VisualAge Java - RMI - Smalltalk
The ATM Sample from A to Z

Markus Muetschard, D. Allen Smith

International Technical Support Organization
http://www.redbooks.ibm.com

SG24-5418-00
## Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Figures</td>
<td>ix</td>
</tr>
<tr>
<td>Tables</td>
<td>xiii</td>
</tr>
<tr>
<td>Preface</td>
<td>xv</td>
</tr>
<tr>
<td>The Team That Wrote This Redbook</td>
<td>xv</td>
</tr>
<tr>
<td>Comments Welcome</td>
<td>xvi</td>
</tr>
</tbody>
</table>

### Part 1. Introducing the ATM Sample

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chapter 1. ATM Application</td>
<td>3</td>
</tr>
<tr>
<td>1.1 Application Dialog Flow</td>
<td>3</td>
</tr>
<tr>
<td>1.2 Object Model</td>
<td>5</td>
</tr>
<tr>
<td>1.3 Data</td>
<td>6</td>
</tr>
<tr>
<td>1.4 The Development Approach</td>
<td>6</td>
</tr>
<tr>
<td>Chapter 2. Some Concepts</td>
<td>7</td>
</tr>
<tr>
<td>2.1 RMI</td>
<td>7</td>
</tr>
<tr>
<td>2.2 SST RMI Concepts and Components</td>
<td>8</td>
</tr>
<tr>
<td>2.2.1 Object Spaces</td>
<td>8</td>
</tr>
<tr>
<td>2.2.2 Application Contexts</td>
<td>8</td>
</tr>
<tr>
<td>2.2.3 Uniform Resource Locators (URLs)</td>
<td>8</td>
</tr>
<tr>
<td>2.2.4 Configurations</td>
<td>8</td>
</tr>
<tr>
<td>2.2.5 Java Stub and Interface Classes</td>
<td>8</td>
</tr>
<tr>
<td>2.2.6 Java Type Information</td>
<td>9</td>
</tr>
<tr>
<td>2.2.7 Required Methods for Smalltalk RMI Objects</td>
<td>10</td>
</tr>
<tr>
<td>Chapter 3. ATM Application Design</td>
<td>11</td>
</tr>
<tr>
<td>3.1 Key Design Features</td>
<td>11</td>
</tr>
<tr>
<td>3.2 Object Distribution, Communication, and Life Cycle</td>
<td>12</td>
</tr>
<tr>
<td>3.3 Withdrow Transaction Detail</td>
<td>14</td>
</tr>
<tr>
<td>Chapter 4. Server and Business Objects</td>
<td>17</td>
</tr>
<tr>
<td>4.1 ATMServer</td>
<td>17</td>
</tr>
<tr>
<td>4.2 ATMCard</td>
<td>18</td>
</tr>
<tr>
<td>4.3 BankAccount</td>
<td>18</td>
</tr>
<tr>
<td>4.4 Customer</td>
<td>19</td>
</tr>
<tr>
<td>4.5 CardValidationException</td>
<td>19</td>
</tr>
<tr>
<td>4.6 AccountOverdrawedException</td>
<td>19</td>
</tr>
</tbody>
</table>
Appendix A. Special Notices ........................................... 189

Appendix B. Related Publications ........................................... 191
B.1 International Technical Support Organization Publications ........ 191
B.2 Redbooks on CD-ROMs .............................................. 191
B.3 Other Publications .................................................... 191

How to Get ITSO Redbooks ............................................... 193
How IBM Employees Can Get ITSO Redbooks ......................... 193
How Customers Can Get ITSO Redbooks ................................ 194
IBM Redbook Order Form ................................................. 195

List of Abbreviations ...................................................... 197

Index ............................................................................ 199

ITSO Redbook Evaluation ................................................. 205
## Figures

1. Application Dialog Flow .................................................. 4
2. ATM Application Class Diagram ........................................ 5
3. Object Distribution, Communication, and Life Cycle Model ........ 13
4. Withdrawal Transaction Detail ........................................... 14
5. CardPanel ................................................................. 21
6. SelectionPanel ............................................................ 22
7. TransactionPanel .......................................................... 23
8. ContinuationPanel .......................................................... 24
9. ATMCard sstRmiClassName Method ..................................... 30
10. ATMCard sstIsRmiSerializable Method ................................. 30
11. Code for Creating ATMCard Java Type Information ................. 31
12. Code for Loading ATMCard Java Type Information into Smalltalk ..................................................................... 31
13. RbiRmiSamplesAtm rbi_rmi_samples_atm_ATMCard Method ........ 31
14. Modified rbi_rmi_samples_atm_ATMCard Method ..................... 32
15. rbi.rmi.samples.atm.ATMServerIf Class Definition .................. 35
16. Code for Creating ATMServer and ATMServer_Stub Java Type Information .......................................................... 36
17. Code for Loading ATMServer and ATMServer_Stub Java Type Information .......................................................... 36
18. RbiRmiSamplesAtm rbi_rmi_samples_atm_ATMServer Method .... 37
19. RbiRmiSamplesAtm rbi_rmi_samples_atm_ATMServer_Stub Method .......................................................... 37
20. Modified rbi_rmi_samples_atm_ATMServer Method ................ 38
21. Modified rbi_rmi_samples_atm_ATMServer Method ................ 39
23. RbiRmiSamplesAtm rbi_rmi_samples_atm_BankAccount Method ... 42
24. Modified rbi_rmi_samples_atm_BankAccount Method .......... 42
26. Code for Creating AccountOverdrawnException Java Type Information .......................................................... 45
27. Code for Loading AccountOverdrawnException Java Type Information into Smalltalk .......................................................... 45
28. Generated RbiRmiSamplesAtm Exception Methods .................. 46
29. Modified RbiRmiSamplesAtm Exception Methods .................. 47
30. AMTServer setupServer:namingPort: Class Method ............... 49
31. Code to Register the Invocation Scheme and Handler ............. 50
32. Code to Create and Set Up the ATMServerRmi Application Context .......................................................... 51
33. Code to Start the ATMServerRmi Application Context ............ 52
34. Code to Create the RMI Naming Application Context .......... 53
35. ATMServer Instantiation and RMI Registration ...................... 54
36. ATMServer Application Launch Code .................................... 55
37. Code to Stop the ATMServer .............................................. 55
38. ATMServer Common Class Methods ..................................... 56
39. ATMServer Unique Class Methods ...................................... 57
40. ATMServer Instance Categories and Methods ....................... 59
41. ATMServer defaultCards Getter and Setter Methods ................................. 61
42. ATMServer defaultAccounts Getter and Setter Method ........................... 62
43. ATMServer registeredBankAccounts Getter and Setter Methods ............... 63
44. ATMServer verifyAndGetAccountNumbers: Method ............................... 63
45. ATMServer getBankAccount: Method .................................................... 64
46. ATMServer Methods to unregister BankAccount Objects .......................... 65
47. ATMServer restartAtm Method .............................................................. 65
48. ATMServer shutDown Method ............................................................... 66
49. ATMServer Basic SST RMI Methods ...................................................... 66
50. BankAccount Instance Methods ............................................................ 67
51. BankAccount accountNumber Getter and Setter Methods ....................... 68
52. BankAccount accountType Getter and Setter Methods ............................ 69
53. BankAccount balance Getter and Setter Methods ................................... 69
54. BankAccount cardNumber Getter and Setter Methods ............................ 69
55. BankAccount customerIld Getter and Setter Methods ............................. 70
56. BankAccount getBalance Method ......................................................... 70
57. BankAccount withdraw: Method ............................................................. 71
58. BankAccount Basic SST RMI Methods .................................................... 71
59. ATMCard fieldCardNumber Getter and Setter Methods ......................... 73
60. ATMCard fieldPin Getter and Setter Methods ....................................... 73
61. ATMCard Basic SST RMI Methods ........................................................ 73
62. RMI Naming Service Registry Check ..................................................... 75
63. Inspecting RMI Naming Service Registry Contents ................................... 75
64. ATMServer Test Script ....................................................................... 76
65. ATMServerRmi Context Local Object Space Export Set ........................... 77
66. Timer addTimeout:receiver:selector:clientData: Method ....................... 80
67. Timer removeTimeout: Method ............................................................. 80
68. ATMServer Timer Related Method ......................................................... 81
69. RbiRmiSamplesAtm packagingRulesFor: Class Method ......................... 84
70. ATMServerProxy atmServer Getter and Setter Methods ......................... 91
71. ATMServerProxy accountNumbers Getter and Setter Methods ............... 92
72. ATMServerProxy bankAccountProxy Getter and Setter Methods ............ 93
73. ATMServerProxy hostName Getter and Setter Methods .......................... 94
74. ATMServerProxy registeredAccounts Getter and Setter Methods .......... 94
75. ATMServerProxy namingServicePort Getter and Setter Methods ............ 95
76. ATMServerProxy lastError Getter and Setter Methods .......................... 96
77. ATMServerProxy verifyCardAndGetAccountNumbers(ATMCard) Method ... 98
78. ATMServerProxy getBankAccount(String) Method ................................ 99
79. ATMServerProxy unregisterAllBankAccounts() Method ....................... 100
80. ATMServerProxy restartAtm() Method ............................................... 100
81. ATMServerProxy getUrl() Method ....................................................... 101
82. ATMServerProxy Features .................................................................. 102
83. BankAccountProxy Features ............................................................... 105
## Tables

1. ATMServer Class Description ........................................ 18
2. ATMCard Class Description. ....................................... 18
3. BankAccount Class Description. ................................. 19
4. Customer Class Description ...................................... 19
5. ATMServerProxy Class Description. ............................. 25
6. BankAccountProxy Class Description ............................ 26
7. Customer Table ...................................................... 160
8. Card Table .......................................................... 160
9. BankAccount Table .................................................. 160
Preface

Remote Method Invocation (RMI) is a great enabler for distributed objects on the Internet and intranet. RMI is also well suited for enabling communication between Java and Smalltalk. Client software—either applets or servlets—can easily talk to Smalltalk servers and vice versa. The RMI implementation in Server Smalltalk (SST) makes legacy VisualAge Smalltalk Enterprise the server of choice for Internet and intranet business.

VisualAge Smalltalk Enterprise enables fast and reliable application development and delivery for workstations and servers. Java, widely used for servlets on Web servers and for downloaded applets at runtime, is the perfect client-oriented component for e-Business applications distributed world wide. As the bridge RMI gives you the choice of combining the strength of both components, enables interoperability between your Java and Smalltalk world, and lets you individually choose the appropriate tool for your business applications.

RMI, which became available in Smalltalk with Version 4.5, enables you to connect in an object-oriented way—and on a relatively high level—client and server objects. RMI is practically transparent for application developers, makes it easy to let the objects live where their “gravity” pounds, and is much easier to setup than a full-fledged object request broker (ORB).

The chapters of this book describe the steps of the ATM sample application development from A to Z: from creating the business model to developing the user interface and coding details in VisualAge for Java and VisualAge Smalltalk Enterprise. The hands-on exercised in this book will make you a VisualAge for Java-RMI-VisualAge Smalltalk Server developer, ready to accept today’s application development challenges for the Web.

The code for the ATM sample application is available as SG245418.zip on:

ftp://www.redbooks.ibm.com/redbooks/SG245418

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.

Markus Muetschard is an IT and Object-Oriented Consultant at the IBM International Technical Support Organization, San Jose Center. He writes
extensively and teaches IBM classes worldwide on all areas of object-oriented technology, with a focus on the VisualAge (Smalltalk) product family. Before joining the ITSO, Markus worked in the IT Consulting department of IBM Switzerland, consulting and mentoring for object-oriented customer projects.

D. Allen Smith works in the VisualAge Smalltalk Support group in Research Triangle Park, NC. He graduated from Wake Forest University with a Bachelor of Science degree in Math-Business. Allen joined IBM in 1984 as an application developer for ISPF and has held a number of positions in software development and marketing. He has been a member of the VisualAge Smalltalk Support group since 1994.

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

Charles R. Trip, Manager Customer Support, Research Triangle Park, NC.

Dan Berg, Chuck Bridgham, Harvey Gunther, Dave Salkeld, and Joe Winchester from the VisualAge development and services, Research Triangle Park, NC.

Maggie Cutler, Joan Bow, Pat McCarthy, Jean Gedeon, Ueli Wahli, and Laymond M. Pon of the International Technical Support Organization, San Jose Center.

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 205 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. Introducing the ATM Sample
Chapter 1. ATM Application

In this chapter we introduce the automatic teller machine (ATM) application. The purpose of the ATM application is to show the interoperability between VisualAge Java and Smalltalk using Remote Method Invocation (RMI). As such, we focus primarily on the communication and distribution issues. The application’s business logic is minimal.

1.1 Application Dialog Flow

Figure 1 illustrates the dialog flow of the ATM application. Like a real ATM, the ATM application begins by requiring a user (customer) to insert his or her ATM card. Insertion of the card is simulated by the user entering a card number and personal identification number (PIN).

The ATM card is verified by checking the card number and PIN. After card verification, the user is presented with a list of accessible accounts and a transaction menu. The user chooses an account and a transaction. To reduce the size of the application, we implemented only the withdraw transaction. Extending the application, however, to include other transactions would be relatively simple.

After choosing an account and a transaction, the user is presented with information about the account, including the account number, account type, and current balance. The user can then enter a withdraw amount and proceed with the transaction.

The withdraw transaction is processed and the results displayed to the user. If the withdraw results in a negative balance, the user is notified that the account is overdrawn.

After the transaction completes, the user can exit the ATM application or perform another transaction.

Note: The user interface of the sample code has two additional panels, (WelcomePanel and ExitPanel) and calls the ContinuationPanel shown in Figure 1 the Confirmation Panel (see Chapter 14, “Building the User Interface” on page 113).
Verify ATM card. If verified: get account numbers accessible by ATMCard object.

Get BankAccount for selected Account Number (BankAccount object is remote).

Perform the Withdraw Transaction on the remote BankAccount object (RMI).

Account Number
Account Type
Balance
Amount to Withdraw

OK CANCEL

ANOTHER TRANSACTION Exit

Figure 1. Application Dialog Flow
1.2 Object Model

Figure 2 shows the high-level class diagram of the ATM application object model.

![ATM Application Class Diagram](image)

We created the diagram with VisualAge UML Designer, IBM's object-oriented modeling tool. We started with defining use cases, and finding things and responsibilities. We generated protocols and the class design, did the diagramming, exported the diagram as a .gif file, and embedded it in this book.

Note: The ATM sample code does not implement the Transaction classes.
1.3 Data

The first iteration of the ATM application does not access any external data. It uses a set of default *ATMCard, BankAccount*, and *Customer* objects, eliminating the requirement for an external data source, such as a database.

1.4 The Development Approach

We will take an iterative approach to building our ATM application. In other words, we will build the application in these stages, testing each along the way, making modifications to the various components as necessary:

- Understanding the concept:
  - Remote Method Invocation (RMI) concepts
  - Server Smalltalk (SST) concepts
- Looking at the ATM application design
- Building and testing the server in SST
- Building the client and testing it with the server
- Adding persistence to the server
Chapter 2. Some Concepts

It is essential that you understand RMI before you consider the next steps: overall application design; client, server, and object layout; object distribution; and object life cycle. Therefore we recommend that you read 2.1, “RMI” on page 7 before proceeding further. You can postpone reading 2.2, “SST RMI Concepts and Components” on page 8, which explains how SST implements RMI and object distribution until you get to Part 2, “Building the Server and Part of the Client” on page 27, where we will actually start implementing the components in SST.

2.1 RMI

RMI allows Java objects running on different virtual machines to communicate with one another. Remote objects implement a remote interface that defines which methods a client can invoke. Clients can send any message defined in this interface to the remote object, and the Java RMI handles the routing of the message sent under the covers. To the client, the object appears local. This type of message sending is often referred to as by-reference.

In addition to remote references, you can also pass copies of an object from one machine to another. This is known as by-value. Passing an object by-value requires that the object be converted to a byte stream before it is passed and then converted from the byte stream back to an object after it is received. In Java this is known as object serialization. In SST this is known as by-value marshaling.

The java.rmi.Naming class provides a naming service registry that maps a name to a remote object. Each entry in the registry has a reference to a remote object and a name associated with the reference. Java clients get the references to remote objects by performing a registry lookup, using the object’s name.

We discuss RMI at this time because it has a direct bearing on where we define our business objects. Some objects are defined in Smalltalk for remote message sends from Java to Smalltalk and some are defined in both Java and SST for a by-value passing of the object.
2.2 SST RMI Concepts and Components

For in-depth information about these and other SST concepts and components, refer to the VisualAge Smalltalk Server Guide.

2.2.1 Object Spaces

In a nondistributed environment, all Smalltalk objects exist in the same image and are local to that image. In a distributed environment, you have both local and remote objects. Object spaces house objects. They assign unique identifiers to objects and expose the identifiers, allowing objects in other object spaces to reference the objects.

The two main types of object spaces are full object spaces and import and export sets. A full object space, an instance of SstLocalSpace, has an import set and an export set. The import set contains proxies for remote objects, and the export set contains a list of exported objects.

2.2.2 Application Contexts

Application contexts manage SST resources for an application. They provide discrete operating environments for different applications in the same image. An application context has a local object space and one or more remote spaces.

2.2.3 Uniform Resource Locators (URLs)

Uniform Resource Locators (URLs) provide addressability to object spaces. They are used to build endpoint objects. An endpoint specifies a location where messages can be sent for processing.

2.2.4 Configurations

SST provides a number of predefined configurations that specify how SST components are created and behave. There are configurations for object spaces, invocation handlers, transports, and other components. Usually you only need to choose the configuration that meets the requirements of the distributed application design rather than creating or modifying your own.

2.2.5 Java Stub and Interface Classes

In a Java RMI-only environment, remote messaging is implemented using stubs and skeletons. The client talks to a stub, which passes the message along to the underlying remote reference and transport layers. On the server side, the underlying remote reference and transport layers pass the message
to a skeleton, which sends the message to the remote object. The stub and skeleton are generated by running the Remote Method Invocation Compiler (RMIC) on the class to which remote messages will be sent.

Our design calls for the client to be written in Java and the server in Smalltalk. The ATMServer and BankAccount classes are defined in Smalltalk. As such, there is no requirement for these classes to exist in Java because they will never be instantiated. From a Java client perspective, this is the same case in the RMI-only environment in that the client talks to a stub. The only classes needed in Java are the stub and interface classes for the ATMServer and BankAccount classes. There are two options for creating these classes.

- In Java, you first create interfaces for the ATMServer and BankAccount classes. Next, you create the ATMServer and BankAccount classes that implement the previously created interfaces. You then run the RMIC on the ATMServer and BankAccount classes, producing the required stub classes. Using this approach, you end up with classes that are not needed. Specifically, you do not need the ATMServer, BankAccount, ATMServer_Skel, and BankAccount_Skel classes.

- In Smalltalk, you can use the RMI type builder to generate the stub and interface classes from dynamically created Java class definitions.

2.2.6 Java Type Information

Java is a typed language. In order for Smalltalk and Java to communicate, Smalltalk has to have access to certain Java type information. This requires mapping a Java class and its methods to Smalltalk equivalents. We use the following approach for generating the Java type information and loading it into a Smalltalk:

- From Java, the generateFor(fileName, classNames) method in com.ibm.sst.JavaTypeRepository can be used to produce a file containing the Java-to-Smalltalk mapping information required by SST.

  Note: The com.ibm.sst.JavaTypeRepository package comes with SST and must be imported into VisualAge for Java. After you produce the type definition files in Java, you can load the information into a Smalltalk application, using the SstRmiTypeBuilder>>#mergeDefinitionsFrom: method.

The Java type information is loaded into a Java type repository when an application context is set up.
2.2.7 Required Methods for Smalltalk RMI Objects

In addition to the Java type information, Smalltalk classes that interact with Java must implement certain methods. All classes that interact with Java must implement the following instance method:

- **sstRmiClassName** - This method answers the Java class name to which the class maps.

Objects that are passed to or from Java by-value (serialized) must implement the following method, which answers true:

- **sstIsSerializable**

Remote objects (by-reference) must implement the following method, which answers true:

- **sstIsRemoteable**

Objects that are passed as an array must implement the following method, which answers true:

- **sstIsArray**
Chapter 3. ATM Application Design

We have discussed at a high level how the ATM application operates and have reviewed the RMI and SST concepts. Now it is time to look at the application design.

Two important factors must be considered when you design the ATM application:
1. Part of the application will be written in Smalltalk and part in Java
2. The application will be distributed

The typical distribution model separates an application into three components:
- User interface
- Application or business logic
- Data access

Given today’s popularity of the World Wide Web and the Java programming language, it seems logical that the ATM user interface should be built as a Java applet (or servlet) to run in a Web browser. So that is our first design decision: we will build the user interface component of the application in Java. We will refer to this part of the ATM application as the *ATM client*.

SST facilitates the development and deployment of server applications. The SST feature of VisualAge Smalltalk Enterprise provides RMI support that allows Java and Smalltalk applications to communicate with one another through RMI. As Java is a logical choice for the user interface component, so too is Smalltalk for building the server component of the ATM application. This server component will implement the business logic and data access functions of the ATM application. So, our second design decision is that the business logic and data access components will be written in Smalltalk. We will refer to this part of the ATM application as the *ATM server*.

### 3.1 Key Design Features

Key design features of the ATM application are:
- Remote objects are registered in Smalltalk through a Java-like naming service that responds to the standard `java.rmi.Naming` protocols. Objects are registered by binding the object to a name. A registry keeps track of all bound objects and the names by which they can be requested.
• The ATMServer class is a singleton. It is instantiated and registered (added to the RMI naming service registry) only once when the server starts. The ATMServer object is bound to its class name. Conceptually, the server is always running and available to the client.

• BankAccount objects are dynamically instantiated and registered on request. BankAccount objects are bound to their account number. They are removed from the registry and the ATMServer export set before a new PIN and card number are entered.

• The balance for a BankAccount object is stored as a ScaledDecimal. The balance must be converted to a String before it is returned to the Java client.

• The withdraw amount is passed as a String by the Java client.

• The ATMServer singleton and BankAccounts objects are remote to the Java ATM client. Message sends to the ATMServer and BankAccounts are by-reference.

• ATMCard instances are passed by-value.

• The Java client uses the lookup() method in java.rmi.Naming to obtain remote references to the ATMServer singleton and BankAccount objects.

3.2 Object Distribution, Communication, and Life Cycle

Figure 3 is an object interaction diagram showing the object distribution, communication, and life cycle model. The life lines (filled vertical bars) reflect the life period of the object over time. Time starts on top.
When it comes to the technical transaction (persistence and consistence), the circled area in Figure 3 is neither detailed nor accurate enough.
3.3 Withdraw Transaction Detail

Figure 4 shows the object communication and life cycle model for the withdraw transaction on the server. It complements the required details Figure 3, but it also raises questions about the approach.

The question regarding the withdraw transaction is: Should the business model implementation include the technical transaction and persistence management? or simply in other words: Should the business model know about the technical transaction and persistence management?

RMI and VisualAge Java Persistence Builder or VisualAge Smalltalk ObjectExtender easily coexist, but—or because—they do not care about
each other, at least not in the current implementation. So the business application design and implementation has to care about them as a mediator. To escape this dilemma and to enable clients to have control over a single RMI in one logical unit of work, some controlling methods must be added to the business objects such as—in the ATM application—the withdrawAsTransaction: added to the BankAccount.

With multiple RMIs in one logical unit of work, the withdraw:inTransaction: method has to be added to the BankAccount, and the beginChildOfTransaction:, commitTransaction:, and rollbackTransaction: have to be added to the (ATM)Server. Further more the client must keep track of the transaction in which it is involved and pass the handle (or name) with every RMI, for example:

    anAccount withdraw: anAmount inTransaction: aTransactionName.

The transaction name is a server-generated handle, for example, a timestamp, and it is returned to the client upon the RMI:

    beginChildOfTransaction: aTransactionName.

For the beginning of a TopLevelTransaction, the RMI argument, aTransactionName, has to be nil.

With multiple RMIs in one logical unit of work, the server has to time out and garbage collect the dangling transactions and throw an exception for a request with an invalid or no longer valid transaction handle. A similar timeout procedure has to be applied to registered and therefore dangling business objects (see Chapter 11., “Garbage Collection of Registered Objects” on page 79).

For more information about persistence and transactions see the ObjectExtender User’s Guide and Reference.
Chapter 4. Server and Business Objects

The ATM application has the following business object classes:

- ATMServer
- ATMCard
- BankAccount
- Customer
- CardValidationException
- AccountOverdrawnException

4.1 ATMServer

The ATMServer class (Table 1) is defined in Smalltalk. It is a singleton class (A class is a singleton class when only one instance of the class will ever exist—the singleton). On startup, it starts an RMI naming service and adds an instance of itself to the naming service registry. The Java client gains access to the ATMServer by looking up its remote reference in the registry. Once the reference is obtained, the Java client can send messages to the ATMServer singleton.

The ATMServer validates an ATMCard and returns a list of account numbers accessible by the card. If card validation fails, a CardValidationException is thrown. Once the user selects an account, the Java client sends a message to the ATMServer to retrieve the account for processing. The ATMServer retrieves the account and adds the BankAccount object to the RMI naming service registry, making it available to the Java client.

BankAccount objects remain registered for the duration of processing against an ATMCard. In other words, if an ATMCard accesses multiple accounts and the user performs transactions against all the accounts, the associated BankAccount objects will remain instantiated, exported, and registered until a new ATM card is "entered" or the application is exited and the ATMServer unregisters the BankAccount objects. The stem garbage collection removes then the no longer referenced instances.
Table 1. ATMServer Class Description

<table>
<thead>
<tr>
<th>Property</th>
<th>Type</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>registeredBankAccounts</td>
<td>Dictionary</td>
<td>Holds all BankAccounts added to the naming service registry</td>
</tr>
<tr>
<td>defaultCards</td>
<td>Dictionary</td>
<td>Holds the default ATMCards used by the ATM application</td>
</tr>
<tr>
<td>defaultAccounts</td>
<td>Dictionary</td>
<td>Holds the default BankAccounts used by the ATM application</td>
</tr>
</tbody>
</table>

4.2 ATMCard

The ATMCard class (Table 2) is defined in both Java and Smalltalk. It is instantiated in Java and is passed by-value to the ATMServer singleton in Smalltalk for validation.

Table 2. ATMCard Class Description

<table>
<thead>
<tr>
<th>Property</th>
<th>Type</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>fieldCardNumber</td>
<td>String</td>
<td></td>
</tr>
<tr>
<td>fieldPin</td>
<td>String</td>
<td></td>
</tr>
</tbody>
</table>

4.3 BankAccount

The BankAccount class (Table 3) is defined in Smalltalk. The ATMServer singleton retrieves instances of this class and adds them to the RMI naming service registry. The Java ATM client looks up the remote reference to the BankAccount object, using the account number.

All account transactions are performed by BankAccount objects. If a withdrawal results in a negative balance, an AccountOverdrawnException is thrown.

The balance is stored as a ScaledDecimal. The ScaledDecimal class does not map to the java.math.BigDecimal Java class. As a result, the balance
must be converted to a *String* before it is returned to the Java client. In addition, the Java ATM client passes the withdraw amount as a *String*.

Table 3. BankAccount Class Description

<table>
<thead>
<tr>
<th>Property</th>
<th>Type</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>accountNumber</td>
<td>String</td>
<td></td>
</tr>
<tr>
<td>accountType</td>
<td>String</td>
<td></td>
</tr>
<tr>
<td>balance</td>
<td>ScaledDecimal</td>
<td>Returned to the Java ATM client as a String</td>
</tr>
<tr>
<td>customerId</td>
<td>String</td>
<td></td>
</tr>
<tr>
<td>cardNumber</td>
<td>String</td>
<td></td>
</tr>
</tbody>
</table>

4.4 Customer

The *Customer* class (Table 4) is defined in Smalltalk. The *Customer* class is used when persistence is added.

Table 4. Customer Class Description

<table>
<thead>
<tr>
<th>Property</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>customerId</td>
<td>String</td>
</tr>
<tr>
<td>firstName</td>
<td>String</td>
</tr>
<tr>
<td>lastName</td>
<td>String</td>
</tr>
</tbody>
</table>

4.5 CardValidationException

The *CardValidationException* class is defined in Java as a subclass of *java.rmi.RemoteException*. It is thrown in Smalltalk by the *ATMserver* singleton if an *ATMCard* contains an invalid PIN or card number. It is caught by the *ATMServerProxy* object in Java.

4.6 AccountOverdrawnException

The *AccountOverdrawnException* class is defined in Java as a subclass of *java.rmi.RemoteException*. It is thrown in Smalltalk by a *BankAccount* object if a withdraw results in a negative balance. It is caught by the *BankAccountProxy* object in Java.
Chapter 5. User Interface

The user interface is kept simple and is designed to closely represent a standard ATM. In our application, the user is required to enter a PIN and card number. In reality the card would be read by the card reader. In this chapter we present the main panels of the ATM application. We cover how they are built and their attendant connections later in Part 3, “Building the Client” on page 87, Chapter 14, “Building the User Interface” on page 113, and Chapter 15, “Putting the Pieces Together” on page 135.

5.1 CardPanel

The CardPanel shown in Figure 5 is where the user enters a PIN and card number.

![CardPanel](image)

*Figure 5. CardPanel*
5.2 SelectionPanel

The SelectionPanel shown in Figure 6 presents the user with a list of accessible accounts. The user selects an account and a transaction to be performed against the account.

**Figure 6. SelectionPanel**
5.3 TransactionPanel

The TransactionPanel shown in Figure 7 displays information about the selected account. On this panel the user can enter amount of money to be withdrawn from the account.

![TransactionPanel](image)

**Figure 7. TransactionPanel**
5.4 ContinuationPanel

The ContinuationPanel shown in Figure 8 displays the results of a transaction. It presents the user with the option of performing another transaction or exiting the ATM application.

![ContinuationPanel](image)

Account Number: 172453
Savings
Current Balance: $14945.13

[Another Transaction] [Exit]

Figure 8. ContinuationPanel
Chapter 6. Proxy Objects

A proxy object acts as a substitute for its remote counterpart, allowing the client to talk to the object as if it were a local object. RMI also allows a client to talk to a remote object as if it were a local object. If this is the case, then why create proxy objects? Proxies are created for the remote business objects in Smalltalk because additional functionality is required on the Java end to assist in the communication between Java and Smalltalk and to control the flow of the application. We do not want to add this specialized behavior to our business objects, so we create proxies and add the behavior to the proxies. The proxies contain remote references to their Smalltalk counterparts. Java RMI uses these remote references to perform remote message sends.

The ATM client has the following proxy object classes:

• ATMServerProxy
• BankAccountProxy

6.1 ATMServerProxy

The ATMServerProxy class (Table 5) is defined in Java and contains a remote reference to the ATMServer singleton instantiated in Smalltalk. Java ATM client objects send messages to the proxy, which uses RMI to perform a remote message send to the ATMServer singleton in Smalltalk.

<table>
<thead>
<tr>
<th>Property</th>
<th>Type</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>atmServer</td>
<td>ATMServerIf</td>
<td>Interface</td>
</tr>
<tr>
<td>fieldAccountNumbers</td>
<td>Vector</td>
<td></td>
</tr>
<tr>
<td>fieldBankAccountProxy</td>
<td>BankAccountIf</td>
<td>Interface</td>
</tr>
<tr>
<td>fieldHostName</td>
<td>String</td>
<td></td>
</tr>
<tr>
<td>registeredAccounts</td>
<td>Hashtable</td>
<td></td>
</tr>
<tr>
<td>fieldNamingServicePort</td>
<td>String</td>
<td></td>
</tr>
<tr>
<td>fieldLastError</td>
<td>String</td>
<td></td>
</tr>
</tbody>
</table>
6.2 BankAccountProxy

The BankAccountProxy class (Table 6) is defined in Java and contains a remote reference to a BankAccount object instantiated in Smalltalk. Java ATM client objects send messages to the proxy, which uses RMI to perform a remote message send to the BankAccount object in Smalltalk.

Table 6. BankAccountProxy Class Description

<table>
<thead>
<tr>
<th>Property</th>
<th>Type</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>fieldAccountNumber</td>
<td>String</td>
<td></td>
</tr>
<tr>
<td>fieldAccountType</td>
<td>String</td>
<td></td>
</tr>
<tr>
<td>fieldBalance</td>
<td>String</td>
<td></td>
</tr>
<tr>
<td>fieldCustomerId</td>
<td>String</td>
<td></td>
</tr>
<tr>
<td>bankAccount</td>
<td>BankAccountIf</td>
<td>Interface</td>
</tr>
<tr>
<td>fieldHostName</td>
<td>String</td>
<td></td>
</tr>
<tr>
<td>atmServer</td>
<td>ATMServerProxy</td>
<td>Proxy</td>
</tr>
</tbody>
</table>
Part 2. Building the Server and Part of the Client
Chapter 7. Creating and Mapping the Server and Server Objects

Constructing the ATM application requires that parts of the client and server be built in tandem. Specifically, we have to:

- Create the \textit{ATMC\textit{Card}} class in Java and in Smalltalk and map the Java type information for the \textit{ATMC\textit{Card}} class into Smalltalk.
- Create the ATMServer and BankAccount classes in Smalltalk, create their associated interface and stub classes in Java, and map the Java type information for \textit{ATMS\textit{erver}} and \textit{Bank\textit{Account}} and their stub classes into Smalltalk.

Before creating any classes, we have to create a Smalltalk application and a Java package to hold the classes. In Smalltalk, create an application called \textit{RbiRmiSamplesAtm}. In Java, add a project called \textit{IBM Java RMI and Smalltalk SST Interoperability}. Add a package called \textit{rbi.rmi.samples.atm} to the project.

\section*{7.1 Creating and Mapping the ATMC\textit{Card} Class}

Our design states that \textit{ATMC\textit{ard}} objects are passed from the Java ATM client to the \textit{ATMS\textit{erver}} singleton by-value. In other words, a copy of the object is made, and the copy is passed. Therefore the \textit{ATMC\textit{ard}} class must be defined in both Java and Smalltalk. The \textit{ATMC\textit{ard}} has the following fields (instance variables):

- fieldCardNumber
- fieldPin

Follow these steps to first create the \textit{ATMC\textit{ard}} class in Java and then add it in Smalltalk:

1. \textbf{In Java}, use the SmartGuide to add the \textit{ATMC\textit{ard}} class to the \textit{rbi.rmi.samples.atm} package. Specify \textit{java.io.Serializable} as the superclass. When we add the user interface component later on, we will be making visual connections to the \textit{ATMC\textit{ard}} fields from other composite beans. To make the fields accessible to other Java beans, we have to add the fields to the \textit{ATMC\textit{ard}} bean interface. To do this, we add the fields as properties. To add the fields as properties, open a Class Browser for \textit{ATMC\textit{ard}} and go to the BeanInfo page. From the Features pop-up menu, select \textbf{New Property Feature...} and specify the \textit{property} as \textit{cardNumber} and the property type as \textit{String}. Repeat this for the \textit{pin}. VisualAge for Java forms the field name by appending \textit{field} to the property name. Adding the
cardNumber and the pin as properties results in the fields fieldCardNumber and fieldPin.

2. **In Smalltalk**, add the ATMCard class to the RbiAtmSamplesAtm application.

   **Note:** If you use the VisualAge Organizer to add the class, make sure you specify Smalltalk class as the part type. This is important because you do not want to inadvertently add application prerequisites that will cause problems when you package the application by using Smalltalk Server Workbench.

   Add two instance variables called fieldCardNumber and fieldPin (we want them to match the Java field names) to the class. Create getter and setter methods for the variables.

   All classes that interact with Java must implement the instance method sstRmiClassName. Add this method and have it return the class name rbi.rmi.samples.atm.ATMCard as a String (Figure 9).

   ```smalltalk
   sstRmiClassName
   ^'rbi.rmi.samples.atm.ATMCard'
   ```

   **Figure 9. ATMCard sstRmiClassName Method**

   Because ATMCard objects are passed by-value from Java to Smalltalk, we have to add the instance method sstIsRmiSerializable, which answers true (Figure 10).

   ```smalltalk
   sstIsRmiSerializable
   ^true
   ```

   **Figure 10. ATMCard sstIsRmiSerializable Method**

3. **In Java**, open the Scrapbook. We will use it to execute code that generates the Java-to-Smalltalk mapping information that SST requires. Figure 11 shows the code to produce the ATMCard mapping information.
Creating and Mapping the Server and Server Objects

Figure 11. Code for Creating ATMCard Java Type Information

Two parameters are passed to the `generateFor(fileName, classNames)` method in `com.ibm.sst.JavaTypeRepository`: The name of a file that will contain the mapping information, and an array of class names. Java type information will be written to the file for each class name in the array.

4. In Smalltalk, open a new workspace. We will use the workspace to execute code that will load the Java type information into the `RbiRmiSamplesAtm` application. Figure 12 shows the code to load the `ATMCard` type information.

Figure 12. Code for Loading ATMCard Java Type Information into Smalltalk

The `mergeDefinitionsFrom:` method stores the Java type information for `ATMCard` in instance methods on the `RbiRmiSamplesAtm` application class. Figure 13 shows the newly created method.

Figure 13. RbiRmiSamplesAtm `rbi_rmi_samples_atm_ATMCard` Method

```java
String fileName = "G:\JavaWorking\rbi\rmi\samples\atm\ATMCard.def";
String classNames[] = new String[1];
classNames[0] = "rbi.rmi.samples.atm.ATMCard";
com.ibm.sst.JavaTypeRepository.generateFor(fileName,classNames);
```

```smalltalk
SstRmiTypeBuilder new
  application: RbiRmiSamplesAtm;
  mergeDefinitionsFrom: 'g:\JavaWorking\rbi\rmi\samples\atm\ATMCard.def'.
```

```smalltalk
rbi_rmi_samples_atm_ATMCard

^#
  (#class 'rbi.rmi.samples.atm.ATMCard' 'java.lang.Object' ##nil
   -6690088125205169385 ##nil
   ()
   ((fieldCardNumber 'Ljava.lang.String;' ##nil ##nil)
    (fieldPin 'Ljava.lang.String;' ##nil ##nil))
  )
```
5. The Smalltalk `rbi.rmi.samples.atm_ATMCard` method returns a two-element array. All Smalltalk elements of the definition are initially set to `##nil`.

The first element is an array with information about the class. The second and third elements of the array contain the Java class and superclass names, respectively. The fourth element is used to hold the Smalltalk class to which the Java class maps. Initially it contains `##nil`. Change `##nil` to `#ATMCard`.

The second element is an array with information about the properties of the class. The first element contains the name of the instance variable, the second contains the variable’s Java type, and the third and fourth elements specify the Smalltalk getter and setter methods. If you leave the third and fourth elements unchanged, the getter and setter methods are derived from the property names.

Figure 14 shows the `rbi.rmi.samples.atm_ATMCard` method after it has been modified.

```
Smalltalk

rbi.rmi.samples.atm_ATMCard
^#
  (#class 'rbi.rmi.samples.atm.ATMCard' 'java.lang.Object' #ATMCard
   -6690088125205169385 ##nil
   ()
   (('fieldCardNumber' 'Ljava.lang.String;' ##nil ##nil)
    ('fieldPin' 'Ljava.lang.String;' ##nil ##nil))
  )
```

Figure 14. Modified rbi.rmi.samples.atm_ATMCard Method

The above method provides Smalltalk with the following mapping information:

- Java `rbi.rmi.samples.atm.ATMCard` class maps to `ATMCard`.
- `rbi.rmi.samples.atm.ATMCard` has two instance variables, `fieldCardNumber` and `fieldPin`, each defined as `java.lang.String`.
- The Smalltalk getter and setter methods are `fieldPin` and `fieldPin:`.
Creating and Mapping the Server and Server Objects

7.2 Mapping Base Java or Smalltalk Classes

We have discussed how to map the ATMCard class from Java to Smalltalk, but to which Smalltalk classes do the java.util.Vector and java.util.Hashtable map? Or how about the reverse, what are the Java classes to which OrderedCollection and Dictionary map?

For Java-to-Smalltalk class mappings, you can go to the SstRmiSupport application class. As in the RbiRmiSamplesAtm application class, the mappings are stored in instance methods. If you look at the java_util_Vector method, you will see that the Smalltalk implementation class is OrderedCollection.

For Smalltalk-to-Java class mappings, you can browse implementors of the sstRmiClassName method. Any class that maps to a Java class must implement this method. This method returns the Java implementation class. So, if you look at this method in OrderedCollection, you will see that it returns the 'java.util.Vector' string.

7.3 Creating and Mapping the ATMServer Class

Our design states that the ATMServer class is defined in Smalltalk. Follow these steps to create and map the ATMServer class:

1. In Smalltalk, add the ATMServer class to the RbiRmiSamplesAtm application
   - Add the following instance variables with getter and their associated setter methods:
     - defaultCards
     - defaultAccounts
     - registeredBankAccounts
   - Add the following required RMI support methods:
     - sstRmiClassName - This method should return the class name rbi.rmi.samples.atm.ATMServer as a String.
     - sstIsRmiRemotable - Because the ATMServer singleton is remote to Java, this method should return true.
   - According to our design, the Java client sends the verifyCardAndGetAccountNumbers(anATMCard) and getBankAccount(accountNumber) messages to the ATMServer
singleton. We have to implement these methods in the ATMServer class.

- Add the `verifyCardAndGetAccountNumbers`: instance method. This method receives one parameter, an instance of `ATMCard`. When we complete this method, it will return an `OrderedCollection` of account numbers.

- Add the `getBankAccount`: instance method. This method receives one parameter, an instance of `String`.

- In addition to the `verifyCardAndGetAccountNumbers` and `getBankAccount` methods, add the following instance methods to the ATMServer class:
  - `unregisterAllBankAccounts`
  - `restartAtm`

2. As indicated in 2.2.5, "Java Stub and Interface Classes" on page 8, there are two ways of creating the Java stub and interface classes required for remote message sends. The first approach is to create the necessary classes in Java. We look at this approach first.

**In Java:**

- Add the interface `ATMServerIf` to the `rbi.rmi.samples.atm` package. The interface should extend `java.rmi.Remote`.

- Add a method called `verifyCardAndGetAccountNumbers`. The method receives one parameter, an instance of `ATMCard`. In Smalltalk, this method returns an `OrderedCollection`. An `OrderedCollection` maps to `java.util.Vector`. Therefore the return type is `java.util.Vector`. The method can throw a `java.rmi.RemoteException` exception.

- Add a method called `getBankAccount`. The method is passed an instance of `String`. The return type is void. The method can throw a `java.rmi.RemoteException`.

- Add a method called `unregisterAllBankAccounts`. The return type is void. The method can throw a `java.rmi.RemoteException`.

- Add a method called `restartAtm`. The return type is void. The method can throw a `java.rmi.RemoteException`.

*Figure 15* shows the `rbi.rmi.samples.atm.ATMServerIf` class definition.
Now add the `ATMServer` class to the `rbi.rmi.samples.atm` package. The class extends `UnicastRemoteObject` and implements `ATMServerIf`.

Because the `ATMServer` class implements `ATMServerIf`, it must implement the same methods. Add the same four methods to the `ATMServer` class that you added to `ATMServerIf`.

After adding the `ATMServer` class and its methods, you are ready to generate the stub and skeleton files. To generate these files, select the `ATMServer` class and then select `Tools->Rmi - Generate Stub and...`
**Skeleton** from the pop-up menu. The RMIC tool generates the following two new classes: `ATMServer_Stub` and `ATMServer_Skel`.

- Once we have created the `ATMServer` class and generated the `ATMServer_Stub` class, we can proceed to generate the Java-to-Smalltalk mapping information that SST requires. Figure 16 shows the code to produce the `ATMServer` and `ATMServer_Stub` mapping information.

```java
String fileName = "G:\JavaWorking\rbi\rmi\samples\atm\ATMServer.defs";
String classNames[] = new String[2];
classNames[0] = "rbi.rmi.samples.atm.ATMServer";
classNames[1] = "rbi.rmi.samples.atm.ATMServer_Stub";
com.ibm.sst.JavaTypeRepository.generateFor(fileName,classNames);
```

*Figure 16. Code for Creating ATMServer and ATMServer_Stub Java Type Information*

3. Now we are ready to load the Java-to-Smalltalk mapping information for the `ATMServer` and `ATMServer_Stub` classes into Smalltalk.

- In Smalltalk open a workspace and execute the following shown in Figure 17.

```smalltalk
SstRmiTypeBuilder new
    application: RbiRmiSamplesAtm;
    mergeDefinitionsFrom: 'g:\JavaWorking\rbi\rmi\samples\atm\ATMServer.defs'.
```

*Figure 17. Code for Loading ATMServer and ATMServer_Stub Java Type Information*

The merge creates and stores the Java type information in two new instance methods of the RbiRmiSamplesAtm application class. Figure 18 shows the newly created `rbi_rmi_samples_atm_ATMServer` method, and Figure 19 shows the newly created `rbi_rmi_samples_atm_ATMServer_Stub` method.
• All Smalltalk elements of the ATMServer definition are initially set to **##nil** (Figure 18). In the `rbi_rmi_samples_atm_ATMServer` method we have to map the class and the methods:

```smalltalk
^#(
    (class 'rbi.rmi.samples.atm.ATMServer' 'java.rmi.server.UnicastRemoteObject'
        ##nil
        5525710884232639318 -6589954860484502580
        ('rbi.rmi.samples.atm.ATMServerIf' 'java.rmi.Remote'))
    ()
    (##nil 'getBankAccount(Ljava.lang.String;)V'
        ('java.rmi.RemoteException'))
    (##nil 'restartAtm()V' ('java.rmi.RemoteException'))
    (##nil 'unregisterAllBankAccounts()V'
        ('java.rmi.RemoteException'))
    (##nil 'verifyCardAndGetAccountNumbers(Lrbi.rmi.samples.atm.ATMCard;
       Ljava.util.Vector;' ('java.rmi.RemoteException'))
) }
```

Figure 18. RbiRmiSamplesAtm rbi_rmi_samples_atm_ATMServer Method

```smalltalk
^#(
    (class 'rbi.rmi.samples.atm.ATMServer_Stub' 'java.rmi.server.RemoteStub'
        ##nil
        3944105270779143624 -6589954860484502580
        ('rbi.rmi.samples.atm.ATMServerIf' 'java.rmi.Remote'))
    ()
    (##nil 'getBankAccount(Ljava.lang.String;)V'
        ('java.rmi.RemoteException'))
    (##nil 'restartAtm()V' ('java.rmi.RemoteException'))
    (##nil 'unregisterAllBankAccounts()V' ('java.rmi.RemoteException'))
    (##nil 'verifyCardAndGetAccountNumbers(Lrbi.rmi.samples.atm.ATMCard;
       Ljava.util.Vector;' ('java.rmi.RemoteException'))
) }
```

Figure 19. RbiRmiSamplesAtm rbi_rmi_samples_atm_ATMServer_Stub Method
• Change the first occurrence of ##nil to #ATMServer to map the class.

• Change the next four occurrences of ##nil to
  
  getBankAccount:
  restartAtm
  unregisterAllBankAccounts
  verifyCardAndGetAccountNumbers:

Remember, the methods already exist in ATMServer because we added them when we created the class. All we are doing here is mapping the Java class and its methods to their Smalltalk equivalents.

Figure 20 shows the modified rbi_rmi_samples_atm_ATMServer method.

```
Smalltalk
rbi_rmi_samples_atm_ATMServer

^#(
  (#class 'rbi.rmi.samples.atm.ATMServer' 'java.rmi.server.UnicastRemoteObject'
    #ATMServer
    552571088423639318 -6589954860484502580
    ('rbi.rmi.samples.atm.ATMServerIf' 'java.rmi.Remote'))
  ()
  (#getBankAccount: 'getBankAccount(Ljava.lang.String;)V'
    ('java.rmi.RemoteException'))
  (#restartAtm 'restartAtm()V' ('java.rmi.RemoteException'))
  (#unregisterAllBankAccounts 'unregisterAllBankAccounts()V'
    ('java.rmi.RemoteException'))
  (#verifyCardAndGetAccountNumbers:
    'verifyCardAndGetAccountNumbers(Lrbi.rmi.samples.atm.ATMCard;
      Ljava.util.Vector;' ('java.rmi.RemoteException'))
  )
```

Figure 20. Modified rbi_rmi_samples_atm_ATMServer Method

• The only modification required to the
  rbi_rmi_samples_atm_ATMServer_Stub method is the mapping of the stub class to its Smalltalk equivalent. All Java stub classes map to the predefined SstRmiGenericStub Smalltalk class. To map the stub class, change the first occurrence of ##nil to #SstRmiGenericStub. Figure 21 shows the modified rbi_rmi_samples_atm_ATMServer_Stub method.

38 VisualAge Java - RMI - Smalltalk ATM Sample
7.4 Creating and Mapping the BankAccount Class

The steps required to create and map the BankAccount class are with one variation (step 5) identical to those required for the ATMServer class.

In Smalltalk...

1. Add the BankAccount class to the RbiRmiSamplesAtm application.

2. Add the following instance variables and their associated getter and setter methods.
   - accountNumber
   - accountType
   - balance
   - customerId
   - cardNumber

3. Add the following required RMI support methods:
   - sstRmiClassName - This method should return the class name rbi.rmi.samples.atm.ATMServer as a String.
   - sstIsRmiRemotable - Because the ATMServer singleton is remote to Java, this method should return true.
4. Our design states that all account transactions are processed by the
*BankAccount* class. To keep things simple, we will only implement the
withdraw function.

Add the *withdraw:* instance method. The withdraw method receives one
parameter indicating the amount of money that should be withdrawn from
the account. Because the *java.math.BigDecimal* Java class does not map
to the *ScaledDecimal* Smalltalk class, the withdraw amount is passed as a
string and converted to a *ScaledDecimal*.

5. As indicated in 2.2.5, "Java Stub and Interface Classes" on page 8, there
are two ways of creating the Java stub and interface classes required for
remote message sends. For the *ATMServer* class, we added the interface
and dummy *ATMServer* class directly to the *rbi.rmi.samples.atm* package
and then used the RMIC to generate the stub. For the *BankAccount* class,
we use the *RMI Type Builder* in Smalltalk to generate the stub and
interface classes, as well as the Java type information required by
Smalltalk.

The
*SstRmiTypeBuilder>>#generateClassesAndJavaFor:interfaces:methods:* method can be used to generate the stub class and mapping information
for the *BankAccount* class. Figure 22 shows the code to generate the stub
class and Java type information for the *BankAccount* class.

```
SstRmiTypeBuilder new
    application: RbiRmiSamplesAtm;
    directory: 'g:\javaworking';
    buildDirectoryStructure: true;
    generateClassesAndJavaFor: 'rbi.rmi.samples.atm.BankAccount'
        interfaces: #('rbi.rmi.samples.atm.BankAccountIf' 'java.rmi.Remote')
        methods: #('getAccountType()Ljava.lang.String;' ('java.rmi.RemoteException'))
                      ('getBalance()Ljava.lang.String;' ('java.rmi.RemoteException'))
                      ('withdraw(Ljava.lang.String;)V ('java.rmi.RemoteException'))
);
```

Figure 22. Code for Generating *rbi.rmi.samples.atm.BankAccount_Stub* Class

This code creates a new instance of *SstRmiTypeBuilder*.

The parameter passed on the *application:* message tells the RMI type
builder where the Java type information will be stored. In this example, the
Java type information for *rbi.rmi.samples.atm.BankAccount* and
*rbi.rmi.samples.atm.BankAccount_Stub* will be added as instance
methods to the *RbiRmiSamplesAtm* application class.
The parameter passed on the `directory:` message specifies the root directory where the generated Java class definitions will be stored.

The parameter passed on the `buildDirectoryStructure:` message indicates whether or not the directory structures for the generated class should be built. In this example, a value of true would store the generated stub class in the `g:\javaworking\rbirmi\samples\atm directory`.

Three parameters are passed on the `generateClassesAndJavaFor:interfaces:methods:` message:

1. The `generateClassesAndJavaFor:` parameter specifies the name of the class for which the stub will be generated. The code will generate a stub for the `rbi.rmi.samples.atm.BankAccount` class.

2. The `interfaces:` parameter specifies the interfaces implemented by the class. This parameter is an array of strings, each representing a specific interface implemented by the class. The `rbi.rmi.samples.atm.BankAccount` class implements the `rbi.rmi.samples.atm.BankAccountIf` and `java.rmi.Remote` interfaces.

3. The `methods:` parameter specifies the methods implemented by the class. This parameter is an array. Each element of the array is itself a two-element array consisting of the method signature and an array of exceptions thrown by the method. This parameter tells the RMI type builder that the `rbi.rmi.samples.atm.BankAccount` class implements the following methods:
   - `getAccountType()` - This method has the return type `java.lang.String`, and can throw a `java.rmi.RemoteException`.
   - `getBalance()` - This method has the return type `java.lang.String` and can throw a `java.rmi.RemoteException`.
   - `withdraw(java.lang.String)` - The method passes one parameter of the `java.lang.String` type, has the return type is `void`, and can throw a `java.rmi.RemoteException`.

Executing the code creates for Java the `BankAccount_Stub.java` file with the `BankAccount_Stub` class and stores the file in the directory—this is our choice—`g:\javaworking\rbirmi\samples\atm`. The code also creates and stores the Java type information in the Smalltalk `RbiRmiSamplesAtm>>#rbi_rmi_samples_atm_BankAccount` application method (Figure 23).
Figure 23. RbiRmiSamplesAtm rbi_rmi_samples_atm_BankAccount Method

6. Modify rbi_rmi_samples_atm_BankAccount to include the Java-to-Smalltalk mappings (Figure 24).

Figure 24. Modified rbi_rmi_samples_atm_BankAccount Method

The mapping changes made to rbi_rmi_samples_atm_BankAccount cause the following to occur:
• When the Java client sends the `getAccountType()` message to the remote `BankAccount` object, the message sent in Smalltalk is `accountType`.

• When the Java client sends the `getBalance()` message to the remote `BankAccount` object, the message sent in Smalltalk is `getBalance`. The `getBalance()` Java method is not mapped to the `balance` Smalltalk method because the balance for a `BankAccount` object is stored as a `ScaledDecimal` object. As previously mentioned, there is no mapping between the `java.math.BigDecimal` Java class and the `ScaledDecimal` Smalltalk class. The `getBalance` Smalltalk method converts the balance to a `String` and returns it.

• When the Java client sends the `withdraw(java.lang.String)` message to the remote `BankAccount` object, the message sent in Smalltalk is `withdraw:`.

7. Generate the `BankAccount` Java interface class. Use the `SttRmiTypeBuilder>> #generateInterfaceAndJavaFor:superclass:methods:` method with the parameters as shown in Figure 25.

```
SttRmiTypeBuilder new
directory: 'g:\javaworking';
buildDirectoryStructure: true;
generateInterfaceAndJavaFor: 'rbi.rmi.samples.atm.BankAccountIf'
superclass: 'java.rmi.Remote'
methods: #( ( 'getAccountType()Ljava.lang.String;' ('java.rmi.RemoteException'))
          ( 'getBalance()Ljava.lang.String;' ('java.rmi.RemoteException'))
          ( 'withdraw(Ljava.lang.String;)V' ('java.rmi.RemoteException')) ).
```

Figure 25. Code for Generating `rbi.rmi.samples.atm.BankAccountIf` Class

Three parameters are passed on the `generateInterfaceAndJavaFor:superclass:methods:` message.

1. The `generateInterfaceAndJavaFor:` parameter specifies the name of the interface class. The code generates the `rbi.rmi.samples.atm.BankAccountIf` Java interface class.

2. The `superclass:` parameter specifies the superclass. Like `ATMServerIf`, `BankAccountIf` extends `java.rmi.Remote`.

3. The `methods:` parameter specifies the methods implemented by the interface. This parameter is an array. Each element is itself an array consisting of the method signature and an array of exceptions thrown by the method.
Executing the code in Figure 25 creates the BankAccountIf.java file and stores it in the g:\javaworking\rbi\rmil\samples\atm directory.

### In Java...

8. Import the BankAccount interface and stub classes into VisualAge for Java. Select the rbi.rmi.samples.atm package on the Projects page of the VisualAge for Java Workbench. From the pop-up menu select Import. Select Directory as the import source and then specify the directory containing the generated BankAccount interface and stub classes.

---

### 7.5 Creating and Mapping the ATM Application Exceptions

Exceptions represent error conditions that arise during code execution. In Java, you have try and catch statements. Using these statements, you try to execute a block of code and catch any exceptions that may be thrown by the system should an error condition arise.

Smalltalk SST supports the catching and throwing of Java exceptions, through the existing Smalltalk exception handling mechanism. Java exceptions are instances of a class. All of these Java classes map to the SstRmiThrowable Smalltalk class or one of its subclasses. Exceptions are thrown by sending the class message throw:message: to SstRmiThrowable or one of its subclasses.

Our design states that the ATMServer singleton throws a CardValidationException when processing an ATMCARD object containing an invalid PIN or card number. BankAccount objects throw an AccountOverdrawnException when a withdraw results in a negative balance.

Because exceptions are instances of classes, we have to provide Java type information for all exceptions thrown in Smalltalk.

### 7.6 Creating and Mapping the ATM Application Exceptions

Use the following steps to create and map the exceptions:

**In Java...**

1. Add the CardValidationException class to the rbi.rmi.samples.atm package. Specify java.rmi.RemoteException as the superclass.

2. Add the AccountOverdrawnException class to the rbi.rmi.samples.atm package. Specify java.rmi.RemoteException as the superclass.
3. Execute the Java code shown in Figure 26 to create the Java-to-Smalltalk mapping information for the exception classes.

```java
String fileName = "G:\JavaWorking\rbi\rmi\samples\atm\Exception.defs";
String classNames[] = new String[2];
classNames[0] = "rbi.rmi.samples.atm.AccountOverdrawnException";
classNames[1] = "rbi.rmi.samples.atm.CardValidationException";
com.ibm.sst.JavaTypeRepository.generateFor(fileName,classNames);
```

*Figure 26. Code for Creating AccountOverdrawnException Java Type Information*

In Smalltalk...

4. Execute the Smalltalk code shown in Figure 27 to load the generated Java-to-Smalltalk mapping definitions.

```smalltalk
SstRmiTypeBuilder new
  application: RbiRmiSamplesAtm;
  mergeDefinitionsFrom: 'g:\JavaWorking\rbi\rmi\samples\atm\Exception.defs'.
```

*Figure 27. Code for Loading AccountOverdrawnException Java Type Information into Smalltalk*

Figure 28 shows the generated and loaded Java type definitions.
5. The only change required is to map the Java classes to Smalltalk classes. Both exception classes will map to \textit{SstRmiDetailedException}. To map the classes, change the first occurrence of ##nil in both methods to ##SstRmiDetailedException (Figure 29).
Figure 29. Modified RbiRmiSamplesAtm Exception Methods
Now that we have created and loaded the required class mappings, we need to add code to the ATMServer class that will set up the server and make it available to the Java ATM client. Specifically, we have to add code to accomplish the following tasks:

- Register the invocation schema and handler.
- Create and set up the application context.
- Start the context and the components it manages.
- Create the naming context that runs the RMI naming service.
- Instantiate the ATMServer singleton and add it to the RMI naming service registry.
- Shut down the service

This `setupServer:namingPort: ATMServer` class method in Figure 30 is responsible for setting up the ATMServer singleton.

```
setupServer: serverAddress namingPort: aPort

" Set up the ATMServer on serverAddress. Register the invocation handler. Create an application context, add a object space, setup the context and start the context. Create and start an RMI naming context. Instantiate the singleton and bind it to a name. "

| url portName|
self registerInvocationConfiguration.
aPort notNil
    ifTrue: [portName := ':', aPort]
    ifFalse: [portName := ''].
self context
    addSpace: self logicalName asSymbol at:
        (Array with: (url := self urlFor: serverAddress ));
    setupFor: self logicalName asSymbol
        using: SstSpaceConfiguration rmiProxyConfiguration.
selct context rmiStartUpWith: self typeRepository.
selct createNamingContextOn: ('rmi://', portName) sstAsUrl.
selct bind: self new as: self logicalName.
```

Figure 30. AMTServer setupServer:namingPort: Class Method
We will look at code fragments from this method when discussing each of the tasks listed above.

### 8.1 Registering the Invocation Schema and Handler

The code in Figure 31 registers the invocation handler for the `ATMServer`. A by-reference invocation scheme with an RMI marshaler is registered to the `atmServerRmi` name. The invocation handler waits for incoming messages. When a message is received, it coordinates the retrieval of the message, the reconstruction of the message, the sending of the message to the local object, and the reply with any required return values.

```smalltalk
setupServer: serverAddress namingPort: aPort

| url portName|
self registerInvocationConfiguration.
....

registerInvocationConfiguration

" Register the invocation scheme and handler. The handler is registered using a by-reference invocation configuration with an RMI marshaler "

SstUrl register: self invocationScheme as: SstRmiUrl.
SstInvocationHandler
   register: SstInvocationConfiguration rmiConfiguration
     asScheme: self invocationScheme

SstInvocationHandler
   register: SstInvocationConfiguration rmiConfiguration
     asScheme: self invocationScheme

invocationScheme

" Return the invocation scheme name "

^'atmServerRmi'`

Figure 31. Code to Register the Invocation Scheme and Handler
8.2 Creating and Setting up the Application Context

The code Figure 32 creates the application context, adds an object space to the context, and sets up the context. The local object space is configured by using an SstSpaceConfiguration, which allows remote references to be imported and exported for Java RMI.

```
Smalltalk

setupServer: serverAddress namingPort: aPort
....
  self context
    addSpace: self logicalName asSymbol at:
      (Array with: (url := self urlFor: serverAddress ));
    setupFor: self logicalName asSymbol
      using: SstSpaceConfiguration rmiProxyConfiguration.
    ....

context

  "Answer the application context. If it doesn’t exist, one is created."

  ^SstApplicationContext
    contextNamed: self contextName
    ifNone: [SstApplicationContext createContextNamed: self contextName]

logicalName

  "Return a unique name. In this example, the class name is returned as a string."

  ^self name asString

customName

  ^self logicalName, 'Rmi'

urlFor: transportSpec

  "Create and return a Url for the supplied transport."

  ^(self invocationScheme , ':/' , transportSpec) sstAsUrl
```

*Figure 32. Code to Create and Set Up the ATMServerRmi Application Context*
The `logicalName` method returns the class name as a string. The context name is the `logicalName` with `Rmi` appended to it. The name of the object space is `#ATMServer`.

The URL specified when the context is set up provides a scheme, a transport, and an address. The scheme is `atmServerRmi`, the transport is `rmi`, and the address is the `serverAddress` parameter passed to the `setupServer:namingPort:` method. This URL will be used to build an endpoint that provides addressability to the object space.

### 8.3 Starting the Context

Application contexts configured for RMI are started through the `rmiStartUpWith:` method (Figure 33). The startup parameter specifies an RMI typing repository that maps Java types to their Smalltalk counterparts. The `typeRepository` method returns an instance of `SstRmiTypingRepository` that is passed on the `rmiStartUpWith:` message. It is initialized with all of the Java type definitions from the `SstRmiSupport` and `RbiRmiAtmSamples` application classes.

```smalltalk
Smalltalk
setupServer: serverAddress namingPort: aPort
...
self context rmiStartUpWith: self typeRepository.
....

typeRepository

"Answer the typing repository to use for this application."

^SstRmiTypingRepository newWithBaseClasses
loadDefinitionsFrom: RbiRmiSamplesAtm.
```

*Figure 33. Code to Start the ATMServerRmi Application Context*
8.4 Creating the Naming Context

The `SstApplicationContext createRmiNamingServiceAt:` class method creates, starts, and returns a context that runs an RMI naming service (Figure 34). Because this method starts the context, there is no need to send the `rmiStartUpWith:` to the context.

The naming service listens on `aPort` passed to the `setupServer:namingPort:`. If `aPort` is `nil`, the naming service listens on the default RMI registry port, 1099.

When the RMI naming context is started, an instance of `SstRmiRegistry` is exported. The `SstRmiRegistry` object maintains the list of registered objects.

The returned context is stored in the class variable `NamingContext`.

```
Smalltalk
setupServer: serverAddress namingPort: aPort
...
self createNamingContextOn: (‘rmi://’, portName) sstAsUrl.
...
createNamingContextOn: aPort
    self namingContext: (SstApplicationContext createRmiNamingServiceAt: aPort)
```

Figure 34. Code to Create the RMI Naming Application Context

8.5 Instantiating and Registering the ATMServer Singleton

The code Figure 35 creates an `ATMServer` object and adds it to the RMI naming service registry. When an object is added to the registry, it is bound to a key. The `ATMServer` object is bound to the ‘ATMServer’ string.

To bind the `ATMServer` object, a URL with addressability to the local object space of the RMI naming context is created. The data field of the URL is set to the ‘ATMServer’ string. This is the key to which the object will be bound in the RMI naming service registry. The `rmiNaming` message is sent to the `ATMServerRmi` context which returns a `SstRmiNaming` object. The `rebind:to:` message is then sent to the `SstRmiNaming` object with the `ATMServer` object and the URL passed as parameters.
Here is a summary of the processing that occurs as a result of sending the rebind:to: message to the SstRmiNaming object:

- A remote object space for the URL is added to ATMServerRmi context.
- A request is sent to the remote object space to process the rebind:to: message.
- The rebind:to: message is sent to the SstRmiRegistry object in the export set of the RMI naming context local object space.

```
Smalltalk
setupServer: serverAddress namingPort: aPort
...
self bind: self new as: self logicalName.

bind: anObject as: aName

" Create a URL with addressability to the RMI naming context local object space. Set the data field of the URL to aName. Get the rmi naming for the server context and send the rebind:to: message to it passing the URL and anObject to be bound. "

| url |

(url := self namingContext space accessibleUrls first copyCore) data: aName.
self context rmiNaming rebind: url to: anObject.
```

*Figure 35. ATMServer Instantiation and RMI Registration*

We have to consider one more ATMServer class method. The method name is setup and it should be specified as the application launch code when the application is packaged (Figure 36). The setup method sends a message to the default SciSocketManager object to get the host name of the machine on which the ATM server application is running. It processes the command line arguments specified when the application was started, looking for the -PORT: argument. If this argument is found, the port number is retrieved. The host name and port number are passed as parameters on the setupServer:namingPort: message.
8.6 Shutting Down the Server

We need to be able to stop the ATM server application and release all of its resources. To do this, we will implement a shutDown class method that unregisters the invocation scheme and handler and sends the clear message to the ATMServerRmi and RMI Naming application contexts. The clear message shuts down an application context and releases all its resources, ensuring that no pointers to complex objects remain. After the SST resources are released, the ATMServer singleton will be garbage collected. Figure 37 shows the code to stop the ATMServer.

```
Smalltalk

setup

| host port |
host := 'rmi:\', SciSocketManager default getHostName.
System commandLine
  do: [:arg | (arg asUppercase indexOfSubCollection: '-PORT:' startingAt: 1) = 1
    ifTrue: [port := arg copyFrom: 7 to: arg size.]].
self setupServer: host namingPort: port

Smalltalk

shutDown

" Unregister the invocation scheme and handler. Clear all application contexts."

self unregisterInvocationConfiguration.
self context clear.
self namingContext notNil
  ifTrue: [self namingContext clear.
    self namingContext: nil].

unregisterInvocationConfiguration

SstUrl unregister: self invocationScheme.
SstInvocationHandler unregisterScheme: self invocationScheme
```

Figure 36. ATMServer Application Launch Code

Figure 37. Code to Stop the ATMServer
8.7 Methods Common and Unique to an SST RMI Server Object

We implemented 15 `ATMServer` class methods. Most of the methods are generic and could be implemented in an abstract class to provide common behavior for setting up SST RMI servers. The generic methods are grouped together in the *Common* category (Figure 38).

![ATMServer - Script Editor](image)

*Figure 38. ATMServer Common Class Methods*

The methods unique to the `ATMServer` are grouped together in the *Unique* category (Figure 39).
### Enabling the ATMServer for SST RMI

#### Figure 39. ATMServer Unique Class Methods

```
ATMServer

Object subclass: #ATMServer
  instanceVariableNames: 'defaultCards defaultAccounts registeredBankAccounts' 
  classVariableNames: "NamingContext" 
  poolDictionaries:""

ATMServer(10/28/98 3:37:26 PM) (Defined in RblRmiSamplesAtm source)
```
Chapter 9. Adding State and Behavior to the Server Objects

To keep the ATM sample application as simple as possible, we add only a minimum of business state and behavior and stage the implementation of the persistence state and behavior. In the first stage of the persistence implementation, the `defaultXYZ` methods provide default data to emulate the database. Real persistence is added in Part 4, “Adding Persistence” on page 153.

9.1 ATMServer State and Behavior

When complete, the ATMServer class has the instance variables and methods shown in Figure 40.

Figure 40. ATMServer Instance Categories and Methods
9.1.1 ATMServer Instance Variables

We added these three instance variables and their associated getter and setter methods when we created the ATMServer class:

**defaultCards** - a Dictionary. Its keys are card numbers, and its values, PIN. This variable serves as a datastore for the ATM application.

**defaultAccounts** - a Dictionary. Its keys are account numbers, and its values BankAccount objects. This variable serves as a datastore for the ATM application.

**registeredBankAccounts** - a Dictionary. Its keys are account numbers, and its values BankAccount objects. This variable holds the BankAccount objects that have been registered and exported.

9.1.2 ATMServer Methods

We added the following methods to the ATMServer:

Six getters and setter methods

- **defaultCards** and **defaultCards**:
- **defaultAccounts** and **defaultAccounts**:
- **registeredBankAccounts** and **registeredBankAccounts**:

Four business-oriented action methods:

- **verifyCardAndGetAccountNumbers**:
- **getAccount**:
- **unregisterBankAccount**:
- **unregisterAllBankAccounts**

Two operation management methods:

- **restartATM**
- **shutDown**

Two basic RMI methods:

- **sstsRmiRemotable**
- **sstRmiClassName**

**Getter and Setter Methods for the Instance Variables**

The getter and setter methods for the **defaultCards**, **defaultAccounts**, and **registeredBankAccounts** ATMServer instance variables are shown in Figure 41, Figure 42, and Figure 43, respectively.
Smalltalk

defaultCards

defaultCards isNil
  ifTrue: [ defaultCards := Dictionary new
    at: '000000' put: '0000';
    at: '111111' put: '1111';
    yourself].

  ^defaultCards

defaultCards: aDictionary

  defaultCards := aDictionary

Figure 41. ATMServer defaultCards Getter and Setter Methods
Smalltalk

defaultAccounts

defaultAccounts isNil
ifTrue: [defaultAccounts := Dictionary new
   at: '098764' put: (BankAccount new
      accountNumber: '098764';
      accountType: 'Checking';
      balance: '5678.55' asScaledDecimal;
      customerId: nil;
      cardNumber: '000000');
   at: '023745' put: (BankAccount new
      accountNumber: '023745';
      accountType: 'Savings';
      balance: '5045.03' asScaledDecimal;
      customerId: nil;
      cardNumber: '000000');
   at: '139483' put: (BankAccount new
      accountNumber: '139483';
      accountType: 'Checking';
      balance: '3523.99' asScaledDecimal;
      customerId: nil;
      cardNumber: '111111');
   at: '172453' put: (BankAccount new
      accountNumber: '172453';
      accountType: 'Savings';
      balance: '15001.00' asScaledDecimal;
      customerId: nil;
      cardNumber: '111111');
   yourself. ].

^defaultAccounts

defaultAccounts: aDictionary

defaultAccounts := aDictionary
Adding State and Behavior to the Server Objects

Smalltalk

registeredBankAccounts

registeredBankAccounts isNil
  ifTrue: [registeredBankAccounts := Dictionary new].
^registeredBankAccounts

registeredBankAccounts: aDictionary

registeredBankAccounts := aDictionary

verifyCardAndGetAccountNumbers:
The verifyCardAndGetAccountNumbers: method receives an ATMCard object and is responsible for verifying the card number and PIN (Figure 45). If the card number is invalid, the Java RMI rbi.rmi.samples.atm.CardValidationException is thrown, and the exception message set to 'Invalid Card Number'. If the PIN number is invalid, the rbi.rmi.samples.atm.CardValidationException is thrown and the message set to 'Invalid PIN'. If the ATMCard object is validated, a list of accessible account numbers is returned in an OrderedCollection.

Smalltalk

verifyCardAndGetAccountNumbers: anATMCard

| accountNumbers |
| anATMCARD fieldPin = (self defaultCards at: anATMCARD fieldCardNumber
  ifAbsent: [SstRmiDetailedException
    throw: 'rbi.rmi.samples.atm.CardValidationException'
    message: 'Invalid Card Number'])
  ifFalse: [SstRmiDetailedException
    throw: 'rbi.rmi.samples.atm.CardValidationException'
    message: 'Invalid PIN'].
accountNumbers := OrderedCollection new.
self defaultAccounts associationsDo: [:assoc |
  assoc value cardNumber = anATMCARD fieldCardNumber
  ifTrue: [accountNumbers add: assoc value accountNumber]].
^accountNumbers

Figure 43. ATMServer registeredBankAccounts Getter and Setter Methods

Figure 44. ATMServer verifyAndGetAccountNumbers: Method
**getBankAccount:**

The `getBankAccount:` method receives a `String` object containing an account number (Figure 45). The `BankAccount` object corresponding to the account number is retrieved and added to the RMI naming service registry.

```
getBankAccount: accountNumber

" Check to see if the BankAccount object is registered. If not, retrieve it, add it to the RMI naming service registry and add it to the list of registered accounts. "

(self registeredBankAccounts includesKey: accountNumber)
  ifFalse: [self class bind: (self defaultAccounts at: accountNumber)
    as: accountNumber.
    self registeredBankAccounts at: accountNumber
    put: (self defaultAccounts at: accountNumber)]
```

*Figure 45. ATMServer getBankAccount: Method*

**Unregister BankAccounts Methods**

When a new `ATMCard` is entered or the ATM Application is restarted, all registered `BankAccount` objects are unregistered. Unregistering a `BankAccount` object requires that it be removed from the RMI naming service registry and from the export set of the `ATMServerRmi` context local object space. Figure 46 shows the `unregisterBankAccount:` and `unregisterAllBankAccounts` methods which unregister `BankAccount` objects.
Adding State and Behavior to the Server Objects

At some point when running the ATM application with the non-persistent data, you may want to reset the balances of the default bank accounts. The `restartAtm` method (Figure 47) unregisters all `BankAccount` objects and sets the `defaultAccounts` instance variable to nil, causing the `defaultAccounts` lazy initialization to run the next time the variable is accessed.

```
Smalltalk
restartAtm

" Unregister all BankAccount objects and reset defaultAccounts. "

self unregisterAllBankAccounts.
self defaultAccounts: nil.
```

---

**Figure 46. ATMServer Methods to unregister BankAccount Objects**

- **unregisterBankAccount: accountNumber**
  ```
  Smalltalk
  unregisterBankAccount: accountNumber
  
  " Unregister a BankAccount object for an accountNumber. To unregister a a BankAccount object is must be removed from the RMI naming service registry and from the local object space export set of the ATMServerRmi context."

  (self registeredBankAccounts includesKey: accountNumber)
  ifTrue: [self class context space
    unexportObject: (self registeredBankAccounts at: accountNumber).
    self class context rmiNaming
    unbind: ((self class namingContext space
      accessibleUrls first copyCore)
      data: accountNumber).
    self registeredBankAccounts removeKey: accountNumber.]
  ```

- **unregisterAllBankAccounts**
  ```
  " Unregister all BankAccount objects. "
  self registeredBankAccounts keysDo: [:key | self unregisterBankAccount: key]
  ```

---

**Figure 47. ATMServer restartAtm Method**
**shutDown**

Previously, we added the shutDown class method that stops the ATMServer singleton and releases all SST resources. Because we may want to make this shutDown method available to our Java ATM client or some other application that remotely administers the ATM server application, we can add a shutDown instance method that sends the shutDown class message (Figure 48). Note that the SST objects created by the ATM server application are required to return control to the calling Java ATM client application. To make sure we do not destroy the SST objects before returning, we execute the code in a separate Smalltalk process. In addition to shutting down the ATMServer singleton, you can add code to exit the image.

```smalltalk
shutDown

"To exit the image, you could add a line of code to the block that sends exit message to System."

[self class shutDown] fork.
```

*Figure 48. ATMServer shutDown Method*

**Basic SST RMI Methods**

The `sstIsRmiRemotable` and `sstRmiClassName` are the last two required methods (Figure 49).

```smalltalk
sstIsRmiRemotable

^true

sstRmiClassName

^'rbi.rmi.samples.atm.ATMServer'
```

*Figure 49. ATMServer Basic SST RMI Methods*
9.2 BankAccount State and Behavior

When complete, the BankAccount class has the instance variables and methods shown in Figure 50.

9.2.1 BankAccount Instance Variables

We added these five instance variables and their associated getter and setter methods when we created the BankAccount class:

- accountNumber
- accountType
- balance
- customerId
- cardNumber.
All of the instance variables with the exception of balance are Strings; balance is a ScaledDecimal.

9.2.2 BankAccount Methods

We added the following methods to the ATMServer:

Ten getter and setter methods
- accountNumber and accountNumber:
- accountType and accountType:
- balance and balance:
- customerId and customerId:
- cardNumber and cardNumber:

Two business-oriented action methods:
- getBalance
- withdraw:

Two basic RMI methods:
- sstIsRmiRemotable
- sstRmiClassName

Getter and Setter Methods for the Instance Variables

The getter and setter methods for the accountNumber, accountType, balance, customerId, and cardNumber BankAccount instance variables are shown in Figure 51, Figure 52, Figure 53, Figure 54, and Figure 55, respectively.

Smalltalk

accountNumber

^accountNumber

accountNumber: aString

accountNumber := aString

Figure 51. BankAccount accountNumber Getter and Setter Methods
Figure 52. BankAccount accountType Getter and Setter Methods

```smalltalk
Smalltalk
accountType

^accountType

accountType: aString

accountType := aString
```

Figure 53. BankAccount balance Getter and Setter Methods

```smalltalk
Smalltalk
balance

^balance

balance: aScaledDecimal

balance := aScaledDecimal
```

Figure 54. BankAccount cardNumber Getter and Setter Methods

```smalltalk
Smalltalk
cardNumber

^cardNumber

cardNumber: aString

cardNumber := aString
```
VisualAge Java - RMI - Smalltalk ATM Sample

Figure 55. BankAccount customerId Getter and Setter Methods

getBalance
A BankAccount balance is stored as a ScaledDecimal. As previously noted, the ScaledDecimal class does not map to java.math.BigDecimal. As a result, a getBalance method is required to get the balance and return it to the Java ATM client as a String (Figure 56). The getBalance method uses the ScaledDecimal>>#printOn:showDigits:pad: method to get a String representation of the balance. The resulting String is parsed to remove all leading zeros.

Smalltalk

customerId

^customerId

customerId: aString

customerId := aString

Figure 56. BankAccount getBalance Method

getBalance

" Convert the balance from a ScaledDecimal to a String and return it "

| stream balString index retString |

stream := WriteStream on: (String new: 10).
balString := stream contents.

retString := ".
((balString at: 1) = $-) ifTrue: [ index := 2.
  retString := '-' ]
ifFalse: [index := 1].
[((balString at: index) = $0] whileTrue: [index := index + 1].
^retString, (balString copyFrom: index to: balString size).
### Smalltalk

#### withdraw: withdrawAmount

```
" Perform the withdrawal. Convert the withdrawAmount to a ScaledDecimal and subtract it from the existing balance. If the result balance is negative, throw the Java exception rbi.rmi.samples.atm.AccountOverdrawnException."

self balance: self balance - (withdrawAmount asScaledDecimal).
self balance < 0 ifTrue: [SstRmiDetailedException
throw: 'rbi.rmi.samples.atm.AccountOverdrawnException'
message: 'Account ", self accountNumber, ´is Overdrawn']
```

**Figure 57.** BankAccount withdraw: Method

### Basic SST RMI Methods

The `sstIsRmiRemotable` and `sstRmiClassName` are the last two required methods (Figure 58).

#### Smalltalk

```
sstIsRmiRemotable

^true

sstRmiClassName

^'rbi.rmi.samples.atm.BankAccount'
```

**Figure 58.** BankAccount Basic SST RMI Methods
9.3 **ATMCard State Behavior**

The *ATMCard* has no actual behavior. The only methods are the setters and getters for the instance variables and the basic SST RMI methods.

9.3.1 **ATMCard Instance Variables**

We added these two instance variables and their associated getter and setter methods when we created the *ATMCard* class:

- `fieldCardNumber`
- `fieldPin`

Both variables are *Strings*.

9.3.2 **ATMCard Methods**

We added the following methods to the *ATMCard*:

Four getter and setter methods

- `fieldCardNumber` and `fieldCardNumber`:
- `pin` and `pin`:
- `customerId` and `customerId`:
- `cardNumber` and `cardNumber`:

Two basic RMI methods:

- `sstIsRmiRemotable`
- `sstRmiClassName`

**Getter and Setter Methods for the Instance Variables**

The getter and setter methods for the *ATMCard* instance variables: `fieldCardNumber` and `fieldPin` are shown below:

The getter and setter methods for the `fieldCardNumber` and `fieldPin` are shown in Figure 59, and Figure 60, respectively.
Adding State and Behavior to the Server Objects

---

**Smalltalk**

```smalltalk
fieldCardNumber

^fieldCardNumber

fieldCardNumber: aString

fieldCardNumber := aString
```

*Figure 59. ATMCard fieldCardNumber Getter and Setter Methods*

---

**Smalltalk**

```smalltalk
fieldPin

^fieldPin

fieldPin: aString

fieldPin := aString
```

*Figure 60. ATMCard fieldPin Getter and Setter Methods*

---

**sstIsRmiRemotable and sstRmiClassName**

The `sstIsRmiRemotable` and `sstRmiClassName` are the last two required methods (Figure 61).

---

**Smalltalk**

```smalltalk
sstIsRmiSerializable

^true

sstRmiClassName

^’rbi.rmi.samples.atm.ATMCard’
```

*Figure 61. ATMCard Basic SST RMI Methods*
Chapter 10. Testing and Inspecting the ATMServer

Now that we have completed the ATMServer and BankAccount classes, we can begin some initial testing.

To start the ATMServer, execute the setup class message. This action will perform the create and start of all required SST objects, instantiate the ATMServer singleton, and add it to the RMI naming service registry.

10.1 RMI Naming Service Registry Inspection

To verify that the ATMServer singleton is accessible from Java, we can execute the java.rmi.Naming static method, list(String) as shown in Figure 62. This method returns an array of strings for the URLs in the RMI naming service registry. The parameter passed should be a string containing an asterisk. To get all the URLs in the RMI naming service registry, inspect the code in a Java Scrapbook.

```java
java.rmi.Naming.list("*")
```

*Figure 62. RMI Naming Service Registry Check*

Inspecting the code should display a Ljava.lang.String; inspector with an entry for the ATMServer as illustrated in Figure 63.

*Figure 63. Inspecting RMI Naming Service Registry Contents*
10.2 ATMServer withdraw(String) Test

Now that the server is up and running and we can look it up in the registry, we can send messages to it. Specifically, we can retrieve the remote reference to the ATMServer and then send it a message to get a BankAccount. After sending the getBankAccount(String) message, we can retrieve the remote reference to the BankAccount from the registry and send it the withdraw(String) message. Figure 64 shows the script to accomplish these tasks.

Java

```java
rbi.rmi.samples.atm.ATMServerIf server = 
    (rbi.rmi.samples.atm.ATMServerIf)java.rmi.Naming.lookup("/ATMServer");
server.getBankAccount("098764");
	rbi.rmi.samples.atm.BankAccountIf account = 
    (rbi.rmi.samples.atm.BankAccountIf)java.rmi.Naming.lookup("098764");
com.ibm.uvm.tools.DebugSupport.inspect(account.getBalance());
account.withdraw("5000");
com.ibm.uvm.tools.DebugSupport.inspect(account.getBalance());
```

Figure 64. ATMServer Test Script

Our test code inspects the balance for BankAccount 098764 before and after a withdraw. The balance is 5678.55 before to the withdraw, and 678.55 after the withdraw.

Now that we have sent messages to the ATMServer singleton and a BankAccount object from the Java ATM client, let us go over to Smalltalk and inspect some of the SST objects. Specifically, let us look in the export set of the ATMServerRmi context local object space and see what is there.

To inspect the ATMServerRmi application context, execute the ATMServer context ATM Server class message, and inspect the results. This should display an SstApplicationContext inspector (Figure 65). Double-click on space instance variable to expand its contents which is an SstLocalSpace object. Next double click on the exportSet instance variable to expand its contents which is an SstDualRegistry object. Next double click on objects instance variable. This will expand its contents which is an IdentityDictionary object. The IdentityDictionary object should contain the ATMServer singleton object and the 098764 BankAccount object as illustrated in Figure 65.
Figure 65. ATMServerRmi Context Local Object Space Export Set
Chapter 11. Garbage Collection of Registered Objects

The distributed garbage collection algorithms used by Java and Smalltalk differ and do not work together. This poses a problem when objects that have been added to the RMI naming service registry and exported are no longer referenced by the client application. Take the ATM application for example. In our design, all BankAccount objects are unregistered when the user does one of the following two things:

1. Click on the Exit button in the ConfirmationPanel.
2. Click on the Cancel button in the TransactionPanel.

Our user interface is designed so that at some point the user’s interaction will result in the unregistering of all BankAccount objects. Suppose, however, that the client application is running in a Web browser and the user closes the Web browser through one of the above two actions. In this scenario, the BankAccount objects will remain in the RMI naming service registry run by the RMI naming context and the ATMServer application export set.

Not exactly an ideal situation. To overcome this problem, we provide some sample code and describe how to change the ATM server application to set timeout values for BankAccount objects.

11.1 Defining a Timer

The idea is to associate a timeout object with a BankAccount object when the BankAccount object is retrieved. When the timeout expires, the BankAccount would be removed from the RMI naming service registry and removed from the export set of the ATMServer application context.

We have already implemented the code necessary to remove a BankAccount object from the registry and export set. The additional code required to implement the timeout feature is a mechanism for creating the timeout objects and some code modifications to the ATM server application to set and remove the timeouts.

To implement the timer function, we have included an additional class in the RbiRmiSamplesAtm application. The name of the class is Timer. The Timer class implements two class methods: one to create a timeout and one to remove a timeout.

The method to create a timeout receives four parameters (Figure 66). The first parameter specifies the timeout duration in seconds. The second and
third parameter specify a receiver and selector. When the timeout elapses, the message specified by the selector is sent to the receiver. The fourth parameter specifies an object to be passed on the message send.

```
Smalltalk
addTimeout: interval
  receiver: receiver
  selector: selector
  clientData: clientData

  ^(Delay forSeconds: interval) wait.
  receiver perform: selector with: clientData.] fork
```

Figure 66. Timer addTimeout:receiver:selector:clientData: Method

The `addTimeout:receiver:selector:clientData:` method creates a `Delay` object with the specified timeout interval. The `wait` message is sent to the `Delay` object. When the wait completes, the message specified by the `selector` is sent to the `receiver`, with the `clientData` passed as a parameter. This code is wrapped in a block and forked, creating a `Process`. The `Process` is returned.

This method may appear familiar to you. It is also implemented by `CwAppContext` class. The reason we created an own method is that the Common Widget classes are not available when you package the application in XD, the cross-development environment of the Server Workbench.

Figure 67 shows the code to remove a timeout.

```
Smalltalk
removeTimeout: aProcess
  aProcess terminate
```

Figure 67. Timer removeTimeout: Method

The `removeTimeout:method` receives one parameter, a `Process`. The `terminate` message is sent to the passed `Process` to remove the timeout.

11.2 Integrating the Timer

To use the timeouts in the ATM server application, you have to modify the code. Here are the changes you have to make:
1. Add an instance variable called `accountTimers` to the `ATMServer` class. This variable is a `Dictionary` that holds the timeouts. The keys are account numbers.

2. Add a `setTimer:` instance method to the `ATMServer` class that creates a timeout. The timeout is added to the `accountTimers Dictionary`. The key for the timeout is the account number.

3. Add a `resetTimer:` instance method that removes the timeout for a `BankAccount` and creates a new timeout.

4. Modify the `getBankAccount:` method to create a timeout for the `BankAccount` being retrieved.

5. Modify the `unregisterBankAccount:` method to remove the timeout key from the `accountTimers Dictionary`.

Figure 68 shows the `accountTimers` getter method, the `setTimer:` method, and the `resetTimer:` method of the `ATMServer`.

```
accountTimers

accountTimers isNil
  ifTrue:[accountTimers := Dictionary new].
^accountTimers

setTimer: accountNumber

  self accountTimers at: accountNumber
  put: (Timer
    addTimeout: 20
    receiver: self
    selector: #unregisterBankAccount:
    clientData: accountNumber)

resetTimer: accountNumber

  Timer removeTimeout: (self accountTimers at: accountNumber).
  self setTimer: accountNumber
```

*Figure 68. ATMServer Timer Related Method*
Some other factors to consider are: what is the duration of the timeout, when the timeouts should be reset, and, most importantly, whether you notify the client that the registered objects no longer exist.
Chapter 12. Packaging the ATM Server Application

We use the cross-development (XD) environment of the VisualAge Smalltalk Enterprise Server Workbench feature to create and package the ATM server application as a headless and passive image; that is, the ATM server application has no user interface components and runs in the SST runtime environment.

12.1 Packaging Rules

Before we package the RbiRmiSamplesAtm application we have to add some packaging rules to include the necessary Java-to-Smalltalk mapping methods in the packaged image.

In addition, we have to add packaging rules to include the setter methods for the ATMCard fieldCardNumber and fieldPin attributes in the packaged image. ATMCard objects are passed by value from the Java ATM client to the ATM server. Therefore an ATMCard object is converted to a series of bytes in Java and passed to Smalltalk where SST marshals (converts) the bytes back into an ATMCard object. SST calls the setter methods for fieldCardNumber and fieldPin when marshaling an ATMCard object. We need packaging rules for these methods because there are no references to them in any of the classes in the RbiRmiSamplesAtm application, and the reduction step of the packager would remove them from the code.

Packaging rules are specified in class methods called packagingRulesFor: of the application class. To add the packaging rules for the ATM server application, we add the packagingRulesFor: class method to the RbiRmiSamplesAtm class. After creating the class method, we add the rules to include the following methods in the packaged application:

- #rbi_rmi_samples_atm_AccountOverdrawnException
- #rbi_rmi_samples_atm_ATMCard
- #rbi_rmi_samples_atm_ATMServer
- #rbi_rmi_samples_atm_ATMServer_Stub
- #rbi_rmi_samples_atm_BankAccount
- #rbi_rmi_samples_atm_BankAccount_Stub
- #rbi_rmi_samples_atm_CardValidationException
- #fieldCardNumber:
- #fieldPin:
The `packagingRulesFor:` method receives one parameter. To include a method in a packaged application, we send the `includeMethod:inClassNamed:` message to this parameter. Figure 69 shows the completed `packagingRulesFor:` class method.

```smalltalk
packagingRulesFor: aRuleCollector
    aRuleCollector
        includeMethod: #rbi_rmi_samples_atm_AccountOverdrawnException
            inClassNamed: #RbiRmiSamplesAtm;
        includeMethod: #rbi_rmi_samples_atm_ATMCard
            inClassNamed: #RbiRmiSamplesAtm;
        includeMethod: #rbi_rmi_samples_atm_ATMServer
            inClassNamed: #RbiRmiSamplesAtm;
        includeMethod: #rbi_rmi_samples_atm_ATMServer_Stub
            inClassNamed: #RbiRmiSamplesAtm;
        includeMethod: #rbi_rmi_samples_atm_BankAccount
            inClassNamed: #RbiRmiSamplesAtm;
        includeMethod: #rbi_rmi_samples_atm_BankAccount_Stub
            inClassNamed: #RbiRmiSamplesAtm;
        includeMethod: #rbi_rmi_samples_atm_CardValidationException
            inClassNamed: #RbiRmiSamplesAtm;
        includeMethod: #fieldCardNumber: inClassNamed: #ATMCard;
```

**Figure 69. RbiRmiSamplesAtm packagingRulesFor: Class Method**

### 12.2 Packaging Steps

To package the ATM server application, we create a passive image, load the SST components and the `RbiRmiSamplesAtm` application into the image, and package the image, using the Browse Packaged Images tool.

To create a passive image, we select **New Image...** from the XD menu choice. From the Image Properties window, we select the following options:

- Image Type: *Windows NT*
- Installed Features: *Server Smalltalk Framework (SST)*

Once we have created the passive image, we have to load the `RbiRmiSamplesAtm` application into the image. From the XD Transcript window for the newly created passive image, we launch an Application Manager by selecting **Tools->Manage Applications** from the XD Transcript window menu bar and load the `RbiRmiSamplesAtm` application.
To package the image, we launch the Browse Packaged Images tool by selecting **Tools->Browse Packaged Images** from the XD Transcript window menu bar. This will display the XD Packager Control Panel. The Packager Control Panel contains a notebook with three pages. Initially, the notebook is opened on the *Create New Instructions* page. On this page, we select the package image type. We will package the ATM server application as an *XD Single Image*.

After selecting the image type, we click on the **Modify Instructions** box to display another notebook in the XD Packager Control Panel. Initially the notebook is opened on the *Applications and ICs* page. We select the applications to be included in the packaged image. We select *RbiRmiSamplesAtm* and move it to the Selected Applications and ICs pane.

Next, we have to specify the application launch code. To specify the application launch code, we go to the *Startup Code* page of the *Modify Instructions* notebook. From the *Startup Code* page, we select *AbtHeadlessRuntimeStartUp* as the *Image Startup Class* and specify *ATMServer setup* as the *Application Entry Point*. The application entry point tells the packager which code should be executed when the image starts.

Now, we reduce the image, check and correct any errors, and output the image. The default image name of the packaged image is *singimg.icx*. You can specify another name or rename the image after it is written to disk.

**Note:** The packager does not create associated .ini and .exe files for the packaged image.

---

### 12.3 Deploying the ATM Client and Server Applications

There are several different configurations for deploying and running the client and server components of the ATM application. Of course the simplest approach is to run both the client and server applications from the VisualAge for Java and Smalltalk development environments.

A more typical deployment would be to install both the client and server components on the same machine on which your Web server is running.

To install the ATM Java client, we export the *RbiRmiSamplesAtm* package. Only the .class files are needed to run the application. The .class files can be exported to a directory or to a .jar file. We check the .html option to create an .html file to launch the *ATMApplet*. When we export to a directory, the export creates a subdirectory structure that matches *rbi/rmi/samples/atm*. We copy this subdirectory to the machine where our Web server is installed. The
subdirectory must be copied to a directory that is in the `CLASSPATH`. If we were to export the package to a .jar file, we would copy the .jar file to the Web server machine. Again, we would copy it to a directory that is in the `CLASSPATH`. Now we copy the .html file to the directory where our Web server looks for HTML files.

To install the ATM server, we first have to install the VisualAge Smalltalk Enterprise Server Runtime feature. We install the runtime feature on our Web server machine. After installing the runtime, we can install our packaged image.

We create the .exe and .ini files with the same name as our packaged image to start the ATM server by executing the .exe file. We specify the `-PORT:` command line parameter to change the port number to which the RMI Naming Service is listening. Without creating the .exe and .ini files for the packaged image, we can start the ATM server by executing `esvio -iIMAGENAME`, where `IMAGENAME` is the name of the packaged image.

To run the Java ATM client, we open a Web browser and specify the URL for our Web server and the .html file generated by the export operation.
Part 3. Building the Client
Chapter 13. Building the Proxy Objects

On the basis of our application design, we have to create proxies for the remote ATMServer and BankAccount objects. The proxies hold references to their remote objects and provide specialized behavior to assist in the communication between Java and Smalltalk.

13.1 ATMServerProxy

Using the SmartGuide, add the ATMServerProxy class to the rbi.rmi.samples.atm package. Uncheck the Compose the class visually option. Import the following packages:

- java.rmi
- java.util

13.1.1 Fields

The ATMServerProxy Java class has seven fields (instance variables). We use the BeanInfo page in the Class Browser to add as properties those fields, to which we have to make connections from other beans when we add the user interface later on (fieldAccountNumbers, fieldBankAccountProxy, fieldHostName, fieldNamingServicePort, and fieldLastError). For the other fields (atmServer and registered) we use the Create Field SmartGuide.

The ATMServerProxy fields we have to add are:

- atmServer - Holds the remote reference to the ATMServer singleton instantiated in Smalltalk. We need not make connections to this field so we will use the Create Field SmartGuide to add the atmServer field as property. Specify the field type as ATMServerIf. Lazy initialization will be used to set the field, so we have to make sure it is initialized to null. Here is the field definition:

  public ATMServerIf atmServer = null;

- fieldAccountNumbers - Holds the collection of account numbers by the ATMServer singleton after an ATMCard is verified. We have to make connections to this field so we will use the BeanInfo page to add the field as a property. Enter accountNumbers as the property name (remember VisualAge prefixes the property name with field to form the field name) and java.util.Vector as the property type. Here is the field definition:

  private Vector fieldAccountNumbers = new Vector();
• **fieldBankAccountProxy** - Holds the current BankAccountProxy object. Use the BeanInfo page to add the field as a property. Enter bankAccountProxy as the property name and BankAccountProxy as the property type. Here is the field definition:
  
  ```java
  private BankAccountProxy fieldBankAccountProxy;
  ```

• **fieldName** - Holds the host name of the machine running the ATM server. Use the BeanInfo page to add the field as a property. Enter hostName as the property name and String as the property type. Here is the field definition:
  
  ```java
  private String fieldName = new String();
  ```

• **registeredAccounts** - A hashtable that contains a list of remote references to BankAccount objects that have been retrieved from the RMI naming service registry. Use the Create Field SmartGuide to add the registeredAccounts field and specify the field type as Hashtable. Here is the definition:
  
  ```java
  private Hashtable registeredAccounts = new Hashtable();
  ```

• **fieldNamingServicePort** - Holds the port number on which the RMI naming service is listening. Use the BeanInfo page to add the field as a property. Enter namingServicePort as the property name and String as the property type. Here is the field definition:
  
  ```java
  private String fieldNamingServicePort = new String();
  ```

• **fieldLastError** - Holds the error message set after an exception is thrown. Use the BeanInfo page to add the field as a property. Enter lastError as the property name and String as the property type. Here is the field definition:
  
  ```java
  private String fieldLastError = new String();
  ```

### 13.1.2 Methods

Now let us look at the ATMServerProxy methods. For fields added as properties, VisualAge automatically generates the getter and setter methods. We list these methods for illustrative purposes. We also look at the methods we will need to communicate with the remote ATMServer singleton.

**Field Getter and Setter Methods**

The `getAtmServer()` method returns the remote reference to the ATMServer singleton. If `atmServer` is null, the `lookup(String)` message is sent to `java.rmi.Naming` to get the remote reference of the ATMServer singleton. If a `RemoteException` occurs, the `setLastError(String)` is invoked to set the error.
The getter and setter methods for the ATMServerProxy fields are shown in Figure 70, Figure 71, Figure 72, Figure 73, Figure 74, Figure 75, and Figure 76, respectively.

```
Java

/**
 * This method was created in VisualAge.
 * @return rbi.rmi.samples.atm.ATMServerIf
 */
public ATMServerIf getAtmServer() {
    if (atmServer == null) {
        try {
            atmServer = (ATMServerIf) Naming.lookup(getUrl() + "/ATMServer");
        } catch (Exception e) {
            setLastError("ATM Server unavailable " + e);
        }
    }
    return atmServer;
}

/**
 * This method was created in VisualAge.
 * @param aSever rbi.rmi.samples.atm.ATMServerIf
 */
public void setAtmServer(ATMServerIf anATMServer) {
    atmServer = anATMServer;
}
```

Figure 70. ATMServerProxy atmServer Getter and Setter Methods
Java

```java
/**
 * Gets the accountNumbers property (java.util.Vector) value.
 * @return The accountNumbers property value.
 * @see #setAccountNumbers
 */
public Vector getAccountNumbers() {
    return fieldAccountNumbers;
}

/**
 * Sets the accountNumbers property (java.util.Vector) value.
 * @param accountNumbers The new value for the property.
 * @see #getAccountNumbers
 */
public void setAccountNumbers(Vector accountNumbers) {
    Vector oldValue = fieldAccountNumbers;
    fieldAccountNumbers = accountNumbers;
    firePropertyChange("accountNumbers", oldValue, accountNumbers);
}
```

Figure 71. ATMServerProxy accountNumbers Getter and Setter Methods
Java

```java
/**
 * Gets the bankAccountProxy property (rbi.rmi.samples.atm.BankAccountProxy) value.
 * @return The bankAccountProxy property value.
 * @see #setBankAccountProxy
 */
public BankAccountProxy getBankAccountProxy() {
    return fieldBankAccountProxy;
}

/**
 * Sets the bankAccountProxy property (rbi.rmi.samples.atm.BankAccountProxy) value.
 * @param bankAccountProxy The new value for the property.
 * @see #getBankAccountProxy
 */
public void setBankAccountProxy(BankAccountProxy bankAccountProxy) {
    BankAccountProxy oldValue = fieldBankAccountProxy;
    fieldBankAccountProxy = bankAccountProxy;
    firePropertyChange("bankAccountProxy", oldValue, bankAccountProxy);
}
```

*Figure 72. ATMServerProxy bankAccountProxy Getter and Setter Methods*
```java

public String getHostName() {
    return fieldHostName;
}

public void setHostName(String hostName) {
    String oldValue = fieldHostName;
    fieldHostName = hostName;
    firePropertyChange("hostName", oldValue, hostName);
}
```

Figure 73. ATMServerProxy hostName Getter and Setter Methods

```java

public Hashtable getRegisteredAccounts() {
    return registeredAccounts;
}

public void setRegisteredAccounts(Hashtable aHashtable) {
    registeredAccounts = aHashtable;
}
```

Figure 74. ATMServerProxy registeredAccounts Getter and Setter Methods
Java

```java
/**
 * Gets the namingServicePort property (java.lang.String) value.
 * @return The namingServicePort property value.
 * @see #setNamingServicePort
 */
public String getNamingServicePort() {
    return fieldNamingServicePort;
}

/**
 * Sets the namingServicePort property (java.lang.String) value.
 * @param namingServicePort The new value for the property.
 * @see #getNamingServicePort
 */
public void setNamingServicePort(String namingServicePort) {
    String oldValue = fieldNamingServicePort;
    fieldNamingServicePort = namingServicePort;
    firePropertyChange("namingServicePort", oldValue, namingServicePort);
}
```

Figure 75. ATMServerProxy namingServicePort Getter and Setter Methods
Java

```java
/**
 * This method was created in VisualAge.
 * @return java.util.Hashtable
 */
public Hashtable getRegisteredAccounts() {
    return registeredAccounts;
}

/**
 * Sets the lastError property (java.lang.String) value.
 * @param lastError The new value for the property.
 * @see #getLastError
 */
public void setLastError(String lastError) {
    String oldValue = fieldLastError;
    fieldLastError = lastError;
    firePropertyChange("lastError", oldValue, lastError);
}
```

Figure 76. ATMServerProxy lastError Getter and Setter Methods

An ATMServerProxy object is a local representation of the remote ATMServer singleton. As such, the ATMServerProxy class has many of the same methods as the Smalltalk ATMServer class. Specifically, there are four common methods:

- `verifyCardAndGetAccountNumbers(ATMCard)`
- `getBankAccount(String)`
- `unregisterAllBankAccounts()`
- `restartAtm()`

Later, we have to make connections to these methods from other beans, so they must be in the ATMServerProxy bean interface. To create the methods and add them to the bean interface, we use the BeanInfo page.

To create a method, on the BeanInfo page:

- Select **New Method Feature...** from the Features pop-up menu.
- Enter the name of the method.
- Enter the return type for the method.
- Click on **Finish**.
When adding the `verifyCardAndGetAccountNumbers(ATMCard)` method as a method feature, specify the following values:

- **Method name** - `verifyCardAndGetAccountNumbers`
- **Return type** - `void`
- **Parameter count** - 1
- **Parameter name** - `anATMCard`
- **Parameter type** - `ATMCard`

When adding the `getBankAccount(String)` as a method feature, specify the following values:

- **Method name** - `getBankAccount`
- **Return type** - `void`
- **Parameter count** - 1
- **Parameter name** - `accountNumber`
- **Parameter type** - `String`

For the `unregisterBankAccounts()` and `restartAtm()` methods, there are no parameters, and the return type is void.

In addition to the above four methods, we also need to add the following method:

- **getUlr()**

Use the Create Method SmartGuide to add the `getUrl()` method. This method should return a `String`. 
**verifyCardAndGetAccountNumbers(ATMCard)**

The `verifyCardAndGetAccountNumbers(ATMCard)` method (Figure 77) is passed an `ATMCard` object and returns a `Vector` containing the numbers of the accounts accessible by the passed card. This method sends the `verifyCardAndGetAccountNumbers(ATMCard)` message to the remote `ATMServer` singleton. The returned account numbers are stored in `fieldAccountNumbers` field. On return, the `validCard` event is fired.

The `CardValidationException` and `RemoteException` exceptions are trapped, and the `setLastError(String)` method is invoked to set the error.

Figure 77 shows the `verifyCardAndGetAccountNumbers(ATMCard)` method.

```java
/**
 * This method was created in VisualAge.
 * @param anATMCard rbi.rmi.samples.atm.ATMCard
 */
public void verifyCardAndGetAccountNumbers(ATMCard anATMCard) {
    if (getAtmServer() != null) {
        try {
            setLastError("");
            setAccountNumbers(
                getAtmServer().verifyCardAndGetAccountNumbers(anATMCard));
            fireProcessValidCard(new ValidCardEvent(this));
        } catch (CardValidationException e) {
            setLastError(e.getMessage());
        } catch (RemoteException e) {
            setLastError("ATM Error Encountered "+ e);
        }
    }
    return;
}
```

*Figure 77. ATMServerProxy verifyCardAndGetAccountNumbers(ATMCard) Method*

**getBankAccount(String)**

The `getBankAccount(String)` method (Figure 78) is passed an account number and returns void. It first calls `getRegisteredAccounts(String)` to find out whether a `BankAccountProxy` object exists for the passed account number.

If the object exists it is retrieved and the `setBankAccountProxy(BankAccountProxy)` method is invoked to set the
BankAccountProxy. After setting the proxy, the accountRegistered event is fired.

If the BankAccountProxy does not exist for the passed account number, the getBankAccount(String) message is sent to the remote ATMServer singleton. On return, a new BankAccountProxy object is created, initialized, and added to the fieldRegisteredBankAccounts, and the accountRegistered event is fired.

The RemoteException exception is trapped, and the setError(String) method is invoked to set the error.

```java
/**
 * Perform the getBankAccount method.
 * @param accountNumber java.lang.String
 */
public void getBankAccount(String accountNumber) {
    if (getRegisteredAccounts().containsKey(accountNumber)) {
        setBankAccountProxy((BankAccountProxy) getRegisteredAccounts().get(accountNumber));
        fireProcessAccountRegistered(new AccountRegisteredEvent(this));
    } else
    try {
        getAtmServer().getBankAccount(accountNumber);
        BankAccountProxy bp = new BankAccountProxy();
        bp.setAtmServer(this);
        setBankAccountProxy(bp);
        getRegisteredAccounts().put(accountNumber, bp);
        fireProcessAccountRegistered(new AccountRegisteredEvent(this));
    } catch (RemoteException e) {
        setError("ATM Error Encountered " + e);
    }
    return;
}
```

Figure 78. ATMServerProxy getBankAccount(String) Method

**unregisterAllBankAccounts()**

The unregisterAllBankAccounts() method (Figure 79) is called to remove all BankAccount objects from the RMI naming service registry. This method sends the unregisterAllBankAccounts() message to the remote ATMServer singleton. On return, the fieldRegisteredBankAccounts instance variable is cleared.
The RemoteException exception is trapped and the method setLastError(String) called to set the error.

```java
/** *
 * Perform the unregisterAllBankAccounts method.
 */
public void unregisterAllBankAccounts() {
    if (!(getRegisteredAccounts().isEmpty())) {
        if (getAtmServer() != null) {
            try {
                getAtmServer().unregisterAllBankAccounts();
                getRegisteredAccounts().clear();
                catch (RemoteException e) {
                    setLastError("ATM Error Encountered " + e);
                }
            }
            return;
        }
    }
}
```

Figure 79. ATMServerProxy unregisterAllBankAccounts() Method

**restartAtm()**

The restartAtm() method (Figure 80) sends the restartAtm() message to the ATMServer singleton to unregister all BankAccount objects and reset the balances for the default BankAccount objects.

```java
/** *
 * Perform the restartAtm method.
 */
public void restartAtm() {
    try {
        getAtmServer().restartAtm();
    } catch (RemoteException e) {
        setLastError("ATM Error Encountered " + e);
        return;
    }
}
```

Figure 80. ATMServerProxy restartAtm() Method
**getUrl()**

The `getUrl()` method (Figure 81) returns a `String` containing the URL specification of the machine running the ATM server.

```java
/**
 * This method was created in VisualAge.
 * @return java.lang.String
 */
public String getUrl()
{
    String url = "//" + getHostName();
    if (getNamingServicePort().length() > 0)
    {
        url = url + ":" + getNamingServicePort();
    }
    return url;
}
```

Figure 81. ATMServerProxy getUrl() Method

### 13.1.3 Events

We have to create the `validCard` event fired by `verifyCardAndGetAccountNUMbers(String)` and the `accountRegistered` event fired by `getBankAccount(String)`. Like the other features added to the bean interface, we use the BeanInfo page to add the events.

To add the `validCard` event, open a Class Browser for the `ATMServerProxy` class and go to the BeanInfo page:

1. From the *Features* pop-up menu, select **New Listener Interface**... to open the New Event Listener SmartGuide.
2. Enter `validCard` as the *Event name*. The remaining fields are automatically filled in for you and require no action.
3. After entering the *Event name*, click on the **Next>** button to go to the Event Listener SmartGuide.
4. Enter `processValidCard` for the *Method Name* and click on the **Add** button. This is the method that other classes must implement in order to listen for the event.
5. Click on **Finish**.

Adding the `validCard` event creates three classes:

- ValidCardEvent class
- ValidCardEventMulticaster class
- **ValidCardListener** interface class

For the `accountRegistered` event, repeat the above steps, entering `accountRegistered` as the event name and `processAccountRegistered` as the method name.

### 13.1.4 Features

After you have added the properties, methods, and events to the `ATMServerProxy` bean interface, the `ATMServerProxy` BeanInfo page should resemble Figure 82.

---

**Figure 82. ATMServerProxy Features**
13.2 BankAccountProxy

Use the SmartGuide to add the BankAccountProxy class to the 
rbi.rmi.samples.atm package. Import the java.rmi package.

13.2.1 Fields

We have to define seven fields in the BankAccountProxy class. Like the 
ATMServerProxy, we will be making connections to some of the fields later on 
when we add the user interface. We will create these fields, using the 
BeanInfo page. The fields we have to add are:

- **fieldAccountNumber** - Holds the remote BankAccount object’s account 
  number. Use the BeanInfo page to add the field as a property. Enter 
  accountNumber as the property name and String as the property type. 
  Here is the field definition:

  private String fieldAccountNumber = new String();

- **fieldAccountType** - Holds the remote BankAccount object’s account type. 
  Use the BeanInfo page to add the field as a property. Enter accountType 
  as the property name and String as the property type. Here is the field 
  definition:

  private String fieldAccountType = new String();

- **fieldBalance** - Holds the remote BankAccount object’s balance. Use the 
  BeanInfo page to add the field as a property. Enter balance as the 
  property name and String as the property type. Here is the field 
  definition:

  private String fieldBalance = new String();

- **fieldCustomerId** - Holds the remote BankAccount object’s customerId. 
  This field is not used in this example; it is included for illustrative purposes 
  only. Use the BeanInfo page to add the field as a property. Enter 
  customerId as the property name and String as the property type. Here is 
  the field definition:

  private String fieldCustomerId = new String();

- **bankAccount** - Holds the reference to its remote BankAccount object 
  instantiated in Smalltalk. Use the Create Method SmartGuide to add the 
  bankAccount field and specify the field type as BankAccountIf. Lazy 
  initialization will be used to set the field so we need to make sure its set to 
  null. Here is the field definition:

  public ATMServerIf atmServer = null;
• **atmServer** - Holds the `ATMServerProxy` object. Use the Create Method SmartGuide to add the `atmServer` field. Specify the field type as `ATMServerProxy`. Here is the field definition:
  
  ```java
  public ATMServerProxy atmServer = null;
  ```

• **fieldLastError** - Holds the error message set after an exception is thrown. Use the BeanInfo page to add the field as a property. Enter `lastError` as the property name and `String` as the property type. Here is the field definition:

  ```java
  private String fieldLastError = new String();
  ```

### 13.2.2 Methods

All but one of the `BankAccountProxy` methods we will look at are getter and setter methods. As we have seen, VisualAge for Java automatically generates getter and setter methods for fields added as properties. These methods get and set fields belonging to the `BankAccountProxy` object. Some of the values we need, however, are held by the remote `BankAccount` object. Specifically, we need to obtain the `accountType` and `balance` from the remote object. For these specific values, we will modify the getter method to retrieve the value from the remote `BankAccount` object.

The only non-getter/setter method we need to implement in the proxy is `withdraw(String)`.

**accountNumber Getter and Setter Methods**

The `ATMServerProxy` object instantiates the `BankAccountProxy` class when the user selects an account number and a transaction. After creating the proxy object, the `ATMServerProxy` object stores the account number in the proxy. As a result, no changes need to be made to the `accountNumber` getter and setter methods (Figure 83).
When the `BankAccountProxy` object is created, the only information known about the remote `BankAccount` object is the account number. The accountType must be obtained from the remote `BankAccount` object. To accomplish this, we simply modify `getAccountType()` to forward the message to its remote `BankAccount` object (Figure 84). If a `RemoteException` occurs, `setLastError(String)` is called to set the error.

Later, when we add the user interface component, we will make connections to the `accountType` property. By doing this (and because we modified the getter method) we will ensure that the correct `accountType` is obtained whenever the connection fires.

Because the getter method obtains the `accountType` from the remote object, there is no need to store it locally. As such, the setter method is not used.

Figure 84 shows the `getAccountType()` method.
getBalance()
As we did with the getAccountType() method, we have to modify the
getBalance() method (Figure 85) to return the balance of the remote
BankAccount object. We make the same change and forward the message to
the remote BankAccount object. If a RemoteException occurs,
setLastError(String) is called to set the error.

As with accountType, the setter method is not used.

/*
* Gets the balance property (java.math.BigDecimal) value.
* @return The balance property value.
* @see #setBalance
*/
public String getBalance() {
    try {
        return(getBankAccount().getBalance());
    } catch (RemoteException e) {
        setLastError("ATM Error Encountered " + e);
    }
    return null;

Figure 85. BankAccountProxy balance Getter Method
**bankAccount Getter and Setter Methods**

The `getBankAccount()` method returns the remote reference to the remote `BankAccount` object.

If `bankAccount` is null, the `lookup(String)` message is sent to `java.rmi.Naming` to get the remote reference. The account number is used to look up the object. If a `RemoteExcpetion` occurs, `setLastError(String)` is called to set the error.

Figure 86 shows the bankAccount getter and setter methods.

```java
/**
 * This method was created in VisualAge.
 * @return rbi.rmi.samples.atm.BankAccountIf
 */
public BankAccountIf getBankAccount() {
    if (bankAccount == null) {
        try {
            setBankAccount((BankAccountIf)
                Naming.lookup(getAtmServer().getUrl() + "\" +
                getAccountNumber()));
        } catch (Exception e) {
            setLastError("ATM Error Encountered \" + e);
        }
    }
    return bankAccount;
}

/**
 * This method was created in VisualAge.
 * @param aBankAccount rbi.rmi.samples.atm.BankAccountIf
 */
public void setBankAccount(BankAccountIf aBankAccount) {
    bankAccount = aBankAccount;
}
```

*Figure 86. BankAccountProxy bankAccount Getter and Setter Methods*
**atmServer Getter and Setter Methods**

Figure 87 shows the *atmServer* getter and setter methods.

```java
/**
 * This method was created in VisualAge.
 * @return rbi.rmi.samples.atm.ATMServerProxy
 */
public ATMServerProxy getAtmServer() {
    return atmServer;
}

/**
 * This method was created in VisualAge.
 * @param anAtmServer rbi.rmi.samples.atm.ATMServerProxy
 */
public void setAtmServer(ATMServerProxy anATMServer) {
    atmServer = anATMServer;
}
```

*Figure 87. BankAccountProxy atmServer Getter and Setter Methods*
customerId Getter and Setter Methods

As previously mentioned, the customerId field is not used; its getter and setter methods are shown in Figure 88 for illustrative purposes only.

```
Java

/**
 * Gets the customerId property (java.lang.String) value.
 * @return The customerId property value.
 * @see #setCustomerId
 */
public String getCustomerId() {
    return fieldCustomerId;
}

/**
 * Sets the customerId property (java.lang.String) value.
 * @param customerId The new value for the property.
 * @see #getCustomerId
 */
public void setCustomerId(String customerId) {
    String oldValue = fieldCustomerId;
    fieldCustomerId = customerId;
    firePropertyChange("customerId", oldValue, customerId);
}
```

Figure 88. BankAccountProxy customerId Getter and Setter Methods
**lastError** Getter and Setter Methods

Figure 89 shows the lastError getter and setter methods.

```java
/**
 * Gets the lastError property (java.lang.String) value.
 * @return The lastError property value.
 * @see #setLastError
 */
public String getLastError() {
    return fieldLastError;
}

/**
 * Sets the lastError property (java.lang.String) value.
 * @param lastError The new value for the property.
 * @see #getLastError
 */
public void setLastError(String lastError) {
    String oldValue = fieldLastError;
    fieldLastError = lastError;
    firePropertyChange("lastError", oldValue, lastError);
}
```

*Figure 89. BankAccountProxy lastError Getter and Setter Methods*
**withdraw(String)**

The `withdraw(String)` method (Figure 90) is the only transaction-oriented method we will implement. Like the `getAccountType()` and `getBalance()` methods, it forwards the message to the remote `BankAccount` object for processing.

If the withdrawal results in a negative balance, the remote `BankAccount` object throws an `rbi.rmi.samples.atm.AccountOverdrawnException`. If this or a `RemoteException` occurs, `setLastError(String)` is called to set the error.

```java
/**
 * This method was created in VisualAge.
 * @param withdrawAmount java.lang.String
 */
public void withdraw(String withdrawAmount) {
    try {
        getBankAccount().withdraw(withdrawAmount);
    } catch (AccountOverdrawnException e) {
        setLastError(e.getMessage());
    } catch (RemoteException e) {
        setLastError("ATM error encountered " + e);
    }
    return;
}
```

_Figure 90. BankAccountProxy withdraw(String) Method_
13.2.3 Features

After you have added the properties and methods to the BankAccountProxy bean interface, the BankAccountProxy BeanInfo page should resemble Figure 91.

![BankAccountProxy Features](image)

Figure 91. BankAccountProxy Features
Chapter 14. Building the User Interface

We will build the ATM application user interface using Java’s Abstract Windowing Toolkit (AWT), to run in a Web browser. AWT provides a convenient mechanism for positioning and displaying the various visual components required by the ATM application.

AWT provides a set of Container classes that, as the name implies, can contain other components. Container objects can be nested, that is, they can include other Container objects. The Container we use to build and assemble the various parts of the ATM application user interface is the Panel class.

AWT provides a set of layout manager classes. These classes can be used to partition a Container into sections and control how components within the container are positioned. Nested containers have their own layout managers. The ATM application user interface will use the following layout manager classes:

- **BorderLayout** - divides a Container into five directional sections: East, West, North, South, and center. Components added to the container will be placed in one of the sections.

- **CardLayout** - stacks components, one on top of the other, like a deck of cards. Only one component is visible at a time. You can navigate through the components in order, either forward or backward, or go to a specific component in the stack.

- **GridLayout** - divides a Container into a number of rows and columns.

The ATM application user interface will have one main applet. This applet, which is itself a subclass of Container, will use a BorderLayout (Figure 92). We will place a label in the North section, a Panel in the center section, and another label in the South section. The labels will be used to display ATM instructions (North) and error information (South).

The Panels added to the applet’s center section will use a CardLayout. All ATM panels displayed to the user will be added to (stacked in) this Panel. The CardLayout will be used to switch from one Panel to another depending on input from the user.

### 14.1 ATMApplet

Add an Applet called ATMApplet to the rbi.rmi.samples.atm package. We will add two labels and another Panel to hold the other ATM panels. To set up the ATMApplet:
1. Open the *ATMApplet* class and switch to the Visual Composition page. The only bean that appears is an *Applet*.

2. Open the Properties dialog for the *Applet*. To open the Properties dialog, double-click on the Applet component. Change the *layout* to *BorderLayout*. Change the height to 290.

3. Add a *Label* bean to the North section. Open the bean’s Properties and change the *beanName* to *InformationLabel*. Set the text to an empty string.

4. Add a *Label* bean to the South section. Open the bean’s Properties and change the beanName to *ErrorLabel*. Set the text to an empty String.

5. Add a *Panel* bean to the center section. Open the bean’s Properties and change the beanName to *MainPanel*. Change the layout to *CardLayout*.

After we have added the above beans, the *ATMApplet* should resemble Figure 92.

![Figure 92. ATMApplet Beans](image)

After adding the *Label* beans and *Panel* bean to the *ATMApplet*, we have to promote some of their features. Promoting a bean’s feature exposes the feature by adding it to the composite bean’s interface. This will allow us to make connections to the feature from other composite beans.
To promote a feature, display the bean’s pop-up menu and select **Promote Bean Feature**. This displays the Promote features dialog where you can select the Property, Method, or Event for the bean that you want to promote.

Here are the bean features we want to promote:

- The `text` property of the `InformationLabel` bean
- The `this` property of the `MainPanel` bean
- The `text` property of the `ErrorLabel` bean

In addition to promoting the above features, we have to add a field to the `ATMApplet` to hold the host name of the machine running the ATM server application. We will need the host name to look up remote references to the `ATMServer` singleton and the `BankAccount` objects.

The `java.awt.Applet` class provides a method called `getCodeBase()` that returns the URL of the applet’s directory. The `getHostName()` message can be sent to the URL to get the host name. Because `java.awt.Applet` implements the `getCodeBase()` method, it makes sense to add the field to `ATMApplet`.

We will add the field as a property so that we can make a property-to-property connection with the `hostName` of the `ATMServerProxy`. To add the field, use the feature’s pop-up menu on the BeanInfo page. Enter `hostName` as the property name and `String` as the property type.

After you have promoted the bean features and added the `hostName` property, the `ATMApplet` BeanInfo page should resemble Figure 93.
Figure 93. ATMApplet Features
14.2 WelcomePanel

The WelcomePanel is the first Panel displayed in the ATMApplet (Figure 94). It has two entry fields. The first entry field allows the user to enter the host name of the machine running the ATM server application. When first displayed, the field contains the host name of the ATMApplet URL. The second field allows the user to enter the port number on which the RMI naming service is listening.

![WelcomePanel Beans](image)

To build the WelcomePanel:

1. Add a class called WelcomePanel to the rbi.rmi.samples.atm package. Specify java.awt.Panel as the superclass.
2. Add a Text bean to the Panel. Open the bean’s Properties and change the beanName to HostNameField.
3. Add a Label bean. Position the bean next to the HostNameField. Open the bean’s Properties and set the text to ‘Enter Server Address: ’.
4. Add a Text bean. Open the bean’s Properties and change the beanName to PortNumberField.
5. Add a Label bean. Position the bean next to the PortNumberField. Open the bean’s Properties and set the text to ‘Enter Naming Service Port:’.

6. Add a Button bean. Open the bean’s Properties and set the label to ‘Begin ATM Application’. Change the beanName to BeginATMAplicationButton.

7. Add a Variable bean to the free-form surface. Variable beans are in the Other category on the bean palette. Change the type from Object to ATMServerProxy. Open the bean’s Properties and change the beanName to ATMServerProxy.

8. Add a Variable bean to the free-form surface. Change the type from Object to CardLayout. Open the bean’s Properties and change the beanName to CardLayout.

9. Add a Variable bean to the free-form surface. Change the type from Object to ATMApplet. Open the bean’s Properties and change the beanName to ATMApplet.

To complete the WelcomePanel, we have to promote the following bean features:

- The this property of the ATMServerProxy
- The this property of the CardLayout
- The this property of the ATMApplet

After you have promoted the bean features, the WelcomePanel BeanInfo page should resemble the Figure 95.
Figure 95. WelcomePanel Features
14.3 CardPanel

The *CardPanel* allows the user to enter ATM card information (Figure 96). It has two entry fields, one for the card number and one for the PIN. It has two buttons, the **OK** button to process the ATM card and the **Cancel** button to cancel the operation.

![CardPanel Beans](image)

*Figure 96. CardPanel Beans*

To build the *CardPanel*:

1. Add a class called *CardPanel* to the `rbi.rmi.samples.atm` package. Specify `java.awt.Panel` as the superclass.

2. Add a **Text** bean to the **Panel**. Open the bean’s Properties and change the `beanName` to *CardNumberField*.

3. Add a **Label** bean. Position the bean next to the *CardNumberField*. Open the bean’s Properties and set the `text` to ‘Card Number’.

4. Add a **Text** bean. Open the bean’s Properties and change the `beanName` to *PINField*.

5. Add a **Label** bean. Position the bean next to the *PINField*. Open the bean’s Properties and set the text to ‘PIN’.

6. Add a **Button** bean. Open the bean’s Properties and set the label to ‘OK’.
7. Add a Button bean. Open the bean's Properties and set the label to 'Cancel'.

8. Add a Variable bean to the free-form surface. Change the type from Object to ATMServerProxy. Open the bean's Properties and change the beanName to ATMServerProxy.

9. Add a Variable bean to the free-form surface. Change the type from Object to CardLayout. Open the bean's Properties and change the beanName to CardLayout.

10. Add a Variable bean to the free-form surface. Change the type from Object to ATMApplet. Open the bean's Properties and change the beanName to ATMApplet.

To complete the CardPanel, we have to promote the following bean features:

- The this property of the ATMServerProxy
- The this property of the CardLayout
- The this property of the ATMApplet

After you have promoted the bean features, the CardPanel BeanInfo page should resemble Figure 97.

---

Figure 97. CardPanel Features
14.4 SelectionPanel

The SelectionPanel displays a list of accessible accounts (Figure 98). It prompts the user to select an account and a transaction to be performed against the account.

To build the SelectionPanel:

1. Add a class called SelectionPanel to the rbi.rmi.samples.atm package. Specify java.awt.Panel as the superclass.

2. Add a List bean to the Panel. Increase the width and height of the List bean to 195 by 160. Open the bean’s Properties and change the beanName to AccountList. We want the user to choose only one account, so make sure the multipleMode property is set to false.

3. Add another Panel bean to the existing Panel. Make the Panel bean the same size as the List bean. Note: You can use the Match Width and Match Height buttons on the toolbar to resize the Panel. Position the bean next to the AccountList. Open the bean’s Properties and change the
layout to GridLayout. Expand the layout property and change rows to 5 and vgap to 4. Change the beanName to TransactionListPanel.

4. Add five Button beans to the TransactionListPanel. As you add the beans, you will notice that they are positioned in a tabular format, because we specified the layout as GridLayout. A GridLayout partitions the container into rows and columns. We specified 5 rows and 0 columns, therefore each bean added to the Panel is added to a row. A space appears between the buttons because we specified a vertical gap (vgap).

5. Open the Properties for each button and change the labels, respectively top button to bottom button, to 'Withdraw', 'Deposit', 'Transfer', 'Balance', 'Cancel'.

6. Rename the button with the Withdraw label to WithdrawButton and the button with Cancel label to CancelButton.

7. We are going to implement only the withdraw function, therefore disable the buttons with Deposit, Transfer, and Balance labels. To disable a button, open its Properties and set the enabled property to false.

8. Add a Variable bean to the free-form surface. Change the type from Object to ATMServerProxy. Open the bean’s Properties and change the beanName to ATMserverProxy.

9. Add a Variable bean to the free-form surface. Change the type from Object to ATMServerProxy. Open the bean’s Properties and change the beanName to ATMServerProxy.

10. Add a Variable bean to the free-form surface. Change the type from Object to CardLayout. Open the bean’s Properties and change the beanName to CardLayout.

11. Add a Variable bean to the free-form surface. Change the type from Object to ATMApplet. Open the bean’s Properties and change the beanName to ATMApplet.

To complete the SelectionPanel, we have to promote the following bean features:
- The this property of the ATMServerProxy
- The this property of the CardLayout
- The this property of the ATMApplet

After you have promoted the bean features, the SelectionPanel BeanInfo page should resemble Figure 99.
Figure 99. SelectionPanel Features
14.5 AccountInfoPanel

The AccountInfoPanel is a reusable visual bean that displays the account number, account type, and balance for a BankAccount object (Figure 100). This bean is used in the TransactionPanel and ConfirmationPanel.

![AccountInfoPanel Beans](image)

Figure 100. AccountInfoPanel Beans

To build the AccountInfoPanel:

1. Add a class called AccountInfoPanel to the rbi.rmi.samples.atm package. Specify java.awt.Panel as the superclass.

2. Open the bean’s Properties and set the width to 273 and the height to 94.

3. Add a Label bean to the Panel. Position the bean in the upper left corner of the Panel. Open the bean’s Properties and set the text to ‘Account Number’

4. Add a Label bean. Position the bean to the right of the label whose text is set to Account Number. Open the bean’s Properties and set the text to an empty string. Change the beanName to AccountNumberLabel.

5. Add a Label bean. Position the bean below the label with its text set to Account Number. Open the bean’s Properties and set the text to an empty string. Change the beanName to AccountTypeLabel.

6. Add a Label bean. Position the bean below the AccountTypeLabel bean. Open the bean’s Properties and set the text to ‘Current Balance $’. 

7. Add a Label bean. Position the bean to the right of the label with its text set to Current Balance $. Open the bean’s Properties and set the text to an empty string. Change the beanName to AccountBalanceLabel.
To complete the AccountInfoPanel, we need to promote the following bean features:

- The text property of the ATMNumberLabel
- The text property of the AccountTypeLabel
- The text property of the AccountBalanceLabel

After you have promoted the bean features, the AccountInfoPanel BeanInfo page should resemble Figure 101.

Figure 101. AccountInfoPanel Features
14.6 TransactionPanel

The TransactionPanel displays account information and allows the user to enter a withdraw amount (Figure 102). This amount will be withdrawn from the current account when the user clicks on the OK button.

![TransactionPanel Beans](image)

Figure 102. TransactionPanel Beans

To build the TransactionPanel:

1. Add a class called TransactionPanel to the rbi.rmi.samples.atm package. Specify java.awt.Panel as the superclass.

2. Add the previously created AccountInfoPanel bean to the Panel bean.

3. Add a Label bean to the Panel. Position the bean below the AccountInfoPanel bean. Open the bean’s Properties and set the text to ‘Amount to Withdraw’.

4. Add a Text bean. Position the bean to the right of the label with its text set to Amount to Withdraw. Open the bean’s Properties and change the beanName to WithdrawAmountField.
5. Add a Button bean. Open the bean’s Properties and set the label to ‘OK’. Change the beanName to OKButton.

6. Add a Button bean. Open the bean’s Properties and set the label to ‘Cancel’. Change the beanName to CancelButton.

7. Add a Variable bean to the free-form surface. Change the type from Object to ATMServerProxy. Open the bean’s Properties and change the beanName to ATMServerProxy.

8. Add a Variable bean to the free-form surface. Change the type from Object to CardLayout. Open the bean’s Properties and change the beanName to CardLayout.

9. Add a Variable bean to the free-form surface. Change the type from Object to ATMApplet. Open the bean’s Properties and change the beanName to ATMApplet.

10. Tear off the bankAccountProxy property from the ATMServerProxy bean. Open the bean’s Properties and change the beanName to BankAccountProxy.

To complete the TransactionPanel, we have to promote the following bean features:

- The this property of the ATMServerProxy
- The this property of the CardLayout
- The this property of the ATMApplet

After you have added the bean features, the TransactionPanel BeanInfo page should resemble Figure 103.
Figure 103. TransactionPanel Features
14.7 ConfirmationPanel

The ConfirmationPanel displays the results of the withdraw transaction (Figure 104). It allows the user to choose to perform another transaction or exit.

![Diagram of ConfirmationPanel](image)

**Figure 104. ConfirmationPanel Beans**

To build the ConfirmationPanel:

1. Add a class called ConfirmationPanel to the rbi.rmi.samples.atm package. Specify java.awt.Panel as the superclass.
2. Add the AccountInfoPanel bean to the Panel bean.
3. Add a Button bean to the Panel. Open the bean’s Properties and set the label to ‘Another Transaction’. Change the beanName to AnotherTransactionButton.
4. Add a Button bean. Open the bean’s Properties and set the label to ‘Exit’. Change the beanName to ExitButton.
5. Add a Variable bean to the free-form surface. Change the type from Object to ATMServerProxy. Open the bean’s Properties and change the beanName to ATMServerProxy.

6. Add a Variable bean to the free-form surface. Change the type from Object to CardLayout. Open the bean’s Properties and change the beanName to CardLayout.

7. Add a Variable bean to the free-form surface. Change the type from Object to ATMApplet. Open the bean’s Properties and change the beanName to ATMApplet.

8. Tear off the bankAccountProxy property from the ATMServerProxy bean. Open the bean’s Properties and change the beanName to BankAccountProxy.

To complete the ConfirmationPanel, we have to promote the following bean features:

- The this property of the ATMServerProxy
- The this property of the CardLayout
- The this property of the ATMApplet

After you have promoted the bean features, the ConfirmationPanel BeanInfo page should resemble Figure 105.
Figure 105. ConfirmationPanel Features
14.8 ExitPanel

The ExitPanel displays a message thanking the user for choosing the XYZ bank (Figure 106). It allows the user to restart the ATM application.

To build the ExitPanel:
1. Add a class called ExitPanel to the rbi.rmi.samples.atm package. Specify java.awt.Panel as the superclass.
2. Add a Label bean to the Panel. Open the bean’s Properties and set the text to ‘Thank You for Choosing XYZ Bank!’.
3. Add a Button bean. Open the bean’s Properties and set the label to ‘Restart ATM’. Change the beanName to RestartATMButton.
4. Add a Variable bean to the free-form surface. Change the type from Object to ATMServerProxy. Open the bean’s Properties and change the beanName to ATMServerProxy.
5. Add a Variable bean to the free-form surface. Change the type from Object to CardLayout. Open the bean’s Properties and change the beanName to CardLayout.
6. Add a Variable bean to the free-form surface. Change the type from Object to ATMApplet. Open the bean’s Properties and change the beanName to ATMApplet.

To complete the ExitPanel, we have to promote the following bean features:

- The this property of the ATMServerProxy
- The this property of the CardLayout
- The this property of the ATMApplet

After you promoted the bean features, the ExitPanel BeanInfo page should resemble Figure 107.

![Figure 107. ExitPanel Features](image)

Figure 107. ExitPanel Features
Chapter 15. Putting the Pieces Together

The assembly process is almost completely visual—a model sample "par excellence" for building applications from parts with IBM's visual application builder technology. The assembly place is the free form surface in the VisualAge for Java tool. The root part—or in java terms: bean—is the applet bean with its MainPanel.

15.1 Adding the Panel Beans to ATMApplet

The MainPanel bean in the ATMApplet will be used to display all of the panels created for the ATM client application. To add the panel beans to the MainPanel in ATMApplet:

1. Open a Class Browser for ATMApplet and switch to the Visual Composition page.

2. Select the Choose Bean button on the parts palette to display the Choose Bean dialog. Specify Class for the Bean type. Enter rbi.rmi.samples.WelcomePanel for the Class name and click on the OK button.

3. Drop the bean on the MainPanel. After dropping the bean, you should see the WelcomePanel appear in the MainPanel as illustrated in Figure 108.

Figure 108. WelcomePanel Added to ATMApplet
Repeat the above steps for the following beans:

- CardPanel
- SelectionPanel
- TransactionPanel
- ExitPanel

15.2 Getting the URL

One of the things we need to know when the ATMApplet starts is the host name of the machine where the applet is running. When we created the ATMApplet class, we added a property called hostName. We added the property to ATMApplet because its superclass, java.awt.Applet, provides a method that returns the URL of the applet’s directory. And from this URL, we can get the host name.

When we added the hostName property, VisualAge for Java created a getter method called getHostName(). In its original state, this method simply returns the fieldHostName field. We need to modify this method to retrieve the host name from the applet’s URL. Figure 109 shows the changes we made.

```java
/**
 * Gets the hostName property (java.lang.String) value.
 * @return The hostName property value.
 * @see #setHostName
 */
public String getHostName() {
    if (fieldHostName == null) {
        fieldHostName = getCodeBase().getHost();
    }
    return fieldHostName;
}
```

*Figure 109. ATMApplet getHostName() Method*
15.3 ATMApplet Connections

Now it is time to start connecting all the pieces. First we have to add an ATMServerProxy bean to the free-form surface of the ATMApplet so that we will have an instance of this class when the ATMApplet starts (Figure 110).

![Diagram of ATMApplet Connections](image)

Figure 110. ATMApplet Connections

The ATMServerProxy needs the host name returned by the ATMApplet bean. To provide this information to the ATMServerProxy, we make a property-to-property connection between the hostName property of ATMApplet and the hostName property of ATMServerProxy we added to the free-form surface.

When we created the panels, we added three Variable beans to each panel. We changed the types of the three variable beans to ATMServerProxy, ATMApplet, and CardLayout. Variable beans are placeholders for objects.

Depending on the user interactions with the ATM panels, we have to send a message to the remote ATMserver singleton. Messages are sent to the ATMserver singleton by an ATMServerProxy object. As a result, each panel
needs an ATMServerProxy bean. Like the remote ATMServer object, there is only one ATMServerProxy object created by the Java ATM client. This is where the ATMServerProxy variable beans added to each panel come into play. The ATMServerProxy object is instantiated when the ATMApplet starts. This ATMServerProxy instance is passed to the other panels and stored in their ATMServerProxy variables. The ATMServerProxy instance will be passed to the panels with connections.

Previously, we added all the ATM panels to the MainPanel bean in the ATMApplet. The layout manager for the MainPanel bean is CardLayout. This class implements a number of methods that enable the navigation through the contents of the MainPanel bean. We use the MainPanel bean’s layout manager to switch from one panel to next. In order to accomplish this panel switching, each panel must access to this layout manager. The CardLayout of the MainPanel will be passed to each ATM panel and stored in its CardLayout variable. Again, this object is passed using connections.

The messages sent to the CardLayout object to switch from one panel to another require the parent of the Container as a parameter. The parent of MainPanel is ATMApplet, so we also need to make the ATMApplet object available to our panel beans.

To pass the ATMServerProxy object from the ATMApplet to our first panel, the WelcomePanel, we make a property-to-property connection between the this property of ATMServerProxy bean on the ATMApplet free-form surface and the ATMServerProxyThis property of the WelcomePanel. Remember, we created the ATMServerProxyThis property by promoting the this property of the ATMServerProxy variable on the WelcomePanel.

To pass the CardLayout object to the WelcomePanel, we make a property-to-property connection between the layout property of the MainPanel and the cardLayoutThis property of the WelcomePanel.

To pass the ATMApplet object to the WelcomePanel, we make a property-to-property connection between the this property of the ATMApplet and the ATMAppletThis property of the WelcomePanel.

After we have made the above four connections, the ATMApplet bean should resemble Figure 110.

We have to make the same property-to-property connections to pass the ATMServerProxy, CardLayout, and ATMApplet objects to the remaining panels.
We pass the `ATMServerProxy`, `CardLayout` and `ATMApplet` objects to the remaining panels, using the same property-to-property connections we made above. You may find it easier to make the connections from the `ATMApplet` Bean List rather than flipping through all the panel beans stored in the `MainPanel`.

After all the connections are complete, you should have a total of 19 connections in the Bean List. There are 3 connections per panel to pass the `ATMServerProxy`, `CardLayout`, and `ATMApplet` objects, and there are 6 Panel beans, making 18 connections. We end up with 19 connections because the first connection made was a property-to-property connection between the `hostName` of the `ATMServerProxy` and the `hostName` of the `ATMApplet`.

### 15.4 WelcomePanel Connections

The `WelcomePanel` (Figure 111) is the first panel displayed by the Java ATM client. It allows the user to specify the host name of the machine where the ATM server application is running and the port number on which the RMI naming service is listening.

![WelcomePanel Connections](Figure 111. WelcomePanel Connections)
To pass the hostname and port number to the `ATMServerProxy` object, we need to make the following connections:

- A property-to-property connection between the `hostName` property (the source) of the `ATMServerProxy` and the `text` property (the target) of the `HostNameField`. After making the connection, open the connection properties. The Target property should be text. Change the Target event from `<none>` to `textValueChanged` to ensure the connection fires when the user enters a value in the `HostNameField`.

- A property-to-property connection between the `namingServicePort` (source) of the `ATMServerProxy` and the `text` (target) of the `PortNumberField`. After you have made the connection, open the connection properties and change the Target event from `<none>` to `textValueChanged`.

To switch from the `WelcomePanel` to the `CardPanel`, we have to send the `next(Container)` message to the `CardLayout` object. The switch to the `CardPanel` occurs when the user clicks the `Begin ATM Application` button. The connections required to switch panels are:

- An event-to-method connection between the `actionPerformed` event of the `BeginATMApplicationButton` and the `next(Container)` method of the `CardLayout`. After we have made the connection, the connection arrow appears as a broken line, indicating the connection requires a parameter.

- To supply the parameter, connect the `mainPanel` property of the `ATMApplet` to the `parent` parameter of the connection.

The last two connections we have to make will clear the `ATMApplet errorLabel` and `informationLabel`. These properties are used by the ATM panels to display error and informational messages. These messages should be cleared if the `WelcomePanel` is redisplayed because a user has canceled a subsequent function. To clear the labels, we use event-to-property connections between the `componentShown` event of the `WelcomePanel` and the `text` property of the labels. Here are the connections we have to make:

- An event-to-property connection between the `componentShown` event of the `WelcomePanel` and the `errorLabelText` property of the `ATMApplet`. This connection requires a parameter. To supply the parameter, open the properties for the connection and click on the `Set parameters...` button. This will open a dialog where you can specify the `value` parameter. Set the parameter to an empty string.

- An event-to-property connection between the `componentShown` event of the `WelcomePanel` and the `informationLabelText` property of the
ATMApplet. This connection requires a parameter. Specify an empty string as the parameter.

15.5 CardPanel Connections

When the CardPanel (Figure 112) is initially displayed, we want to instruct the user to enter a PIN and card number. To do this, we will display a message in ATMApplet informationLabel. To display the message, make the following connection:

- An event-to-property connection between the componentShown event of the CardPanel and the informationLabelText of the ATMApplet. Open the connection properties and set the value parameter to the 'Enter PIN and Card Number' string.

Figure 112. CardPanel Connections

The CardPanel is used to input a PIN and card number. This information has to be stored in an ATMCARD object and passed to the ATMServer singleton on the verifyCardAndGetAccountNumbers(ATMCARD) message. To create the
ATMCard object, we have to add an ATMCard bean to the free-form surface of the CardPanel. When the ATMApplet starts, the ATMCard will be instantiated. We will use the following property-to-property connections to place the PIN and card number in the ATMCard object:

- A property-to-property connection between the text (source) of the CardNumberField and the cardNumber (target) of the ATMCard. Change the Source event from <none> to textValueChanged.
- A property-to-property connection between the text (source) of the PINField and the pin (target) of the ATMCard. Change the Source event from <none> to textValueChanged.

The CardPanel has two buttons, OK and Cancel. If the user clicks on the OK button, we need to send a message to the ATMServer singleton to verify the ATMCard object and return a list of the accessible accounts. To accomplish this, we have to make the following connections:

- An event-to-method connection between the actionPerformed event of the OKButton and the verifyCardAndGetAccountNumbers(ATMCard) method of the ATMServerProxy. This method requires an ATMCard object as a parameter. To pass the ATMCard object, connect the this property of the ATMCard bean to the anATMCard property of the event-to-method connection just made.

If the ATMCard object contains valid PIN and card numbers, the ATMServerProxy object fires the validCard event. This event will be used to switch from the CardPanel to the SelectionPanel. To switch panels, we have to make the following connections:

- An event-to-method connection between the validCard event of the ATMServerProxy and the next(Container) method of the CardLayout. This method requires a parameter. To supply the parameter, connect the mainPanel property of ATMApplet to the parent parameter of the connection.

If the user clicks on the Cancel button, we need to switch back to the WelcomePanel. To switch back, make the following connections:

- An event-to-method connection between the actionPerformed event of the CancelButton and the first(Container) method of the CardLayout.
- Connect the mainPanel property of ATMApplet to the parent parameter of the event-to-method connection.

Finally, we need to make sure error information from the ATMServerProxy object is displayed. To display this error information, make a
property-to-property connection between the lastError of ATMServerProxy and the errorLabelText of ATMApplet.

### 15.6 SelectionPanel Connections

When the SelectionPanel (Figure 113) is initially displayed, we want to instruct the user to select an account and transaction. To do this, we will display a message in the ATMApplet informationLabel. To display the message, make the following connection:

- An event-to-property connection between the componentShown event of the SelectionPanel and the informationLabelText of ATMApplet. Open the connection properties and set the value parameter to the ‘Select an Account and Transaction’ string.

![SelectionPanel Connections](image-url)
The SelectionPanel displays a list of accessible account numbers. The account numbers returned by the ATMServer singleton are stored in the accountNumbers property of the ATMServerProxy. Unfortunately, we cannot make a simple property-to-property connection between accountNumbers and the items property of the AccountList. The AccountList property of the SelectionPanel is a java.awt.List bean. The java.awt.List class implements a getItems() method but not a setItems(String[]) method. Therefore you can retrieve the array of Strings displayed in a List object but you cannot set the items with an array of Strings.

To workaround this problem, we have to add the account numbers individually. To this, we have to add a setItems(Vector) method (Figure 114) to the SelectionPanel class. This method is passed a Vector containing the account numbers. It loops through the contents of the Vector, sending the add(String) message to the AccountList.

```java
/**
 * Perform the setItems method.
 * @param accountNumbers java.util.Vector
 */
public void setItems(java.util.Vector accountNumbers) {
    /* Perform the setItems method. */
    int i = 0;
    while (i < accountNumbers.size())
    {
        getAccountList().add((String) accountNumbers.elementAt(i++));
    }
    return;
}
```

Figure 114. SelectionPanel setItems(Vector) Method

Because items are added to the AccountList individually, we have to remove any existing items before adding the new items. If we did not remove the existing items, we will end up with a growing list of account numbers after switching ATM cards. To clear the items, make the following connection:

- An event-to-method connection between the accountNumbers event of the ATMServerProxy and the removeAll() method of the AccountList.

After clearing the items, we can add the new account numbers to the AccountList. To invoke the setItems(Vector) method, make the following connections:
• An event-to-method connection between the accountNumbers event of the ATMServerProxy and the setItems(Vector) method of the SelectionPanel. This connection requires a parameter. Connect the accountNumbers property of the ATMServerProxy to the accountNumbers parameter of the connection.

Note: The sequence of the above connections is important. We want to make sure the items are not cleared as soon as they are added. So the connection to the removeAll() method must precede the connection to setItems(Vector). Reorder the connections if required.

When the user selects an account and clicks the Withdraw button, a message is sent to the ATMServer singleton to retrieve and register the BankAccount object. To send the getBankAccount(String) message to the ATMServer singleton, make the following connections:
• Make an event-to-code connection between the actionPerformed event of the WithdrawButton and the getBankAccount(String) method of the ATMServerProxy. Connect the selectedItem property of the AccountList to the accountNumber parameter of the connection.

The ATMServerProxy getBankAccount(String) method calls the ATMServer singleton to retrieve the BankAccount object. When control returns to the proxy, it fires the accountRegistered event. This event will be used to switch from the SelectionPanel to the TransactionPanel. To switch panels, make the following connections:
• An event-to-method connection between the accountRegistered event of the ATMServerProxy and the next(Container) method of the CardLayout. Connect the mainPanel property of the ATMApplet to the parent parameter of the connection.

If the user clicks on the Cancel button, we need to switch back to the CardPanel. To switch back, make the following connections:
• An event-to-method connection between the actionPerformed event of the CancelButton and the first(Container) method of the CardLayout. Connect the mainPanel property of ATMApplet to the parent parameter of the event-to-method connection.

In addition to switching back to the CardPanel, a message has to be sent to the ATMServer singelton to unregister all the BankAccount objects. As stated in the application design section, BankAccount objects remain instantiated, exported, and registered for the duration of ATM card processing. If the user clicks on the Cancel button, we assume no further transactions against
accounts for that ATM card. To unregister the BankAccount objects when the user clicks on the Cancel button, make the following connection:

- An event-to-method connection between the actionPerformed event of the CancelButton and the unregisterAllBankAccounts() method of the ATMServerProxy.

15.7 TransactionPanel Connections

When the TransactionPanel (Figure 115) is displayed, we want to instruct the user to enter a withdrawal amount. To do this, we will display a message in the ATMApplet informationLabel. To display the message, make the following connection:

- An event-to-property connection between the componentShown event of the SelectionPanel and the informationLabelText of ATMApplet. Open the connection properties and set the value parameter to the 'Enter Account to Withdraw' string.

![Figure 115. TransactionPanel Connections]
The TransactionPanel displays a BankAccount object’s accountNumber, accountType, and balance. If you remember, we added identical fields as properties to the BankAccountProxy class and modified the accountType and balance getter methods to retrieve the values from the associated remote BankAccount object. We have to make connections between these properties to display their values. To display the accountNumber and accountType, make the following connections:

- A property-to-property connection between the accountNumber of the BankAccountProxy and the accountNumberLabelText of the AccountInfoPanel. Remember, the reusable AccountInfoPanel bean was added to the TransactionPanel.
- A property-to-property connection between the accountType of the BankAccountProxy and the accountTypeLabelText of the AccountInfoPanel.

The property-to-property connections work fine for the accountNumber and accountType, but not for the balance. The BankAccountProxy bean is a Variable bean that holds a BankAccountProxy object. When the contents of the variable change, the connections from the variable are fired. When the user selects an account and transaction, the ATMServerProxy sets the BankAccountProxy object which causes the connections to fire. The first time the TransactionPanel is displayed, you will see the accountNumber, accountType, and balance. If, however, the user makes a withdraw and then returns to the SelectionPanel and chooses the same account, you will not see an updated balance when the TransactionPanel is redisplayed, because of the way the property changed events are fired. If the property changes but its new value is the same as the old, the event is not fired. Therefore, if a user reselects the same account number on the SelectionPanel, the property changed event will not signal when the ATMServerProxy sets the BankAccountProxy object. This is not a problem for the accountNumber and accountType because their values do not change. To work around the problem of the changing balance, we will force the balance getter method to be called every time the TransactionPanel is shown. To force the retrieval of the balance, make the following connection:

- An event-to-property connection between the componentShown event of the TransactionPanel and the balanceLabelText property of the AccountInfoPanel. This connection appears as a broken line indicating that it requires a parameter. To supply the parameter, connect the balance property of the BankAccountProxy to the value parameter of the connection.
When the user clicks on the **OK** button, we want to send the `withdraw(String)` message to the remote `BankAccount` object, switch to the next panel, and clear the `WithdrawAmountField`. To accomplish these tasks, make the following connections:

- An event-to-method connection between the `actionPerformed` event of the `OKButton` and the `withdraw(String)` method of the `BankAccountProxy`. Connect the `text` property of the `WithdrawAmountField` to the `withdrawAmount` parameter of the connection.

- An event-to-method connection between the `actionPerformed` event of the `OKButton` and the `next(Container)` method of the `CardLayout`. Connect the `mainPanel` property of the `ATMApplet` to the `parent` parameter of the connection.

- An event-to-property connection between the `actionPerformed` event of the `OKButton` and the `text` property of the `WithdrawAmountField`. Open the connection properties and set the parameter to an empty string.

If the user clicks on the **Cancel** button, we need to switch back to the `SelectionPanel`. To switch back, make the following connections:

- An event-to-method connection between the `actionPerformed` event of the `CancelButton` and the `previous(Container)` method of the `CardLayout`. Connect the `mainPanel` property of the `ATMApplet` to the `parent` parameter of the connection.

### 15.8 ConfirmationPanel Connections

When the `ConfirmationPanel` (Figure 116) is displayed, we want to clear the text in the `ATMApplet informationLabel`. To clear the text, make the following connection:

- An event-to-property connection between the `componentShown` event of the `ConfirmationPanel` and the `informationLabelText` of the `ATMApplet`. Open the connection properties and set the `value` parameter to an empty string.

The `ConfirmationPanel` displays a `BankAccount` object's `accountNumber`, `accountType`, and an updated `balance` resulting from the withdraw transaction. These values are obtained by calling the getter methods for these properties in the `BankAccountProxy` object. Because the `BankAccountProxy` is stored in a `Variable` bean and is unchanged since the `getBankAccount(String)` message was processed, the property-to-property connections made to the variable containing the `BankAccountProxy` object will not fire. This is similar to the problem we had earlier with the changing
balance in the `TransactionPanel`. As a result, we need a different approach than the property-to-property connections to get the values. To get the values, we use the same approach to get the value of the changing balance in the `TransactionPanel`. To get these values when the `ConfirmationPanel` is shown, make the following connections:

![ConfirmationPanel Connections](image)

- An event-to-property connection between the `componentShown` event of the `ConfirmationPanel` and the `accountNumberLabelText` property of the `AccountInfoPanel`. Connect the `accountNumber` property of the `BankAccountProxy` to the `value` parameter of the connection.

- An event-to-property connection between the `componentShown` event of the `ConfirmationPanel` and the `accountTypeLabelText` property of the `AccountInfoPanel`. Connect the `accountType` property of the `BankAccountProxy` to the `value` parameter of the connection.

- An event-to-property connection between the `componentShown` event of the `ConfirmationPanel` and the `balanceLabelText` property of the
AccountInfoPanel. Connect the balance property of the BankAccountProxy to the value parameter of the connection.

When the user clicks on the **Another Transaction** button, we have to redisplay the SelectionPanel. To redisplay the SelectionPanel, make the following connection:

- An event-to-method connection between the `actionPerformed` event of the AnotherTransactionButton and the `show(Container, String)` method of the CardLayout. This connection requires two parameters. The first parameter—`Container`—is the parent container and the second parameter—`String`—is the name of the panel. For the first parameter, connect the `mainPanel` property of the ATMApplet to the `parent` parameter of the connection. For the second parameter, open the connection properties and set the `name` parameter to the `String` ‘SelectionPanel’.

When the user clicks the **Exit** button, we have to display the ExitPanel and unregister all remote BankAccount objects. To switch to the ExitPanel and unregister the BankAccount objects, make the following connections:

- An event-to-method connection between the `actionPerformed` event of the ExitButton and the `last(Container)` method of the CardLayout. Connect the `mainPanel` property of the ATMApplet to the `parent` parameter of the connection.

- An event-to-method connection between the `actionPerformed` event of the ExitButton and the `unregisterAllBankAccounts()` method of ATMServerProxy.

There are two final connections that we need to make to ensure that error messages from the BankAccountProxy are displayed and cleared as appropriate. The connections are:

- A property-to-property connection between the `lastError` property of the BankAccountProxy and the `errorLabelText` property of the ATMApplet.

- An event-to-property connection between the `componentHidden` event of the ConfirmationPanel and the `lastError` property of the BankAccountProxy. We want to clear the last error message when the ConfirmationPanel is hidden, therefore open the connection properties and set the `value` parameter to an empty string.
15.9 ExitPanel Connections

The ExitPanel is the last panel displayed by the ATM application (Figure 117). It has one button that allows the user to restart the ATM application. When the ATM application is restarted, the default BankAccount objects are reset, and the WelcomePanel is redisplayed.

Figure 117. ExitPanel Connections

Here are the connections we have to make for the ExitPanel:

- An event-to-method connection between the actionPerformed event of the RestartATMButton and the first(Container) method of the CardLayout. Connect the mainPanel property of the ATMApplet to the parent parameter of the connection.

- An event-to-method connection between the actionPerformed event of the RestartATMButton and the restartAtm() method of ATMServerProxy.

Congratulations! You have now completed the ATM application!
15.10 Test the Client

Start the ATM server, start your ATM applet, and test with the dummy data, for example (Figure 41 on page 61 and Figure 42 on page 62):

Card Number 000000
Pin 0000
Account Number 098764
Withdraw 5000
Part 4. Adding Persistence
Chapter 16. Object Persistence in the ATM Application

The first iteration of the ATM application used a set of default objects for its data source. In this chapter, we will add an object persistence layer, using the VisualAge ObjectExtender tools to access stored data. All modifications required to add the persistence support will be made to the ATM server application; no changes will be made to the ATM client application.

16.1 Local Image Datastore and External Datastore

Again, we take an iterative approach to add object persistence:

One of the nice things about the ObjectExtender is that we do not actually need a database management system to implement persistence support. ObjectExtender provides a local image datastore that can be populated with persistent data. From an application point of view, there is no difference between a local image datastore and an external datastore. So we first implement the local image datastore that is immediately ready for use after defining the model and generating the model classes, the local image schema, and the (data access) services classes.

After adding the local image datastore, we can use our ATM Model (without any modifications) to generate a schema, map, (data access) services classes, and the DDL for an external datastore such as DB2.

16.2 SST, RMI, and ObjectExtender Limitations

The ATMCard class is defined in both Java and Smalltalk. ATMCard objects are instantiated by the ATM Java client and passed by-value to the ATM Smalltalk server. When an object is passed by-value, the sender basically converts the object to a series of bytes and passes them to the receiver. The receiver reassembles the bytes into an object. The transfer may include the object’s tail (like that of a comet), that is, all objects that are referenced by the initial object and not declared as transient.

Currently, both SST—and therefore RMI—and ObjectExtender lack the necessary functionality that would allow persistent objects to be passed by-value. Persistent objects are instantiated by sending a message to the object’s home collection. This presents a problem for the SST marshaler when reconstructing an object passed by-value and the object’s class is a business model class. On the ObjectExtender side, there is no function to transform an existing object into a persistent object. For example, suppose an
instance of the Java class `rbi.rmi.samples.atm.ATMCard` is created with a value of “111111” for the card number and “1111” for the PIN and then passed by-value to Smalltalk. SST receives the bytes, creates an instance of `ATMCard`, and fills in its instance data. SST does not instantiate the class by sending a message to the home collection. In addition, the data in the passed object may or may not exist in the datastore. If it does not exist, you want to raise an error condition (invalid card number or PIN). If the object’s data does exist in the datastore, there is no way of making the object persistent.

Given this limitation, we need to add another class that contains information similar to that contained in `ATMCard` objects. The Java ATM client will continue to pass `ATMCard` objects to the Smalltalk ATM server, but now the object data will be used as keys to find a persistent object.

Further impacts of the lack of distribution awareness of ObjectExtender and the lack of transaction awareness of RMI (or SST) are covered in Chapter 20, “Changes to the Model Classes” on page 169.
Chapter 17. Data Model

In order to add persistence support to the ATM application, we have to create a data model for the ATM application. If we model the current ATM datastore, we end up with three distinct entities: ATMCard, BankAccount, and Customer. As previously discussed, we cannot pass persistent objects by-value so we will need a new entity that represents the ATM cards. If we replace the ATMCard entity with a new entity called Card and diagram the relationships between the entities, we end up with the entity-relationship diagram shown in Figure 118.

![Figure 118. ATM Entity-Relationship Diagram](image-url)
17.1 Customer Entity

The Customer entity has three attributes:

- customerNumber
- firstName
- lastName

17.2 Card Entity

The Card entity has three attributes:

- cardNumber
- pin
- customerId

17.3 BankAccount Entity

The BankAccount entity has five attributes:

- accountNumber
- accountType
- balance
- cardNumber
- customerId

17.4 Relationships

The customerId property associates a Card with a Customer. A Customer can have many Cards, but a Card can belong to only one Customer.

The customerId property associates a BankAccount with a Customer. A Customer can have many BankAccounts, but a BankAccount can belong to only one Customer.

The cardNumber property associates a BankAccount with a Card. A Card can access many BankAccounts, but a BankAccount can be accessed by only one Card.
Chapter 18. Database Design

The entities defined in the entity-relationship diagram represent tables in the physical database (Figure 119). Therefore the ATM application will have three tables: Customer, Card, and BankAccount. Relationships between tables are established using referential constraints.

![Diagram of ATM database tables with referential constraints]

The last step in designing our ATM database is to assign data types and lengths to columns, determine whether the columns can be empty, and define the primary keys.
18.1 Customer Table

Table 7 lists the specifications for the Customer table columns.

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Length</th>
<th>Key</th>
<th>Nulls</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUSTOMERID</td>
<td>CHAR</td>
<td>4</td>
<td>Yes</td>
<td>No</td>
<td>Customer ID</td>
</tr>
<tr>
<td>FIRSTNAME</td>
<td>VARCHAR</td>
<td>30</td>
<td>No</td>
<td>No</td>
<td>First name</td>
</tr>
<tr>
<td>LASTNAME</td>
<td>VARCHAR</td>
<td>30</td>
<td>No</td>
<td>No</td>
<td>Last name</td>
</tr>
</tbody>
</table>

18.2 Card Table

Table 8 lists the specifications for the Card table columns.

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Length</th>
<th>Key</th>
<th>Nulls</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CARDNUMBER</td>
<td>CHAR</td>
<td>6</td>
<td>Yes</td>
<td>No</td>
<td>Card number</td>
</tr>
<tr>
<td>PIN</td>
<td>CHAR</td>
<td>4</td>
<td>No</td>
<td>No</td>
<td>PIN</td>
</tr>
<tr>
<td>CUSTOMERID</td>
<td>CHAR</td>
<td>4</td>
<td>No</td>
<td>No</td>
<td>Customer ID</td>
</tr>
</tbody>
</table>

18.3 BankAccount Table

Table 9 lists the specifications for the BankAccount table columns.

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Length</th>
<th>Key</th>
<th>Nulls</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACCOUNTNUMBER</td>
<td>CHAR</td>
<td>4</td>
<td>Yes</td>
<td>No</td>
<td>Account number</td>
</tr>
<tr>
<td>ACCOUNTTYPE</td>
<td>VARCHAR</td>
<td>8</td>
<td>No</td>
<td>No</td>
<td>Account type</td>
</tr>
<tr>
<td>BALANCE</td>
<td>DECIMAL</td>
<td>(10,2)</td>
<td>No</td>
<td>No</td>
<td>Balance</td>
</tr>
<tr>
<td>CUSTOMERID</td>
<td>CHAR</td>
<td>4</td>
<td>No</td>
<td>No</td>
<td>Customer ID</td>
</tr>
<tr>
<td>CARDNUMBER</td>
<td>CHAR</td>
<td>6</td>
<td>No</td>
<td>No</td>
<td>Card number</td>
</tr>
</tbody>
</table>
Chapter 19. Object Model for the Persistence

Object models are used to define persistent classes and the relationships among them. Based on our data model, the ATM application will have the following persistent classes and relationships (associations):

- **Customer**
- **Card**
- **BankAccount**
- **Card** to **Customer** relationship. Each Card belongs to one Customer. A Customer can have many Cards.
- **BankAccount** to **Customer** relationship. A BankAccount belongs to one Customer. A Customer can have many BankAccounts.
- **BankAccount** to **Card** relationship. A BankAccount is accessed by one Card. A Card can access multiple BankAccounts.

19.1 Defining the Model Classes and Attributes

Now that we have identified the persistent classes and their relationships, we can create an object model and define the model classes. To create the object model for the persistence:

- Select **Browse Models** from the **ObjectExtender Tools** menu choice on the System Transcript.
- Select **New Model...** from the **Models** pop-up menu.
- Enter **ATM** for the model name.

To add classes to the ATM Model, select ATM in the Models pane and then select **New Class...** from the **Model Classes** pop-up menu. This displays the Class Editor where you define a class and its characteristics.

To add the **Customer** class:

- Specify **Customer** for the class name. The Customer class has three attributes: customerId, firstName, and lastName. To add attributes to the class, go to the Attributes page in the Class Editor and click on the **New...** button.

The Customer attributes have the following values:

- **customerId** - type is String; value required is true. Each model class must have an object ID. Object IDs are used to retrieve objects from the datastore. Make customerId the object ID.
• **firstName** - type is **String**; value required is **true**.
• **lastName** - type is **String**; value required is **true**.

To add the **Card** class:

• Specify **Card** for the class name. According to our data model, the **Card** class has three attributes: **cardNumber**, **pin**, and **customerId**. Because we are creating persistent classes and the relationships between the classes, we will not need to define the **customerId** attribute. The link between a **Card** object and a **Customer** object will be defined using a class association.

• The **cardNumber** and **pin** attributes have the following values:
  • **cardNumber** - type is **String**; value required is **true**.
  • **pin** - type is **String**; value required is **true**.
  • Make **cardNumber** the object ID.

To add the **BankAccount** class:

• Specify **BankAccount** for the class name. The **BankAccount** class has five attributes: **accountNumber**, **accountType**, **balance**, **customerId**, and **cardNumber**. We only need to add the first three attributes; we will use class associations to define the links between the **BankAccount** object and **Card** and **Customer** objects.

  The **BankAccount** attributes have the following values:
  • **accountNumber** - type is **String**; value required is true. Make **accountNumber** the object ID.
  • **accountType** - type is **String**; value required is true.
  • **balance** - type is **ScaledDecimal**; value required is true.

### 19.2 Defining the Associations

At this point, there are no links between our model classes; if we have a **Card** object there is no way of getting its associated **Customer** object. On the base of our object model, we identified three separate relationships between our persistent classes. We will use these relationships to define class associations to establish the necessary links between our persistent classes.

To add an association, select **New Association**... from the **Class Associations** pop-up menu. This will display the Association Editor.
To define an association for the Card to Customer relationship:

- Specify Card-Customer as the association name.
- The Association Editor has two drop-down lists containing the classes eligible to participate in the association. Select Card from the left-hand list and Customer from the right-hand list.
- Specify customer for Role of Customer and card for Role of Card.
- Because a Customer can have more than one Card, check the Many box under Customer class (right-hand group box).

Figure 120 shows the completed Card-Customer class association.

![Association Editor](image)

Figure 120. Card-Customer Class Association

To define an association for the BankAccount to Customer relationship:

- Specify BankAccount-Customer as the association name.
- Select the BankAccount class from the left-hand drop-down list and the Customer class from the right-hand list.
- Specify customer for Role of Customer and bankAccount for Role of BankAccount.
- Because a Customer can have more than one BankAccount, check the Many box under Customer class (right-hand group box).
Figure 121 shows the completed BankAccount-Customer class association.

![Figure 121. BankAccount-Customer Class Association](image)

To define an association for the BankAccount to Card relationship:

- Specify BankAccount-Card as the association name.
- Select the BankAccount class from the left-hand drop-down list and the Card class from the right-hand list.
- Specify card for Role of Card and bankAccount for Role of BankAccount.
- Because a Card can access more than one BankAccount, check the Many box under Card class (right-hand group box).

Figure 122 shows the completed BankAccount-Card class association.
19.3 Reviewing the Model Classes

The completed Customer, Card, and BankAccount model classes should resemble those shown, respectively, in Figure 123, Figure 124, and Figure 125.
19.4 Saving the ATM Model

After defining the model classes and their associations, we have to save the ATM model. Models, schemas, and maps created with the ObjectExtender tools are stored in classes and applications when saved. To save the model, select **Save Model** from the **Models** pop-up menu, specify **ATMMetadataApp** for the storage application, and accept **ATMModel** for the storage class. **ATMMetadataApp** and **ATMModel** will be created, and the ATM Model will be stored in the latter.

Note: The **ATMMetadataApp** application was created when you saved the model is a Smalltalk application. Therefore it will not show up in the
19.5 Generating the Model Classes

At this point, the Card, Customer, and BankAccount classes added to the ATM Model exist only as definitions in the model itself. The persistent classes for these definitions will be generated using the ObjectExtender tools. A number of persistent classes will be generated by the tools including the Card, Customer, and BankAccount classes.

This presents a problem because the current ATM server application, RbiRmiSamplesAtm, already has a BankAccount class. So before we generate any of the persistent classes from our model, we have to create a new edition of the RbiRmiSamplesAtm application and delete the BankAccount class. After generating the BankAccount class with the ObjectExtender tools, we will have to go back and implement the withdraw:, getBalance, sstIsRmiRemotable, and sstRmiClassName methods.

Because we will be using a persistent datastore, we no longer need the default datastore provided by the ATMServer class. The following methods and instance variables can be deleted from the ATMServer class:

- methods - defaultCards, defaultCards:, defaultAccounts, defaultAccounts:
- instance variables - defaultCards, defaultAccounts

We also have to modify the verifyCardAndGetAccountNumbers: and getBankAccount: to use the new persistent datastore. We will look at these modifications Chapter 20, “Changes to the Model Classes” on page 169.

After deleting the BankAccount class from RbiRmiSamplesAtm, we can generate the persistent classes. To generate the classes, select the ATM model and then select Generate... from the Models pop-up menu. The generated classes will be added to an application called ATMMModelApp.

Generating the model classes produces the following kinds of classes:

- BusinessObject classes - created for each class defined in the model. The classes have accessors for each of the defined attributes and associations (relationships).
- HomeCollection classes - used to instantiate and perform lookups of the BusinessObject classes.
- Key classes - used to perform lookups of BusinessObjects using a key.
• *Relationship* classes - created for the associations defined in the model. These classes manage the relationships between BusinessObjects.
Chapter 20. Changes to the Model Classes

After generating the model classes, we need to add the RMI and ATM specific methods to them. Basically, they are the same methods added to the setters and getters for the sample without persistence, but some of them require modifications. Also, some of the generated methods may require modifications. The changes are required because of the new situation of combining RMI and persistence with transactions.

Assuming that a transaction is completed—it began and has either committed or rolled back—in one RMI, all methods belong to basically four groups:

1. Methods that are not affected by persistence are methods that simply have nothing to do with persistence, such as the RMI required method, \textit{sstRMIremoveClassName}.

2. Methods implicitly affected by persistence are methods that can run under the default—read-only and data caching \textit{SharedTransaction} of ObjectExtender’s persistence—when invoked directly with RMI, or that are part and only part of a sequence of methods in any kind of a transaction. The \textit{getBalance} method and all of the other generated model class methods belong to that group.

3. Methods locally affected by persistence are methods that perform a stand-alone and only a stand-alone business transaction requiring actual data from persistent storage or making and committing changes to persistent storage (beginning and committing their own read-write \textit{TopLevelTransaction}). An example would be the \textit{withdraw} method in the ATM sample application as currently implemented in this book.

4. Methods globally affected by persistence are methods that perform a stand-alone business transaction requiring actual data from persistent storage or making and committing changes to persistent storage (beginning and committing their own read-write \textit{TopLevelTransaction}, and are part of a sequence of methods in any kind of transaction in whose scope they must run. An example would be the \textit{withdraw}; if used as a stand-alone business transaction and as part of, for example, the \textit{transfer:to}: standalone-business transaction.

The methods in group 4 result from the conflict of a method owning the control over a read-write transaction or a method being directly or indirectly called from a method that owns the control over the actual read-write transaction. Remember that a transaction also defines the isolation policy, which affects the repeatable read aspect.
Actually, there is a fifth group of methods. These methods begin and commit a read-write transaction that must have its own transaction scope. For example, the logging of each (or only each failed) withdraw request. Note, however, that the logging of a successfully completed withdraw request must be part of the very withdraw transaction. You may argue that the backout of the already successfully completed withdraw request due to the rollback, which is caused by the log write, is not desirable. For the first cut this may be true, but it violates the transaction principle. Successfully completed requests are also successfully logged and vice versa, of course.

Methods of the group 3 and 4 require modifications. Methods in group 3 have to begin and end their own TopLevelTransaction, whereas methods in group 4 actually spawn into two methods:

1. A method that falls into group 3—invokable from a method in any circumstances
2. A method that falls into group 4.

The second consists of a transaction shell that calls the first one.

Furthermore we have to consider which methods are allowed to throw an RMI remote exception. Of course, the answer is: Only the method with (top-level) transaction ownership is allowed to throw an RMI exception, and it must first complete the transaction by either a commit or a rollback. All other methods must either return an error or fire an exception caught by the server or application environment, which must in turn first handle the application situation and then throw its own RMI exception.

### 20.1 Methods Not Affected by Persistence

The \texttt{sstIsRmiRemotable} method of the \texttt{BankAccount} class (Figure 126) has nothing to do with persistence and is unchanged from the nonpersistent version (Figure 126).

```smalltalk
sstIsRmiRemotable
^true
```

*Figure 126. BankAccount sstRmiRemotable Method*
20.2 Methods Implicitly Affected by Persistence

The `getBalance` method of the `BankAccount` class (Figure 127) is—in the sample application as implemented—only implicitly affected by persistence and is unchanged from the nonpersistent version of the `BankAccount` class. We could argue that the balance has to be actual. We would then move the `getBalance` method into the group of locally affected methods (group 3).

```
Smalltalk
getBalance

" Convert the balance from a ScaledDecimal to a String and return it "

| stream balString index retString |

stream := WriteStream on: (String new: 10).
balString := stream contents.

retString := ".
((balString at: 1) = $-)
  ifTrue: [ index := 2.
       retString := '.']
  ifFalse: [ index := 1 ].
[(balString at: index) = $0]
  whileTrue: [index := index + 1].
^retString, (balString copyFrom: index to: balString size).
```

Figure 127. BankAccount getBalance Method

20.3 Methods Locally Affected by Persistence

The `withdraw` method of the `BankAccount` class—as implemented in the ATM sample application—can belong to this group of locally affected methods because it is directly called only by RMI. If the sample application were extended with a `transfer:to:`, the `withdraw` would certainly fall within the group of globally affected methods (group 4), and the server and the client would then be as described in 3.3, “Withdraw Transaction Detail” on page 14. Furthermore—if we take `withdraw` as a locally impacted method—we do not have to change the client.

The modification required for locally affected methods is relatively trivial. We simply have to create a transaction before the crucial passage of the method and perform a commit after it.
For the actual `withdraw:` method we have to update the balance within a transaction. So we simply have to create and start a transaction before updating the balance and then commit the transaction after the update. Figure 128 shows the `withdraw:` method with transactional support. Note that the RMI exception is thrown after the transaction ends.

**Smalltalk**

```smalltalk
withdraw: withdrawAmount

" Perform the withdrawal. Convert the withdrawAmount to a ScaledDecimal and subtract it from the existing balance. If the result balance is negative, throw the Java exception rbi.rmi.samples.atm.AccountOverdrawnException."

| trans |
trans := Transaction begin: 'withdraw'.
self balance: self balance - withdrawAmount asScaledDecimal.
trans commit.
self balance < 0
ifTrue: [SstRmiDetailedException
throw: 'rbi.rmi.samples.atm.AccountOverdrawnException'
message: 'Account ', self accountNumber, ' is Overdrawn']
```

*Figure 128. BankAccount Class withdraw: Method with Persistence*

The transaction may throw a business-relevant exception but the method will not catch it or promote it to the client. Instead SST catches the exception and promotes it as a generic remote RMI exception, as SST does with any other exception in an RMI. To pass context information to the ATM user, the method should catch exceptions, figure out the reason, and throw an exception similar to the `rbi.rmi.samples.atm.AccountOverdrawnException`.

### 20.4 Methods Globally Affected by Persistence

Globally affected methods must be split up into a primary method that is usable under any circumstances and a shell method that cares about persistence and has as its core the invocation of the primary method. Accordingly, there are two major options for naming these methods:

1. The primary method gets the `prim` prefix.
2. The shell method gets either the `shell` prefix, the purpose-oriented `transactionally` prefix, or the `AsTransaction` suffix.

Option 2 is more natural, because it leaves the original business model name, for example, `withdraw:`, unchanged and adds "things", such as prefixes or
suffixes, to the additional methods, which have to be added anyway. Therefore the additional methods will be called `transactionallyWithdraw:` or `withdrawAsTransaction:`.

Because we could handle the one and only potential candidate, `withdraw:`, in the group of the locally affected methods, the sample application has no globally affected methods.
Chapter 21. Local Image Datastore

The local image datastore requires no data modeling or database design. When persisted, the datastore—actually the contents of the one and only service object serving the datastore—is dumped as a whole and in binary format into a file and is loaded as a whole from a file. (The service object of all local image datastores is the ImageServiceObject class and responds to the save, load, and reset protocol; save and load ask through the standard system file dialog for a file, and reset clears all data in the service object and all instances in the local image datastore.)

21.1 Generating the Local Image Datastore

To generate the local image data store, select the ATM model and then select Generate Image Schema from the Models pop-up menu. Actually, the local image schema generation does not generate a schema or a map—as it would for an external datastore. Instead it directly generates the services classes and the datastore class, which would be generated in an extra step from a schema for a database. The services classes and the datastore class are added to a newly created application called ATMServicesApp.

Generating the services classes produces the following kinds of classes:

- A Datastore class - manages all service classes for this particular datastore. If the datastore accesses data in a database management system, it also manages the database connections.
- ServiceObject classes - provide the necessary services to create, update, and delete DataObjects associated with a BusinessObject.
- DataObject classes - hold the actual data for a BusinessObject.

21.2 Populating the Local Image Datastore

To populate the local image datastore, you have to create Customer, Card, and BankAccount objects and link them together. When we created the ATM Model, we added the following associations:

- Card-Customer
- BankAccount-Card
- BankAccount-Customer
These associations establish links between the classes. The Card class has accessor methods for Customer objects. The BankAccount class has accessor methods for Customer and Card objects.

To connect the objects:

• You associate a Customer object with a Card object.
• You associate a Customer object with a BankAccount object.
• You associate a Card object with a BankAccount object.

Figure 129 lists some sample code to create a Customer object, a Card object, and two BankAccount objects.

```smalltalk
| customer card |
ATMDataStore singleton activate.
Transaction begin.
customer := (CustomerHome singleton create)
customerId: '0000';
firstName: 'Johnny';
lastName: 'Appleseed'.
card := (CardHome singleton create)
cardNumber: '000000';
pin: '0000';
customer: customer.

(BankAccountHome singleton create)
accountNumber: '098764';
accountType: 'Checking';
balance: ('5678.55' asScaledDecimal);
customer: customer;
card: card.

(BankAccountHome singleton create)
accountNumber: '023745';
accountType: 'Savings';
balance: ('5045.03' asScaledDecimal);
customer: customer;
card: card.
Transaction current commit.
```

Figure 129. Code to Populate the Persistent Datastore

The Customer object is associated with the Card object and both BankAccount objects. The Card object is associated with both BankAccount objects. The entire code sequence occurs within the scope of a transaction.
After this code is executed, you can retrieve BankAccounts 098764 and 023745, using card number 000000.

To populate the datastore with additional Customer, Card, and BankAccount objects, you simply repeat the sequence of steps in the code.

21.3 Changes to the ATMServer Class

As previously mentioned, we have to modify the verifyCardAndGetAccountNumbers: and getBankAccount: methods to use the new local image datastore. To make the actual changes the same considerations apply as described in Chapter 20., “Changes to the Model Classes” on page 169.

The verifyCardAndGetAccountNumbers: method (Figure 130) receives an ATMCard object passed by the ATM Java client application. The cardNumber contained in the ATMCard object will be used to find a Card object in the local image datastore. If no Card object can be found with the passed cardNumber, a CardValidationException is thrown with the ‘Invalid Card Number’ message. If a Card object is found, a check is made to determine whether its PIN matches that of the ATMCard object. If the PINs do not match, a CardValidationException is thrown with the ‘Invalid PIN’ message.

---

Smalltalk

verifyCardAndGetAccountNumbers: anATMCard

" Verify the PIN and card number for anATMCard. Throw an exception for invalid PIN and card numbers. If anATMCard is valid, return a list of accounts accessible by anATMCard. "

| card |

card := CardHome singleton findByCardNumber: anATMCard fieldCardNumber.
card isNil ifTrue: [SstRmiDetailedException throw: 'rbi.rmi.samples.atm(CardValidationException' message: 'Invalid Card Number'].
card pin = anATMCard fieldPin ifFalse: [SstRmiDetailedException throw: 'rbi.rmi.samples.atm(CardValidationException' message: 'Invalid PIN'].
^card bankAccount collect: [:ea | ea accountNumber]

Figure 130. ATMServer verifyCardAndGetAccountNumbers: Method for Persistence

Local Image Datastore 177
If the *ATMCARD* is valid, a list of account numbers accessible by the persistent *Card* object are returned. When we created the ATM Object Model, we added an association between a *Card* and a *BankAccount*. This association creates a link between the two classes. This link is used to retrieve *BankAccount* objects associated with a *Card* object.

The *getBankAccount:* method (Figure 131) is passed an *accountNumber*. This *accountNumber* is used to find a *BankAccount* object in the local image datastore. The persistent *BankAccount* object is added to the RMI naming service registry and the list of registered accounts.

```smalltalk
getBankAccount: accountNumber

" Check to see if the BankAccount object is registered. If not, retrieve it, add it to the RMI naming service registry and add it to the list of registered accounts. "

| account |

(self registeredBankAccounts includesKey: accountNumber)
ifFalse: [ account := BankAccountHome singleton
findByAccountNumber: accountNumber.
self class bind: account as: accountNumber.
sel registeredBankAccounts at: accountNumber put: account]
```

*Figure 131. ATMServer getBankAccount: Method for Persistence*

The last modification of the *ATMServer* class is to activate the datastore when the ATM server is started and reset it when the server is shut down. We will activate the datastore in the *setup* class method and reset it in the *shutDown* class method (Figure 132 and Figure 133).
21.4 Testing the ATM Application with the Local Image Datastore

After creating and generating the model classes, generating the service classes, making the necessary modifications to the BankAccount and ATMServer classes, and populating the datastore, you should be able to run the ATM application as before.
Chapter 22. External Datastore

Now we will add an external datastore to our ATM application. There are several approaches to creating the external datastore and the persistent classes required to access it. One approach would be to build an external database and import the schema into VisualAge Smalltalk using the ObjectExtender tools. After importing the schema, we could generate a model and map from the schema and then use the model and map to generate the model and services classes.

Another approach would be to take the existing ATM model and use it to generate a schema and map. The schema could then be modified to match the table definitions in Chapter 18, “Database Design” on page 159. From the schema, we can generate the DDL necessary to create the Card, Customer, and BankAccount tables in a database. Finally, the map could be used to generate the service classes required to access data stored in the external database.

We use the latter approach. The new model and service classes will be generated into the existing ATMModelApp and ATMServicesApp applications. This will allow access to either the local image datastore or the external datastore, depending on which versions of the ATMModelApp and ATMServicesApp applications are loaded.

22.1 Generating the Database Schema

To generate the schema from the ATM model, select ATM in the Models pane of the Model Browser and then select Generate schema from model from the Models pop-up menu. Generating the schema from the ATM model creates a schema named ATM and a map named ATMAtm.

After generating the ATM schema, we have to modify the table definitions to match those listed in Chapter 18, “Database Design” on page 159. To edit the table definitions, open a Schema Browser from the ObjectExtender tools and select the ATM schema. Select the table you want to edit and then select Edit Table...

Make the following modifications to the BankAccount table definition:

- Change the qualifier from vap to the qualifier of your choice. We chose rbi.
- Change the type and length of accountNumber to CHAR(6) and the converter to VapTrimStringConverter. To make the changes, select accountNumber in the Table columns group box and click on the Edit...
push button. This will display the Column Editor where you can make the necessary changes.

• Change the type and length of accountType to VARCHAR(8) and the converter to VapTrimStringConverter.

• Change the length and precision of balance to 10,2.

Make the following modifications to the Card table definition:

• Change the qualifier from vap to the qualifier of your choice. We chose rbi.

• Change the type and length of cardNumber to CHAR(6) and the converter to VapTrimStringConverter.

• Change the type and length of pin to CHAR(4) and the converter to VapTrimStringConverter.

Make the following modifications to the Customer table definition:

• Change the qualifier from vap to the qualifier of your choice. We chose rbi.

• Change the type and length of customerId to CHAR(4) and the converter to VapTrimStringConverter.

• Change the length of firstName to 30 and the converter to VapTrimStringConverter.

• Change the length of lastName to 30 and the converter to VapTrimStringConverter.

You may have noticed that the BankAccount table has column definitions for Card_cardNumber and Customer_customerId and the Card table has a column definition for Customer_customerId. These column definitions were added as a result of the associations defined in the ATM model. They are used in foreign key relationships. The ATM schema has three foreign key relationships:

• BankAccount BankAccount-Card Card
• BankAccount BankAccount-Customer Customer
• Card Card-Customer Customer

As previously mentioned, we are going to generate DDL from the ATM schema and use it to create tables in a database. The foreign key relationships will be used to generate foreign key constraints. Each foreign key relationship has a name and a physical name. The physical name is used to construct the constraint name in the generated DDL. If the physical name is omitted, the logical name is used. Before generating the DDL, we have to specify a physical name for each relationship to ensure that the constraint names in the generated DDL do not exceed any maximum limit set by the
database management system. In the case of DB2, the maximum length of a constraint name is 18 characters.

Specify the following physical names in the foreign key relationships:

- For the BankAccount BankAccount-Card Card relationship, specify BANKACCOUNT_CARD as the physical name. Make sure that the Constraint exists in database box is checked.
  
  **Note:** To edit the relationship, select BankAccount BankAccount-Card Card in the Foreign Key Relationships pane and select **Edit Foreign Key Relationship...** from the pop-up menu.

- For the Account BankAccount-Customer Customer relationship, specify BANKACCOU_CUSTOMER as the physical name. Make sure that the Constraint exists in database box is checked.

- For the Card Card-Customer Customer relationship, specify CARD_CUSTOMER as the physical name. Make sure that the Constraint exists in database box is checked.

After making the above modifications, save the ATM schema. When prompted, specify ATMMetaDataApp as the storage application.

Now, we are ready to generate DDL from the ATM Schema. To generate the DDL, select the ATM schema and then select **Generate DDL script for Schema Creation** from the Schemas pop-up menu. The generated DDL should appear as shown in Figure 134.
After reviewing the DDL, create the tables in DB2. Either copy the DDL and paste it into DB2’s command center—where you interactively execute it, or select Import / Export->Export entire Schema to the Database to let ObjectExtender create directly the tables.
22.2 Mapping the Classes, Attributes, and Associations

The last step in adding an external datastore is to map the tables to persistent classes. A map called ATMAtm was automatically created when we generated the ATM schema from the ATM model. To map the tables to persistent classes, open a Map Browser from the ObjectExtender tools and select the ATMAtm map. The BankAccount, Card, and Customer classes should appear in the Persistent Classes pane. Each persistent class has a table associated with it, and we have to map the attributes to table columns.

To map the BankAccount attributes to the BankAccount table columns, select the BankAccount class and BankAccount table map and then select Edit Property Maps... from the Table Maps pop-up menu. The BankAccount class should have the following attribute and association mappings:

- The accountNumber attribute is mapped to the accountNumber column. The map type is Simple.
- The accountType attribute is mapped to the accountType column. The map type is Simple.
- The balance attribute is mapped to the balance column. The map type is Simple.
- The card association is mapped to the BankAccount-BankAccount-Card foreign key relationship.
- The customer association is mapped to the BankAccount-BankAccount-Customer foreign key relationship.

The Card class should have the following attribute and association mappings:

- The cardNumber attribute is mapped to the cardNumber column. The map type is Simple.
- The pin attribute is mapped to the pin column. The map type is Simple.
- The bankAccount association is mapped to the BankAccount-BankAccount-Card foreign key relationship.
- The customer association is mapped to the Card-Card-Customer Customer foreign key relationship.

The Customer class should have the following attribute and association mappings:

- The customerId attribute is mapped to the customerId column. The map type is Simple.
The `firstName` attribute is mapped to the `firstName` column. The map type is `Simple`.

The `lastName` attribute is mapped to the `lastName` column. The map type is `Simple`.

The `bankAccount` association is mapped to the `BankAccount BankAccount-Customer Customer` foreign key relationship.

The `card` association is mapped to the `Card Card-Customer Customer` foreign key relationship.

After completing the mappings, we can generate the service and model classes required to access the external datastore. The new service classes and the updated model classes will be stored in the `ATMServiceApp` and `ATMModelApp` applications. These applications currently contain the local image versions of these classes. Our plan is to have two editions of these applications; one for the local image datastore and one for the external datastore. Before we generate the service and model classes, we have to version these applications.

After versioning the `ATMServiceApp` and `ATMModelApp` applications for local image access, create open editions of the applications. Previously, the service class names for local image access were prefixed with `ATM`. When we generate the service classes from the `ATMAtm` map, the service class names will now be prefixed with `ATMAtm`. Before generating the new service classes, you may want to delete the existing classes from the `ATMServiceApp` application.

Before generating the service classes, we have to set the generation options. To set the generation options, select `ATMAtm` map in the Datastore Maps pane and then select `Generation Options...` from the pop-up menu. Specify `ATMServiceApp` as the service application name and the desired database connection information.

To generate the services classes, select `ATMAtm` map in the Datastore Maps pane and then select `Generate Services` from the pop-up menu. When the generation completes, the `ATMServiceApp` application will contain all new service classes. In addition, some of the model classes will have been updated.

One of the changes made to the `ATMServer` class was to activate the datastore on setup (`ATMDataStore singleton activate`) and reset the datastore (`ATMDataStore reset`) on shutdown. The newly created datastore class, `ATMAtmDataStore`, is different from the local image datastore class. Our goal is to load different versions of the service and model applications to access...
different datastores. To eliminate the need for making additional changes to
the ATMServer class, we can copy the ATMAtmDataStore class and name it
ATMDaDataStore. Note: This assumes you have deleted the local image
ATMDaDataStore class. If not, delete it before you copy the ATMAtmDataStore
class.

Another approach would be to create the abstract datastore class,
ATMAbstractDataStore, with the one and only class method singleton that
redirects the singleton request to the actual datastore class. Different
versions of a hardcoded redirection or a variable consulting implementation
would make the selection of the actual datastore class variable. The variable
value could be derived from a command line parameter at server startup time.
A second and last change to the ATMServer class is required to access the
actual datastore by sending the singleton message to the new abstract
datastore class, ATMAbstractDataStore.

22.3 Populating the External Datastore

After generating the service and model classes, you can populate the
datastore, using the same code you used to populate the local image
datastore (Figure 129 on page 176).

22.4 Testing the ATM Application with the External Datastore

After generating the new service classes, copying the ATMAtmDataStore
class to ATMDaDataStore or implementing a redirection to the
ATMAtmDataStore, and populating the datastore, you should be able to run
the ATM application as before.
Appendix A. Special Notices

This publication is intended to help application developers to design and implement Web application solutions using the Remote Method Invocation (RMI) technique for Java and Smalltalk interoperability. The information in this publication is not intended as the specification of any programming interfaces that are provided by VisualAge for Java or VisualAge Smalltalk Enterprise. See the PUBLICATIONS section of the IBM Programming Announcement for the VisualAge for Java and VisualAge Smalltalk Enterprise products for more information about what publications are considered to be product documentation.

References in this publication to IBM products, programs or services do not imply that IBM intends to make these available in all countries in which IBM operates. Any reference to an IBM product, program, or service is not intended to state or imply that only IBM’s product, program, or service may be used. Any functionally equivalent program that does not infringe any of IBM’s intellectual property rights may be used instead of the IBM product, program or service.

Information in this book was developed in conjunction with use of the equipment specified, and is limited in application to those specific hardware and software products and levels.

IBM may have patents or pending patent applications covering subject matter in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to the IBM Director of Licensing, IBM Corporation, 500 Columbus Avenue, Thornwood, NY 10594 USA.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact IBM Corporation, Dept. 600A, Mail Drop 1329, Somers, NY 10589 USA.

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The information contained in this document has not been submitted to any formal IBM test and is distributed AS IS. The use of this information or the implementation of any of these techniques is a customer responsibility and depends on the customer’s ability to evaluate and integrate them into the customer’s operational environment. While each item may have been
reviewed by IBM for accuracy in a specific situation, there is no guarantee
that the same or similar results will be obtained elsewhere. Customers
attempting to adapt these techniques to their own environments do so at their
own risk.

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 terms are trademarks of the International Business Machines
Corporation in the United States and/or other countries:

IBM ©
VisualAge
DB2

The following terms are trademarks of other companies:

Java and HotJava are trademarks of Sun Microsystems, Incorporated.

Microsoft, Windows, Windows NT, and the Windows 95 logo are trademarks
or registered trademarks of Microsoft Corporation.

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.
Appendix B. Related Publications

The publications listed in this section are considered particularly suitable for a more detailed discussion of the topics covered in this redbook.

B.1 International Technical Support Organization Publications

For information on ordering these ITSO publications see “How to Get ITSO Redbooks” on page 193.

- *Programming with VisualAge for Java Version 2*, SG24-5264
- *Application Development with VisualAge for Java Enterprise*, SG24-5081
- *Using VisualAge UML Designer*, SG24-4997

B.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.

<table>
<thead>
<tr>
<th>CD-ROM Title</th>
<th>Subscription Number</th>
<th>Collection Kit Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>System/390 Redbooks Collection</td>
<td>SBOF-7201</td>
<td>SK2T-2177</td>
</tr>
<tr>
<td>Networking and Systems Management Redbooks Collection</td>
<td>SBOF-7370</td>
<td>SK2T-6022</td>
</tr>
<tr>
<td>Transaction Processing and Data Management Redbook</td>
<td>SBOF-7240</td>
<td>SK2T-8038</td>
</tr>
<tr>
<td>Lotus Redbooks Collection</td>
<td>SBOF-6899</td>
<td>SK2T-8039</td>
</tr>
<tr>
<td>Tivoli Redbooks Collection</td>
<td>SBOF-6898</td>
<td>SK2T-8044</td>
</tr>
<tr>
<td>AS/400 Redbooks Collection</td>
<td>SBOF-7270</td>
<td>SK2T-2849</td>
</tr>
<tr>
<td>RS/6000 Redbooks Collection (HTML, BkMgr)</td>
<td>SBOF-7230</td>
<td>SK2T-8040</td>
</tr>
<tr>
<td>RS/6000 Redbooks Collection (PostScript)</td>
<td>SBOF-7205</td>
<td>SK2T-8041</td>
</tr>
<tr>
<td>RS/6000 Redbooks Collection (PDF Format)</td>
<td>SBOF-8700</td>
<td>SK2T-8043</td>
</tr>
<tr>
<td>Application Development Redbooks Collection</td>
<td>SBOF-7290</td>
<td>SK2T-8037</td>
</tr>
</tbody>
</table>

B.3 Other Publications

These publications are also relevant as further information sources:

- *Java Remote Method Invocation (RMI)*


© Copyright IBM Corp. 1998
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 BookManager 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:
  
<table>
<thead>
<tr>
<th>In United States</th>
<th>IBMMAIL</th>
<th>Internet</th>
</tr>
</thead>
<tbody>
<tr>
<td>In Canada</td>
<td>caibmbkz at ibmmail</td>
<td><a href="mailto:lmannix@vnet.ibm.com">lmannix@vnet.ibm.com</a></td>
</tr>
<tr>
<td>Outside North America</td>
<td>dkibmbsh at ibmmail</td>
<td><a href="mailto:bookshop@dk.ibm.com">bookshop@dk.ibm.com</a></td>
</tr>
</tbody>
</table>

- **Telephone Orders**
  
  United States (toll free) 1-800-879-2755
  Canada (toll free) 1-800-IBM-4YOU
  
  Outside North America 1-800-IBM-4YOU (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
  Publications Customer Support 144-4th Avenue, S.W.
  P.O. Box 29570 Calgary, Alberta T2P 3N5
  Raleigh, NC 27626-0570 DK-3450 Allerød
  USA Canada Denmark

- **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

---

**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:

<table>
<thead>
<tr>
<th>Title</th>
<th>Order Number</th>
<th>Quantity</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>APA</td>
<td>all points addressable</td>
</tr>
<tr>
<td>ATM</td>
<td>Automatic Teller Machine</td>
</tr>
<tr>
<td>AWT</td>
<td>Abstract Windowing Toolkit</td>
</tr>
<tr>
<td>DDL</td>
<td>Data Definition Language</td>
</tr>
<tr>
<td>IBM</td>
<td>International Business Machines Corporation</td>
</tr>
<tr>
<td>ITSO</td>
<td>International Technical Support Organization</td>
</tr>
<tr>
<td>PIN</td>
<td>Personal Identification Number</td>
</tr>
<tr>
<td>ITSO</td>
<td>International Technical Support Organization</td>
</tr>
<tr>
<td>RMI</td>
<td>Remote Method Invocation</td>
</tr>
<tr>
<td>RMIC</td>
<td>Remote Method Invocation Compiler</td>
</tr>
<tr>
<td>SST</td>
<td>Server Smalltalk</td>
</tr>
<tr>
<td>ST</td>
<td>Smalltalk</td>
</tr>
</tbody>
</table>
Symbols

.Axe 85
.ini 85

A
abbreviations 197
AbtHeadlessRuntimeStartUp 85
account 22, 23
AccountBalanceLabel 125
AccountInfoPanel 125, 127, 130
AccountList 122
accountNumber 39, 67, 68, 104
AccountNumberLabel 125
AccountOverdrawnException 19, 44
accountType 39, 67, 68
AccountTypeLabel 125, 126
AccountBalanceLabel 126
acronyms 197
addressability 8, 53
amount 19, 23
applet 11
application
  context 8, 49, 51, 52
  entry point 85
  launch code 54
  metadata 166
  RbiRmiSamplesAtm 29
  server application 11
  Smalltalk 29
application: 40
array 10
array of exceptions 43
assembly 135
association 161, 162
  BankAccount-Card 165
  BankAccount-Customer 164
  Card-Customer 163
ATM 3
ATMApplet 113, 121, 123, 128, 131, 134
  connections 137
ATMCard 18, 29, 72
ATMNumberLabel 126
ATMServer 17, 25, 33, 49, 59, 75, 89
  atmServer 89, 104, 108
ATMServerRmi 34
ATMServerProxy 19, 25, 89, 121, 123, 128, 131, 134, 137, 142, 146, 151
ATMServerRmi 53
atmServerRmi 50
AWT 113

B
balance 39, 44, 67, 68
  button 123
BankAccount 18, 26, 39, 67, 89, 161
    class 162
    entity 158
    table 160
bankAccount 103, 107
BankAccount-Card association 165
BankAccount-Customer association 164
BankAccountProxy 26, 103, 147
  base class 33
  behavior 25, 59
  browser 11
buildDirectoryStructure: 41
  builder
    type 9
  business 25
business objects 17
by-reference 7, 10
by-value 7, 10, 29, 155

C
cancel
  button 79, 123
Card 161
  class 162
  entity 158
  table 160
Card-Customer association 163
CardLayout 121, 123, 128, 131, 134
cardNumber 21, 30, 39, 67, 68, 72
CardNumberField 120
CardPanel 21, 120, 145
  connections 141
CardValidationException 19, 44
catch 44
category 56
class
  BankAccount 162

© Copyright IBM Corp. 1998
Card 162
Customer 161
diagram 5
client 7, 9, 25, 29
test 152
com.ibm.sst.JavaTypeRepository.generateFor 31
command line argument
-POR: T 54
common methods 56
communication 3, 7, 9, 12, 25
concept 7
configuration 8, 51
ConfirmationPanel 130
  connections 148
conflict 169
connections
  ATMApplet 137
  CardPanel 141
  ConfirmationPanel 148
  ExitPanel 151
  SelectionPanel 143
  TransactionPanel 146
  WelcomePanel 139
context 8, 49, 52
  application 49, 51
  naming 49, 52, 53
  start 49, 52
conversion 83
create
  application context 49
  naming context 52, 53
  object 29
  table 184
createRmiNamingServiceAt: 53
cross-development (XD) 83
Customer 161
  class 161
  entity 158
  table 160
customer 3
Customer (Class) 19
customerId 39, 67, 68, 72, 109

data 6, 59, 152, 175
  model 157
database 6, 59, 155
  design 159
physical 159
  schema 181
datastore 155
  activate 178, 186
  external 181
  local image 175
  populate 175, 187
DB2 155, 184
DDL 155, 183, 184
defaultAccounts 33, 60
defaultCards 33, 60
deploy 85
deployment 11
deposit
  button 123
design 7, 9, 11, 44
database 159
diagram
  class 5
  entity-relationship 157, 159
dialog
  application 3
Dictionary 33
directory: 41
distributed garbage collection 79
distribution 3, 7, 12

E
endpoint 8
entity
  BankAccount 158
  Card 158
  Customer 158
entity-relationship
diagram 157, 159
entry point 85
error condition 44
ErrorLabel 115
event
  proxy 101
exception 19, 34, 43, 44
transaction 170
exit
  button 79
ExitPanel 133, 150
collections 151
expire 79
export 8, 17
exportSet 76, 79
external datastore 155, 181

F
feature
  proxy 102, 112
field 29
  proxy 89, 103
fieldAccountNumber 103
fieldAccountNumbers 89
fieldAccountType 103
fieldBalance 103
fieldBankAccountProxy 90
fieldCardNumber 30, 72
fieldCustomerId 103
fieldHostName 90, 136
fieldLastError 90, 104
fieldNamingServicePort 90
fieldPin 30, 72
foreign key relationship 183

G
garbage collection 55, 79
generateClassesAndJavaFor: 41
generated
  DDL 183
    model classes 167
    services classes 186
generateInterfaceAndJavaFor: 43
generic stub 38
getAccount: 60
getAccountType 41, 105
getBalance 41, 68, 70, 106, 169
getBankAccount 33, 96, 98, 177
getCodeBase 115
getHostName 115, 136
getter 30
getter methods
  include 83
getUrl 97, 101

H
handler 49
  invocation 50
HostNameField 117

I
IBM Java RMI and Smalltalk SST Interoperability
  project 29
  identifier 8
  IdentityDictionary 76
image 8, 84
  single 85
import 8, 44
include
  setter methods 83
incoming message 50
InformationLabel 115
inspect 75, 76
instance 17
instance variable 29
interaction 12
interface 7, 40, 43
interfaces: 41
interoperability 3
invocation 8
  handler 49, 50
  schema 49
invocation handler 49
invocation schema 49, 50

J
Java 3, 9
java.awt.Applet 115
java.io.Serializable 29
java.math.BigDecimal 18, 43
java.rmi.Naming 7, 75
java.rmi.RemoteException 19, 34
java.util.Hashtable 33
java.util.Vector 33
java_util_Vector 33

K
key
  foreign 183

L
language 9
lastError 110
launch code 54, 85
life cycle 7, 12
limitations 155
Ljava.lang.String 75
local 7, 25, 53, 96
local image datastore 155, 175
lookup 7

**M**
MainPanel 115, 135
map 10, 33
object 29
persistence 185
marshaling 7, 83
message 7
incoming 50
meta model 161
metadata application 166
method
affected by persistence and distribution 169
category 56
common methods 56
include when packaging 83
proxy 90, 104
unique methods 56
method signature 43
methods: 41, 43
model 5, 12
class 167, 169
classes 155
data 157
generated classes 167
meta 161
modified classes 169
object 161
review 165
visual 135
modified model classes 169

**N**
naming 7, 18, 49
context 49, 52, 53
service 49, 53, 75
naming context 79
create 52, 53
naming service 11, 49
NamingContext 53

**O**
object 5, 7, 25, 53
create 29
identifier 8
model 5, 161
name 7
persistence 155
proxy 89
register 53
serialize 7
space 8, 51
ObjectExtender 155, 169
limitations 155
objet
map 29
OrderedCollection 33

**P**
package
Java 29
rbi.rmi.samples.atm 29
packaging 83
rules 83
packagingRulesFor 83
Panel 113
passive 84
persistence 59, 155, 161, 169
getBankAccount 177
mapping 185
services classes 186
setup 178
shut down 178
test 179
verifyCardAndGetAccountNumbers: 177
PIN 21
pin 30, 72
PINField 120
port 53
command line argument 54
port default 53
-PORT: 54
PortNumberField 117
project
IBM Java RMI and Smalltalk SST Interoperability 29
Java 29
property feature 29
proxy 8, 25, 89
event 101
feature 102, 112
field 89, 103
R
rbiRmi Samples.atm 29
RbiRmi Samples Atm 29
rebind:to: 53
reference 7, 25
referential constraints 159
register 11, 17, 53
invocation schema 49
object 53
registeredAccounts 90
registeredBankAccounts 33, 60
port 53
port default 53
relationship 158, 161, 183
relationships 158
release 55
release resource 55
remote 7, 10, 25, 96
repository
  type 31
reset 186
resetATM 60
resource 55
resources 55
restartAtm 34, 65, 96, 100, 151
result 24
RMI 3, 7, 8, 25, 49, 155, 169
  limitations 155
registry port 53
rmi
  transport 52
RMIC 9
rmiNaming 53
rules 83

S
ScaledDecimal 18, 43
schema 49
  database 181
  export to database 184
  invocation 49, 50
  local image 155
SciSocketManager 54
scrapbook 30
SelectionPanel 22, 122, 148, 150
connections 143
serialization 7
serialize 10
server 7, 9, 17, 29
  application 11
server workbench 80
service 7
  naming 53
services classes 155, 186
servlet 11
setter 30
setter methods
  include 83
setup 178, 186
  application context 49
  naming context 49
setupServer:namingPort: 54
shut down 49, 55, 178
shutDown 60, 66
shutdown 186
single image 85
singleton 17, 25, 34, 49
skeleton 8, 35, 36
Smalltalk 3, 9, 10
space 51, 76
SST 7, 8, 49, 155
SstApplicationContext 76
SstApplicationContext createRmiNamingServiceAt: 53
SstDualRegistry 76
sstsArray 10
sstIsRemoteable 10
sstIsRmiRemotable 60, 68, 72
sstIsRmiSerializable 30
sstIsSerializable 10
SstLocalSpace 76
sstRMIClassName 169
sstRMIClassName 10, 30, 33, 60, 68, 72
SstRmiDetailedException 46
SstRmiGenericStub 38
SST RmiNaming 53
SstRmiRegistry 53
SstRmiSupport 33
SstRmiThrowable 44
SstRmiTypeBuilder generateClassesAndJavaFor:interfaces:methods: 40
SstRmiTypeBuilder generateInterfaceAndJavaFor:superclass:methods: 43
VisualAge Java - RMI - Smalltalk ATM Sample

SstRmiTypeBuilder mergeDefinitionsFrom 31
SstRmiTypingRepository 52
SstSpaceConfiguration 51
start
  context 49, 52
startup 17
state 59
stub 8, 35, 40
superclass: 43

table
  BankAccount 160
  Card 160
  create 184
  Customer 160
tandem 29
test 75
  client 152
  with external datastore 187
  with local image datastore 179
throw 44
throw:message: 44
timeout 79
timer 79
transaction 22, 24
  ownership 169
  RMI remote exception 170
t  technical 156
TransactionPanel 23, 127
  connections 146
transactions
  technical 169
transcript 84
transfer
  button 123
transport 8
  rmi 52
try 44
type 31
  builder 9, 31
  information 9
  repository 31
typeRepository 52

unregister 64
unregister bankAccounts 64
unregisterAllBankAccounts 34, 60, 96, 99
unregisterBankAccount: 60
URL 8, 52, 53, 115, 136
user 3
user interface 11, 21, 113

V
verifyCardAndGetAccountNumbers 33, 96, 98
verifyCardAndGetAccountNumbers: 60, 177
visual 135
VisualAge 3, 155

W
Web browser 11, 79
WelcomePanel 117, 135, 140, 142
  connections 139
withdraw 19, 23, 40, 41, 76, 111
  button 123
withdraw: 68, 71
WithdrawAmountField 127
workbench 80

X
XD (cross development) 84
ITSO Redbook Evaluation

VisualAge Java - RMI - Smalltalk. The ATM Sample from A to Z
SG24-5418-00

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