
- ❑ Connect the property of CardId to the *ce1InputATM__CARDID* property of AccountsCommand. Set the target event to *none* (2).
- ❑ Connect the *executionSuccessful* event of AccountsCommand to the *checkReturnCode* method of AtmHeader and pass the *ceOutput1ATMH__RETURN__CODE__N* property as parameter (3).

❑ Connect the *normalResult* of the connection to a new *createAccountsVector* method (event-to-code) (4).

The createAccountsVector method has to create the result vector of accounts that belong to the card given as input. Define the method with a return type of *java.util.Vector* and these two parameters:

- noOfAccounts, type int
- accounts, type AccountsRecord_IO__COMMAREA_ATM__TRAN...[] (an array of the very long class name)

We will develop the code of the method later.

❑ The call to the createAccounts method requires two parameters. Pass the number of accounts (property *ceOutput1ATM__NO__ACCOUNTS*) and the accounts array (property *ceOutput1ATM__ACCOUNT__DETAILS*) as parameters (5).

❑ Connect the *normalResult* of the createAccountsVector method call to the *this* of the AccountsVector variable. Pass the event data (the vector) as a parameter. (6)

❑ Connect the *executionSuccessful* (and *executionUnsuccessful*) event of the AccountsCommand to the *returnExecutionSuccessful* (and *returnExecutionUnsuccessful)* method of the navigator. (7)

Creating the Vector of Accounts

The objective of the CICS transaction is to populate a card object with a vector of bank accounts. On completion of the ATMACCNT CICS program, an array of accounts is returned. The array is only a data stream, however. Something is needed to format this array into a vector of bank accounts where the individual elements are either a savings account or a checking account object. This is the role of the createAccountsVector method.

The array of accounts and the number of occurrences in the array are in the CICS output COMMAREA. The COMMAREA is represented by the AccountsRecord bean. We promote the two properties in the output bean and pass them as parameters into the createAccountsVector method.

The createAccountsVector method creates a vector of accounts from the array of accounts in the output bean of the command. The method runs through the number of accounts retrieved in the accounts array and creates a new checking account or savings account object according to the account type.

The class that represents an account in the array of accounts provides the required methods to extract the properties of an account, for example, getATM__ACCID and getATM__BALANCE. These properties are used in the constructor of the checking or savings account.

The new bank account objects are added to the result vector, which in turn is returned to the caller.

Figure 151 shows the code of the createAccountsVector method.

```
public java.util.Vector createAccountsVector
   (int noOfAccounts,
    AccountsRecord_IO__COMMAREA_ATM__TRANSACTION__SPECIFIC__ATM__ACCOUNTS__
       REQRESP_ATM__ACCOUNTS_ATM__ACCOUNT__DETAILS[] accounts) {
   itso.entbk2.atm.model.BankAccount bankAccount;
   String acctype = null;
   String acctid = null;
   java.math.BigDecimal balance = null;
   java.math.BigDecimal minAmt = null;
   java.math.BigDecimal overDraft = null;
   java.util.Vector vecAccounts = new java.util.Vector();
   System.out.println("Creating accounts vector ...");

   for (int i=0 ; i < noOfAccounts ; i++) {
      acctype = accounts[i].getATM__ACCTYPE();
      acctid = accounts[i].getATM__ACCID();
      balance = new java.math.BigDecimal(accounts[i].getATM__BALANCE());
      minAmt = new java.math.BigDecimal(accounts[i].getATM__MINAMT());
      overDraft = new java.math.BigDecimal(accounts[i].getATM__OVERDRAF());

      if (acctype.equalsIgnoreCase("C"))
         bankAccount = new itso.entbk2.atm.model.CheckingAccount
            (acctid, balance, overDraft);
      else
         bankAccount = new itso.entbk2.atm.model.SavingsAccount
            (acctid, balance, minAmt);

      vecAccounts.addElement(bankAccount);
      System.out.println(" - added account: "+bankAccount);
   }
   return vecAccounts;
}
```

Figure 151. Code Listing of createAccountsVector Method

Promote External Features

The *this* property of CardId and the AccountsVector must be promoted for external access. This generates public methods *setCardIdThis* and *getAccountsVectorThis,* which will be used in the extGetAccounts method of the AtmCICS bean.

Implement the extGetAccounts Method

The next step is to include the CICSAccountsNavigator class in the AtmCICS class and code the extGetAccounts method to invoke the transaction.

Add CICSAccountsNavigator Bean to AtmCICS

Open the Visual Composition Editor for the AtmCICS class. Add a CICSAccountsNavigator bean to the free-form surface and label it CICSAccountsNavigator. Close the window and save.

Code the extGetAccounts Method

Figure 152 shows how the extGetAccounts method initializes the CICSAccountsNavigator with the card ID and then calls the execute method of the CICSAccountsNavigator. The createAccountsVector method of the navigator creates the vector of associated accounts that is retrieved and assigned to the card object.

```
public void extGetAccounts(itso.entbk2.atm.model.Card card) throws Exception {
   getCICSAccountsNavigator().setCardIdThis(card.getCardNumber());
   getCICSAccountsNavigator().execute();
   card.setAccounts(getCICSAccountsNavigator().getAccountsVectorThis());
   if (getCICSAccountsNavigator().getAccountsVectorThis() == null)
      System.out.println("No accounts for "+card);
}
```

Figure 152. extGetAccounts Method

At this stage the CICS function to prime a card object with its associated accounts can be invoked.

10.11 Testing the CICS Accounts Transaction

Before we test the CICS transaction to retrieve the accounts of a card, let's prepare for testing.

Prepare Test Output for Accounts Transaction

We modify the AccountsCommand class with test output as we did for the CardCommand and CardMapperCommand classes:

❑ Change the handleException method according to the help documentation by adding one line of code:

```
this.internalExceptionHandler(exception);
```

❑ Overwrite the checkInputState method that is inherited from the CommunicationCommand class to display the cardId in the input record:

```
public void checkInputState()
                throws com.ibm.ivj.eab.command.InvalidInputStateException {
   System.out.println("Card Command Input State:");
   System.out.println("cardid="+getCeInputATM__CARDID());
}
```

❑ Overwrite the afterInternalExecution method to display the output record data after the execution of the command:

```
public void afterInternalExecution
                (com.ibm.ivj.eab.command.CommandEvent param1) {
   System.out.println("Command execute: ");
   System.out.println("retcod="+getCeOutput1ATMH__RETURN__CODE__N());
   System.out.println("noAcct="+getCeOutput1ATM__NO__ACCOUNTS());
   if (getCeOutput1ATM__NO__ACCOUNTS() > 0)
   System.out.println("acct1 ="+getCeOutput1ATM__ACCOUNT__DETAILS(0));
}
```

Testing the Accounts Transaction with a Scrapbook Script

We use a Scrapbook script to test the accounts transaction (Figure 153).

```
// Test the CICS accounts transaction

itso.entbk2.atm.cics.AtmCICS atmcics = new itso.entbk2.atm.cics.AtmCICS();
itso.entbk2.atm.model.Card card;

card = new itso.entbk2.atm.model.Card("1111111","1111",null);
System.out.println(card);

atmcics.extGetAccounts(card);
System.out.println(card);
try {
   java.util.Enumeration enum = card.getAccounts().elements();
   while (enum.hasMoreElements())
     { System.out.println( (itso.entbk2.atm.model.BankAccount)enum.nextElement() ); }
} catch (Exception e) { System.out.println("card has no accounts"); }
```

Figure 153. Scrapbook Script for CICS Accounts Transaction Testing

10.12 Testing the ATM Application with CICS

To test the sequence of the two transactions with the CICS implementation we use the ATM application controller.

We can write a small Scrapbook script for initial testing and then use the real ATM servlet application for final testing.

Figure 154 shows a Scrapbook script for the two transactions using the ATM application controller.

```
// set Page->Run in to ATMApplicationController class in itso.entbk2.atm.model

ATMApplicationController ctl = new ATMApplicationController();
itso.entbk2.atm.cics.AtmCICS atmcics = new itso.entbk2.atm.cics.AtmCICS();
ctl.setATMPersistenceLayer(atmcics);

atmcics.setCardMapperSwitch(false);   // set to true for CardMapperCommand

Card card1 = ctl.getCard("1111111");
System.out.println(card1);

if (card1==null) card1 = new Card("1111111","1111",null);

System.out.println("PIN OK "+card1.checkPin("1111"));

Card card2 = ctl.getAccounts(card1);
System.out.println(card2);

try {
   java.util.Enumeration enum = card2.getAccounts().elements();
   while (enum.hasMoreElements())
     { System.out.println( (BankAccount)enum.nextElement() ); }
} catch (Exception e) { System.out.println("card has no accounts"); }
```

Figure 154. Scrapbook Script for CICS Application Testing

Note that we set the persistence layer in the application controller to be the CICS implementation, that is, the AtmCICS class.

Testing the Real Application

You can test the CICS transactions by using the servlet implementation described in Chapter 9, “ATM Application Using Servlets.”

Change the getApplicationController method of the ATMServletController to use the AtmCICS class for persistence:

```
public .... getApplicationController() {
   if (applicationController == null) {
      applicationController = new itso.entbk2.atm.model.ATMApplicationController();
      applicationController.setATMPersistenceLayer
         ( new itso.entbk2.atm.cics.AtmCICS() );
      setTransferToServiceHandler(getCardView());
   }
   return applicationController;
}
```

Note that you can only test the first steps of the application, namely, retrieving the card information, checking the PIN, and retrieving the accounts for the card.

10.13 Using an Advanced Navigator

A navigator enables us to build a complex business transaction that consists of multiple commands with branching logic between the commands.

The design of the application controller with individual methods for each step of the application enabled us to build simple navigators that perform one command, but with a slight variation of the controller we can at least demonstrate how an advanced navigator could be used.

The current design of the application controller involves three steps:

- CICS transaction to get the card information for a card number entered
- Check the PIN entered against the card object
- CICS transaction to get the accounts associated with the card

We implemented both the first and the last step with a CICS command executed from the getCard and getAccounts method of the application controller.

Let us assume that the user enters the card number and the PIN together. In such a case we can implement a navigator that invokes the two CICS commands and checks the PIN in between.

Design of a Navigator

The navigator that implements the CICS card and account transactions in one class performs these three steps:

- ❑ Invokes the card command passing the card ID entered by the user. Creates a card object for the card information returned by the command.
- ❑ Check the PIN entered by the user against the PIN stored in the card.
- ❑ Invokes the accounts command passing the card ID. Creates a vector of accounts from the information returned by the command and adds the vector to the card object.

The navigator has to check whether each command was successful and set the proper return condition.

Implementation of the Navigator

We build the navigator in the Visual Composition Editor by combining the designs of the CICSCardNavigator and CICSAccountsNavigator classes.

Navigator Class

Create a class named CICSCardAccountsNavigator as a subclass of CommunicationNavigator. On the BeanInfo page, add two properties, cardNumber and pinNumber, both of string type. Using properties is an alternative to using variables and then promoting them.

Visual Composition of the Navigator

Let us now implement the visual composition of the navigator:

- ❑ Drop three beans on the free-form surface: a CardCommand, an AccountsCommand, and an AtmHeader.
- ❑ Drop two factories on the free-form surface and change the types to Customer and Card (Figure 155).

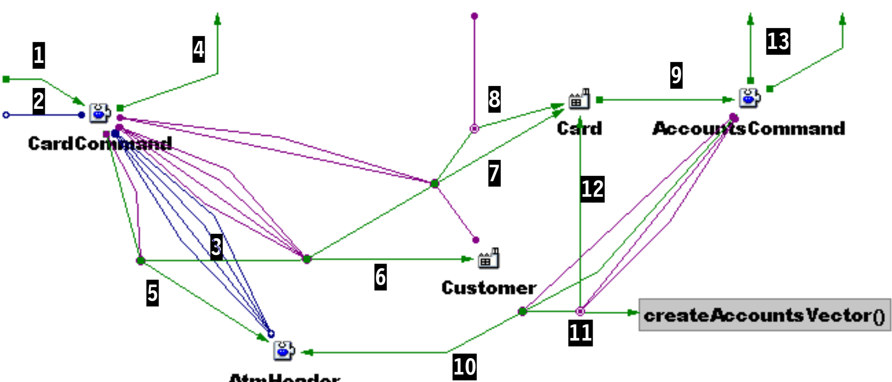


Figure 155. Visual Composition of the Navigator

Connections

First we set up the execution of the card command:

- ❑ Connect the *internalExecutionStarting* event of the navigator to the *execute* method of CardCommand to invoke the first CICS transaction (1).
- ❑ Connect the *cardNumber* property of the navigator to the *ceInputATM__CARDID* property of CardCommand (2).
- ❑ Connect the four properties of the AtmHeader to the matching *ceInputxxxx* properties of CardCommand (3).

After the card command is executed we check for its status and return code. Most of these connections are identical to the CICSCardAccess class.

- ❑ Connect the *executionUnsuccessful* event of CardCommand to the *returnExecutionUnsuccessful* method of the navigator and pass the event data as a parameter (4).
- ❑ Connect the *executionSuccessful* event of CardCommand to the *checkReturnCode* method of the AtmHeader and pass the *ceOutput1ATMH__RETURN__CODE__N* property as a parameter (5).
- ❑ Connect the *normalResult* of the connection to the constructor (with parameters) of Customer (6).
- ❑ Connect each constructor parameter to the corresponding *ceOutput1xxxx* properties of CardCommand.
- ❑ Connect the *normalResult* of the instantiation of Customer to the constructor (with parameters) of Card (7).
- ❑ Connect the *cardNumber* and *pinCard* parameters to the corresponding *ceOutput1xxxx* properties of CardCommand.
- ❑ Connect the *customer* parameter or the Card constructor to the *this* property of Customer.

Now we can check the PIN entered by the user and invoke the accounts command if the PIN is correct.

- ❑ Connect the *normalResult* of the Card constructor to the *checkPin* method of the Card and pass the *pinNumber* property of the navigator as a parameter (8).
- ❑ Connect the *pinCheckedOk* event of the Card to the *execute* method of AccountsCommand (9).

After the accounts command is executed we check for its status and return code. Most of these connections are identical to the CICSAccountsAccess class.

❑ Connect the *executionSuccessful* event of AccountsCommand to the *checkReturnCode* method of AtmHeader and pass the *ceOutput1ATMH__RETURN__CODE__N* property as a parameter (**10**).

❑ Connect the *normalResult* of the connection to a new *createAccountsVector* method (event-to-code) (**11**).

Copy the createAccountsVector method from the CICSAccountsAccess class. It has the same two parameters and returns a vector of accounts.

❑ Connect the number of accounts and the accounts array (properties *ceOutput1ATM__NO__ACCOUNTS and xxx__ACCOUNT__DETAILS*) as parameters to the createAccountsVector method.

❑ Connect the *normalResult* of the createAccountsVector method call to the *accounts* property of the Card. Open the connection and pass the event data (the vector) as a parameter (**12**).

❑ Connect the *executionSuccessful* (and *executionUnsuccessful*) event of AccountsCommand to the *returnExecutionSuccessful* (and *returnExecutionUnsuccessful*) method of the navigator and pass the event data as a parameter to set the final return code (**13**).

Testing the Navigator

We cannot easily use the AtmCICS class or the application controller to test the navigator because we did not design a method that takes both a card number and a PIN as input and retrieves the card and its accounts.

We can test the navigator by itself with a Scrapbook script, however (Figure 156).

```
// Test the CICS navigator, set Page -> Run in to itso.entbk2.atm.cics.AtmCICS

CICSCardAccountsNavigator navig = new CICSCardAccountsNavigator();
itso.entbk2.atm.model.Card card;
navig.setCardNumber("1111111");
navig.setPinNumber("1111");
navig.execute();
card = navig.getCard();
System.out.println(card);
try {
   java.util.Enumeration enum = card.getAccounts().elements();
   while (enum.hasMoreElements())
     { System.out.println( (itso.entbk2.atm.model.BankAccount)enum.nextElement() ); }
} catch (Exception e) { System.out.println("card has no accounts"); }
```

Figure 156. Scrapbook Script for Advanced Navigator

We can add a suitable method for the navigator to the AtmCICS class, (Figure 157), but to use it we would have to modify the ATM application controller.

```
public itso.entbk2.atm.model.Card extGetCardAccounts(String cardId, String pin)
                          throws Exception {
   CICSCardAccountsNavigator navig = new CICSCardAccountsNavigator();
   navig.setCardNumber(cardId);
   navig.setPinNumber(pin);
   navig.execute();
   itso.entbk2.atm.model.Card card = navig.getCard();
   if (card == null) throw new java.lang.Exception("Card not found");
   return card;
}
```

Figure 157. Method to Invoke the Advanced Navigator

10.14 Implementation of the Back-End Programs

We implemented the back-end CICS transaction with COBOL programs. The setup of a real CICS transaction server is described in "Setup for the CICS Connector" on page 365 and the COBOL programs are described in "COBOL Sample Programs" on page 366.

10.15 Conclusion

We have shown how two CICS transactions can be invoked with the CICS Connector.

We have not explored the full functionality of the CICS Connector. We have limited the options on record generation, only looking at custom records. We have only considered ECI calls and have not looked at BMS. The use of the mapper still needs much further exploration. Given a common header, we still would have to investigate reuse of its mapping across various commands. Business objects were not used.

In short we have only touched the tip of the proverbial iceberg. More examples are provided with the VisualAge for Java product. Sample packages to examine are in com.ibm.ivj.eab.sample.eci.*.

We hope we have given you a sense of the CICS Connector and how it simplifies the use of the underlying middleware. This is indeed powerful. The ECI and EPI can be used without having to acquire knowledge about their details. The fact that the e-business connectors conform to a Common Connector Framework augures well for the incorporation of other connectors in the future.

11 ATM Application Using MQSeries

In this chapter we explain how VisualAge for Java can incorporate MQSeries to access enterprise business and data services. The MQSeries support for Java enables MQSeries access from Java applications, applets, and servlets.

We discuss the classes that have to be created to interface with MQSeries. Some of these classes are application specific, whereas others pertain to MQSeries access per se.

To illustrate message-driven processing we use examples from the ATM application described in Chapter 5, “ATM Application Requirements and ATM Database.”

We show a partial implementation of the persistence interface of the ATM application with MQSeries to demonstrate how ATM transactions can access enterprise data through messaging and queuing.

© Copyright IBM Corp. 1998

11.1 A Brief Overview of MQSeries

MQSeries is IBM's middleware for commercial messaging and queuing. It runs on more than 20 hardware and software platforms from mainframe to desktop. MQSeries allows programs to communicate with each other across different platforms. These programs use message queuing to participate in message-driven processing. Message delivery is assured even across temporary network or system failures. A common, platform-independent API called the message queue interface (MQI) is implemented. The exchange of messages between the sending and receiving programs is time independent. The sender can continue processing without having to wait for the receiver to acknowledge the receipt of the message.

This section is taken from Chapter 1, "Introduction to IBM MQSeries," of the sixth edition of the *MQSeries Planning Guide*.

Messages and Queues

Messages and queues are the core components of a message queuing system.

Messages

Messages are used to transfer information from one application to another. These applications can be running on the same or different platforms.

MQSeries messages have two parts: the application data and a message descriptor. The content and structure of the application data is defined by the application programs that use the data. The message descriptor identifies the message and contains control information, such as the type of message and the name of the queue for the reply.

Queues

A queue is a data structure that stores messages. The messages may be put on the queue by either applications or a queue manager. Queues exist independently of the applications that use them. Each queue belongs to a queue manager, which is responsible for maintaining it.

Queues can exist either in a local system, in which case they are called *local queues*, or at another queue manager, in which case they are called *remote queues*.

Applications use MQI calls to send and receive messages. For example, an application can put a message on a queue and another application can retrieve the message from the same queue.

MQSeries Objects

An MQSeries object is a recoverable resource managed by MQSeries. Commands are available for manipulating objects. Each object is associated with a name and can be referenced in MQSeries commands and MQI calls. Objects include:

- Queue manager
- Queues
- Named lists
- Distribution lists
- Processes
- Channels
- Storage classes

Not all these objects are available on all platforms. For example, storage classes pertain to MVS only. We discuss some of these objects in this section.

Queue Manager

The queue manager is the heart of an MQSeries system. It provides queuing services to applications and manages the queues that belongs to it. It ensures that messages are put on the correct queue, as requested by the application making the MQPUT call.

The functions of a queue manager include:

- Management of queues
- Transfer of messages to other queue managers
- Generation of trigger and instrumentation events when appropriate conditions are met
- Object attribute management

Queues

A queue is an MQSeries object that can store messages. Each queue has queue attributes that determine what happens when applications reference the queue in MQI calls. An example of an attribute is whether the queue is put enabled or not. If it is, a message can be put on the queue. If it is not, an MQPUT request will fail. Another example pertains to whether the queue is associated with triggering. If triggering is on, depending on the type of triggering, the presence of a message or messages is associated with the initiation of a process. This process is defined in a process object.

Figure 158 shows the queue attribute of a local queue. It was captured by entering the *dis ql(ATM.REQUEST.QUEUE)* command.

```
C:\>runmqsc
84H2004,6539-B43 (C) Copyright IBM Corp. 1994, 1997.  ALL RIGHTS RESERVED.
Starting MQSeries Commands.

dis ql(atm.request.queue)
     1 : dis ql(atm.request.queue)
AMQ8409: Display Queue details.
   DESCR(ATM application request queue)    PROCESS(ATM.CICS.PROCESS)
   BOQNAME( )                              INITQ(SYSTEM.CICS.INITIATION.QUEUE)
   TRIGDATA( )                             QUEUE(ATM.REQUEST.QUEUE)
   CRDATE(1998-11-29)                      CRTIME(22.13.22)
   GET(ENABLED)                            PUT(ENABLED)
   DEFPRTY(0)                              DEFPSIST(NO)
   MAXDEPTH(5000)                          MAXMSGL(4194304)
   BOTHRESH(0)                             SHARE
   DEFSOPT(SHARED)                         NOHARDENBO
   MSGDLVSQ(PRIORITY)                      RETINTVL(999999999)
   USAGE(NORMAL)                           TRIGGER
   TRIGTYPE(FIRST)                         TRIGDPTH(1)
   TRIGMPRI(0)                             QDEPTHHI(80)
   QDEPTHLO(20)                            QDPMAXEV(ENABLED)
   QDPHIEV(DISABLED)                       QDPLOEV(DISABLED)
   QSVCINT(999999999)                      QSVCIEV(NONE)
   DISTL(NO)                               DEFTYPE(PREDEFINED)
   TYPE(QLOCAL)                            SCOPE(QMGR)
   IPPROCS(0)                              OPPROCS(0)
   CURDEPTH(0)
```

Figure 158. Local Queue Attributes

Queue Object Types

In MQSeries, there are several types of queue objects. This does not mean that there are several different types of queues; essentially there is only one type of queue. Each type of queue object can be manipulated by MQSeries commands and is associated with queues in different ways:

- A local queue object defines a local queue belonging to the queue manager to which the application is connected.
- A remote queue object identifies a queue belonging to another queue manager. The remote queue is usually given a local definition. The definition specifies the name of the remote queue manager where the queue exists as well as the name of the remote queue itself. The information on a remote queue definition enables the queue manager to find the remote queue manager, so that any messages destined for the remote queue go to the correct queue manager.
- An alias queue object enables applications to access queues by referring to them indirectly in MQI calls. When an alias queue name is used in an MQI call, the name is resolved to the name of a message queue at run time.

- The model queue object defines a set of queue attributes that are used as a template for a dynamic queue. Dynamic queues are created by the queue manager when an application makes an open queue request specifying a queue that is a model queue. The dynamic queue that is created in this way is a local queue whose name is specified by the application and whose attributes are those of the model queue.

Processes

A process definition object defines an application to an MQSeries queue manager. Typically, in MQSeries, an application puts or gets messages from one or more queues and processes them. A process definition object is used for defining applications to be started by a trigger monitor. The definition includes the application ID, the application type, and application-specific data. For example, an application type can be CICS, and the application ID can be the name of the CICS transaction associated with the process.

Channels

If a message is destined for a remote queue manager, it needs to be transmitted to that queue manager. Messages are transmitted through channels. If an MQSeries client needs to communicate with a queue manager, such communication is achieved through channels. A channel is a logical communication link. Several transport protocols are supported, including SNA LU 6.2, TCP/IP, NetBIOS, and SPX. There are two types of channels, message channels and MQI channels.

Message Channels

A message channel provides a communication path between two queue managers. The software that handles the sending and receiving of messages is called the message channel agent. The message channel is used for the transmission of messages from one queue manager to another and shields the application programs from the complexities of the underlying networking protocols.

A message channel can transmit messages in one direction only. Each end of a channel has a separate definition, defining it, for example, as the sending end or the receiving end. The definition of each end of a message channel can be one of four types, namely sender, receiver, server, and requester. A message channel is defined using one of these types defined at one end and a compatible type at the other end. Possible combinations are:

- Sender-receiver
- Requester-server
- Requester-sender (callback)
- Server-receiver

As an example, a channel could consist of a sender channel definition at the local queue manager and a receiver channel definition at the remote queue manager. These two definitions must have the same name, and together they constitute one channel.

Figure 159 shows a message flow across a channel from queue manager QM1 to queue manager QM2. The figure shows a transmission queue, which is a special type of local queue on which messages are stored until they can be successfully transmitted and stored at the remote queue manager.

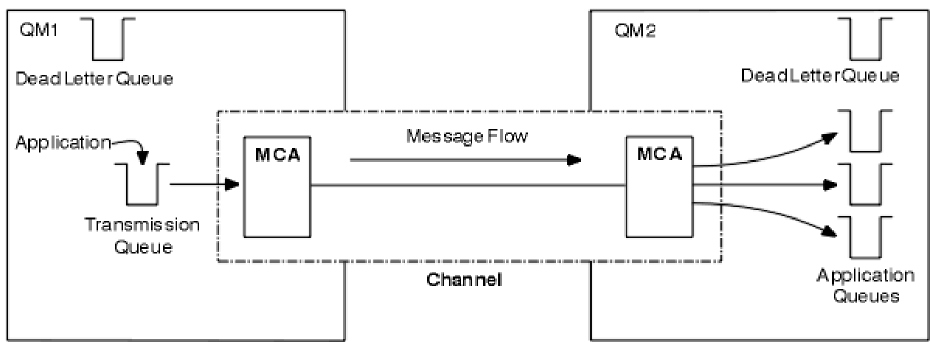


Figure 159. Message Flow across a Channel

To send a message from queue manager QM2 to queue manager QM1, another channel is required, with each end of the channel having a separate definition. In this case the sender end would be at QM2 and the receiver end would be at QM1 (Figure 160).

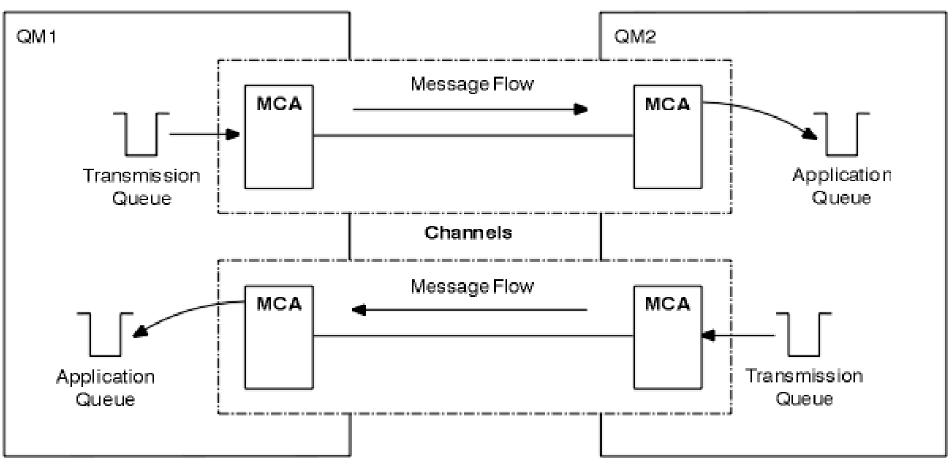


Figure 160. Two-way Message Channel Communication

MQI Channels

An MQI channel connects an MQSeries client to a queue manager on a server machine. It is used only for the transfer of MQI calls and responses and is bidirectional and synchronous.

The MQI channel is established when an MQCONN or MQCONNX call is issued. To create any new channel, two channel definitions, one for each end of the connection, are needed. The channel definition associated with a server is known as a *server* connection, and the channel definition associated with the client is known as a *client* connection. There are two different ways of creating the channel definitions, one where the server definition is created at the server and the client definition is set up on the client, and the other where both definitions are set up at the server. In the latter case, the definitions are stored in a binary file known as the *client channel definition table*. (MQSeries Version 5 provides an auto-definition capability where channel definitions are created automatically from a model definition.)

Figure 161 shows definitions for a server connection and a client connection of channel CHAN2, and Figure 162 shows the connection using this channel.

```
DEFINE CHANNEL(CHAN2) CHLTYPE(SVRCONN) TRPTYPE(TCP) +
       DESCR('Server connection to Client_2')

DEFINE CHANNEL(CHAN2) CHLTYPE(CLNTCONN) TRPTYPE(TCP) +
       CONNAME(9.20.4.26) QMNAME(QM2) DESCR('Client connection to Server_2')
```

Figure 161. MQI Channel Definition

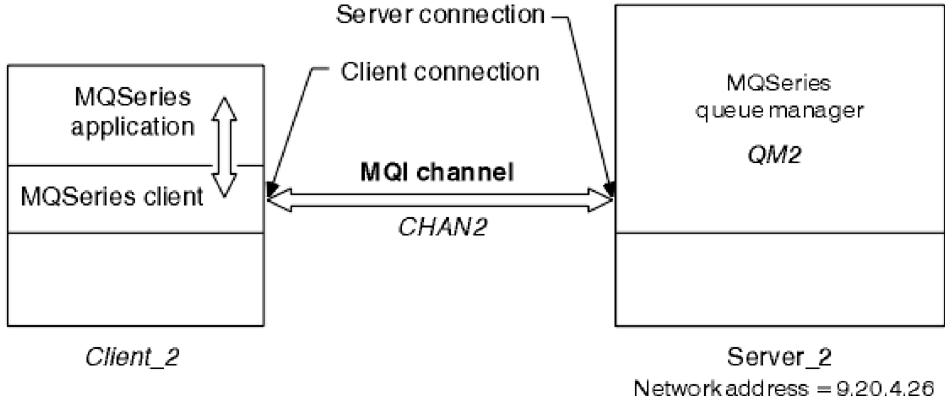


Figure 162. Use of an MQI Channel

MQ connection tables are not supported by the MQSeries Client for Java. Therefore, we have not used the client channel definition table.

MQSeries Clients and Servers

An MQSeries client is a part of an MQSeries product that can be installed on a machine without installing the full queue manager. It accepts MQI calls from application programs and passes MQI requests across an MQI channel to an MQSeries server executing on another processor (see Figure 163).

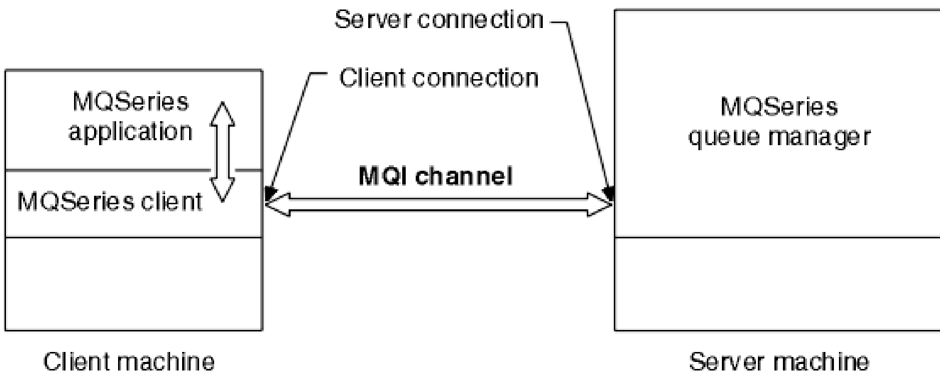


Figure 163. MQSeries Client to Server Flow

The MQSeries server is a full queue manager that can accept MQI calls directly from application programs running on the server processor as well as requests from MQSeries clients.

The MQSeries client allows a configuration where an application running on a client machine can use the MQI to access a queue manager running on a different machine.

When an application program in the client issues an MQI call, the client formats the parameter values of the call into an MQI request and sends the request to the server. The server receives the request, performs the action specified in the request, and sends back a response to the client. The response is used by the client to generate information that is returned to the application program through the normal MQI return mechanism.

An MQSeries client communicates with an MQSeries server, using an MQI channel, which is used to transfer MQI call requests from the client to the server and responses from the server to the client.

The examples discussed in this chapter use the MQSeries client for Java (see “MQSeries Client for Java” on page 287).

11.2 MQSeries Version 5

The latest release of MQSeries is Version 5. New features include:

- ❑ Database-message resource coordination
- ❑ Smart distribution of multicast messages
- ❑ Improvements to distributed performance through fast messages, trusted bindings, and improved internal architecture
- ❑ Support for messages of up to 100 MB
- ❑ Support for reference messages where the MQ message is a logical pointer to external data such as a file
- ❑ Use with additional languages such as C++, COBOL on Windows NT, PL/I, and Java
- ❑ IBM Software Server Integration
- ❑ Enhancements to simplify administration and problem solving
- ❑ Enhancements to security; DCE authentication can be used

For more information see:

```
http://www.software.ibm.com/ts/mqseries/v5
```

11.3 About MQSeries and Java

The MQSeries support for Java enables application developers to use the power of the Java programming language to create applets, servlets, and applications that interface with MQSeries and run on any platform that supports the Java run-time environment. This support helps reduce the development time for multiplatform MQSeries applications. Changes to applets are automatically picked up by end users as the applet code is downloaded.

MQSeries for Java provides two types of support:

- ❑ MQSeries Client for Java to enable Java applets, servlets, and applications to use MQSeries applications through a Web browser, Web server, or applet viewer.
- ❑ MQSeries Bindings for Java to enable Java applications to connect directly to an MQSeries queue manager.

MQSeries Client for Java

MQSeries Client for Java is an MQSeries client written in the Java programming language for communicating through TCP/IP. It enables Web

browsers, Java applets, and servlets to issue calls and queries to MQSeries. Thus host-based applications can be accessed over the Internet without the need for any other MQSeries code on the client machine. With MQSeries Client for Java the user of an Internet terminal can become a true participant in transactions, rather than just a giver and receiver of information.

The client can be installed on either a local hard disk or a Web server. Installation on a Web server has the advantage of allowing the MQSeries client applications to be run on machines that do not have the MQSeries Client for Java installed locally.

The client can be run in four different modes:

- From within any Java-enabled Web browser. If running as a Java applet from a browser, the MQSeries Client for Java classes, the Web server, and the MQSeries server are installed on a server.
- As a servlet in any Java-enabled Web server. If the servlet needs to connect to a queue manager on a different machine, the MQSeries Client for Java is used. The servlet and MQSeries Client for Java classes run on the Web server. The MQSeries server is on another machine. If the MQSeries server is on the same machine as the Web server, MQSeries Bindings for Java can be used.
- Using an applet viewer. The JDK must be installed on the client machine.
- As a stand-alone Java program. The JDK must be installed on the client machine. The MQSeries Client for Java classes are on the client and an MQSeries server is on a server.

To use the MQSeries Client for Java include the following import statement:

```
import com.ibm.mq*
```

For more information about the MQSeries Client for Java, see the redbook *Internet Application Development with MQSeries and Java*, SG24-4896.

MQSeries Bindings for Java

The MQSeries Bindings for Java enable MQSeries applications and servlets to be written with the Java programming language. These applications communicate directly with MQSeries queue managers. The MQSeries Bindings for Java provide the same programming interface as the MQSeries Client for Java, but they use Java native methods to call directly into the existing queue manager API rather than communicating through an MQSeries server connection channel. Unlike the MQSeries Client for Java, applications written using the MQSeries Bindings for Java cannot be

downloaded as applets and they cannot be run inside an applet viewer or Web browser. Provided that the queue manager is installed on the same machine as the Web server, MQSeries Bindings for Java can be used for servlets.

To use the MQSeries Bindings for Java include the following import statement:

```
import com.ibm.mqbind.*
```

The MQSeries Java Programming Interface

The Java programming interface conforms to the MQSeries object model. A program using the Java programming interface consists of a set of MQSeries objects that are acted on by calling methods on them. There are two packages for interfacing with MQSeries: com.ibm.mq for use with the MQSeries Client for Java and com.ibm.mqbind for use with the MQSeries Bindings for Java. These packages are described in the *MQSeries for Java Programmer's Reference Manual.*

Here are some of the more commonly used classes found in both packages:

- MQEnvironment

  MQEnvironment contains static data members that control the environment in which an MQQueueManager object (and its corresponding connection to a queue manager) is constructed. The values in the MQEnvironment class should be set before constructing an MQQueueManager instance. An example of a static data member is *channel*, which is the name of the channel for connecting to the target queue manager.

  All methods and attributes of this class apply to the MQSeries Client for Java, but only enableTracing and disableTracing apply to the MQSeries Bindings for Java.

- MQManagedObject

  MQManagedObject is a superclass for MQQueueManager, MQQueue, and MQProcess. It provides the ability to inquire about and set attributes of these objects.

- MQQueueManager (extends MQManagedObject)

  To connect to a queue manager, an MQQueueManager object must be instantiated.

  To open a queue, invoke the accessQueue method of this class. This establishes access to an MQSeries queue on this queue manager. It returns an object of type MQQueue.

- MQQueue (extends MQManagedObject)

  MQQueue provides inquire, set, put, and get operations for MQSeries queues. The inquire and set capabilities are inherited from MQManagedObject.

  To put a message on a queue, use the put(MQMessage, MQPutMessageOptions) method. To get a message from a queue, use the get(MQMessage, MQGetMessageOptions) method.

- MQMessage (implements DataInput, DataOutput)

  MQMessage represents both the message descriptor and the data for an MQSeries message. There are a group of readXXX methods for reading data from a message and a group of writeXXX methods for writing data into a message. An instance of this class is passed as an argument to the MQQueue put and get methods.

- MQPutMessageOptions

  This class contains options that control the behavior of the MQQueue put method. An instance of this class is passed as an argument to the put method.

- MQGetMessageOptions

  This class contains options that control the behavior of the MQQueue get method. An instance of this class is passed as an argument to the get method.

- MQException (extends Exception)

  Methods in the Java interface do not return a completion code and reason code. Instead, they throw an MQException whenever the completion code and reason code resulting from an MQSeries call are not both zero. To test the completion and reason code, use try and catch blocks. By default, exceptions are logged to System.err. This can be changed by altering the value of MQException.log. Constants beginning with MQCC_ are MQSeries completion codes, and constants beginning with MQRC_ are MQSeries reason codes.

11.4 Implementing the ATM Application with MQSeries

MQSeries can be used to send transaction requests to back-end processes and receive responses from such processes. When a transaction is initiated by some user action, such as the clicking of an OK button, an MQSeries request message is sent to a queue manager. The message is retrieved by an application monitoring the message queue. On the basis of the contents of the message, an appropriate program is called to execute the business and data services associated with the message. This program returns response data. The data is put to a response MQSeries queue. The response message is retrieved, formatted, and presented to the user through an HTML page.

The ATM application design architecture consists of three layers, namely the user interface layer, the business object layer, and the persistence layer. The persistence layer is used to separate the data access from the rest of the application. This separation allows for different services to be used to access the data without affecting the rest of the application. Managing these layers is a controller. A Java interface, ATMPersistenceInterface, is called from the application controller. AtmPersistenceInterface specifies the interface to the persistence layer.

To illustrate how enterprise data can be accessed through MQSeries, the persistence layer of the ATM model is extended to include an MQSeries implementation of the ATMPersistenceInterface interface. The class that implements this interface is called AtmMQ. This class controls the MQSeries access used by each ATM transaction.

The methods of the interface are shown in Figure 85 on page 157.

Not all methods are implemented in the examples. Only extGetCard and extGetAccounts are detailed. However, a methodology is described for how the other methods can be implemented.

In these examples the back-end programs are designed as CICS programs. We could have selected other application types. In this case we chose CICS to reuse the CICS COBOL programs written for the CICS Connector samples.

In the rest of this chapter we discuss what you have to configure and develop in order to have ATM transactions access enterprise data through MQSeries.

11.5 MQSeries Queue Manager and Objects

We assume that MQSeries Version 5 has been installed. In this section we describe the MQSeries queue manager and objects that need to be set up to implement the sample ATM transactions.

Create a Queue Manager

In this example the queue manager is called VAJEQMGR. It is created as the default queue manager. The dead letter queue associated with this queue manager is called VAJE.DEAD.LETTER.QUEUE.

To create the queue manager run this command:

```
crtmqm -q -u VAJE.DEAD.LETTER.QUEUE VAJEQMGR
```

Start the queue manager: strmqm `VAJEQMGR`

Define MQSeries Objects

To use the ATM transactions it is necessary to define some MQSeries objects. These include:

- ❑ Local queue definitions for:
  - VAJE.DEAD.LETTER.QUEUE
  - SYSTEM.CICS.INITIATION.QUEUE
  - ATM.REQUEST.QUEUE
  - ATM.RESPONSE.QUEUE
- ❑ Process definition for:
  - ATM.CICS.PROCESS
- ❑ ServerConnection channel definition for:
  - A.CLCHL.ATM

These definitions can be set up in a file that is input to the *runmqsc* command.[9] Assuming that the file is called atmobj.def, issue the following command to define the objects to the queue manager:

```
runmqsc <file-path:\atmobj.def
```

Figure 164 shows the command file.

[9] MQSeries provides the RUNMQSCutility to create and delete queue manager objects and manipulate them.

```
************************************************************************
* define MQ objects for ATM application
************************************************************************

* server connection channel definition

define channel('A.CLCHL.ATM') +
       chltype(SVRCONN) +
       trptype(TCP)  +
       replace +
       descr('Client channel for ATM application') +
       mcauser(' ')

* dead letter queue defintion

define ql('VAJE.DEAD.LETTER.QUEUE') +
       replace +
       descr('VAJEQMGR dead letter queue ')

* ATM request queue
*    triggered on first, associated with process ATM.PROCESS
*    associated with initiation queue ATM.INIT.QUEUE

define ql('ATM.REQUEST.QUEUE') +
       replace +
       descr('ATM application request queue') +
       initq(SYSTEM.CICS.INITIATION.QUEUE) +
       trigger +
       process(ATM.CICS.PROCESS) +
       share +
       trigtype(first) +
       usage(normal)

* ATM initiation queue
*       Uncomment this if SYSTEM.CICS.INITIATION.QUEUE is not defined
* define ql('SYSTEM.CICS.INITIATION.QUEUE') +
*       replace +
*       descr('Default Init queue associated with CICS triggering ')

* ATM process

define process('ATM.CICS.PROCESS') +
       replace +
       descr('process for ATM applications') +
       applicid(ATMQ) +
       appltype(CICS)

* ATM response queue holding replies to requests

define ql('ATM.RESPONSE.QUEUE') +
       replace +
       descr('ATM application responsequest queue')
```

Figure 164. MQSeries Objects for ATM Application

Command File to Start the Queue Manager

It is advisable to set up in a command file the commands needed to start the queue manager. This file can be run either automatically or manually, depending on deployment requirements.

The command file includes the starting of a queue manager, the command server, and listeners. Normally the starting of trigger monitors could also be included in the command file. However, in our case the trigger monitor will be started in CICS. The exact contents of such a file is operating system and deployment dependent. The command file shown in Figure 165 is used on Windows NT, if the queue manager has not been added as a service.

```
strmqm VAJEQMGR
strmqcsv VAJEQMGR
start "MQ TCP Listener" runmqlsr -t TCP -p 1414 -m VAJEQMGR
start "MQ chinit" runmqchi -q SYSTEM.CHANNEL.INITQ -m VAJEQMGR
REM start "MQ triggermon" runmqtrm -m VAJEQMGR -q ATM.INIT.QUEUE
```

Figure 165. MQSeries Startup Command File

After you run the command file, the queue manager should be started and ready for use.

11.6 Importing MQSeries into VisualAge for Java

It is necessary to import the MQSeries for Java packages into VisualAge for Java. MQSeries Version 5 Java classes are located at mqdisk:\MQM\Java\Lib.

Define a project for MQSeries for Java classes, for example, MQSeries, and import these packages:

- com.ibm.mq - needed for MQSeries Client for Java
- com.ibm.mqbind - needed for MQSeries Bindings for Java
- com.ibm.mqservices - needed for MQSeries service functions

The MQSeries for Java property files must be copied into the project resource directory of VisualAge for Java. These property files are located at mqdisk:\MQM\Java\Lib*.properties and must be copied into the resource directory d:\IBMVJava\ide\project_resources\MQSeries.

Version the MQSeries project (with all packages and classes), for example, with a version identifier of 5.0.

11.7 Create an MQAccess Bean

Because VisualAge for Java does not have MQSeries beans, it is necessary to create one and write some methods that invoke the MQSeries Java APIs. These methods need to connect to a queue manager, disconnect from a queue manager, open a queue, close a queue, put a message on a queue, and get a message from a queue.

Sample MQSeries Package

We create a package called **itso.entbk2.atm.mq** for all classes and interfaces relevant to the MQSeries implementation of the ATM application.

For discussion purposes, we create a new class called MQAccess. In reality two new classes might have to be created. One would import com.ibm.mq.* and would be used with the MQSeries Client for Java. The other would import com.ibm.mqbind.* and would be used with the MQSeries Bindings for Java. For our ATM application we use the MQSeries Client for Java.

MQAccess Bean

Create a new class called MQAccess as a subclass of Object in the itso.entbk2.atm.mq package. Add the com.ibm.mq.*package to the import list.

Properties

Before you connect to a queue manager from a MQSeries client, you have to set MQEnvironment variables. For this example the values of these variables are derived from bean property values. Therefore, the environment variables for which the default value is not used are set up as bean properties of the MQAccess bean.

Other properties pertaining to MQSeries classes also need to be added. These are qMgr (an MQQueueManager object), queue (an MQQueue object), and three strings, requestQueue, responseQueue, and queueManagerName, used in the instantiation of MQSeries objects.

Table 27 shows the MQAccess bean properties.

Table 27. MQAccess Bean Properties

Property Name	Property Type	Option
hostName	java.lang.String	read/write, bound
channelName	java.lang.String	read/write, bound
queueManagerName	java.lang.String	read/write, bound
qMgr	ibm.com.mq.MQQueueManager	read/write, bound
queue	ibm.com.mq.MQQueue	read/write, bound
requestQueue	java.lang.String	read/write, bound
responseQueue	java.lang.String	read/write, bound
mqOpenOutput	int	read-only
mqOpenInputShared	int	read-only

The generated code for the qMgr and queue properties is flagged with errors because the com.ibm.mq.MQException is not handled. Alter the generated code from:

```
private MQQueueManager fieldQMgr = new com.ibm.mq.MQQueueManager(" ");
private MQQueue fieldQueue = new MQQueue(null, "", 0, "", "", "");
```

to:

```
private MQQueueManager fieldQMgr = null;
private MQQueue fieldQueue = null;
```

The last two properties return constant values. Change their definitions to:

```
int fieldMqOpenOutput = MQC.MQOO_OUTPUT;
int fieldMqOpenInputShared = MQC.MQOO_INPUT_SHARED;
```

Method Features

Several methods are needed to put messages on and get messages from MQSeries queues. The bodies of these methods are described in sections “Connect to a Queue Manager” on page 297 through “Getting a Message from a Queue” on page 302.

Define the method features listed in Table 28.

Table 28. MQAccess Bean Method Features

Method	Return Type	Parameters	Remarks
connectToQmgr	void	-	Connects to queue manager
disconnectFromQmgr	void	-	Disconnects
openQueue	void	String	Opens the given queue for input and output
openQueue	void	String int	Opens the given queue with the specified open mode (int)
closeQueue	void	-	Closes a queue
putRequestMessage	MQ Message	String String	Given a message string and a queue name, puts a message to an MQ queue
putRequestMessage	MQ Message	AtmRequest String	Tailored method for the ATM application to queue a request of type AtmRequest (see note)
retrieveSpecific Message	String	MQMessage	Returns a message for a given correlation ID
Note: The AtmRequest type is not defined at this time. For now, create an abstract class named AtmRequest. We will define its details later.			

Implement the MQAccess Methods

In this section we describe the function and implementation of each method.

Connect to a Queue Manager

There is a difference in connecting to a queue manager when you use the MQSeries Client for Java and the MQSeries Bindings for Java. When you use the client, you must set MQEnvironment variables before creating an instance of the MQQueueManager class. The MQSeries Bindings for Java ignore most of the parameters provided by the MQEnvironment class and use only those connected with tracing. As the Bindings attach directly to a queue manager, the *port* and *channel* parameters are not required, and the *user ID* and *password* are obtained from the MQSeries environment variables. All examples assume the use of the MQSeries Client for Java. In this example the values of the environment variables are derived from pertinent MQAccess bean property values.

connectToQmgr Method

The connectToQmgr method is used to connect the application to a queue manager. Try and catch blocks are used. All caught exceptions are rethrown (Figure 166).

```
public void connectToQmgr() throws Exception {
   /* Perform the connectToQmgr method. */
   // setup MQEnvironment
   try {
      MQEnvironment.hostname = getHostName();
      MQEnvironment.channel = getChannelName();
      // connect to Queue Manager
      setQMgr(new MQQueueManager(getQueueManagerName()));
   }
   catch (MQException e) {
      System.out.println("An MQ exception occurred in connectToQmgr: CC: " +
               e.completionCode + " Reason Code: " + e.reasonCode );
      throw e;
   }
   return;
}
```

Figure 166. connectToQmgr Method

Disconnect from a Queue Manager

It is good practice to disconnect from the queue manager when the connection is no longer needed.

disconnectFromQmgr Method

The disconnectFromQmgr method uses try and catch blocks to handle any MQExceptions. All caught exceptions are rethrown (Figure 167).

```
public void disconnectFromQmgr() throws Exception {
   /* Perform the disconnectFromQmgr method. */
   try {
      getQMgr().disconnect();
      // fireHandleDisconnected(new DisconnectedEvent (this) );
   }
   catch (MQException e) {
      System.out.println("An MQ exception occurred when disconnecting: CC: " +
               e.completionCode + " Reason Code: " + e.reasonCode );
      throw e;
   }
   return;
}
```

Figure 167. disconnectFromQmgr Method

Open a Queue

Before messages can be put to or retrieved from a queue, the queue must be opened. To open a queue, you supply the name of the queue. You also have to specify open options. You can either use the defaults or code the options explicitly.

Initially we had only one openQueue method in which we allowed the open mode to cater for both input and output. However, this caused problems with MQSeries triggering. Because the queue was already open for input, no triggering occurred. We therefore added another openQueue method in which we parameterized the open mode.

openQueue(String, int) Method

This method has two parameters, a sting that represents the queue name, and an int that determines whether the queue is opened for input, output, or both. The queue is opened and the result is stored in the queue property (Figure 168).

```
public void openQueue(java.lang.String queueName, int openMode) throws Exception {
   /* Perform the openQueue method. */
   try {
      // Note: we will access the system default queue
      setQueue(getQMgr().accessQueue(queueName, openMode,
         null,            // default q manager
         null,            // no dynamic queue
         null) );         // no alternate user id
   }
   catch (MQException e) {
      System.out.println("An MQ exception occurred when opening queue " +
         queueName  + " : CC: " +
         e.completionCode + " Reason Code: " + e.reasonCode );
      throw e;
   }
   return;
}
```

Figure 168. openQueue(String, int) Method

openQueue(String) Method

This method has only one parameter, a string that represents the queue name. The method calls the previously defined openQueue method with a default open option set to input and output (Figure 169).

```
public void openQueue(java.lang.String queueName) throws Exception {
   /* Perform the openQueue method. */
   int openMode = MQC.MQOO_INPUT_SHARED | MQC.MQOO_OUTPUT;
   openQueue(queueName, openMode);
}
```

Figure 169. openQueue(String) Method

Close a Queue

Once all queue processing is complete, it is good practice to close the queue.

closeQueue Method

The closeQueue method invokes the close method of the MQManagedObject class (Figure 170).

```
public void closeQueue() throws Exception {
   /* Perform the closeQueue method. */
   try {
      getQueue().close();
   }
   catch (MQException e) {
      System.out.println("An MQ exception occurred when closing queue: CC: " +
               e.completionCode + " Reason Code: " + e.reasonCode );
   }
   return;
}
```

Figure 170. closeQueue Method

Put a Message to a Queue

To put a message to a queue, call the MQQueue put method. This method has two parameters, namely, MQMessage and MQPutMessageOptions. MQMessage contains the user message that is added to the MQMessage object through a writeXXX method. The put options control the behavior of the put. In our method we use default put options.

putRequestMessage(String, String) Method

The putRequestMessage method is used to put a message to a queue. Two parameters are passed: a string that is the user portion of the message, and a string that contains the name of the reply-to queue name. The reply-to queue name represents the name of the queue to which a response associated with this request message must be sent.

The putRequestMessage method (see Figure 171):

- ❑ Creates an MQMessage object and initializes its replyToQueueName (parameter), messageType (request), and format (string).
- ❑ Adds the user message by calling the writeString method (Originally the writeUTF method was used, but the generated 2-byte length field caused problems when putting a corresponding reply from a COBOL program.)
- ❑ Creates an MQPutMessageOptions object with default options
- ❑ Invokes the put method on the queue property with the constructed message and options parameters
- ❑ Returns the result message object

```
public com.ibm.mq.MQMessage putRequestMessage(String request, String aReplyToQueue)
                                                            throws Exception {
   /* Perform the putRequestMessage method. */
   MQMessage message = null;
   try {
      // create a new message object
      message = new MQMessage();
      message.replyToQueueName = aReplyToQueue;
      message.messageType = MQC.MQMT_REQUEST;
      message.format = MQC.MQMTFMT_STRING;
      message.writeString(request);
      MQPutMessageOptions pmo = new MQPutMessageOptions();   // default options
      // put the message on the queue
      getQueue().put(message,pmo);
   }
   catch (MQException e) {
      System.out.println("An MQ exception occurred when putting a message on a queue "
      + " : CC: " + e.completionCode + " Reason Code: " + e.reasonCode );
      throw e;
   }
   catch (java.io.IOException e) {
      System.out.println("An IO error occurred when putting a message on a queue " +
            e.toString());
      throw e;
   }
   catch (Exception e) {
      System.out.println("An exception has occurred " + e.toString());
      throw e;
   }
   return message;
}
```

Figure 171. putRequestMessage Method

putRequestMessage(AtmRequest, String) Method

Originally we only passed strings to the putRequestMessage method. During the course of development we created a second method with a parameter that is an object representing a request message. This new method uses the toString method to convert the request object into a string and calls the previously defined putRequestMessage method (Figure 172).

```
public com.ibm.mq.MQMessage putRequestMessage(itso.entbk2.atm.mq.AtmRequest request,
                                String aReplyToQueue) throws Exception {
   /* Perform the putRequestMessage method. */
   return putRequestMessage( request.toString(), aReplyToQueue);
}
```

Figure 172. putRequestMessage Method for ATM Requests

Getting a Message from a Queue

To get a message from a queue you have to call the MQQueue get method. This method has two parameters, namely, MQMessage and MQGetMessageOptions. These types must be instantiated and possibly primed if you do not want to use the default options associated with the MQGetMessageOptions object. In our method, changed defaults are hard-coded and not passed as parameters. This is not the best approach and restricts the reusability of the method. Once the get method has executed, the user portion of the message can be extracted by using one of the MQMessage readXXX methods. In our method the user portion of the message is returned as a string.

retrieveSpecficMessage(MQMessage) Method

The retrieveSpecificMessage method is used to get a specific message from a queue. By specific message we mean that the message we want to retrieve is identified by a specific correlation ID. The value of this correlation ID is associated with the message ID of a request message, so an object of type MQMessage is passed as a parameter.

In this method we hard-coded the get options:

- MQC.MQGMO_WAIT
- MQC.MQGMO_NO_SYNCPOINT
- MQC.MQGMO_FAIL_IF_QUIESCING

We also hard-coded the wait interval time for a response to the request to 30 seconds. During testing, we found that 30 seconds was not necessarily sufficient during the first two or three invocations.[10]

Also note the choice of MQGMO_NO_SYNCPOINT. We point out in “Unit of Work Considerations” on page 305 that we have largely ignored unit of work considerations. Of course in the real world such omissions could lead to poor designs that do not meet business objectives. Whether or not MQGMO_NO_SYNCPOINT is a reasonable option would depend largely on business unit of work requirements.

[10] This problem has been addressed by corrective service.

The getSpecificMessage method (see Figure 173):

- Creates an MQMessage object and an MQGetMessageOptions object with our hard-coded options
- Sets up the message correlation ID from the message parameter
- Invokes the get method on the queue property to retrieve a message
- Checks the message length and reads the user message string
- Returns the message as a string.

```
public String retrieveSpecificMessage(MQMessage msg) throws Exception {
   /* Perform the retrieveSpecificMessage method. */
   String aString = null;
   MQMessage aMessage = null;
   try {
      // create a new message object
      aMessage = new MQMessage();
      MQGetMessageOptions gmo = new MQGetMessageOptions();
      gmo.options  = MQC.MQGMO_WAIT + MQC.MQGMO_NO_SYNCPOINT +
              MQC.MQGMO_FAIL_IF_QUIESCING;
      gmo.waitInterval = 60000;
      aMessage.correlationId = msg.messageId; // set up correlation ID
      // get the message from the qeueue
      getQueue().get(aMessage, gmo);
      try {
         if (aMessage.getMessageLength() > 0 ) {
            int i = aMessage.getMessageLength();
            aString = aMessage.readString(i);
            return aString;
         }
         else {  System.out.println("message length is <= 0  "  );
                     return null; }
      } catch (java.io.IOException e) {
         System.out.println("An IO exception occurred in retrieveSpecificMessage”
                       + “ method" + e.toString());
         throw e;
      }
   } catch (MQException e) {
      // failed or MQRC_NO_MSG_AVAILABLE
      if ((e.reasonCode  != e.MQRC_NO_MSG_AVAILABLE) ) {
         System.out.println("An MQ exception occurred in retrieveSpecificMessage”
             +“ method"+": CC: "+e.completionCode+" Reason Code: "+e.reasonCode );
         throw e;
      }
      else { System.out.println("There are no more messages on the queue ");
                throw e;
      }
   }
}
```

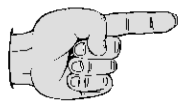
Figure 173. retrieveSpecificMessage Method

11.8 ATM MQSeries Design Choices

The ATM application consists of a set of transactions. Each transaction is associated with a request and a response. From an MQSeries perspective a request is represented by a message on a queue that is retrieved by a back-end process. When the request is processed, the back-end application returns a response by putting a reply message on an MQSeries queue.

We made a design decision to have a common header that is associated with every request and every response. In addition, in keeping with the protocol of the MQSeries-CICS/ESA bridge, the first 8 bytes of every message are associated with a program name.[11] To represent the common methods and data members associated with each request, we created an ATM request class called AtmRequest. This class is associated with the ATM header. Similarly we created an ATM response class called AtmResponse. This class checks the response return code.

The request and response for each ATM transaction has a well-defined format that needs to be adhered to by both sender and receiver applications. We decided to represent each of these formats as a separate class. This is analogous to having a separate COBOL copy book for each format. Ideally, there should be only one definition for each format from which language-specific views can be generated. We did not implement this for the ATM application.

Attention 	For the CICS implementation, we used the Java Record Framework and dynamic record types that can be generated from COBOL copy books. It would be interesting to see whether we could use the generated record beans for formatting the application data portion of MQSeries messages. If this were possible we could probably avoid having to create our own home-grown classes representing each message request and each message response. Records are described in “Records and the Java Record Framework” on page 105. The Java Record Framework and e-Connectors will be available for MQSeries in future updates of VisualAge for Java.

[11] The MQSeries-CICS/ESA bridge is supplied with MQSeries for MVS/ESA Version 1.2. Documentation can be obtained at http://www.software.ibm.com/ts/mqseries/txppacs/supportpacs/ma1e.pdf.

When formatting a request message, the ATM request class associated with the relevant transaction is invoked. Once the message is formatted, a method in the MQAccess class is invoked to put the message on a queue. When a response message is available on a queue, a method in the MQAccess class is invoked to retrieve the message. The ATM response class associated with the relevant transaction is invoked to parse the message into its field components.

Conforming to the ATM Model

In the first development cycle the MQSeries access to enterprise data was invoked directly in a servlet. The way response messages were formatted did not necessarily conform to any predefined layout. Thus MQSeries access was developed in isolation, without taking any architectural models into consideration.

This quick approach proved to be a rather poor choice but provides a valuable lesson. Whatever access mechanism is chosen to implement an end-to-end transaction flow, it should conform to an architected strategy represented by a model. Conversely the model should be comprehensive enough to utilize the strengths of the various access mechanisms.

The ATM application was designed to be a layered architecture with a separation of user interface, business, and persistence layers. To interface with this model, the MQSeries access classes need to take cognizance of it and its specifications. Not considering the architecture in the first place caused considerable redevelopment effort.

To reiterate, it is critical to understand the interfaces of the design model, before you undertake major design decisions. It is also important to note that it is only necessary to understand the model interfaces. The implementation details should be hidden.

Unit of Work Considerations

MQSeries provides transactional messaging. Over and above this, the decision to run back-end applications under CICS provides a transaction-based framework. It should therefore be possible to achieve transaction atomicity and integrity.

To a large extent this key concept has not been addressed sufficiently in the work presented here. The home-grown back-end program developed to feed MQSeries messages to CICS programs is not sophisticated enough to deal with the multiplicity of options when coordinating MQSeries and CICS resource management. It is assumed that in reality a more sophisticated tool

would be used to enable this. This omission of comprehensive unit of work design can cause disparity between the view presented to the user and the actual data stored on the enterprise databases. For example, it is possible for a user to be shown a balance without that balance having been stored on the enterprise database.

Unit of work issues are essential to any MQSeries and CICS design considerations. These issues exist regardless of the development environment and tools chosen. Specific implementation of design decisions, however, can be environment dependent. For example, if a transaction manager such as CICS is used, the unit of work is coordinated by it. If an X/Open XA compliant database is used, MQSeries itself can take on the role of coordinator of the unit of work.

Comprehensive unit of work analysis in terms of using MQSeries with the ATM application has not been done. This would prove to be an interesting undertaking that we have not addressed in this book. It is also deemed that such analysis is essential for any robust MQSeries implementation. The omission here is due to time constraints, not to the importance that should be attributed to the task.

11.9 ATM Request Classes

In this section we discuss the creation of the request classes of the ATM application. We do not discuss the creation of each request class associated with every ATM transaction. We discuss only the creation of the classes associated with the instantiation of a customer and a card.

AtmRequest Class

To deal with those features that are common to all requests, an abstract class called *AtmRequest* is created. Primarily this class is responsible for formatting the common header.

All ATM MQSeries messages consist of an 8-byte program name followed by an ATM header. We use the same header properties here as in “ATM Header for the COMMAREA” on page 245 in the CICS implementation:

- ❑ atmTxProgram, of type java.lang.String, read/write, bound (<== added)
- ❑ atmHdrId, of type java.lang.String, read/write, bound
- ❑ atmHdrDate, of type java.lang.String, read-only
- ❑ atmHdrReturnCode, of type short, read/write, bound
- ❑ atmHdrOutputLength, of type int, read/write, bound

To implement the AtmRequest class we do not implement the properties, only the methods that return the formatted information. This is sufficient for the MQSeries implementation of the back-end because we chose to use a string format for all our messages. Table 29 shows the methods of the AtmRequest class.

Table 29. Methods of the AtmRequest Class

Method	Return Type	Para-meter	Description
abstract getTxPgm	String	-	Abstract method that returns the name of the program associated with the back-end transaction
getHeader	String	-	Returns the AtmHeader as a string
abstract appendData	String	-	Abstract method that returns the transaction-specific portion of a request message
toString	String	-	Returns the MQSeries message as a string consisting of the header and the transaction-specific data

getHeader Method

The getHeader method formats an ATM header with a transaction program prefix (Figure 174).

```
protected final String getHeader() {
   String date = java.util.Calendar.getInstance().getTime().toString();
   date  = (date + "        ").substring(0,28);
   String header = getTxPgm() + "ATM "  + date + "9999" + "00000052";
   return header;
}
```

Figure 174. getHeader Method

toString Method

The toString method concatenates the header and the transaction-specific portion of the message (Figure 175).

```
public String toString() {
   return (getHeader() + appendData());
}
```

Figure 175. toString Method

Execution Trigger

To trigger the execution of the request we add a boolean write-only property named *trigger* and change the setTrigger method so that the trigger is reset immediately after being set (Figure 176). The trigger event is used to start processing.

```
public void setTrigger(boolean trigger) {
   boolean oldValue = fieldTrigger;
   fieldTrigger = trigger;
   firePropertyChange("trigger", new Boolean(oldValue), new Boolean(trigger));
   fieldTrigger = false;            // reset the trigger
}
```

Figure 176. Request Trigger Method

Card Request

In the first step of the ATM application, a card and a customer object must be instantiated in the business object layer from enterprise data. In the MQSeries implementation we send an MQSeries request message to get the data, and we retrieve a corresponding MQSeries response message. This response message contains the information necessary to create the business layer objects.

Create the CardReq class as a subclass of AtmRequest (Table 30).

Table 30. Card Request Class

Class	**CardReq**	**Definition**
Property	cardId	String, read/write, bound

Method	getTxPgm	`String getTxPgm() {`
		`   return "ATMCARDI";`
		`}`

	appendData	`String appendData() {`
		`   return getCardId();`
		`}`

Accounts Request

A card object is also associated with account information. A card can have multiple accounts. Once the card object has been instantiated, another MQSeries request is sent to retrieve the account information.

Create the AccountsReq class as a subclass of AtmRequest (Table 31).

Table 31. Accounts Request Class

Class	AccountsReq	Definition
Property	cardId	String, read/write, bound

Method	getTxPgm	`String getTxPgm() {`
		`   return "ATMACCNT";`
		`}`

	appendData	`String appendData() {`
		`   return getCardId();`
		`}`

11.10 ATM Response Classes

In this section we discuss the creation of the response classes of the ATM application. As with the request classes, we discuss only the creation of the classes associated with the instantiation of a customer and a card.

AtmResponse Class

To deal with the data content that is common to all responses, we create an abstract class called *AtmResponse*. This class is primarily responsible for checking the response return code. Because this class parses a retrieved MQSeries message, we create an additional constructor with the message as a parameter.

Create the AtmResponse class as an abstract class and declare a private string, responseMsg.

Table 32 shows the methods of the AtmResponse class.

Table 32. Methods of the AtmResponse Class

Method	Return Type	Description and Code

AtmResponse	-	Constructor to save the response message string:
		`public AtmResponse(String str) {`
		`   responseMsg = str;`
		`}`

getUserData	String	Returns the response message without the header:
		`public String getUserData() {`
		`   return responseMsg.substring(52);`
		`}`

Method	Return Type	Description and Code

checkRCOfHeader	void	Checks the ATM header return code and throws an exception if the return code is not 0:
		`public final void checkRCOfHeader() throws Exception{`
		`String returnCode = responseMsg.substring(40,44);`
		`if (!(returnCode.equals("0000")))`
		`throw (new Exception("Transaction failed - " +`
		`"return code is " + returnCode ));`
		`}`

Card Response

In response to the CardReq message, enough information is sent back in the response message to create a card and customer object. We provide a method in the response class that instantiates the card and customer objects and returns the card object.

Remember that the card object contains a reference to the customer object.

Create the CardResp class as a subclass of AtmResponse (Table 33).

Table 33. Card Response Class

Class	CardResp	Definition
Method Feature	getCard	Constructs a customer object and a card object and returns the card object

getCard Method

Figure 177 shows the getCard method.

```
public itso.entbk2.atm.model.Card getCard()  {
   itso.entbk2.atm.model.Customer cust = new itso.entbk2.atm.model.Customer(
      getUserData().substring(11,15),      // customer ID
      getUserData().substring(15,18),      // title
      getUserData().substring(18,48),      // first name
      getUserData().substring(48,78) );    // last name
   itso.entbk2.atm.model.Card card = new itso.entbk2.atm.model.Card(
      getUserData().substring(0,7),        // card ID
      getUserData().substring(7,11),       // card PIN
      cust );                              // cutomer object
   return card;
}
```

Figure 177. getCard Method

Accounts Response

In response to the AccountsReq message, enough information is sent back in the response message to create a vector of accounts associated with the card object. The account classes are defined in the business object layer of the ATM model. An account can be either a CheckingAccount class or a SavingsAccount class. Both classes are subclasses of the BankAccount class.

Create the AccountsResp class as a subclass of AtmResponse (Table 34).

Table 34. Accounts Response Class

Class	**AccountsResp**	**Definition**
Method Feature	getAccounts	Extracts account details in the response message to create a vector whose elements are objects of either the CheckingAccount or the SavingsAccount class

getAccounts Method

Figure 178 shows the getAccounts method.

```
public java.util.Vector getAccounts() throws Exception {
   itso.entbk2.atm.model.BankAccount account = null;
   java.util.Vector vectorAccounts = new java.util.Vector();
   int nbrOfAccounts = (new Integer(getUserData().substring(7,9)).intValue());
   for (int i=0 ; i < nbrOfAccounts ; i++) {
      String str = getUserData().substring(9+i*33,(9+i*33+33));
      String acctype = str.substring(8,9);
      if (acctype.equalsIgnoreCase("C"))
         account = new itso.entbk2.atm.model.CheckingAccount(
            str.substring(0,8),       // account ID
          new java.math.BigDecimal(new java.math.BigInteger(str.substring(9,17)),2),
          new java.math.BigDecimal(new java.math.BigInteger(str.substring(25,33)),2));
      else
         account = new itso.entbk2.atm.model.SavingsAccount(
            str.substring(0,8),       // account ID
          new java.math.BigDecimal(new java.math.BigInteger(str.substring(9,17)),2),
          new java.math.BigDecimal(new java.math.BigInteger(str.substring(17,25)),2));
      vectorAccounts.addElement(account);
   }
   return vectorAccounts;
}
```

Figure 178. getAccounts Method

11.11 ATM Access Classes

Before we can implement the ATM persistence interface, we encapsulate individual MQSeries interactions in ATM access classes. Such an approach will make it easy to code the class that implements the persistence interface.

An ATM access class embodies the flow of MQSeries requests and responses for one user transaction. These MQSeries requests trigger back-end applications that return sufficient information in MQSeries response messages to prepare the response for the user interface.

To begin, we concentrate on two interactions, retrieving ATM card information and retrieving the accounts for one ATM card. We encapsulate these two interactions in two classes called MQCardAccess and MQAccountsAccess.

Each class implements an execute method that triggers the processing.

Card Access Class

Create a class called MQCardAccess as a subclass of Object. Define the properties shown in Table 35, all read/write and bound.

Table 35. Properties of the Card Access Class

Property	Type	Description
cardId	String	The input card number for retrieving
card	Card	The resulting card object of the operation

Visual Composition

We construct the processing of the card access class in the Visual Composition Editor (Figure 179).

- ❑ Drop a bean of type CardReq and a factory of type CardResp on the free-form surface.
- ❑ Add two variables of type MQAccess and name them mqRequest and mqResponse. These variables will be set from outside before execution.
- ❑ Promote the *this* property of the two variables as *mqRequestThis* and *mqResponseThis*.

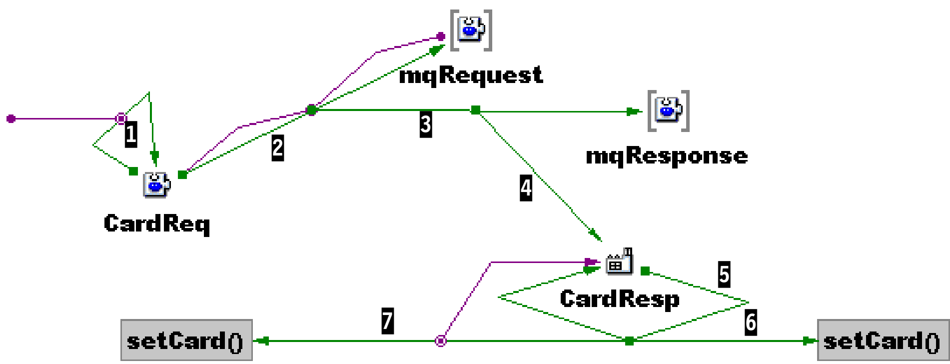


Figure 179. Visual Composition of Card Access Class

Connections

The processing is accomplished through these connections:

- ❑ Connect the *trigger* event of the CardReq bean to the *cardId* property of the CardReq bean and pass the *cardId* property of the MQCardAccess class as a parameter (1).
- ❑ Connect the *trigger* event of the CardReq bean to the *putRequestMessage(AtmRequest, String)* method of the mqRequest variable. Connect the *request* parameter to the *this* of the CardReq bean and the *aReplyToQueue* parameter to the *responseQueue* property of the mqRequest variable (2). This sends the message to MQSeries.
- ❑ Connect the *normalResult* of putRequestMessage to the *retrieveSpecificMessage* of the mqResponse variable and pass the event data (the MQMessage) (3). This retrieves the response from MQSeries.
- ❑ Connect the *normalResult* of retrieveSpecificMessage to the *CardResp(String)* constructor of the CardResp factory and pass the event data (the retrieved message) (4).
- ❑ Connect the *this* event of the CardResp factory to the *checkRCOfHeader* of the factory (5). Here we check the return code and throw an exception if not zero.
- ❑ Connect the *exceptionOccurred* of the return code check to the *setCard* method of the MQCardAccess class and pass *null* as a parameter (6).
- ❑ Connect the *normalResult* of the return code check to the *setCard* method of the MQCardAccess class and use the *getCard* method of CardResp as a parameter (7). This constructs the card object, our result.

Execute Method

The execute method sets the trigger in the CardReq bean:

```
public void execute() {
   getCardReq().setTrigger(true);
}
```

Account Access Class

Create a class called MQAccountAccess as a subclass of Object. Define the properties shown in Table 36, all read/write and bound.

Table 36. Properties of the Account Access Class

Property	Type	Description
cardId	String	The input card number for retrieving
accountsVector	Vector	The resulting vector of accounts for a given card

The MQAccountsAccess class is constructed in the same sequence as the MQCardAccess class (see Figure 180). The only differences are:

- ❑ Use an AccountsReq bean and an AccountsResp factory.
- ❑ The method to assign the result is setAccountsVector.

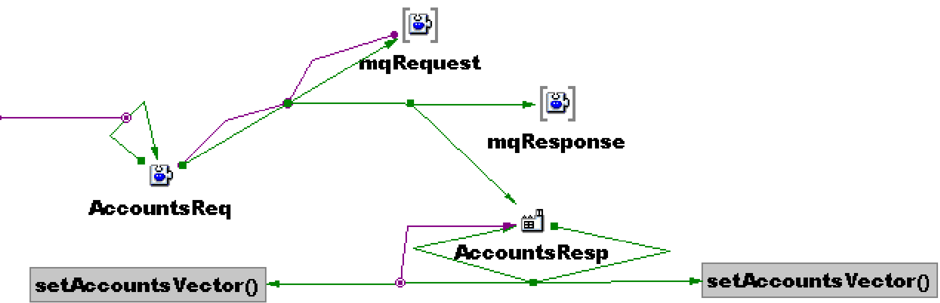


Figure 180. Visual Composition of Accounts Access Class

Do not forget to promote the *this* of the mqRequest and mqResponse variables.

Execute Method

The execute method sets the trigger in theAccountsReq bean:

```
public void execute() {
   getAccountsReq().setTrigger(true);
}
```

11.12 Persistence Interface with MQSeries

The ATM model has an application controlling class, the AtmApplicationController. When an event occurs, such as the user clicking on an Ok button, a relevant method in this class is invoked.

The controller delegates the access to enterprise data to a class that implements the ATMPersistenceInterface. In previous chapters we show implementations of the persistence interface using data access beans and the CICS Connector. In this section we implement this interface with MQSeries.

The controller contains the ATMPersistenceLayer property that holds the implementation of the persistence interface. This object is instantiated in the default constructor of the ATMApplicationController class:

```
setATMPersistenceLayer(new ...nameOfImplementationClass...());
```

For access through MQSeries we implement a class called *AtmMQ* and we initialize the controller with:

```
setATMPersistenceLayer(new itso.entbk2.atm.mq.AtmMQ());
```

AtmMQ Class

Create the AtmMQ class and implement the ATMPersistenceInterface:

```
class AtmMQ implements itso.emtbk2.atm.model.ATMPersistenceInterface
```

This creates skeleton bodies for all the methods. We discuss the contents of the methods of interest for the MQSeries implementation in “Methods of the Persistence Interface” on page 317.

Setting up the MQ Connections

We decided to have two MQ connections per instantiation of the AtmMQ class. One connection is used for request messages, and the other is used for response messages.

If more connections are needed for workload management, several instances of the AtmMQ class could be instantiated. We use only one instance for the ATM application.

Visual Components

We build the AtmMQ class in the Visual Composition Editor:

❑ Add two beans of type MQAccess to the free-form surface and name them MQRequest and MQResponse.

❑ Open the two MQAccess beans and set the value for the channelName, hostName, queueManagerName, requestQueue, and responseQueue properties according to your MQSeries configuration (Figure 181).

❑ Add a bean for every class that needs to use the MQSeries connections, in our case, MQCardAccess and MQAccountAccess.

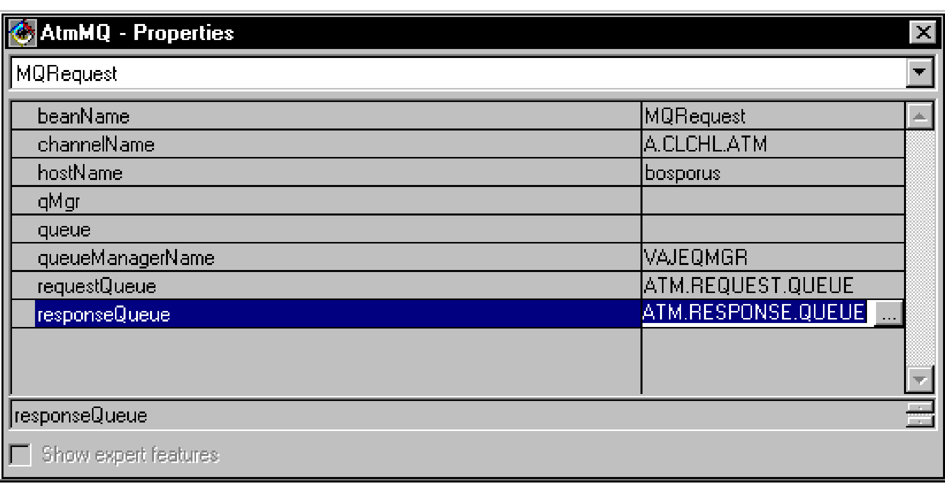


Figure 181. Properties of MQAccess Beans

Connections

Figure 182 shows the visual composition of the AtmMQ class.

❑ Connect the *initialize* event of AtmMQ to the *connectToQmgr* method of MQRequest (1).

❑ Connect the *normalResult* of connectToQmgr to the *openQueue(String,int)* method of MQRequest. Connect the *queueName* parameter to the *requestQueue* property of MQRequest, and connect the *openMode* parameter to the *MQOpenOutput* property of MQRequest (2).

❑ Connect the *initialize* event of AtmMQ to the *connectToQmgr* method of MQResponse (3).

❑ Connect the *normalResult* of connectToQmgr to the *openQueue(String)* method of MQResponse. Connect the *queueName* parameter to the *responseQueue* property of MQResponse (4).

- ❑ For every class that uses the MQRequest connection, connect the *this* property of MQRequest to the *mqRequestThis* property of the class. (This assumes that the variable was named mqRequest.)

  For example, connect the *this* property of MQRequest to the *mqRequestThis* property of MQCardAccess and MQAccountAccess (5).

- ❑ For every class that uses the MQResponse connection, connect the *this* property of MQResponse to the *mqResponseThis* property of the class (6).

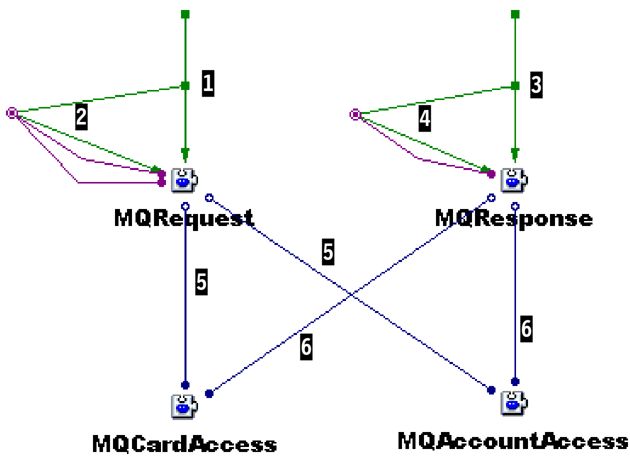


Figure 182. Visual Composition of AtmMQ Class

Methods of the Persistence Interface

AtmMQ must implement all methods of the ATM persistence interface. For now we only deal with the instantiation of a card object with its associated accounts, so we implement the extGetCard and extGetAccounts methods (Table 37).

Table 37. AtmMQ Methods Implemented for ATM Application

Method	Return Type	Parameters	Description
extGetCard	Card	String	Retrieves card and customer data and constructs a card object
extGetAccounts	void	Card	Retrieves all accounts of a card and stores the accounts in the card object

extGetCard Method

Figure 183 shows the extGetCard method. This method sets the cardId variable in MQCardAccess and calls the execute method. This triggers the MQSeries flows, culminating in the creation of a card object that is returned by this method. We discuss the details of these MQSeries request and response flows in “Card Access Class” on page 312.

```
public itso.entbk2.atm.model.Card extGetCard(String cardId) throws Exception {
   getMQCardAccess().setCard(null);
   getMQCardAccess().setCardId(cardId);
   getMQCardAccess().execute();           // MQ processing
   if (getMQCardAccess().getCard() == null)
      throw new java.lang.Exception("Card not found");
   return getMQCardAccess().getCard();
}
```

Figure 183. extGetCard Method

extGetAccounts Method

Figure 184 shows the extGetAccounts method. The extGetAccounts method retrieves the accounts associated with a card object and stores them in a vector of the card object. The vector is created in the MQAccountAccess class. An exception is thrown if no accounts are found.

```
public void extGetAccounts(itso.entbk2.atm.model.Card card) throws Exception {
   card.setAccounts(null);
   getMQAccountAccess().setAccountsVector(null);
   getMQAccountAccess().setCardId( card.getCardNumber() );
   getMQAccountsAccess().execute();           // MQ processing
   if (getMQAccountAccess().getAccountsVector() == null)
      throw new java.lang.Exception("Card has no accounts");
   card.setAccounts( getMQAccountAccess().getAccountsVector() );
}
```

Figure 184. extGetAccounts Method

11.13 Adding Additional Transactions

Up to now we have concentrated on showing how MQSeries can be used to enable the creation of a card and customer object. The ATM application consists of several transactions. Any of these could be implemented using a similar MQSeries access mechanism.

The explicit implementation of these transactions, using MQSeries, is not shown. Instead the focus is on how the classes already discussed are modified

to incorporate other transactions. To illustrate the discussion, we use the updateBalance function.

The updateBalance function of the ATM persistence interface gets the updated account as a parameter. In the account is the new balance, and a new transaction record is added to the transaction property.

We construct an update balance message containing:

- account ID
- account type
- new balance (BigDecimal converted to a string of 10 characters, in cents)
- transaction ID
- transaction type
- transaction amount (BigDecimal converted to a string of 10 characters)

This invokes a back-end application that updates the external resources of the account and the transaction and returns a response message indicating whether the update was successful or not.

Create a Class for the MQSeries Request

Every request message is represented by a class whose superclass is AtmRequest. The two methods of this class that affect the message content are getTxPgm and appendData.

Relevant classes that have already been discussed are CardReq and AccountsReq.

Update Balance Request

Create the UpdateBalanceReq class as a subclass of AtmRequest (Table 38).

Table 38. Update Balance Request Class

Class	**UpdateBalanceReq**	**Definition**
Property	account	type: BankAccount, read/write, bound

Method	getTxPgm	`String getTxPgm() {`
		`   return "ATMBALUP";`
		`}`

	appendData	Formats the message string
	bigdecToString	Converts a BigDecimal value to a 10-character string

bigdecToString Method

```
private String bigdecToString(java.math.BigDecimal amount) {
   // returns a string of 10 chars, BigDecmial amount in cents
   java.math.BigDecimal baldec;
   java.math.BigInteger balint;
   baldec = amount.multiply(new java.math.BigDecimal(100));
   baldec = baldec.add(new java.math.BigDecimal(.5));  // rounding
   balint = baldec.toBigInteger();
   String str = "0000000000" + balint.toString();
   str = str.substring( str.length()-10 );
   return str;
}
```

appendData Method

```
public String appendData() {
   itso.entbk2.atm.model.Transaction trans = getAccount().getLastTransaction();
   return getAccount().getAccountId() +
          getAccount().getAccountType().substring(0,1) +
          bigdecToString( getAccount().getBalance() ) +
          trans.getTransId() +
          trans.getTransType() +
          bigdecToString( trans.getTransAmount() );
}
```

Create a Class for the MQSeries Response

Every response message is represented by a class whose superclass is AtmResponse. The methods that must be created depend on the processing that occurs with the response message. Relevant classes that have already been discussed are CardResp and AccountsResp.

Update Balance Response Class

Create the UpdateBalanceResp class as a subclass of AtmResponse.

Because only the header return code is checked for successful execution, no additional methods are needed to format the response message.

Create a Transaction-Specific Access Class

This class essentially controls the MQSeries ATM transaction interface. It depicts how a request message is sent to trigger an enterprise transaction. On getting a response message back from the transaction, the message is formatted to return an object or data type.

The message flows in this class must be activated by an event triggered in one of the persistent interface methods. Because this class accesses MQSeries, the connections established in the AtmMQ class must be referenced. This class can be created visually.

The following components are needed for an access class:

- ❑ Properties that hold the values required for the execution of the transaction and that are set from the persistent interface method
- ❑ Two variables that reference the MQ connection established for request and response messages (mqRequest and mqResponse) with their *this* properties promoted for external access
- ❑ A bean representing the request message (for example, UpdateBalanceReq)
- ❑ A factory for the response message, with a constructor that accepts a string (for example, UpdateBalanceResp)
- ❑ Any variables or properties that are required to supply parameters or return values
- ❑ An execute method that sets the trigger property in the request bean

The components of the class can be connected visually. The specifics of these connections can only be detailed on a transaction by transaction basis, but in general the following applies:

- ❑ Connect the *trigger* property event of the request message bean (UpdateBalanceReq) to set up its properties.
- ❑ Connect the *trigger* property event to the *putRequestMessage* method of the mqRequest variable.
- ❑ Connect the *request* parameter of the putRequestMessage to the *this* property of the request class (UpdateBalanceReq).
- ❑ Connect the *aReplyToQueue* parameter of the putRequestMessage to the *responseQueue* property of the mqRequest variable.
- ❑ Connect the *normalResult* event of the putRequestMessage to the *retrieveSpecificMessage* method of the mqResponse variable and pass the event data.
- ❑ Connect the *normalResult* event of the retrieveSpecificMessage to the *constructor(String)* method of the response class factory (UpdateBalanceResp).
- ❑ Connect the *this* event of the response class to its own *checkRCOfHeader* method.
- ❑ Connect the *normalResult* event of checkRCOfHeader to whatever method(s) are needed to format the response message.
- ❑ Connect any return object to a property that can be retrieved externally.

The MQCardAccess and MQAccountsAccess classes are examples of this type of class.

Update Balance Access Class

Create the MQUpdateBalance class as a subclass of Object. Define the properties shown in Table 39, all read/write and bound.

Table 39. Properties of the Update Balance Access Class

Property	Type	Description
account	BankAccount	The updated bank account
errorCode	int	A return code indicating success (0) or failure (1)

The MQUpdateBalance class is constructed visually, similar to the MQCardAccess and MQAccountsAccess classes (see Figure 185). Note that the trigger event sets the account property of the UpdateBalanceReq class.

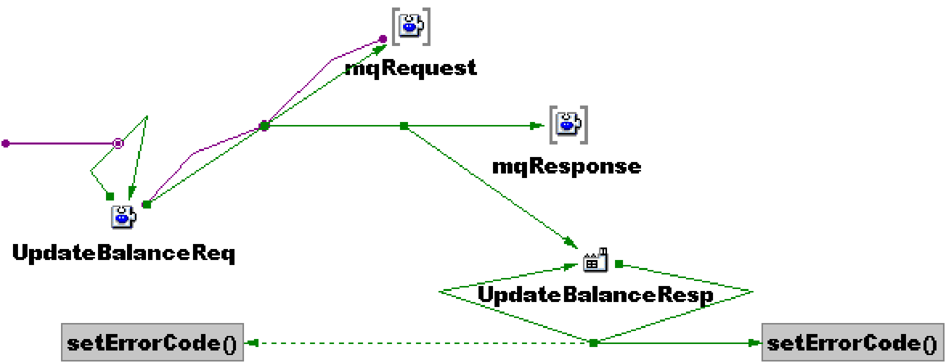


Figure 185. Visual Composition of Update Balance Access Class

Modify the AtmMQ Class

It is necessary to modify the AtmMQ class to include processing for the additional transaction.

Connect the Additional Access Bean

Add the additional access bean (MQUpdateBalance) to the free-form surface of the AtmMQ class. Connect the *this* property of the MQRequest and MQResponse beans to the promoted *mqRequestThis* and *mqResponseThis* properties of the additional access bean.

Figure 186 shows the updated visual composition.

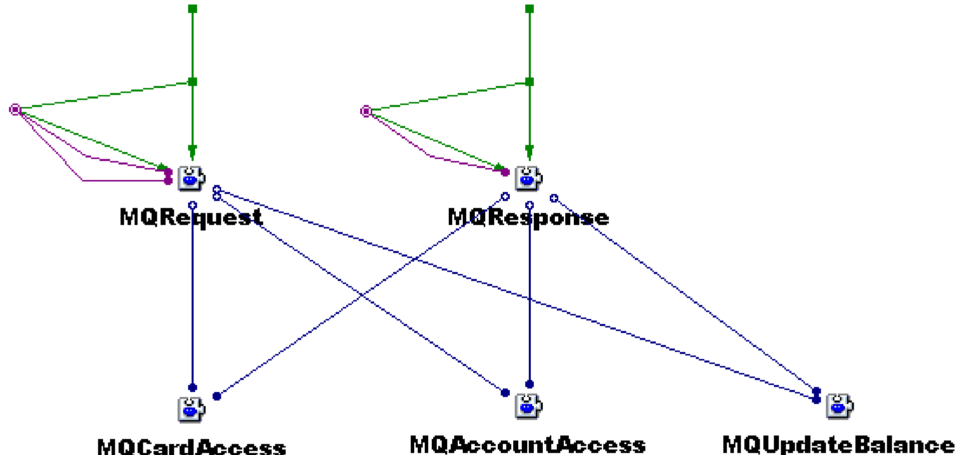


Figure 186. Visual Composition of AtmMQ with Update Balance

Modify the Transaction-Specific Interface Method

Every ATM transaction is associated with a method of the ATM persistence interface method. It is in such a method that the transaction-specific processing must be invoked.

Modify the extUpdateBalance Method

The method associated with the update balance transaction is extUpdataBalance(BankAccount). This method sets the error code property to 1, indicating failure. The bank account is then set into the property of MQUpdateBalance and the execute method is invoked to start the processing. After processing the error code is checked and an exception is thrown if it is not zero (Figure 187).

```
public void extUpdateBalance(itso.entbk2.atm.model.BankAccount account)
                               throws Exception {
   getMQUpdateBalance().setErrorCode(1);
   getMQUpdateBalance().setAccount(account);
   getMQUpdateBalance().execute();       // MQ processing
   if (getMQUpdateBalance().getErrorCode() != 0)
      throw new java.lang.Exception("Update of balance failed");
}
```

Figure 187. extUpdateBalance Method

Create a Back-End Application Program

To complete the end-to-end flow we require a back-end transaction to carry out the designated business and data services.

The existence of such a transaction is assumed. The name of the program associated with this transaction is reflected in the getTxPgm() method of the relevant subclass of AtmRequest. We have coded the transaction program ATMBALUP (see “Program ATMBALUP” on page 368).

The CICS COBOL programs invoked through the MQSeries messages are briefly discussed in the next section.

11.14 Back-End Programs

The enterprise business services required for the ATM applications use MQSeries queues to retrieve request data from MQSeries messages. Response data, created using data services, is put on MQSeries queues. These response messages are retrieved and the user interface (HTML for servlets, or applet) is updated with the appropriate information.

In a real application the back-end could be implemented through multiple CICS transaction programs. For testing we can also implement the back-end as a Java program that reads transactions from the MQSeries request queue and sends responses back through the MQSeries response queue.

Java Back-End Server Program

To test our design the easiest implementation of the back-end is a Java program that accesses the queues defined for the ATM application. This program reads the messages sent by the front-end application and constructs suitable response data, without accessing a database or transaction system.

ATM MQSeries Server Class

Create the AtmMQServer class as a subclass of Object.

Define two properties, read/write and bound:

- card, of type Card, to hold the current card object
- mqCorrelationId, of type byte[], to hold the incoming message ID to be put in the reply message (the retrieveSpecificMessage of the MQAccess class tests for a matching correlation ID)

Visual Composition

The visual composition is similar to that of the AtmMQ class with two MQAccess beans named MQRequest and MQResponse (Figure 188).

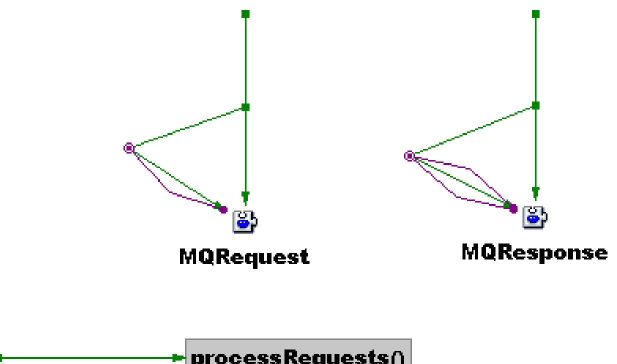


Figure 188. Visual Composition of the ATM MQSeries Server Class

The properties and connections for the two beans are set up in the same way as for the AtmMQ class (Figure 181 on page 316 and Figure 182 on page 317). One additional connection from the *initialize* event to a new *processRequests* method starts the processing cycle of the ATM MQ Server.

ATM MQSeries Server Methods

Table 40 shows the processing methods of the server class.

Table 40. Methods of the ATM MQSeries Server Class

Method	Description
processRequests	Reads a message from the queue, extracts the user portions, and calls one of the tailored processing routines
processCard	Prepares the response for the card request
processAccounts	Prepares the response for the accounts request
processUpdate	Prepares the response for the update balance request
sendReply	Sends a positive response with return code zero
sendError	Sends an error message with return code nonzero
getAtmHeader	Returns an ATM header
bigdecToString	Returns a string representation of a BigDecimal number

processRequests Method

The processRequests method performs a loop to retrieve a message from the queue, check it, save the message ID for correlation in the response message, and call the appropriate specific processing routine.

```
private void processRequests() throws Exception{
   String msgString = null;
   MQMessage aMessage = null;
   for (int i=1; ; i++) {
      System.out.println("Get message "+ i);
      try {
         msgString = null;
         aMessage = new MQMessage();
         MQGetMessageOptions gmo = new MQGetMessageOptions();
         gmo.options  = MQC.MQGMO_WAIT + MQC.MQGMO_NO_SYNCPOINT +
               MQC.MQGMO_FAIL_IF_QUIESCING;
         gmo.waitInterval = 30000;
         getMQRequest().getQueue().get(aMessage, gmo);
         try {
            if (aMessage.getMessageLength() > 0 ) {
               int lg = aMessage.getMessageLength();
               msgString = aMessage.readString(lg);
            }
            else { System.out.println("message length is <= 0  "  ); }
         } catch (java.io.IOException e) {
              System.out.println("An IO exception occurred in ” +
                    “retrieveSpecificMessage method" + e.toString());
            throw e;
         }
      } catch (MQException e) {
         // failed or MQRC_NO_MSG_AVAILABLE
         if ((e.reasonCode  != e.MQRC_NO_MSG_AVAILABLE) ) {
            System.out.println("An MQ exception occurred in “ +
                  “retrieveSpecificMessage method" +
                  ": CC: " + e.completionCode + " Reason Code: " + e.reasonCode );
            throw e;
         }
         else { System.out.println("Waiting for messages ... "); }
      }
      if (msgString != null) {
         System.out.println("-> Msg: " + msgString);
         setMqCorrelationId( aMessage.messageId );
         if ( msgString.substring(0,8).equals("ATMCARDI") )
                  processCard(msgString.substring(52));
         if ( msgString.substring(0,8).equals("ATMACCNT") )
                  processAccounts(msgString.substring(52));
         if ( msgString.substring(0,8).equals("ATMBALUP") )
                  processUpdate(msgString.substring(52));
      }
   }
}
```

processCard Method

The processCard method creates a customer and a card object with two accounts. The card is saved in a property for further transactions. A response message is generated and sent to the response queue.

```
private void processCard(String msg) {
   String reply = null;
   String blank30 = "                              ";
   String cardId = msg.substring(0,7);
   java.math.BigDecimal amt1 = new BigDecimal(1000);
   java.math.BigDecimal amt2 = new BigDecimal(100);
   String custId = cardId.substring(3,7);
   Customer cust = new Customer(custId,"Dr.","VA","Java");
   System.out.println("   cust " + cust);
   CheckingAccount acct1 = new CheckingAccount(custId+"-601",amt1,amt2);
   SavingsAccount  acct2 = new SavingsAccount(custId+"-602",amt1,amt2);
   Card card = new Card(cardId,cardId.substring(0,4),cust);
   System.out.println("   card " + card);
   card.addAccount(acct1);
   card.addAccount(acct2);
   setCard(card);        // save for next transactions
   // prepare reply
   reply = cardId + cardId.substring(0,4) + custId + "Dr." +
          ("VA"+blank30).substring(0,30) + ("Java"+blank30).substring(0,30);
   sendReply(reply,"00000130");
}
```

processAccounts Method

The processAccounts method prepares a response message with the two accounts of the card. The card ID is checked against the stored card from the card request.

```
private void processAccounts(String msg) {
   String reply = null;
   BankAccount account = null;
   String cardId = msg.substring(0,7);
   if (cardId.equals( getCard().getCardNumber() )) {
      System.out.println("   card " + cardId + " 02 accounts");
      reply = cardId + "02";
      for (int i=0; i<2; i++) {
         account = (BankAccount) getCard().getAccounts().elementAt(i);
         System.out.println("   acct " + account);
         String accountId = (account.getAccountId() + "        ").substring(0,8);
         String accType = account.getAccountType().substring(0,1);
         reply = reply + accountId + accType +
              bigdecToString( account.getBalance() );
         if (accType.equalsIgnoreCase("C"))
            reply = reply + "00000000" +
                bigdecToString( ((CheckingAccount)account).getOverdraft() );
         else
            reply = reply +
                bigdecToString( ((SavingsAccount)account).getMinAmount() ) +
                "00000000" ;
      }
      sendReply(reply,"00000127");
   }
   else sendError("Bad Card ID for account retrieval");
}
```

processUpdate Method

The processUpdate method prepares an empty response message (no data is sent back). The input account is checked against the stored card and a mismatch in the calculated balance is reported in the Console window.

```
private void processUpdate(String msg) {
   String accountId = msg.substring(0,8);
   BankAccount account = getCard().getAccount(accountId);
   if (account == null) {
      sendError("Bank account " + accountId + " not found");
      return;
   }
   BigDecimal balance = new BigDecimal( new BigInteger( msg.substring(9,17) ),2 );
   String tranType = msg.substring(17,18);
   String amount  = msg.substring(18,24)+"."+msg.substring(24,26);
   System.out.println("   trn: " + tranType + amount + " = " + balance);
   if (tranType.equalsIgnoreCase("C"))
      account.deposit(amount);
   else account.withdraw(amount);
   if (account.getBalance().compareTo(balance) != 0)
      System.out.println("   bal: server = " + account.getBalance() + " client = " +
                               balance);
   sendReply("","00000052");
}
```

sendReply Method

The sendReply method adds the ATM header to a response message, sets the return code to zero, and sends the message to the response queue.

```
private void sendReply(String userReply, String lg) {
    MQMessage message = null;
    String totalReply = getAtmHeader() + "0000" + lg + userReply;
    try {
      // create a new message object
      message = new MQMessage();
      message.messageType = MQC.MQMT_REPLY;
      message.writeString(totalReply);
      message.correlationId = getMqCorrelationId();
      MQPutMessageOptions pmo = new MQPutMessageOptions();   // default options
     // put the message on the queue
      System.out.println("-> Rep: " + totalReply);
      getMQResponse().getQueue().put(message,pmo);
   }
   catch (MQException e) {
      System.out.println("An MQ exception occurred when putting a message on a queue "
               + " : CC: " + e.completionCode + " Reason Code: " + e.reasonCode );
   }
   catch (java.io.IOException e) {
      System.out.println("An IO error occurred when putting a message on a queue " +
                   e.toString());
   }
   catch (Exception e) {
      System.out.println("An exception has occurred " + e.toString());
   }
   return;
}
```

sendError Method

The sendError method adds the ATM header to a response message, sets the return code to 8, and sends the message to the response queue.

```
private void sendError(String errorMsg) {
   MQMessage message = null;
   String paddedMsg = (errorMsg+" ...50 blanks ...").substring(0,50);
   String totalReply = getAtmHeader() + "0008" + "00000102" + paddedMsg;
   try {
      // create a new message object
      message = new MQMessage();
      message.messageType = MQC.MQMT_REPLY;
      message.writeString(totalReply);
      message.correlationId = getMqCorrelationId();
      MQPutMessageOptions pmo = new MQPutMessageOptions();   // default options
      // put the message on the queue
      System.out.println("-> Err: " + totalReply);
      getMQResponse().getQueue().put(message,pmo);
   }
   catch (MQException e) {
      System.out.println("An MQ exception occurred when putting a message on a queue "
               + " : CC: " + e.completionCode + " Reason Code: " + e.reasonCode );
   }
   catch (java.io.IOException e) {
      System.out.println("An IO error occurred when putting a message on a queue " +
                    e.toString());
   }
   catch (Exception e) {
      System.out.println("An exception has occurred " + e.toString());
   }
   return;
}
```

getAtmHeader Method

The getAtmHeader method returns a formatted ATM header.

```
private String getAtmHeader() {
   return "ATMREPLY" + "ATM "+ java.util.Calendar.getInstance().getTime().toString();
}
```

bigdecToString Method

The bigdecToString method converts a BigDecimal into a string of eight characters. The code is the same as in the UpdateBalanceReq class (see "bigdecToString Method" on page 320).

Testing the ATM MQSeries Server

To test the ATM MQSeries server, start the MQSeries queue manager and listener (see “Command File to Start the Queue Manager” on page 294).

Start the ATM MQSeries server by clicking on the *Run* button in the tool bar. The server waits for messages and prints output to the Console window.

The easiest way to test the server is through a Scrapbook script (Figure 189).

```
// set Page->Run in to ATMApplicationController class in itso.entbk2.atm.model

java.util.Enumeration enum;
BankAccount acct1;
BankAccount acct2;

ATMApplicationController ctl = new ATMApplicationController();
itso.entbk2.atm.mq.AtmMQ atmmq = new itso.entbk2.atm.mq.AtmMQ();
ctl.setATMPersistenceLayer(atmmq);

Card card1 = ctl.getCard("1234567");
System.out.println(card1);

boolean pinok = ctl.checkPin(card1,"1234");
System.out.println("PIN OK " + pinok);

ctl.getAccounts(card1);
enum = card1.getAccounts().elements();
while (enum.hasMoreElements())
  { System.out.println( (BankAccount)enum.nextElement() ); }

acct1 = (BankAccount)card1.getAccount("4567-601");
acct1 = ctl.deposit(acct1,"400");
acct1 = ctl.withdraw(acct1,"300");
System.out.println(acct1);

acct2 = (BankAccount)card1.getAccount("4567-602");
acct2 = ctl.deposit(acct2,"200");
acct2 = ctl.withdraw(acct2,"100");
System.out.println(acct2);
```

Figure 189. Scrapbook Script for Testing the ATM MQSeries Server

Testing the ATM Application with MQSeries

Instead of the Scrapbook script you can use the ATM GUI application or the ATM servlet application together with the ATM MQSeries server. You only have to set the AtmMQ bean as the persistence interface.

Preparing the ATM GUI Application

One way to change the GUI application to use the MQSeries persistence interface is to make a copy of the ATMApplet class and change it to use MQSeries:

- ❑ Copy the ATMApplet class (select *Reorganize -> Copy* in the pop-up menu) and name it AtmMQApplet.
- ❑ Delete the main method. It will be regenerated.
- ❑ Open the Visual Composition Editor and change the PersistenceDefault bean into an AtmMQ bean (use the *Morph into* pop-up selection).
- ❑ Change the *Memory* button label to *MQSeries* and save the application.

Running the ATM GUI Application with MQSeries

Make sure that the class path for the applet is set properly. Use *Run->Check Class Path* in the context menu of the class and click on *Compute Now.*

Start the MQSeries queue manager and the ATM MQSeries server. Now you can start the ATM GUI Application and experiment with the MQSeries connection.

Enter any seven-digit card number. A card object with two accounts is created by the ATM MQSeries server. Use the first four digits as the PIN. Note that the getTransaction method for the History button was not implemented in the ATM MQSeries server.

Running the ATM Servlet Application with MQSeries

To test the ATM servlet application with the ATM MQSeries server, change the getApplicationController method of the ATMServletController to use the AtmMQ class for persistence:

```
public .... getApplicationController() {
   if (applicationController == null) {
      applicationController = new itso.entbk2.atm.model.ATMApplicationController();
      applicationController.setATMPersistenceLayer
         ( new itso.entbk2.atm.mq.AtmMQ() );
      ...
```

CICS COBOL Back-End Programs

If it is necessary to maintain atomicity between MQSeries and database resources, a transaction management service is required. Some MQSeries Version 5 queue managers can act as transaction managers and coordinate updates made by external resource managers that comply with the X/Open XA interface. For example, MQSeries for Window NT can coordinate a unit of work involving access to MQSeries and DB2 resources. This function makes it possible to achieve transactional atomicity without the use of a transaction server. MQSeries can play the role of the unit of work coordinator. This function is available on server configurations, but it is not available to client applications.

It is also possible to run MQSeries applications under a transaction manager such as CICS, IMS, Encina, Tuxedo, or Top End. In this case the transaction manager is responsible for coordinating the unit of work, and MQSeries is just another resource that it manages.

Regardless of which method is selected, transactional integrity is guaranteed. It is therefore possible to decide how a back-end application requiring atomicity is written and deployed.

For this book we decided to run back-end applications under CICS. Enterprise services accessed through MQSeries or CICS clients were written as CICS COBOL programs. The samples discussed in this book were run on TXSeries Version 4.2 on Windows NT.

We used the same COBOL programs for MQSeries as for the CICS Connector. See “COBOL Sample Programs” on page 366 for more information about the COBOL programs.

We implemented one extra program as a bridge between MQSeries and CICS. See “MQSeries CICS Bridge Program” on page 369 for more information about the bridge program.

12 Deployment of the ATM Application Implementations

Up to now we have dealt only with the ATM application within the VisualAge for Java development environment. We developed several versions of the application, but we have not discussed how to extract the final code from VisualAge for Java to run it in a real environment.

In this chapter we cover the general process of deployment of applications, applets, and servlets. You can apply this process to deploy the different versions of the ATM application:

- ATM GUI implementation as a stand-alone application with memory or database persistence
- ATM GUI implementation as an applet
- ATM servlet implementation
- ATM CICS implementation
- ATM MQSeries implementation

© Copyright IBM Corp. 1998

12.1 Deployment of Applications

The basic characteristics of applications are:

- They run in a platform JVM.
- They must be installed on a client or server machine.
- They have normal access to the machine they run on and to other machines on the network.

Prerequisites for Applications

To run applications deployed from VisualAge for Java, the machine must have:

- A JVM; VisualAge for Java is now at the JDK 1.1.6 level.
- VisualAge for Java run-time libraries if you used Enterprise Access Builders. Copy the files to the run-time machine and add them to the CLASSPATH environment variable.

VisualAge for Java Access Builder Jar Files

VisualAge for Java Version 1 Access Builders are in:

```
d:\IBMVJava\eab\runtime\ivjeab.zip
```

VisualAge for Java Version 2 Access Builders are in:

```
d:\IBMVJava\eab\runtime20\
    ccf.jar - common connector framework
    eablib.jar - enterprise access builder library
    ivjdab.jar - data access beans
    ivjdab20.jar - data access builder
    ivjpb20.jar - persistence builder
    ivjsap11.jar - SAP/R3 access
    ivjsb20.jar - servlet builder
    jdebug.jar - remote debugger
    recjava.jar - record framework
```

Exporting an Application from VisualAge for Java

The steps to export an application are:

- Create a master directory for the exported classes (d:\Export).
- Select the classes in the Workbench and export the class files to the master directory. Subdirectories for the packages are automatically created.

Deployment Process for Applications

Copy the exported directories to the machine where the applications will run. Set up a master directory that is in the class path and copy the package subdirectories into the master directory.

Figure 190 shows the process of deploying applications from a VisualAge for Java development machine to an execution machine.

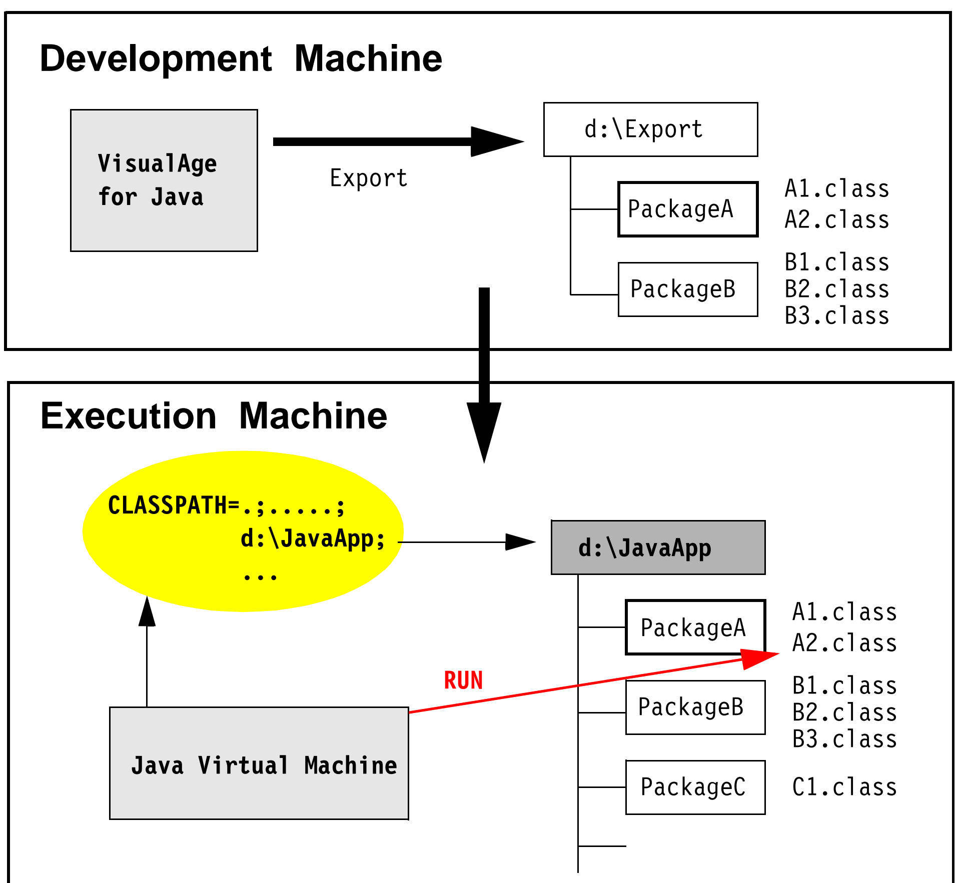


Figure 190. Deployment Process for Applications

This setup guarantees that the JVM will find all classes. A proper setup that follows the Java class naming rules is required for successful operation. Run an application with this command:

```
java PackageA.A2
```

12.2 Deployment of Applets

The basic characteristics of applets are:

- They run in a Web browser on a client machine.
- They are downloaded from a server machine.
- They have limited access to the client machine.
- They have limited access to the network—only to the server machine from which they come.

Some of the restrictions can be lifted for trusted applets; such security facilities are, however, beyond the scope of this book.

The biggest advantage of applets is that they are automatically distributed to the client machine. There is no maintenance burden; new versions of applets are installed on Web servers.

The Web browser provides the JVM; the platform JVM is not required.

Exporting Applets from VisualAge for Java

The steps to export an applet are the same as for exporting an application:

- Create a master directory for the exported classes.
- Select the classes in the Workbench and export the class files to the master directory.

Deployment Process for Applets

Copy the exported directories to the Web server machine. Set up a master directory where the Web server looks for HTML files and applets and copy the package subdirectories into the master directory.

Expand the VisualAge for Java enterprise library and the DB2 JDBC library into a subdirectory called COM\ibm of the master directory. See "VisualAge for Java Access Builder Jar Files" on page 334 for a list of files.

In the master directory, create an HTML file for each applet that points to the applet's class. Note that export can create a skeleton applet file for each applet.

Figure 191 shows the process of deploying applets from a VisualAge for Java development machine to a Web server and a Web browser.

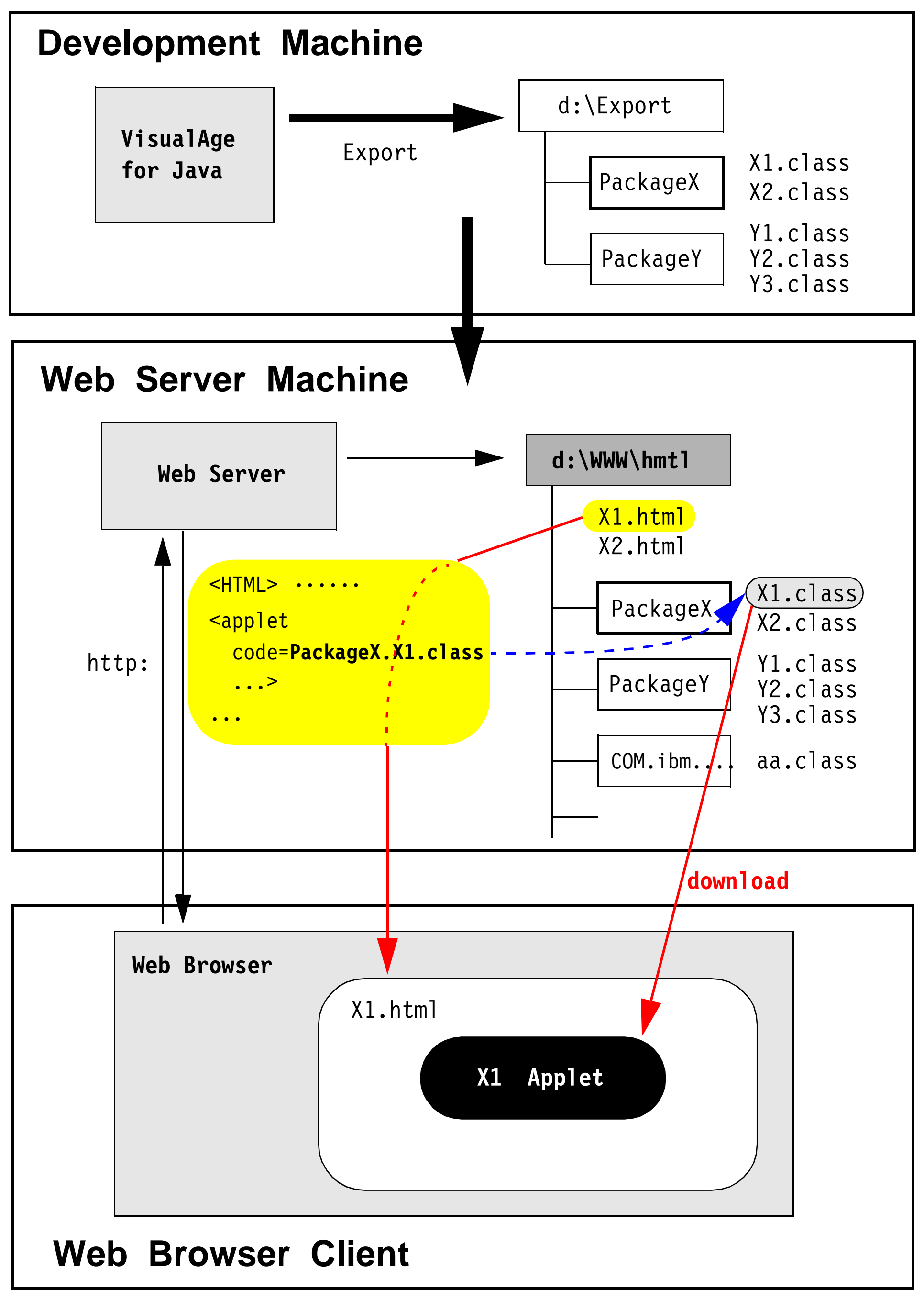


Figure 191. Deployment Process for Applets

12.3 Deployment of Servlets

Most Web servers contain a servlet run-time facility that enables the execution of servlets. Some Web servers have their own JVM, others use the JVM of the underlying operating system.

Modern Web servers have optimized servlet run-time facilities that cache servlet code and keep servlets active after their first use. This optimization improves performance considerably compared to CGI programs that are loaded at each invocation.

We cover only the Windows NT platform and two configurations of Web servers with a servlet run time in this book:

- Lotus Domino Go Webserver without WebSphere
- Lotus Domino Go Webserver with WebSphere

WebSphere contains an improved servlet run-time facility over Lotus Domino Go and completely replaces the servlet run-time facility during installation.

In the near future, WebSphere will run on top of the Apache Web server on Windows NT. We assume that the WebSphere configuration will be very similar to what we describe here.

Because servlets run on the Web server in a trusted environment, they have no real restrictions about what they can access and with which other machines they can communicate. Communication with other servers is more a question of which protocols and products are installed on the Web server.

Possible communication protocols include:

- JDBC to a relational database on the same or a different database server
- RMI to another server
- CICS Transaction Gateway on the same or a different server
- MQSeries on the same or a different server

The basic deployment process is the same with or without WebSphere installed:

- Export the servlet classes
- Copy the exported files to a suitable target directory on the Web server
- Set up the Web server class path

Deployment of Servlets for Lotus Domino Go Webserver

Figure 192 shows the process of deploying servlets from a VisualAge for Java development machine to Lotus Domino Go Webserver.

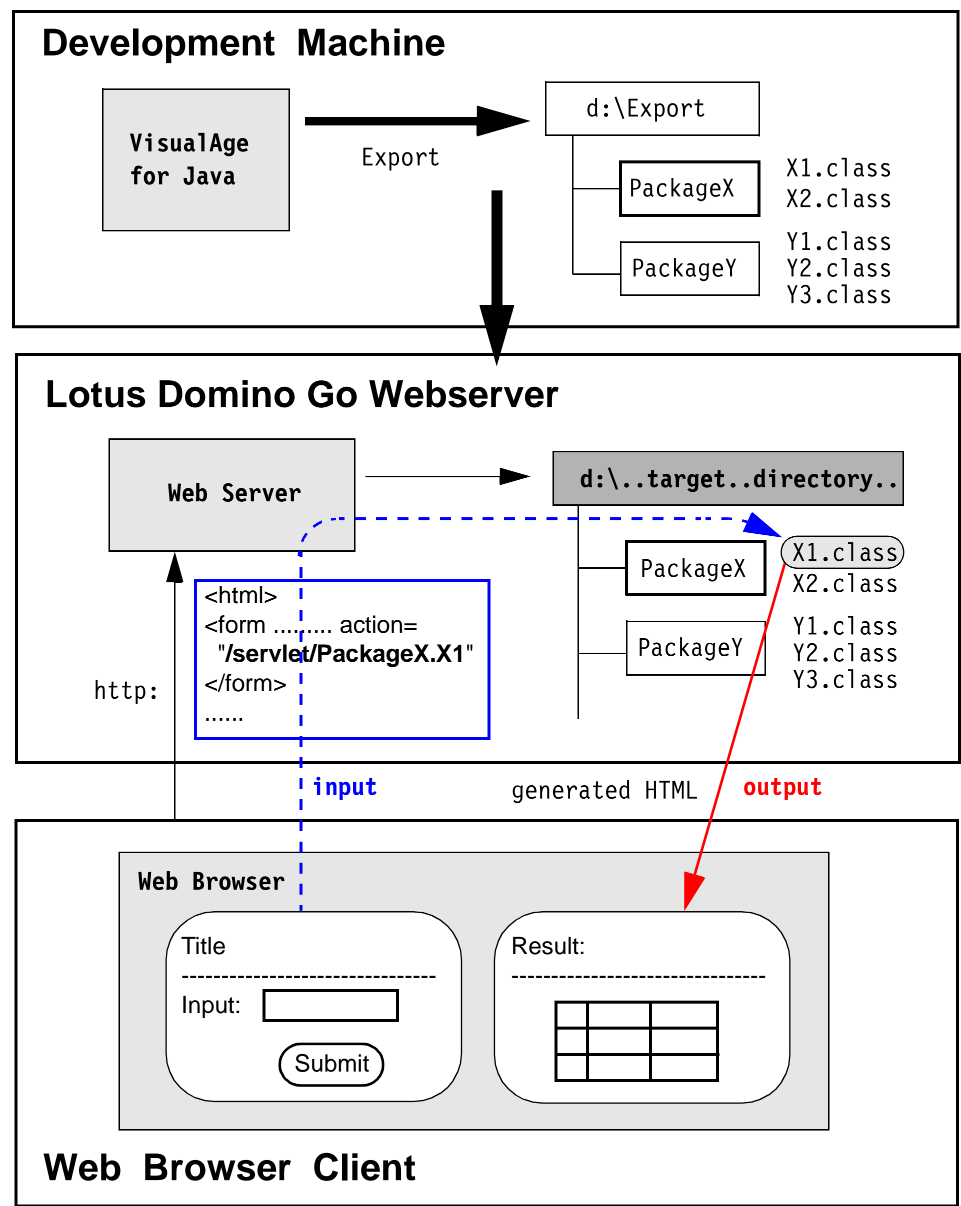


Figure 192. Deployment Process for Servlets in Lotus Domino Go Webserver

Lotus Domino Go Webserver is configured in a configuration file called HTTPD.CNF. We do not go into details here about Web server configuration.

The servlet run-time facility of Lotus Domino Go Webserver is an optional component. To run servlets you install either the servlet runtime or the WebSphere Application Server on top of Lotus Domino Go Webserver.

The differences for deployment of servlets with or without WebSphere are:

- Location of the target directory
- Class path setting

Table 41 gives an overview of the location and class path settings.

Table 41. Servlet Deployment Specifications

Option	Lotus Domino Go	WebSphere
Target location	d:\www\servlets\public	d:\WebSphere\AppServer\classes
Class path	Specified in servlet.cnf file (in c:\Winnt)	Specified using administration applet

Target Location

The target location given in Table 41 is the installation default. You can chose any target directory that is part of the class path setting.

Class Path Setting for Web Server

The most important configuration activity for servlet deployment is the setup of the class path. Every Web server has its unique way of defining its class path. Because many servlets use some of the Enterprise Access Builder classes of VisualAge for Java, it is mandatory that the jar files of the Enterprise Access Builders are deployed to the Web server and added to the class path.

Class Path Setting of Lotus Domino Go Webserver

The class path of the servlet run time of Lotus Domino Go Webserver is specified in the servlet configuration file, servlet.cnf, in the Windows master directory. At the end of the file you find the JavaClassPath specification:

```
JavaClassPath    D:\WWW\Bin\Java\lib\classes.zip;D:\WWW\Servlets\Public;
                 D:\WWW\HTML;D:\WWW\Classes;D:\WWW\CGI-Bin\icsclass.zip;
                 D:\SQLLIB\java\db2java.zip;
                 D:\IBMVJAVA\EAB\RUNTIME20;D:\IBMVJAVA\EAB\RUNTIME;
```

In this example we added the DB2 JDBC drivers and the VisualAge for Java run-time directories. Depending on the use of VisualAge for Java Enterprise Access Builders or other products (CICS, MQSeries), additional jar files might have to be added to the class path.

Class Path Setting for WebSphere

WebSphere does not have a configuration file per se. All configuration activity is performed using an administration applet.

The administration applet is started through a special HTTP command:

```
http://...hostname....:9090/
```

Figure 193 shows the logon form for the WebSphere Application Server.

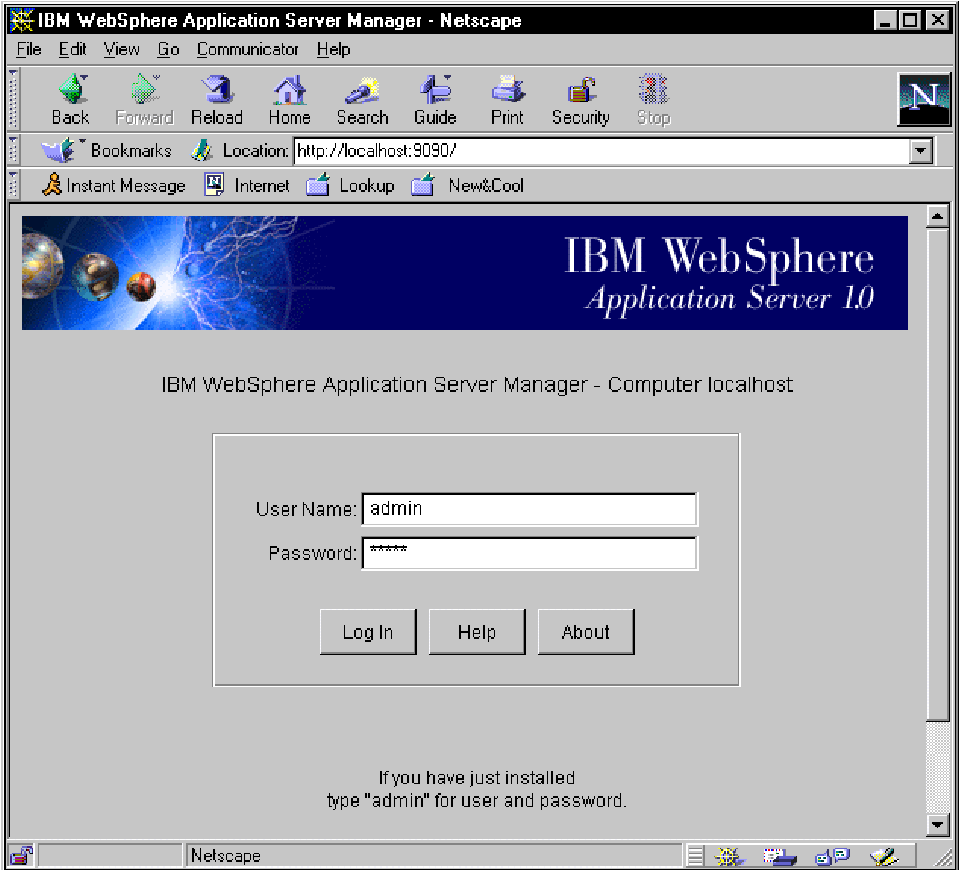


Figure 193. WebSphere Application Server: Administration

WebSphere displays the next administration form, and you click on the *Manage* button to start servlet configuration (Figure 194).

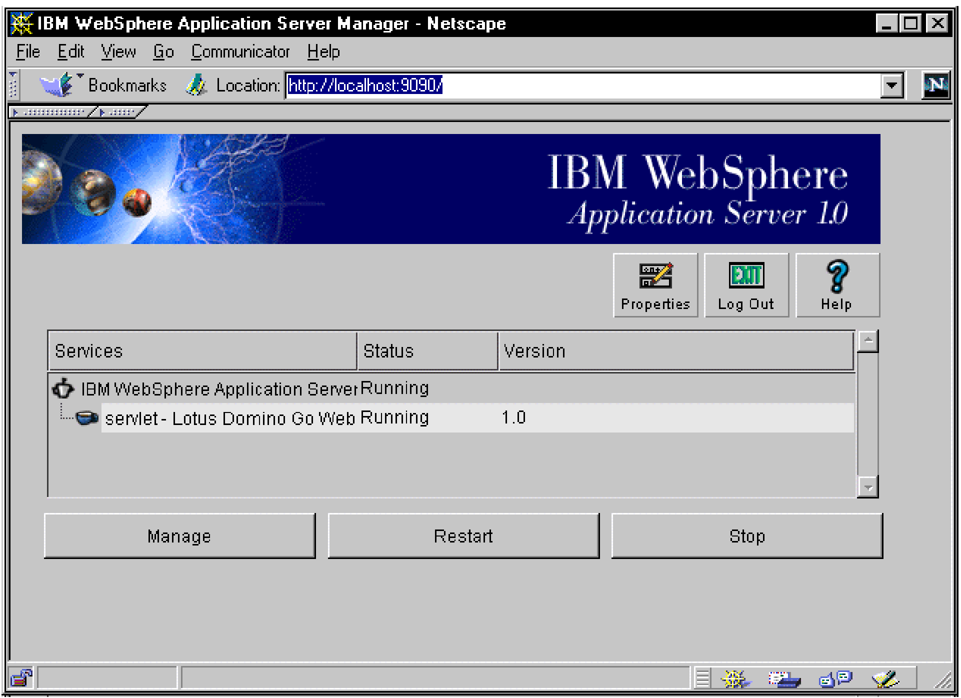


Figure 194. WebSphere Application Server: Manage Configuration

Most of the configuration activities involve the servlet run-time engine of WebSphere.

We do not go through all of the WebSphere administration pages. For now the most important setting to get servlets created with VisualAge for Java to work is the class path setting, and that is on the Basic page.

Clicking on the *Servlets* icon in the tool bar displays another set of pages where you can control individual servlets:

- ❑ Add a servlet
- ❑ Assign an alias (short name)
- ❑ Preload
- ❑ Specify parameters

Figure 195 shows the Basic page of the WebSphere servlet configuration dialog.

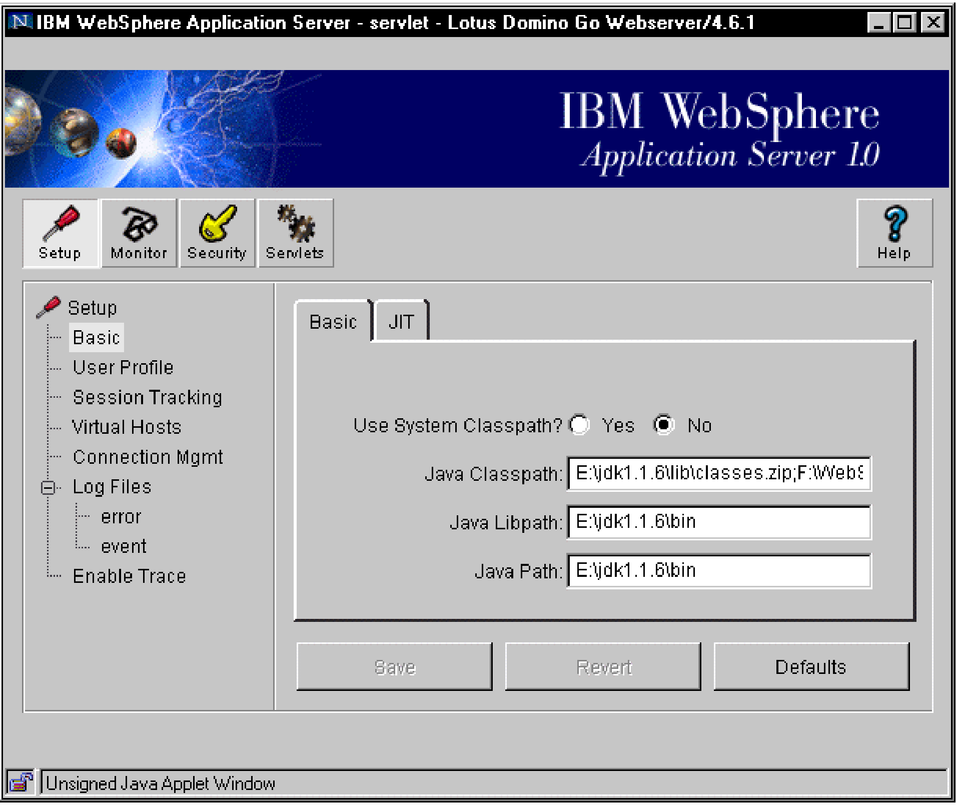


Figure 195. WebSphere Application Server: Servlet Configuration Basic Page

It is difficult to edit the class path setting in that small field, so we suggest copying the text into an editor.

We used the following class path:

```
E:\jdk1.1.6\lib\classes.zip;                    <=== JDK
F:\WebSphere\AppServer\lib\ibmwebas.jar;
F:\WebSphere\AppServer\lib\jst.jar;F:\WebSphere\AppServer\lib\jsdk.jar;
F:\WebSphere\AppServer\lib\x509v1.jar;F:\WebSphere\AppServer\lib;
F:\WebSphere\AppServer\web\admin\classes\seadmin.jar;
F:\WebSphere\AppServer\web\classes\ibmjbrt.jar;
F:\WebSphere\AppServer\web\classes;             <=== export directory
F:\WebSphere\AppServer\servlets;
D:\SQLLIB\java\db2java.zip;                     <=== DB2 JDBC Drivers
D:\IBMVJava\eab\runtime20\ivjdab.jar;           <=== data access beans
D:\IBMVJava\eab\runtime20\ivjsb20.jar;          <=== servlet builder
D:\IBMVJava\hpj\lib\swingall.jar;               <=== swing
```

12.4 Deployment of Applications with Swing

Many new applications use Swing function. Even if you do not use a Swing GUI, there is a good chance that your application uses Swing function; for example, data access beans and visual servlets with HTML result tables use the Swing table model.

If you use any part of Swing, you must make sure that the Swing classes are in the class path. Swing provides several jar files with subsets of the classes and one file with all the classes. This complete file, swingall.jar, is also provided with VisualAge for Java in the IBMVJava\hpj\lib directory.

To make it simple, add the swingall.jar files to every class path.

12.5 Tailoring the Web Browser

Web browsers find classes in several ways. First they look through their own directory of classes, then they check the class path of the platform, and last they ask the Web server for classes.

The jar files of VisualAge for Java are quite large and, instead of exploding them in the Web server, you can install the jar files in the browser's directory.

Netscape

The Netscape browser locates jar files in this directory:

```
d:\...NetscapeInstallDirectory...\Communicator\Program\Java\Classes
```

for example:

```
c:\Program Files\Netscape\Communicator\Program\Java\Classes
```

Microsoft Internet Explorer

Internet Explorer locates jar files in this directory:

```
c:\Winnt\Java\classes                <=== Windows NT
c:\Windows\java\classes              <=== Windows 95
```

13 High-Performance Compiler and Remote Debugger

VisualAge for Java Enterprise Version 2 includes a high-performance compiler and a remote debugger.

In this chapter we briefly describe the high-performance compiler and compile and run the ATM GUI application.

We also briefly describe the remote debugger and apply it to the ATM application.

© Copyright IBM Corp. 1998

13.1 High-Performance Compiler

We do not describe all the facilities and options of the high-performance compiler in this book. We rather choose a practical approach and just show how the ATM application is compiled and run. This experience demonstrates a number of the steps that are necessary and highlights potential problems you may encounter when compiling your own applications.

Compiler Options

The basic command to invoke the high-performance compiler is:

```
hpj [options] input-file
```

Table 42 shows the major compiler options that we will use:

Table 42. High-Performance Compiler Options

-exe	Create an executable (default)
-jll	Create one DLL of all input files (extension .jll)
-jlc	Create a DLL for each input file (extension .jlc)
-O	Optimize the code (default is no optimization)
-verbose	Display compile progress in the window
-o filename	Name of the generated output file
-nofollow	Do not compile referenced classes (default is -follow)
-main class	Name of main program (default is first with *main* method)
-g	Include debugging hooks (default -nog)

The input-file specification can be one file or a file name pattern with asterisks. Files can be source files (.java), class files (.class), jar files (.jar or .zip), or object files from a previous compilation.

The high-performance compiler creates an object file (.obj) for each class, and result file(s) according to the specification (.exe, .jll, .jlc).

Base Java Classes

The base Java classes are precompiled into DLLs. Check out the d:\IBMVJava\hpj\lib\classes directory.

Swing Classes

The Swing classes are precompiled as well. The Swing classes are in the d:\IBMVJava\hpj\lib directory:

```
swingall.jar: Swing classes for class path
swingall.jll: Compiled Swing classes for high-performance compiler
```

Execution

To execute a compiled program, just run the executable, or, for a DLL use the hpjava command:

```
hpjvava -load dll-name;dll-name classname parameters
```

To pass execution options to a .exe file, use an environment variable:

```
set IBMHJ_OPTS=execution-options
```

If you get an error message about a Swing class not being found, be sure to include the swingall.jll in the -load option:

```
set IBMHJ_OPTS=-load d:\IBMVJava\hpj\lib\swingall.jll
```

13.2 Compiling the ATM Application

To compile the ATM application we have to perform a series of steps:

- ❑ Export the code from VisualAge for Java
- ❑ Compile the ATM application
- ❑ Compile the data access beans
- ❑ Compile the DB2 JDBC drivers

Export the ATM Application

To export the ATM application from VisualAge for Java, we could use the same directory as discussed in “Deployment of Applications” on page 334, but to separate the files we create a new directory:

```
md d:\atmgui
```

We export the three packages of the ATM application:

```
itso.entbk2.atm.model
itso.entbk2.atm.databean
itso.entbk2.atm.gui
```

We export only the class files. The high-performance compiler can run from source Java files or from class files. Exporting the packages creates the package subdirectories under the d:\atmgui directory.

Test the Exported Application

Before compiling we can test whether the application works. Check that DB2 is started, and that the Java daemon (db2jstrt 8888) is started as well. Remember that we use the net driver in the ATM data bean implementation.

```
cd atmgui
set classpath=d:\atmgui;d:\SQLLIB\java\db2java.zip;%classpath%
java itso.entbk2.atm.gui.ATMApplet
```

The application should work without a problem. You may have to resize the window to have the panel displayed properly.

Compile the ATM Application

In the d:\atmgui directory we invoke the high-performance compiler for the ATM application. We can only compile one directory at a time. Therefore, we compile the GUI part into an executable, and each of the two supporting packages into a DLL. We specify the -nofollow option so that no other classes are compiled. We also add the atmgui directory to the class path:

```
cd atmgui
set classpath=d:\atmgui;%classpath%
hpj -exe -O -o atmgui.exe -verbose -nofollow itso\entbk2\atm\gui\*.class
hpj -jll -O -o atmdata.jll -verbose -nofollow itso\entbk2\atm\databean\*.class
hpj -jll -O -o atmmodel.jll -verbose -nofollow itso\entbk2\atm\model\*.class
```

Compile the Data Access Beans

The data access beans are not available in compiled format. Therefore we have to compile those classes ourselves. The classes are available in a jar file:

```
d:\IBMVJava\eab\runtime20\ivjdab.jar
```

In a first try we used the -nofollow option but ran into a problem when trying to execute later. The data access beans require a few classes of the com.ibm.uvm.edit package, and that package is not available outside VisualAge for Java.

Exporting the com.ibm.uvm Classes

We export the entire IBM Java Implementation project into a jar file and add the jar file to the class path so that its classes can be found:

```
d:\IBMVJava\eab\runtime20\ibmuvm.jar
set classpath=d:\IBMVJava\eab\runtime20\ibmuvm.jar;%classpath%
```

Compile the Data Access Beans

Now we compile the data access beans with the -follow option:

```
cd d:\IBMVJava\eab\runtime20
hpj -jll -O-o ivjdab.jll -verbose -follow ivjdab.jar
```

Note that we could compile the complete ibmuvm.jar file, but the data access beans actually require only three classes.

Compile the DB2 JDBC Drivers

We know that the data access beans use the JDBC drivers to access the relational database. Therefore we compile the DB2 JDBC driver classes into a DLL:

```
cd d:\SQLLIB \java
hpj -jll -O -verbose -o db2java.jll db2java.zip
```

Remove the Object Files

The compiler creates an object (.obj) file for every compiled class. You can safely remove those classes:

```
cd d:\IBMVJava\eab\runtime20
del *.obj /s
cd com\ibm
rd uvm /s                    <=== remove uvm directory structure
cd d:\SQLLIB\Java
rd com /s                    <=== remove com\ibm\ directory structure
```

13.3 Run the Compiled ATM Application

Now let's try to run the compiled application.

Check that DB2 and the Java daemon (db2jstrt 8888) are started. Remember that we use the net driver in the ATM data bean implementation.

We have to load all DLLs before starting the executable:

```
d:\IBMVjava\hpj\lib\swingall.jll
d:\sqllib\java\db2java.jll
d:\IBMVJava\eab\runtime20\ivjdab.jll
atmdata.jll
atmmodel.jll
```

One approach is to create a small .bat file for the execution (the -load option must fit on one line):

```
set IBMHPJ_OPTS=-load d:\IBMVjava\hpj\lib\swingall.jll;
          d:\sqllib\java\db2java.jll;d:\IBMVJava\eab\runtime20\ivjdab.jll;
          atmdata.jll;atmmodel.jll
atmgui
```

The ATM application should run perfectly. You may have to resize the window so that the panel appears.

13.4 Alternative Compile Approach

Another way of compiling is to compile the main class with the -follow option and have the high-performance compiler find all referenced classes.

To make this approach work we have to set the class path so that all classes can be found.

```
set classpath=d:\atmgui;d:\SQLLIB\java\db2java.zip;%classpath%
```

Now we compile the main class:

```
hpj -exe -O -o atmgui2.exe -verbose itso\entbk2\atm\gui\ATMApplet.class
IVJH3220(S) Stopping because an error was detected:
IVJH3260(E) Class com.ibm.uvm.abt.edit.GenericBeanInfo not found.
```

We immediately run into the problem of the com\ibm\uvm classes not being found. We change the class path to include the jar file we exported previously:

```
set classpath=d:\atmgui;d:\SQLLIB\java\db2java.zip;%classpath%
set classpath=d:\IBMVJava\eab\runtime20\ibmuvm.jar;%classpath%
```

Let’s try again to compile the main class:

```
hpj -exe -O -o atmgui2.exe -verbose itso\entbk2\atm\gui\ATMApplet.class
itso.entbk2.atm.gui.ATMApplet
itso.entbk2.atm.model.ATMApplicationController
itso.entbk2.atm.gui.TransactionPanel
......
itso.entbk2.atm.databean.AtmDB
itso.entbk2.atm.model.ATMPersistenceDefault
itso.entbk2.atm.model.ATMPersistenceInterface
itso.entbk2.atm.model.Customer
......
itso.entbk2.atm.databean.UpdateBalance
itso.entbk2.atm.databean.Transactions
......
com.ibm.ivj.db.uibeans.Select
......
com.ibm.uvm.abt.edit.GenericBeanInfo
com.ibm.uvm.abt.edit.DiscriminatedPropertyDescriptor
......
atmgui2.exe
```

This approach looks very promising because only the required classes are compiled. Note that we still have to load the Swing and DB2 DLLs.

However, execution fails with an exception when accessing DB2:

```
set IBMHPJ_OPTS=-load d:\IBMVjava\hpj\lib\swingall.jll;d:\sqllib\java\db2java.jll
atmgui2
The SQL type specified for the column is invalid or unsupported.
The SQL type specified for the column is invalid or unsupported.
The SQL type specified for the column is invalid or unsupported.
The specified parameter index or name is not defined.
java.lang.IllegalArgumentException:
    at java.lang.Class.forName (Offset 0x0000000e)
    at com.ibm.db.base.JDBCConnectionManager.registerDriver (Offset 0x0000003d)
    at com.ibm.db.base.JDBCConnectionManager.getJDBCConnectionManager (Offset..)
    at com.ibm.db.base.DatabaseConnectionSpec.connect (Offset 0x00000066)
    at com.ibm.db.DatabaseConnection.connect (Offset 0x000001f0)
    at com.ibm.db.SelectStatement.execute (Offset 0x000002c1)
    at com.ibm.ivj.db.uibeans.Select.execute (Offset 0x0000000f)
    at itso.entbk2.atm.databean.Accounts.execute (Offset 0x00000011)
    at itso.entbk2.atm.databean.AtmDB.extGetAccounts (Offset 0x0000003f)
    at itso.entbk2.atm.model.ATMApplicationController.getAccounts (Offset ...)
    at itso.entbk2.atm.gui.PinPanel.connEtoM9 (Offset 0x00000060)
    at itso.entbk2.atm.gui.PinPanel.handlePinCheckedOk (Offset 0x0000007b)
    at itso.entbk2.atm.model.Card.fireHandlePinCheckedOk (Offset 0x00000065)
    at itso.entbk2.atm.model.Card.checkPin (Offset 0x00000065)
    at itso.entbk2.atm.gui.PinPanel.connEtoM7 (Offset 0x00000070)
    at itso.entbk2.atm.gui.PinPanel.actionPerformed (Offset 0x00000032)
    at com.sun.java.swing.AbstractButton.fireActionPerformed (Offset 0x000002d1)
    at com.sun.java.swing.AbstractButton$ForwardActionEvents.actionPerformed (Offset)
    at com.sun.java.swing.DefaultButtonModel.fireActionPerformed (Offset ...)
    at com.sun.java.swing.DefaultButtonModel.setPressed (Offset 0x00000119)
    at com.sun.java.swing.plaf.basic.BasicButtonListener.mouseReleased (Offset ...)
    at java.awt.Component.processMouseEvent (Offset 0x00000104)
    at java.awt.Component.processEvent (Offset 0x000000e3)
    at java.awt.Container.processEvent (Offset 0x00000072)
    at java.awt.Component.dispatchEventImpl (Offset 0x0000022d)
    at java.awt.Container.dispatchEventImpl (Offset 0x000000ac)
    at java.awt.Component.dispatchEvent (Offset 0x00000008)
    at java.awt.LightweightDispatcher.retargetMouseEvent (Offset 0x000000fe)
    at java.awt.LightweightDispatcher.processMouseEvent (Offset 0x0000008e)
    at java.awt.LightweightDispatcher.dispatchEvent (Offset 0x00000067)
    at java.awt.Container.dispatchEventImpl (Offset 0x00000026)
    at java.awt.Component.dispatchEvent (Offset 0x00000008)
    at java.awt.EventDispatchThread.run (Offset 0x000000cd)
```

It looks like we are missing some other classes!

13.5 Remote Debugger

The remote debugger can be used to debug an application that runs in its native execution environment. The user interface of the remote debugger runs on a Windows platform, whereas the program executes on another Intel machine, an AS/400, an OS/390, or an AIX machine.

The remote debugger is part of the Enterprise Toolkit. You need the toolkits for both the debugging and the execution platform.

In this short visit to the remote debugger we only show the user interface part without actually running the program on another machine.

Reasons for Remote Debugging

You have to use a remote debugger if:

- ❑ You have to debug an application that uses graphics or has a GUI and you need to keep the debugger user interface separate from the application's GUI
- ❑ The program you are debugging was compiled for a platform on which the debugger user interface does not run
- ❑ You cannot duplicate the environment for the program on your local machine

Running the Remote Debugger

In a real case you would run in a client/server environment. Two programs provide the function of remote debugging:

- ❑ On the execution platform you start the JDBUG program:

```
jdbug -qport=xxxx
```

- ❑ On the debugging platform (for example, Windows NT) you start the JDEBUG program:

```
jdebug -qport=xxxx -qhost=remotehostname programclass parameters
```

Note that you specify the program to execute on the debugging machine and not on the execution machine.

To debug a local program you start the JDBUG program directly:

```
jdbug programclass parameters
```

13.6 Remote Debugging of the ATM Application

Let's start the remote debugger for the ATM application.

```
set classpath=d:\atmgui;d:\SQLLIB\java\db2java.zip
cd d:\atmgui
jdbug itso.entbk2.atm.gui.ATMApplet
```

The first window that opens is the Debugger - Session Control window (Figure 196).

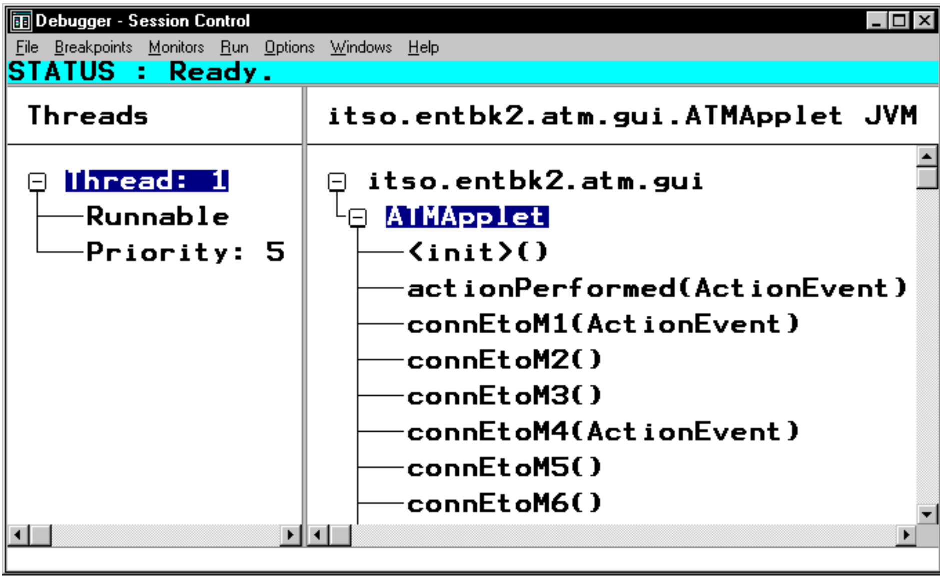


Figure 196. Remote Debugger: Session Control Window

You can use the Session Control window to load additional source programs so that you can set breakpoints. You can also set up a number of debugging options.

After the Session Control window is opened, the source window of the application opens (Figure 197).

Source: ATMApplet - Thread :1

File View Breakpoints Monitors Run Options Windows Help

```
1043  * @param args an array of command-line arguments
1044  */
1045 public static void main(java.lang.String[] args) {
1046         ATMApplet applet = new ATMApplet();
1047         java.awt.Frame frame = new java.awt.Frame("Applet");
1048
1049         frame.addWindowListener(applet);
1050         frame.add("Center", applet);
1051         frame.setSize(350, 250);
1052         frame.show();
1053
1054         applet.init();
1055         applet.start();
1056 }
1057 /**
1058  * Paints the applet.
1059  * If the applet does not need to be painted (e.g. if it is o
1060  * awt components) then this method can be safely removed.
1061  *
1062  * @param g  the specified Graphics window
1063  * @see #update
1064  */
1065 public void paint(Graphics g) {
1066         super.paint(g);
1067
1068         // insert code to paint the applet here
```

Figure 197. Remote Debugger: Source Window

In the Source window you control the execution by setting breakpoints and stepping through the code, using the buttons in the tool bar. You can execute line by line, step into methods within one line, execute to the method's return point, or run to the next breakpoint.

For example, you can click on the *Run* button (in the tool bar) and wait until the ATM applet window (where you can enter the card number) appears. You then halt the execution, using the *Halt* button in the tool bar, load the Card class from the itso.entbk2.atm.model package, and set a breakpoint in the checkPin method. You restart the execution, enter the card number and PIN in the applet's window, and the program stops in the Card class at the breakpoint (Figure 198).

Source: Card - Thread :2

File View Breakpoints Monitors Run Options Windows Help

```
55 }
56 /**
57  * Perform the checkPin method.
58  * @param pinEntered java.lang.String
59  */
60 public boolean checkPin(String pinEntered) {
61         /* Perform the checkPin method. */
62         if ( getPinCard().trim().equals(pinEntered.trim()) ) {
63             fireHandlePinCheckedOk(new PinCheckedOkEvent(this));
64             return true;
65         }
66         else {
67             fireHandlePinCheckedNotOk(new PinCheckedNotOkEvent(this));
68             return false;
69         }
70 }
71 /**
72  * Perform the clearAccounts method.
73  */
74 public void clearAccounts() {
75         /* Perform the clearAccounts method. */
76         setAccounts(new java.util.Vector());
77 }
78 /**
```

Figure 198. Remote Debugger: Source Window with Breakpoint

To display the values of program variables, you click on a variable and select *Popup expression* or *Add to program monitor* in the pop-up window. Figure 199 shows the Program Monitor window with the value of a variable.

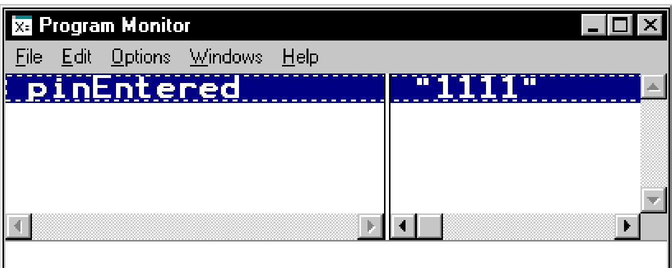


Figure 199. Remote Debugger: Program Monitor Window

The debugger remembers breakpoints if you debug the same program multiple times, and the Program Monitor automatically opens with the last shown variables.

13.7 Debugging a Compiled Program

The remote debugger facility is also available for compiled programs. Two programs provide the function of remote debugging for compiled programs:

- On the execution platform you start the JRMTDBG program:

```
jrmtdbg -qport=xxxx
```

- On the debugging platform (for example, Windows NT) you start the JDEBUG program:

```
jdebug -qport=xxxx -qhost=remotehostname program parameters
```

To debug a local compiled program you start the JDBUG program:

```
jdebug programclass parameters
```

To debug the compiled ATM application:

```
cd d:\atmgui
set IBMHPJ_OPTS=-load d:\IBMVjava\hpj\lib\swingall.jll;
         d:\sqllib\java\db2java.jll;d:\IBMVJava\eab\runtime20\ivjdab.jll;
         atmdata.jll;atmmodel.jll
jdebug atmgui.exe
```

If debugging hooks were not generated into the compiled program, the debugger displays a disassembly of the program in the source window.

Appendixes

© Copyright IBM Corp. 1998

A Installation, Setup, and Configuration

In this appendix we describe the installation of VisualAge for Java Version 2 and the setup that is necessary for the different chapters of this book:

- ❑ Data access beans
- ❑ Servlet Builder
- ❑ CICS Connector
- ❑ MQSeries

We also describe the installation and content of the sample code distributed with this book.

Please review the setup for each chapter before starting to develop or run sample code.

© Copyright IBM Corp. 1998

A.1 Setup of VisualAge for Java Enterprise Version 2

Follow the standard installation procedure for VisualAge for Java Enterprise Version 2. Select all the optional features except for the Enterprise Toolkit/400 (ET/400) that is not used in this book.

You can install the team server and work in a client/server configuration, but it is not necessary for any of the samples discussed in this book.

We strongly suggest that you install the latest fixpack for the product. Check out the information about fixpacks on the Web page:

```
http://www.software.ibm.com/ad/vajava
```

Workspace Recovery

After installing the product and the fixpacks, make a copy of the workspace file, IDE.ICX in d:\IBMVJava\ide\program. If you ever need to recover the workspace, you can always go back to this copy and then load the features and your own code from the repository.

Repository Recovery

We suggest making a copy of the repository file, IVJ.DAT in d:\IBMVJava\ide\repository, every week. Should you ever lose the repository, you can go back to the copy.

Saving Your Own Work

To save all applications you develop, version the project or the packages by selecting *Manage->Version* in the context menu. Then export the project or the packages in repository format to a new repository file (ITSOENTBK2.dat). You can always reload your saved code into the repository.

A.2 Setup for Data Access Beans

To use the data access beans you must have a relational database with a JDBC driver. The IBM product is DB2 V5 (UDB) or DB2 V2.12 with one of the newer service packs. Only very simple examples work without applying a service pack. The sample application discussed in Chapter 2, “Relational Database Access with Data Access Beans” on page 9 requires the service pack to handle the BLOBs.

VisualAge for Java

Add the data access beans feature to the Workbench:

- ❑ Select *File->Quick Star*t, or press F2
- ❑ Select *Features->Add Feature* and select *IBM Data Access Beans 1.0* in the dialog that is displayed

DB2 JDBC Driver Classes

Add the db2java.zip file to the workspace class path:

- ❑ In the Workbench select *Window-> Options*
- ❑ Select *Resources*
- ❑ Click on *Edit* for the workspace class path
- ❑ Add the file db2java.zip that is in d:\SQLLIB\JAVA

There is no need to load the DB2 JDBC driver classes into the Workbench.

Applications

Make sure that the class path for each executable application is set properly. In case of error use *Run->Check Class Path* in the context menu of the class and click on *Compute Now.*

DB2

Make sure that DB2 is started. If you are using the net driver (COM.ibm.db2.jdbc.net.DB2Driver), make sure that the DB2 Java daemon is started (db2jstrt 8888) and that the database URL specifies the same port (jdbc:db2://hostname:8888/databasename).

A.3 Setup for the Servlet Builder

Add the Servlet Builder feature to the Workbench:

- ❑ Select *File->Quick Star*t, or press F2
- ❑ Select *Features->Add Feature* and select *IBM Servlet Builder 2.0* in the dialog that is displayed

Test the help facility using the menu option *Help->Help Home Page*. The Web browser should start and display the help menu. When testing servlets within VisualAge for Java, the Web browser configured for the help facility is started. Testing the help facility confirms that a suitable browser was configured at installation time.

Make sure that the class path for each executable application is set properly. In case of error use *Run->Check Class Path* in the context menu of the class and click on *Compute Now.*

Web Server

You need a Web server that has run-time support for servlets. Most current Web servers have this support.

We used Lotus Domino Go Webserver to deploy the servlets. We also installed the WebSphere Application Server on top of Lotus Domino Go.

For more information about deployment of servlets, see “Deployment of Servlets” on page 338.

A.4 Setup for the CICS Connector

Using the CICS Connector to develop applications requires four features loaded into the Workbench:

- ❑ IBM Java Record Library 2.0
- ❑ IBM Common Connector Framework 2.0
- ❑ CICS Connector 3.0
- ❑ IBM Enterprise Access Builder Library 2.0

Make sure that the class path for each executable application is set properly. In case of error use *Run->Check Class Path* in the context menu of the class and click on *Compute Now.*

Setup of the CICS Server and Client

TXSeries Version 4.2 is used for VisualAge for Java and CICS.

Prepare CICS for the ATM application:

❑ Create a CICS region:

```
cicscp -v create dce -R
cicscp -v create region ATMCICS
cicscp -c ld -r ATMCICS -P ATMTCP Protocol=TCP
```

❑ Add the programs to the CICS definition:

```
cicsadd -c pd -r ATMCICS -P ATMCARDI ActivateOnStartup=yes
      PathName="d:\va2entbk\sampcode\cics\ATMCARDI"
      Resident=yes RSLKey=public
```

❑ Start the CICS Administration utility, highlight the `ATMCICS` region, pop-up, Start. Select *cold start* and wait for this message:

```
Region ATMCICS started successfully.
```

Stop the region again.

❑ Find the CICS client initialization file (CICSCLI.INI) and edit it with the Notepad editor:

- Change: MaxServers=10
- Change: Server = ATMTCP
- Change: NetName=*CICS server's host name*

❑ To test or run the ATM application:

- Start the `ATMCICS` region. Wait for "started successfully" message.
- Start the CICS Client and the CICS Transaction Gateway.

COBOL Sample Programs

The main focus of this book is to show how different enterprise access software can be integrated into VisualAge for Java. Because the intention is to show how a message-based application can be invoked from a servlet or application, the actual design and details of the back-end programs have been simplified. The sample programs are really skeleton programs. The aim of these programs is to illustrate a concept. A sophisticated and all-encompassing design has not been implemented. For example, DB2 access is simulated, and data information is hard-coded.

Table 43 lists the CICS COBOL programs that we wrote to implement some of the ATM transactions. All programs were written using VisualAge for COBOL Version 2.2 on NT.

Table 43. CICS COBOL Programs

Program	Function	Remarks
ATMCARDI	Uses the card ID in the COMMAREA to look up card and customer information. Returns card and customer information in the COMMAREA.	All database access is simulated. There is no explicit UOW control.
ATMACCNT	Uses the card ID in the COMMAREA to look up associated accounts. Stores account information in the COMMAREA.	All database access is simulated. There is no explicit UOW control. A limit of 10 accounts is imposed.
ATMBALUP	Updates the account balance based on the information in the COMMAREA.	All database access is simulated. Updates occur only in memory. There is an explicit rollback in the event of failure.
ATMMQBR	Acts as a bridge between MQSeries and CICS.	See “MQSeries CICS Bridge Program” on page 369 for more information.

ATM Business Transaction Programs

Every ATM business transaction is associated with one or more programs. Only the programs associated with the instantiation of a card and customer object are discussed here.

These programs do not issue MQSeries calls. They are CICS COBOL programs. In reality they should also have issued DB2 calls. These are simulated, and data is stored in memory.

All programs are passed the ATM header discussed in “ATM Header for the COMMAREA” on page 245. It is imperative that the return code in this header be set to indicate the success or failure of the transaction.

Compiling and Defining CICS COBOL Programs

Compiling and defining programs is operating system, Transaction Server, and COBOL compiler dependent. For an explanation of how to do this, please refer to the CICS Application Programming guide relevant to your platform.

If program autoinstall is not available or is not being used, a CICS program definition must be defined and installed per program being loaded.

Program ATMCARDI

The program to implement the business function of retrieving information associated with a card is called ATMCARDI. When the user enters a card number, sufficient enterprise data is retrieved in order to instantiate a card and customer object. All database accesses are simulated.

ATMCARDI retrieves the card based on the card ID in the COMMAREA. If a valid card is found, the output COMMAREA is set up with the card ID, PIN, and the customer information (number, title, first name, last name). For an invalid card ID an error message is set up in the COMMAREA, and the return code is set to nonzero.

Program ATMACCNT

Every card is associated with one to many accounts. In this program, however, we placed an artificial limit of a maximum of 10 accounts per card. The account information must be retrieved from the enterprise database. Database access is simulated.

ATMACCNT retrieves the accounts for the given card. The account information is placed in the COMMAREA. If the access fails, an error message is set up in the COMMAREA, and the return code is set to nonzero.

Program ATMBALUP

Program ATMBALUP implements the business function of updating an account balance. This is truly a skeleton program. No actual update takes place as no database calls are issued. All data is simulated in memory. In addition a balance update should be associated with a transaction history update. This is not done.

ATMBALUP updates the balance of the given account. If the update is successful, a SYNCPOINT is issued; if unsuccessful, a ROLLBACK is issued, an error message is set up in the COMMAREA, and the return code is set to nonzero. (The program should also add the new transaction to the transaction table, but this was not implemented.)

A.5 Setup for MQSeries on Windows NT

Development of VisualAge for Java applications with MQSeries requires Version 5 of the product.

Installation Considerations

When you install MQSeries Version 5, select the MQSeries server, client, toolkit, and bindings for Java features. After rebooting, open the user manager (*Programs -> Administrative Tools -> User Manager*) and add your Windows NT user ID to the group named *mqm*. (This is for administration purposes only; MQSeries users need not be in the mqm group.)

Queue Manager and Queue Setup

Set up the queue manager and MQSeries objects for the ATM application as described in "MQSeries Queue Manager and Objects" on page 292.

After setting up the objects, start the queue manager and the listener.

VisualAge for Java Setup

Import the three MQSeries packages, com.ibm.mq, com.ibm.mqbind, and com.ibm.mqservices, into a new project, for example, MQ Series. Version the code to a version identifier of 5.0 (for MQSeries Version 5).

MQSeries CICS Bridge Program

Conceptually we decided to implement a bridge type of design by having a program that would interface with MQSeries and control the business UOW. However, the actual program written to do this task is written and designed extremely simply and is a somewhat trivial example. As stated in "Unit of Work Considerations" on page 305, the UOW control issues have been largely ignored. It is assumed that organizations using MQSeries and CICS have implemented some form of MQSeries CICS broker or bridge. IBM supplies the MQSeries CICS/ESA bridge as part of MQSeries MVS/ ESA 1.2.

Conceptually the design calls for each business transaction to be independent of the original invocation mechanisms. The bridge hides the details of the source and destination end points. Whether the retrieve card information business function is invoked through an MQSeries message or a CICS client should be irrelevant to the process performing this function. Each business

transaction is invoked through a well-defined protocol. In this sample it is linked to and data is passed through the COMMAREA.

Sample Program ATMMQBR

In these examples the MQSeries-CICS bridge is implemented in the ATMMQBR program. This program determines the business transaction it must link to from the first 8 bytes of the request MQSeries message. The program performs the following functions:

- Issues a Handle Abend, and, should an abend occur, an EXEC CICS SYNCPOINT ROLLBACK
- If triggered, gets the name of the request queue from the MQTMC2 structure
- Connects to the queue manager
- Opens the request queue
- Gets a request message out of syncpoint
- Calls the program specified in message. This program is responsible for the business and data services.
- Puts (out of syncpoint) a reply message containing enterprise data[12]
- If all is OK, issues an EXEC CICS SYNCPOINT to commit the business unit of work
- Gets the next message

In reality the UOW control would be very different. The MQGET and MQPUT would more than likely occur within syncpoint, and the bridge would control the UOW for all recoverable resources. It has not been done here. The processing shown would need to be a lot more sophisticated and design decisions on what to do with backed-out messages would have to be made. In addition it is extremely unlikely that access to request messages would be single streamed as implied by this design.

Compiling the ATMMQBR Program

To compile an MQSeries CICS COBOL program refer to the *MQSeries Application Programming Guide*. The steps to follow depend on the operating system and compiler.

For the ATM application we used VisualAge for COBOL 2.2, MQSeries Version 5, and TXSeries Version 4.2.

[12] We really should have put using the MQPMO-SYNCPOINT option, but the definitions enabling two-phase commit between MQSeries and TXSeries were not set up.

- ❑ Set up environment variables:
  - USERLIB=MQMCBB.LIB
  - CICS_IBMCOB_FLAGS=mqm-drive\mqm\tools\cobol\copybook\VA cobol;%CICS_IBMCOB_FLAGS%

  Note. The setting of CICS_IBMCOB_FLAGS is documented in Chapter 28, “Building Your Application on Windows NT, Preparing COBOL Programs, Preparing CICS and Transaction Server Programs,” of the *MQSeries Application Programming Guide*. When we set the CICS_IBMCOB_FLAGS as suggested, we got an error in the compile step. To compile successfully we did not set this variable. This problem is probably the result of faulty installation. We suggest you set the CICS_IBMCOB_FLAGS variable, and only if you get an error, try unsetting it.

- ❑ Assuming your current path points to your source program directory, to translate, compile, and link a COBOL program, issue the following command:

  cicstcl -l IBMCOB atmmqbr.ccp

  As an aid to using the translate, compile, and link facilities offered with TXSeries, choose the development option when installing the product.

Running ATMMQBR under CICS

You can invoke ATMMQBR under CICS in various ways, for example, as a PLT program, from a terminal, or triggered through MQSeries.

For this discussion we assume that the program is triggered through MQSeries. A trigger type of first is assumed. When we first developed the MQSeries access code with the MQSeries Client for Java, we coded the method to open an MQSeries queue generically so that the queue would open for both input and output. This prevents triggering on first. The method had to be refined to take the open options as a parameter.

CICS MQSeries Trigger Monitor

This monitor is platform dependent. The sample CICS trigger monitors are discussed under “Trigger Monitors” in the chapter entitled “Starting MQSeries Applications Using Triggers” in the *MQSeries Application Programming Guide*.

The sample trigger monitor for TXSeries Version 4.2 for Windows NT is AMQLTMC4. If you use this monitor, you probably have to make the CICS group a member of the mqm group.

CICS Resource Definitions

To support triggering and running of ATMMQBR the CICS resources listed below need to be defined. Note that the transaction names were chosen arbitrarily. The definitions assume a Windows NT platform. They would be different for other platforms.

- Program AMQLTMC4
- Transaction MQTM associated with program AMQLTMC4
- Program ATMMQBR
- Transaction ATMQ associated with program ATMMQBR

Starting the CICS Trigger Monitor

Because the trigger monitor is platform dependent, different trigger monitors can be started in different ways.

For our testing, we started the NT CICS trigger monitor with CECI:

```
START TRANSID(MQTM)
```

A.6 Installation of the Redbook Samples

The samples used in this book are distributed in a zip file called sg245265.zip. This file is available from the ITSO home page:

```
ftp://www.redbooks.ibm.com/redbooks/SG245265
```

Unzipping the file on a hard drive creates an SG245265 directory with subdirectories as listed in Table 44.

Table 44. Redbook Sample Code

Subdirectory	Description
AtmDB	DDL and SQL to setup and load the ATM database in DB2 - AtmDB.ddl: database and table definitions - AtmDB.sql: SQL insert statements with sample data
BusModel	Supporting files for ATM business model - atmBusModel.scrap: Scrapbook to test business model
Databean	Supporting files for data access beans - atmDatabean.scrap: Scrapbook to test relational persistence
Servlet	Supporting files for servlets - xxxxx.java: sample servlet source code - xxxxx.html, xxxxx.shtml: sample HTML/SHTML files
CICS	Supporting files for CICS - atmCicsXxxxx.scrap: Scrapbooks to test CICS transactions
COBOL	CICS COBOL programs for ATM application - xxxCommarea: COMMAREA of COBOL programs - atmXxxxxx.ccp: COBOL source programs
MQ	Supporting files for MQSeries - atmobj.def: ATM object definition file - atmobj.bat: run ATM object definition - atmqueue.bat: define the queue for ATM application - startmq.bat, stopmq.bat: start/stop queue manager/listener - atmMQ.scrap: Scrapbook to test MQSeries persistence
HpjComp	Supporting files for the high-performance compiler and the remote debugger - skeletoncompile.cmd: compile ATM applet - hpjcompile.txt: compile console text with -follow - atmGuiX.bat: run Java applet, class file or executable - atmDebugX.bat: remote debugger, class file or executable
Dat	Repository import file for VisualAge for Java - sg245265.dat: see Table 45

The repository file can be imported into VisualAge for Java. We used a project named *ITSO VAJ Enterprise Book V2*. The samples are structured into the packages listed in Table 45.

Table 45. Packages of the Redbook Sample Applications

Package	**Description**
itso.entbk2.atm.cics	ATM persistence with CICS Connector
itso.entbk2.atm.databean	ATM persistence with data access beans
itso.entbk2.atm.gui	ATM applet or application with Swing GUI
itso.entbk2.atm.model	ATM business model classes
itso.entbk2.atm.mq	ATM persistence with MQSeries
itso.entbk2.atm.servlet	ATM HTML application with servlets
itso.entbk2.sample.databean	Data access beans examples
itso.entbk2.sample.servlet	Servlet Builder examples

B Special Notices

This publication is intended to help VisualAge for Java developers develop enterprise applications with VisualAge for Java Enterprise Version 2. The information in this publication is not intended as the specification of any programming interfaces that are provided by VisualAge for Java Enterprise. See the PUBLICATIONS section of the IBM Programming Announcement for VisualAge for Java Enterprise 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

© Copyright IBM Corp. 1998

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 information about non-IBM ("vendor") products in this manual has been supplied by the vendor and IBM assumes no responsibility for its accuracy or completeness. 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.

Any pointers in this publication to external Web sites are provided for convenience only and do not in any manner serve as an endorsement of these Web sites.

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

Reference to PTF numbers that have not been released through the normal distribution process does not imply general availability. The purpose of including these reference numbers is to alert IBM customers to specific information relative to the implementation of the PTF when it becomes available to each customer according to the normal IBM PTF distribution process.

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

IBM	AIX
CICS	CICS\ESA
DATABASE 2	DB2
IMS	MQ
MQSeries	MVS\ESA
NetRexx	OS/2
S/390	ThinkPad
VisualAge	WebSphere

The following terms are trademarks of other companies:

C-bus is a trademark of Corollary, Inc.

Java and HotJava are trademarks of Sun Microsystems, Incorporated.

Microsoft, Windows, Windows NT, and the Windows 95 logo are trademarks are registered trademarks of Microsoft Corporation.

PC Direct is a trademark of Ziff Communications Company and is used by IBM Corporation under license.

Pentium, MMX, ProShare, LANDesk, and ActionMedia are trademarks or registered trademarks of Intel Corporation in the U.S. and other countries.

UNIX is a registered trademark in the United States and other countries licensed exclusively through X/Open Company Limited.

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

C Related Publications

The publications listed in this section are considered particularly suitable for a more detailed discussion of the topics covered in this redbook.

© Copyright IBM Corp. 1998

C.1 International Technical Support Organization Publications

For information on ordering these ITSO publications see "How to Get ITSO Redbooks" on page 383.

- *Programming with VisualAge for Java Version 2,* SG24-5264
- *VisualAge for Java Enterprise Version 2 Team Support,* SG24-5245
- *Using VisualAge for Java Enterprise Version 2 to Develop CORBA and EJB Applications*, SG24-5276
- *VisualAge for Java - RMI - Smalltalk, The ATM Sample from A to Z*, SG24-5418
- *Using VisualAge UML Designer,* SG24-4997
- *Programming with VisualAge for Java*, published by Prentice Hall, ISBN 0-13-911371-1, 1998
- *Application Development with VisualAge for Java Enterprise,* SG24-5081
- *Creating Java Applications with NetRexx*, SG24-2216
- *Unlimited Enterprise Access with Java and VisualAge Generator*, SG24-5246
- *From Client/Server to Network Computing, A Migration to Java*, SG24-2247
- *CBConnector Overview*, SG24-2022
- *CBConnector Cookbook Volume 1*, SG24-2033
- *Connecting the Enterprise to the Internet with MQSeries and VisualAge for Java*, SG24-2144
- *Internet Application Development with MQSeries and Java*, SG24-4896
- *Factoring JavaBeans in the Enterprise*, SG24-5051
- *JavaBeans by Example*: *Cooking with Beans in the Enterprise*, SG24-2035, published by Prentice Hall, 1997
- *World Wide Web Programming: VisualAge for C++ and Smalltalk*, published by Prentice Hall, ISBN 0-13-612466-6, 1997

C.2 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.

CD-ROM Title	Subscription Number	Collection Kit Number
System/390 Redbooks Collection	SBOF-7201	SK2T-2177
Networking and Systems Management Redbooks Collection	SBOF-7370	SK2T-6022
Transaction Processing and Data Management Redbook	SBOF-7240	SK2T-8038
Lotus Redbooks Collection	SBOF-6899	SK2T-8039
Tivoli Redbooks Collection	SBOF-6898	SK2T-8044
AS/400 Redbooks Collection	SBOF-7270	SK2T-2849
RS/6000 Redbooks Collection (HTML, BkMgr)	SBOF-7230	SK2T-8040
RS/6000 Redbooks Collection (PostScript)	SBOF-7205	SK2T-8041
RS/6000 Redbooks Collection (PDF Format)	SBOF-8700	SK2T-8043
Application Development Redbooks Collection	SBOF-7290	SK2T-8037

C.3 Other Publications

These publications are also relevant as further information sources:

❑ *Developing JavaBeans Using VisualAge for Java*, Dale Nilsson and Peter Jakab, published by John Wiley, ISBN 0-471-29788-7, 1998

❑ *Design Patterns: Elements of Reusable Object-Oriented Software*, Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, published by Addison-Wesley Professional Computing Series, ISBN 0-201-63361, 1995

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. A form for ordering books and CD-ROMs is also provided.

This information was current at the time of publication, but is continually subject to change. The latest information may be found at `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:

- **Redbooks Web Site on the World Wide Web**

  `http://w3.itso.ibm.com/`

- **PUBORDER – to order hardcopies in the United States**
- **Tools Disks**

  To get LIST3820s of redbooks, type one of the following commands:

  ```
  TOOLCAT REDPRINT
  TOOLS SENDTO EHONE4 TOOLS2 REDPRINT GET SG24xxxx PACKAGE
  TOOLS SENDTO CANVM2 TOOLS REDPRINT GET SG24xxxx PACKAGE (Canadian users only)
  ```

  To get BokkManager BOOKs of redbooks, type the following command:

  ```
  TOOLCAT REDBOOKS
  ```

  To get lists of redbooks, type the following command:

  ```
  TOOLS SENDTO USDIST MKTTOOLS MKTTOOLS GET ITSOCAT TXT
  ```

  To register for information on workshops, residencies, and redbooks, type the following command:

  ```
  TOOLS SENDTO WTSCPOK TOOLS ZDISK GET ITSOREGI 1998
  ```

- **REDBOOKS Category on INEWS**
- **Online – send orders to: USIB6FPL at IBMMAIL or DKIBMBSH at IBMMAIL**

Redpieces

For information so current it is still in the process of being written, look at "Redpieces" on the Redbooks Web Site (`http://www.redbooks.ibm.com/redpieces.html`). Redpieces are redbooks in progress; not all redbooks become redpieces, and sometimes just a few chapters will be published this way. The intent is to get the information out much quicker than the formal publishing process allows.

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 – send orders to:**

	IBMMAIL	**Internet**
In United States	usib6fpl at ibmmail	usib6fpl@ibmmail.com
In Canada	caibmbkz at ibmmail	lmannix@vnet.ibm.com
Outside North America	dkibmbsh at ibmmail	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) 4810-1020 - German
(+45) 4810-1420 - Dutch	(+45) 4810-1620 - Italian
(+45) 4810-1540 - English	(+45) 4810-1270 - Norwegian
(+45) 4810-1670 - Finnish	(+45) 4810-1120 - Spanish
(+45) 4810-1220 - French	(+45) 4810-1170 - Swedish

- **Mail Orders – send orders to:**

IBM Publications	IBM Publications	IBM Direct Services
Publications Customer Support	144-4th Avenue, S.W.	Sortemosevej 21
P.O. Box 29570	Calgary, Alberta T2P 3N5	DK-3450 Allerød
Raleigh, NC 27626-0570	Canada	Denmark
USA		

- **Fax – send orders to:**

United States (toll free)	1-800-445-9269
Canada	1-800-267-4455
Outside North America	(+45) 48 14 2207 (long distance charge)

- **1-800-IBM-4FAX (United States) or (+1) 408 256 5422 (Outside USA) – ask for:**

  Index # 4421 Abstracts of new redbooks
  Index # 4422 IBM redbooks
  Index # 4420 Redbooks for last six months

- **On the World Wide Web**

Redbooks Web Site	http://www.redbooks.ibm.com
IBM Direct Publications Catalog	http://www.elink.ibmlink.ibm.com/pbl/pbl

Redpieces

For information so current it is still in the process of being written, look at "Redpieces" on the Redbooks Web Site (`http://www.redbooks.ibm.com/redpieces.html`). Redpieces are redbooks in progress; not all redbooks become redpieces, and sometimes just a few chapters will be published this way. The intent is to get the information out much quicker than the formal publishing process allows.

IBM Redbook Order Form

Please send me the following:

Title	Order Numer	Quantity

First name | Last name

Company

Address

City | Postal code | Country

Telephone number | Telefax number | VAT number

☐ Invoice to customer number

☐ Credit card number

Credit card expiration date | Card issued to | Signature

We accept American Express, Diners, Eurocard, Master Card, and Visa. Payment by credit card not available in all countries. Signature mandatory for credit card payment.

List of Abbreviations

API	application programming interface
ASP	Active Server Pages
ATM	automated teller machine
AWT	Abstract Windowing Toolkit
BMS	basic mapping support
CB	Component Broker
CCF	Common Connector Framework
CGI	Common Gateway Interface
CORBA	Common Object Request Broker Architecture
DBMS	database management system
DLL	dynamic link library
DPL	distributed program link
ECI	external call interface
EPI	external presentation interface
GUI	graphical user interface
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
IBM	International Business Machines Corporation
IDE	integrated development environment
IDL	interface definition language
IIOP	Internet Inter-ORB Protocol
ITSO	International Technical Support Organization
JAR	Java archive
JDBC	Java Database Connectivity
JDK	Java Developer's Kit
JFC	Java Foundation Classes
JNI	Java Native Interface
JSDK	Java Servlet Development Kit
JVM	Java Virtual Machine
MFS	message format services
MQI	message queue interface
ODBC	Open Database Connectivity
PIN	personal identification number
RDBMS	relational database management system
RMI	Remote Method Invocation
SQL	structured query language
TCP/IP	Transmission Control Protocol/Internet Protocol
UOW	unit of work
URL	uniform resource locator
WWW	World Wide Web

© Copyright IBM Corp. 1998

Index

Numerics

3270 emulation 98

A

Abstract Windowing Toolkit 20
account
- access class 314
- CICS transaction 263, 270
- class 147
- command 266
- navigator 267
- panel 198
- record type 264
- request 308
- response 311
- selection 198, 227
- servlet 212
- table 131

Active Server Pages 48
administration
- CICS 365
- WebSphere 342

AIX 8
Apache 48, 338
applet 75, 102
- deployment 336

application
- controller
  - see controller
- deployment 334, 335
- layers 138

ATM application
- access classes 312
- applet 202
- business model 137
- CICS Connector 239
- COBOL programs 367
- data access beans 163
- database 130
- deployment 333
- flow 129, 221
- GUI 191
- high-performance compiler 347
- MQSeries 279, 291
  - objects 293
  - server 325
- panels 193
- remote debugger 354
- request 306
- requirements 128
- response 309
- run executable 350
- run GUI 204
- sample data 133
- servlet views 206
- servlets 205
- test CICS 272
- test MQSeries 330
- test servlet 235

ATM header 245
ATMApplet 202
ATMApplicationController 158
AtmCICS 241, 245
AtmDB 186, 223
AtmMQ 315
ATMPersistenceDefault 162
ATMPersistenceInterface 154, 241, 245, 291
ATMServletController 206, 223

B

bank account 140
basic mapping support 107
BeanInfo 72, 142, 143
BigDecimal 147
BLOB 43, 363
business model 140
business object 119

C

card
- access class 312
- CICS transaction 246, 261
- class 143
- command 250
- layout 192
- mapper 255
- navigator 252
- panel 195
- record type 248

© Copyright IBM Corp. 1998

request 308
response 310
servlet 206
table 131
CGI 48, 56, 338
channel
MQI 285
CICS
client 244
customization 96
programs 242
server 243
Telnet 99
CICS Connector 91, 93
ATM application 239
setup 365
CICS Transaction Gateway 93, 100, 242, 245
CICS Universal Client 93, 95
class
ATMApplet 202
ATMApplicationController 158
AtmCICS 245
AtmDB 186
AtmHeader 245
AtmMQ 315
ATMPersistenceDefault 162
AtmRequest 307
AtmResponse 309
ATMServletController 206
BankAccount 147
BigDecimal 147
Card 143
CheckingAccount 150
Customer 142
GenericServlet 52
HttpServlet 52
MQAccess 295
SavingsAccount 151
Transaction 146
class path 334, 335, 340, 343
ClearCase 5
COBOL
programs 243, 247, 263, 277, 324, 332, 366
record type 248
command 92, 111
editor 112, 113, 122, 250
mapper 257
COMMAREA 94, 98, 104, 109, 124, 245, 263
commit 148
Common Connector Framework 91, 92
Common Gateway Interface
see CGI
communication interface 93
compiler
options 346
Component Broker 7, 8
connection alias 25, 166
ConnectionSpec 93, 111
controller 139, 154, 193, 222, 315
events 156
implementation 158
interface 155
methods 156
servlet 205, 223
cookie 60, 65, 66, 82
counter 55, 71
currency
conversion 78
customer
class 142
table 130
verification 225

D

data access beans 4, 9
ATM application 164
business objects 165
high-performance compiler 348
setup 363
Data Access Builder 9, 11
window 22
database
access class 25
DB2 236, 363
Client Application Enabler 26
Java daemon 27, 204, 223, 236
JDBC 26, 363
DB2 Universal Database 10
db2jstrt 27, 204, 223, 236, 363
db2start 27, 236
DBNavigator bean 10, 15, 19
deployment 333
applet 336
application 335
servlet 338
Swing 344
deposit 148, 160

E
ECI 94, 98, 102
embed 75
Encina 6
Enterprise Access Builder 6, 91, 92
Enterprise JavaBeans 7
Enterprise Toolkit 6, 362
EPI 94, 99, 102
event 158
  aboutToGenerateOrTransfer 208
  componentShown 195
  executionSuccessful 253
  listener 144
events
  ATM business model 143
event-to-code 179, 181, 219
export 334, 336, 362

F
FormData 65
framework 139

G
gateway
  see CICS Transaction Gateway
GenericServlet 52, 53
get 57
gif 34
GUI 138

H
hidden field 81, 218
high-performance compiler 6, 345, 346
HTML
  button 62
  form 56, 65
  page 62
  result table 88
  table 62
HTTP
  session 49
HttpServlet 52, 57

I
IDL 7
installation
  redbook samples 373
  VisualAge for Java 362
integrated development environment 4
InteractionSpec 93, 111
Internet Explorer 344

J
JApplet 202
jar 340, 349
  Internet Explorer 344
  Netscape 344
  VisualAge for Java Access Builders 334
Java Development Kit
  see JDK
Java Foundation Classes
  see Swing
Java programming interface
  MQSeries 289
Java Record Framework 105
Java Record Library 91
Java Servlet Development Kit
  see JSDK
JavaScript 76, 212, 214
JDBC 10, 88, 338
  driver 26
  high-performance compiler 349
JDBUG 353
JDEBUG 353
JDK 4, 51, 334
JFC
  see Swing
JList 198, 200
join 32
JPort 6
JRMTDBG 357
JSDK 48, 51, 52
JTable 20, 45
JVM 334, 336

L
layout 192
list model 198
listener interface 143
Lotus Domino Go Webserver 48, 338, 339, 364
Lotus Notes 8

M
mapper 92, 113, 119, 255

builder 120
mapping
SQL data type 38
message
channel 283
MQSeries 280
receive 302
send 300
message format service 107
method
afterInternalExecution 261
bigdecToString 320
checkInputState 261
checkPin 159
closeQueue 300
connect 161
connectToQmgr 298
deposit 160
destroy 53
disconnect 161
disconnectFromQmgr 298
doGet 57
doPost 57
extGetAccounts 187, 270, 318
extGetCard 162, 186, 254, 318
extGetTransactions 187
extUpdateBalance 187
getAccounts 160
getCard 159
getTransactions 161
handleException 260
init 53
openQueue 299
putRequestMessage 300
requestFocus 195
retrieveSpecficMessage 302
returnExecutionSuccessful 254
service 54
toString 152
withdraw 160
MQI
channel 285
MQSeries
access bean 295
ATM application 292
Bindings for Java 288
CICS/ESA bridge 369
Client for Java 287
Java programming interface 289
objects 281
overview 280
persistence 315
server 325
setup 369
VisualAge for Java 294

N

navigator 92, 117, 252
advanced 273
test 276
Netscape 344

O

object model 141
ODBC 10

P

persistence
data access beans 164
interface 154, 186, 240, 315
layer 154, 162, 164
MQSeries 315
test 188
personal identification number
see PIN 128
PIN 128, 159, 165, 178
panel 196
servlet 209
verification 226
post 57, 58
promotion 194
property
ATM business model 141
cookieName 82
cookieValue 82
MQSeries 294
propertyValue 83
query 17, 166
ServerName 98
styles 75
PVCS 5

Q

query 17
queue
close 300

manager 281
MQSeries 280
open 299
queue manager
ATM application 292
start 294
Quick Start 11

R

record
custom 105
dynamic 105
generation 109
recovery 362
redbook
samples 373
remote debugger 6, 345, 353
repository 362
result set 14
RMI 6, 338
rollback 148
runmqsc 292

S

sample database 19
SanFrancisco 8
SAP R/3 6
Scrapbook script 152
business objects 152
CICS accounts transaction 271
CICS card transaction 261
CICS navigator 276
data access beans 188
MQSeries server 330
Select bean 10, 12, 21, 45, 88, 166
server
MQSeries 325
server-side
application 48
include 55, 68, 71
servlet 47
advanced 75
ATM application 205
branch 84
chaining 77
CICS Java gateway 102
class hierarchy 60
condition control 85
configuration 67
controller 205, 234
deployment 338
JDBC 88
parameter 56
persistence 189
router 85
terminal 101
test 70
transfer 67
visual 61
Servlet Builder 7, 47, 48, 51, 59, 62, 68, 364
Servlet Runner 59
session data 60, 65, 67, 82, 211
SHTML 54
SmartGuide
COBOL Record Type 108, 248
Create Servlet 52, 59, 69
Generate Records 109
New Event Listener 158
SQL Assist 10, 169
source code management 5
SQL Assist SmartGuide 10, 30
SQL specification 29
style sheet 75
submit button 57, 62
Swing 4, 15, 20, 41, 347
deployment 344

T

team programming 5
TeamConnection 5
Telnet 99
terminal servlet 101
tool integration 5
transaction
class 146
deposit 228
history 177, 230
panel 200
servlet 216
table 131
withdraw 229
TXSeries 243, 332, 365

U

unit of work 242, 305
URL 26, 50, 52, 363

user interface 41
    layer 138

V

Visual Composition Editor 4
    command 112
    command with mappers 123
    controller servlet 234
    navigator 117
visual servlet 60, 68
VisualAge for Java
    Enterprise 5
    installation 362
    MQSeries 294
    package 21
    products 10, 19
    Professional 4
    project 21
VisualServlet
    see visual servlet

W

Web
    browser 70, 336, 344
    server 49, 52, 336, 364
WebSphere 7, 48, 52, 235, 340, 341
where clause 34
withdrawal 148, 160
Workbench 334
workspace 362

ITSO Redbook Evaluation

VisualAge for Java Enterprise Version 2: Data Access Beans - Servlets - CICS Connector
SG24-5265-00

Your feedback is very important to help us maintain the quality of ITSO redbooks. **Please complete this questionnaire and return it using one of the following methods:**

- Use the online evaluation form found at http://www.redbooks.ibm.com
- Fax this form to: USA International Access Code + 1 914 432 8264
- Send your comments in an Internet note to redbook@us.ibm.com

Which of the following best describes you?
_ Customer _ Business Partner _ Solution Developer _ IBM employee
_ None of the above

Please rate your overall satisfaction with this book using the scale:
(1 = very good, 2 = good, 3 = average, 4 = poor, 5 = very poor)

Overall Satisfaction __________

Please answer the following questions:

Was this redbook published in time for your needs? Yes___ No___

If no, please explain:

____

____

____

____

What other redbooks would you like to see published?

____

____

____

Comments/Suggestions: (THANK YOU FOR YOUR FEEDBACK!)

____

____

____

____

© Copyright IBM Corp. 1998

SG24-5265-00
Printed in the U.S.A.

VisualAge for Java Enterprise Version 2: Data Access Beans - Servlets - CICS Connector

SG24-5265-00