Using VisualAge for Java Enterprise Version 2 to Develop CORBA and EJB Applications

Joaquin Picon, Craig Edwards, Gianni Scenini

International Technical Support Organization

http://www.redbooks.ibm.com

SG24-5276-00
Using VisualAge for Java Enterprise Version 2 to Develop CORBA and EJB Applications

December 1998
Contents

Figures ......................................................... ix
Tables ......................................................... xiii
Preface ........................................................ xv
How This Document Is Organized ........................................ xvi
The Team That Wrote This Redbook ....................................... xvi
Acknowledgments ...................................................... xvii
Comments Welcome .................................................... xvii

Part 1. Technologies and Products ........................................... 1

Chapter 1. Introduction .................................................. 3
  1.1 Technologies and Products ........................................... 3
  1.2 VisualAge for Java and WebSphere ................................ 4
  1.3 VisualAge for Java and Component Broker ......................... 5
  1.4 Integration ..................................................... 6

Chapter 2. Why Distributed Objects? .................................... 9
  2.1 Monolithic Applications ............................................ 9
  2.2 Client/Server Applications ....................................... 9
  2.3 Distributed Applications ....................................... 10
  2.4 Major Distributed Technologies ................................ 12
    2.4.1 Distributed Computing Environment ....................... 12
    2.4.2 Remote Method Invocation ................................ 13
    2.4.3 Common Object Request Broker Architecture .............. 14
    2.4.4 Distributed Common Object Model ......................... 14
    2.4.5 Common Object Model Plus ................................ 14
  2.5 DCOM/COM+/DNA and CORBA .................................... 14

Chapter 3. What Is CORBA? .............................................. 17
  3.1 Object Management Group ......................................... 17
  3.2 Components of a CORBA Implementation .......................... 17
    3.2.1 Object Request Broker .................................... 18
    3.2.2 Object Services ............................................ 19
    3.2.3 Common Facilities .......................................... 21
    3.2.4 Application Objects ....................................... 21
  3.3 Internet Inter-ORB Protocol (IIOP) ............................... 22

Chapter 4. Why Java? .................................................... 23
  4.1 Java Attributes ................................................ 23
Chapter 4. CORBA and EJB Applications

4.1 Object Oriented ................................................. 23
4.1.1 Object Oriented ............................................. 23
4.1.2 Architecture Neutral ....................................... 23
4.1.3 Multithreaded ................................................. 24
4.1.4 Dynamic ....................................................... 24
4.1.5 Secure ......................................................... 24
4.1.6 Portable ....................................................... 25

Chapter 5. Why Enterprise JavaBeans?

5.1 Roles .......................................................... 27
5.1.1 Provider ...................................................... 27
5.1.2 Deployer ..................................................... 28
5.1.3 Application Assembler ....................................... 28
5.1.4 Container Provider .......................................... 28
5.1.5 Server Provider ............................................. 28
5.1.6 System Administrator ....................................... 28
5.2 Components ..................................................... 28
5.2.1 Container ..................................................... 29
5.2.2 Session and Entity JavaBeans .............................. 29
5.3 An Overall Picture ............................................. 31

Chapter 6. VisualAge for Java Enterprise

6.1 Dynamic and Iterative Development ......................... 35
6.1.1 Simplified Source Control ................................ 35
6.1.2 Interactive Execution ...................................... 36
6.1.3 Incremental Compilation and Linking .................... 36
6.1.4 Debugging .................................................. 38
6.2 Construction-from-Parts Paradigm ....................... 39
6.2.1 Visual Programming ....................................... 39
6.2.2 JavaBeans Support ......................................... 40
6.3 Code Generation ............................................... 40
6.3.1 SmartGuides .............................................. 40
6.3.2 Code Generation for Enterprise Access .................. 41
6.4 Additional Features .......................................... 41
6.5 Conclusion .................................................... 42

Chapter 7. CORBA Environments .......................... 43
7.1 WebSphere .................................................... 43
7.1.1 Servlet Runtime Environment ........................... 43
7.1.2 Java Server Pages .......................................... 43
7.1.3 CORBA Development and Runtime Support .......... 44
7.1.4 WebSphere Performance Pack .......................... 45
7.1.5 WebSphere Developer Studio ............................ 46
7.1.6 Enterprise Java Server for EJBs ....................... 46
7.2 Component Broker .......................................... 46
Figures

1. TEA System View .................................................. 8
2. Client / Server Architecture ....................................... 10
3. Multitiered Environment ........................................... 12
4. CORBA Components ................................................. 18
5. Global Enterprise JavaBeans View from Client and Server .... 32
6. Component Broker Building Blocks ............................... 47
7. TEA Object Model .................................................. 62
8. Adding Features .................................................... 65
9. Selecting Features to Install ...................................... 66
10. Subclassing the WebSphere Web Server ......................... 68
11. Class Path Properties of Web Server ............................ 69
12. Manually Adding Projects to Classpath .......................... 70
13. server.properties File ............................................. 71
14. doc.properties File ................................................ 71
15. A Simple HTML File: D:\WWW\HTML\Test.html ............... 72
17. Console Showing Web Server Starting Successfully .......... 73
18. Debugger Thread List Showing Web Server Starting Successfully ............................................. 74
19. tea.servlet.TEAServlet Snippet ................................ 77
20. Adding a Servlet to WebSphere Configuration ................. 78
21. Autoloading a Servlet ............................................. 79
22. servlet.properties File .......................................... 80
23. Setting WebSphere Classpath ..................................... 81
24. Simple Factory .................................................... 83
25. CORBA-Aware Factory ............................................ 84
26. Sample IOR ......................................................... 87
27. Example of Using Naming Services ............................... 88
28. IDL Development Environment .................................... 89
29. IDL Compile Options ............................................. 90
30. TEA: Java IIOP Client ............................................. 93
31. Java Applet Client Architecture .................................. 94
32. Overriding the getCodeBase() Method .......................... 97
33. TEA: Adding Pure HTML Client .................................. 99
34. HTML Client Architecture ....................................... 100
35. Using Servlet Builder from the Visual Composition Editor .. 103
36. NewTrip Panel Created by the Servlet .......................... 104
37. TEA: Adding Component Broker Object ......................... 109
38. Component Broker Core ......................................... 111
39. Using Component Broker Frameworks within Rational Rose ... 113
40. New EmployeeID Object in the Component Broker Object Architecture ............................................. 114

© Copyright IBM Corp. 1998
Using VisualAge for Java Enterprise Version 2 to Develop CORBA and EJB Applications
## Tables

1. Product Installation Directories .................................................. 63
2. Summary of Browser-Web Server Scenarios ................................. 64
3. IDL to Java Basic Type Mapping ................................................. 208
4. List of IDL Exceptions .............................................................. 216
Using VisualAge for Java Enterprise Version 2 to Develop CORBA and EJB Applications
Preface

VisualAge for Java is the first enterprise-aware, incremental Java application development environment designed to connect Java clients to existing server data, transactions, and applications.

This redbook shows how Component Broker Series and WebSphere IBM products are used to create a CORBA application.

A sample application is used throughout the book to illustrate how to develop a CORBA application and access it with Java clients using IIOP or with pure HTML clients with the help of servlets.

After reading this redbook, you will have a broad understanding of the new VisualAge for Java features that facilitate implementing an enterprise application based on CORBA and servlets. You will also have a sense of the power of Enterprise JavaBeans (EJB).

How This Document Is Organized

The document is organized as follows:

- **Part 1** contains an introduction to the different technologies and products used in this book. We introduce distributed technology, CORBA, Java and Enterprise JavaBeans and provide an overview of VisualAge for Java and two CORBA environments: WebSphere and Component Broker.

- **Part 2** describes a sample CORBA application developed for this book. We address the development of a CORBA application using the IDL Development Environment of VisualAge for Java and the CORBA and Servlet runtime support provided by WebSphere. We create Java IIOP and pure HTML clients, using servlets and VisualAge for Java’s Servlet Builder, to use server CORBA objects.

- **Part 3** covers how to use VisualAge for Java when you develop client and server sides for Component Broker. We focus on the debugging options that can be used for remote debugging.

- **Part 4** covers integration. We add to our application the extension developed with Component Broker in part 3. Once we get the interoperability between WebSphere and Component Broker working, we conclude by integrating a new technology: Enterprise JavaBeans.

- **Appendixes** contain information about IDL to Java mapping, snapshots of Java and HTML clients, some troubleshooting guidelines, information
about workarounds we used, and a description of the sample application packages.

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.

Joaquin Picon is a consultant at the International Technical Support Organization, San Jose Center. He writes extensively and teaches IBM classes worldwide on application development, object technology, and CORBA. Before joining the ITSO, Joaquin worked at the IBM Application Enabling Center Of Competency in France.

Craig Edwards is a Client/Server Specialist in IBM Global Services Australia. He has six years of experience in the client/server development field. His areas of expertise include object-oriented design and development, Internet development, and enterprise e-business solutions. Craig has written several other publications on remote and distributed relational database access.

Gianni Scenini is an Object Technology/Application Development specialist at IBM Italy. He has seven years of experience in the information technology field. Gianni’s areas of expertise include object-oriented, networking, security, and emerging technologies. He has written, several articles on Java security, object-oriented design, and network protocols.

Acknowledgments

This book would not have been possible without the help of the following people who contributed to the content of the book:

Ueli Wahli, International Technical Support Organization, San Jose Center, who started this project.

Maggie Cutler who knows how to explain things in a short and meaningful way, and staff at the International Technical Support Organization, San Jose Center.

Dave Ings, Vito Spatafora, Paul Yu, Victor Chan, Leonard Theivendra VisualAge for Java Development, IBM Toronto Lab, Toronto

Barbara Hampson, Hany Salem, Liane Acker, Jim Knutson Component Broker Support and Development, IBM Austin Lab, Austin
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 255 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. Technologies and Products
Using VisualAge for Java Enterprise Version 2 to Develop CORBA and EJB Applications
Chapter 1. Introduction

In the past few years, we have seen architectures built from huge monolithic applications moving to client/server. This first split of functionality did not provide sufficient flexibility, however, so the distributed object architecture became the architecture of choice.

With the advent of the Internet, having distributed objects spread over the Web makes more and more sense. Once your business objects, or someone else’s, are running somewhere in the network, you can quickly assemble new applications that reuse not only existing code but also running code, ready for use, and executing on servers and save tremendous amounts of time and money.

In the next few years, we can easily imagine some enterprises selling services based on the functionality of objects with appropriate quality of service such as 7 x 24 availability on the Web, and other enterprises renting access to these objects to build additional functionality.

1.1 Technologies and Products

In the first part of this book, we start by looking at the different technologies and products that can help develop such a scenario.

After reviewing the different distributed technologies, we focus on the most mature distributed object architecture, the Common Object Request Broker Architecture (CORBA). The CORBA specifications address the interoperability of objects developed with any programming language. Java appears to be the language of choice to have your CORBA objects running everywhere, from back-end servers to your cellular phone. Enterprise JavaBeans (EJB) are the end result of using Java and CORBA. These new specifications promise to combine the power of CORBA with the ease of use of Java.

If you are convinced that distributed applications based on Java and CORBA are definitely the way to go, the next step is to determine which products support them.

Because most development will be done in Java, our favorite tool for creating Java applets, applications, CORBA objects, Java beans, and Enterprise JavaBeans is VisualAge for Java.
For CORBA environments, we have two choices: IBM WebSphere and Component Broker. From a CORBA perspective, we can characterize WebSphere as a lightweight implementation of CORBA, and the Component Broker as an implementation that addresses enterprise mission-critical applications. For readers who are not familiar with these products, we provide an overview of them in Chapter 7, “CORBA Environments” on page 43.

### 1.2 VisualAge for Java and WebSphere

To illustrate the possibilities offered by the different technologies and products used in this book, in the second part of this book, we define a sample application.

In today’s business environment, traveling interstate or overseas to perform work duties has become a commonplace activity. One of the more onerous tasks associated with traveling on business is submitting expense claims to recoup the costs of the business trip.

The Travel Expense Application (TEA) enables users to define a business trip and a collection of expenses associated with it. At any stage, a business trip can have zero or more expenses. A trip must have a justification, a start date, and an end date. Once users have entered all of the expenses associated with their trips, they can submit the claim for approval. (See Chapter 8, “The Travel Expense Application” on page 57 for detailed information about the application.)

We implement the TEA as CORBA objects running inside the WebSphere environment. To develop the application, we used the new VisualAge for Java’s IDL Development Environment, which helps you create and manage stubs and skeletons from interface definition language (IDL) files. By creating and managing stubs and skeletons directly from VisualAge for Java, you get all of the benefits of versioning that the tool provides.

We develop our first client of the TEA as a Java Internet Inter-ORB Protocol (IIOP) client. It runs as an applet inside any Java-enabled Web browser. Although this solution is appropriate for intranets with adequate network bandwidth, most of the time it is not appropriate for Internet users. You cannot assume that Internet users run a Java-enabled Web browser. Even if they are, however, they still use telephone lines with too low a bandwidth to make the Java code download time acceptable. If your application is aimed at electronic business (e-business), you do not want your customers to have to wait until the required code is loaded in their machine before they can place an order. Until these network problems are solved, by increasing bandwidth...
and providing a quality of service that guarantees a decent response time, the only solution is to send an HTML page to a user’s Web browser. Obviously we are talking here about building dynamic HTML pages with a personalized user content.

This solution is implemented in this book as a pure HTML client (see Chapter 12, “Pure HTML Client Development” on page 99). To address the HTML client, we introduce the use of servlets and make extensive use of the Servlet Builder, which is part of VisualAge for Java Enterprise V2.0.

### 1.3 VisualAge for Java and Component Broker

Having a first draft of a CORBA application running in the WebSphere environment and, given the goal of CORBA to build applications consisting of distributed objects running in different environments, in the third part of this book, we extend the TEA to incorporate a server object running in Component Broker.

This additional object acts as an extension of the Authenticator object used to identity users of the TEA application. Obviously, this is not a way to provide security; it is just an example in our application. If we want to authorize a user by checking his or her existence in a database, we use the DB2 SAMPLE database, which has a table containing a list of employees and their corresponding properties. Therefore, if we find the user in this table, we consider him or her as authorized to use the TEA.

The addition of this functionality to our application has two facets: the client side and the server side. For both sides, we want to use VisualAge for Java. For the client side, Component Broker provides a Java object request broker (ORB) that can be easily imported into VisualAge for Java, providing the power of rapid application development (RAD). However, the use of VisualAge for Java to develop the Java business objects on the server side is far from being straightforward. The development process and tools required when developing Component Broker business objects is not yet well integrated with VisualAge for Java. This lack of integration and the inability to use the local VisualAge for Java debugger forced us to add a chapter describing how to use the remote debugger to debug Component Broker Java objects.
1.4 Integration

The fourth part of this book focuses on integration and considers Enterprise JavaBeans as the future for distributed objects.

Now we want to extend the TEA by incorporating the Component Broker object. The Component Broker object runs on a different machine and is accessed from the first server running the Hypertext Transfer Protocol (HTTP) server and the WebSphere servlet and CORBA environments. Here we run into difficulties related to the interoperability of different CORBA environments. For example, we end up with two different naming services, and, even though the ORB implementations are similar, they still contain differences that we had to solve to have the WebSphere CORBA objects coexist with the client side of Component Broker. By solving some of these problems, we give you an idea of a distributed application built on top of two different products.

As the time devoted to producing the material for this book neared an end, we had access to the beta version of VisualAge for Java 2.1, which is bundled with WebSphere 2.0 and Enterprise JavaBeans support. Therefore we added a chapter about Enterprise JavaBeans, which seem to be a possible evolution for CORBA.

When developing Enterprise JavaBeans, you have to provide an implementation and additional interfaces. With VisualAge for Java you only have to create the implementation, and the tool generates the interfaces as defined in the specifications.

Once you have finished developing your enterprise Bean, in order to run it, you need to create a Deployment Descriptor that defines the various parameters related to persistence, transactional behavior, and security. Then you have to package all of the different classes with the Deployment Descriptor and a Manifest File inside a jar file. VisualAge for Java generates the requested files and creates the final jar file for you.

The jar file is then given to a person with a deployment role who uses the target Enterprise JavaBeans server and the container’s tools to read the jar file and generate additional classes that create the link between the enterprise Bean business logic and the quality of service specified in the Deployment Descriptor before you can run your enterprise Bean inside the given container. VisualAge for Java in combination with the integrated WebSphere environment does all this for you. It creates all deployment classes, creates an Enterprise JavaBeans server, installs your enterprise Beans, and executes them. You are now ready to develop the client side,
using these ready-to-use enterprise Beans. You can develop, debug, and test a complete Enterprise JavaBeans environment with all the benefits of VisualAge for Java, such as debugging and versioning.

Obviously, VisualAge for Java is able to import and export EJB jar files.

With all of these promising features, we could not resist transforming the EmployeeID object created for Component Broker into an enterprise Bean.

When developing enterprise Beans, you have to choose among these possibilities:

- Session bean
- Bean-managed persistence entity bean
- Container-managed persistence entity bean

Because our EmployeeID is persistent, we naturally created an entity enterprise Bean. Even though WebSphere supports container-managed entity beans, in the version we used it creates a table to map columns with persistent attributes. We could not find any schema mappers that enabled us to reuse an existing table, in our case the SAMPLE DB2 table. So, we implemented the enterprise bean as a bean-managed persistence entity bean.

Our final system view of the TEA (Figure 1) integrates a Java IIOP and pure HTML clients, and on the server side, WebSphere CORBA objects, servlets creating dynamic HTML pages, Component Broker objects, and an enterprise Bean.
If you already know about the technologies and products used in this book, you can go directly to Part 2.
Chapter 2. Why Distributed Objects?

Over the last 30 years we have seen many changes in how we design, develop, and maintain an enterprise information system. Today the computer is the network. In the near future we will see distributed applications across several heterogeneous networks, computers, and operating systems using different languages, technologies, and data storage.

The separation of the interface ("what it is,"') from its implementation ("how it is,"') will be the way to achieve true cross-platform, cross-language, and cross-design applications.

2.1 Monolithic Applications

In monolithic, so-called one-tier, applications all data access and business and presentation logic are in the same machine, namely, the mainframe. We cannot share common data. But our applications have to share data and communicate with other applications, so we have to have multiple copies of data replicated in each mainframe. We also need to have the same functions to access them across the applications. This raises such problems as data synchronization and performance.

For such applications we have to have very powerful machines, and when they become overloaded the only solution is to upgrade them. Monolithic applications are costly and hard to maintain.

2.2 Client/Server Applications

To improve on one-tier applications, we developed two-tier applications, also called client/server applications. Basically the client/server approach divides a monolithic application into two monolithic components located in different machines. The server machine provides access to the relational database management system (RDBMS) and implements some of the business logic. The client machine implements the presentation and business logic. Thus the two sides can share data in common, but the client is a fat client because it has to implement not only the presentation logic but also some business logic. Figure 2 shows the architecture of the client/server application.
2.3 Distributed Applications

With the introduction of object-oriented application development we can design and develop flexible and interoperable applications, and the development process becomes less costly and shorter than ever before. Objects can be distributed across different networks and operating systems, and each component can interoperate with each other no matter what the operating system or the physical network is. This brings us several benefits, such as flexible infrastructures, portability, code reuse, and interoperability.

But we have to consider the old technologies and legacy systems. We cannot throw away all the old applications. Too much time and investment have gone into designing, developing, maintaining, and testing them. Object-oriented
development enables us to integrate old technologies with new technologies, using *wrapping or encapsulation*. For example, you can view a module written in a procedural language as an object and then use it. Using encapsulation for integration and reengineering enables us to use the functionality of the legacy software while hiding the legacy application behind a new object model. Because the repository for the objects is located on the server, the client application can be kept small and lightweight. This has a particular advantage for application maintenance. We do not have to redistribute all of our modified code because we have the objects on the server. The client uses an interface of that object. So when you modify the object’s implementation, its interface is not affected.

Another advantage of object-oriented programming is the ability to reuse code. Distributed objects can be divided into separate, self-contained modules that can be worked on independently. Each module or component can communicate with other components. So you do not have to reinvent the wheel. Developers can work synchronously, and when they have created the new objects, they can build up the application using these objects and reusing existing objects.

Another advantage of the distributed object framework relates to code distribution. With the majority of the distributed object application located on the servers, distribution of code to the client is effectively handled by simply swapping the modified components on the server side, thus eliminating the need for massive updates to all clients. Figure 3 shows a three tier architecture environment.
2.4 Major Distributed Technologies

There are several distributed technologies. Some vendors have their own technology, and others implement the standard technologies. Standards such as CORBA embrace the major information and technology companies. All of the distributed computing solutions have their advantages and disadvantages. We are moving to a common solution to make our applications interoperable and portable. We have to keep the design and the architecture as abstract as possible, so that companies are free to implement the services differently. That is one of the goals of CORBA. Today we have many distributed technologies, some well proven, some emerging, and others becoming the standard. In this section we review the major distributed technologies to understand how they work, what they provide, and their pros and cons.

2.4.1 Distributed Computing Environment

The Distributed Computing Environment (DCE), from the Open Software Foundation, supports both portability and interoperability by providing the developer with capabilities to hide differences among the various hardware, software, and networking elements an application deals with in a large
network. DCE is called *middleware* because it consists of multiple components that have been integrated to work closely together:

<table>
<thead>
<tr>
<th><strong>Remote Procedure Call</strong></th>
<th>Provides a way to communicate with other modules situated in different machines across the network</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Time service</strong></td>
<td>Provides a mechanism to monitor clocks in a distributed environment</td>
</tr>
<tr>
<td><strong>Directory service</strong></td>
<td>Provides full X.500 support and a naming model to allow the identification of distributed resources</td>
</tr>
<tr>
<td><strong>Thread service</strong></td>
<td>Provides a way of supporting concurrent applications</td>
</tr>
<tr>
<td><strong>Security service</strong></td>
<td>Provides the network with authentication, authorization, and user account management services</td>
</tr>
<tr>
<td><strong>Data-sharing service</strong></td>
<td>Provides end users with capabilities built on the fundamental distributed services</td>
</tr>
<tr>
<td><strong>Distributed file system</strong></td>
<td>Interoperates with the network file system to provide a high-performance, scalable, and secure file access system</td>
</tr>
<tr>
<td><strong>Diskless service</strong></td>
<td>Allows low-cost workstations to use disks on servers.</td>
</tr>
</tbody>
</table>

### 2.4.2 Remote Method Invocation

Remote Method Invocation (RMI) is Sun’s implementation of a distributed object design. RMI provides a way for client and server applications to invoke methods across a distributed network of clients and servers running the Java Virtual Machine (JVM). RMI makes use of two other protocols for its on-wire-the-wire format: Java Object Serialization and HTTP. The Object Serialization protocol is used to marshal call and return data. The HTTP protocol is used to “POST” a remote method invocation and obtain return data when circumstances warrant. It is cross-platform but not cross-language. Because RMI is not CORBA-IIOP compliant, you cannot invoke a method of an object not written in Java. The Object Management Group (OMG) is planning to map RMI on top of IIOP, so that RMI can be interoperable. RMI has been introduced with Java Development Kit (JDK) 1.1. RMI is easy to use but it offers fewer distributed services than CORBA offers.
2.4.3 Common Object Request Broker Architecture
Developed by the OMG in 1990, CORBA enables invocations of methods on distributed objects located anywhere on a network. It uses a neutral-definition language, the IDL, to describe the interface of an object. The use of IDL makes CORBA a cross-platform and cross-language architecture. Messages between components are handled by the ORB, which hides low-level details from the user interface. You can invoke a method of an object with no knowledge of where it is located, on which operating system it is running, and which language has been used to implement it.

The growth of the Internet has led the OMG to specify the General Inter-ORB Protocol (GIOP) to allow different CORBA-ORB compliant vendor implementations to interoperate with each other. GIOP messages are carried by IIOP, which uses Transmission Control Protocol/Internet Protocol (TCP/IP) to communicate.

2.4.4 Distributed Common Object Model
The Distributed Component Object Model (DCOM) is Microsoft's solution to distributed object architectures. It uses Object Remote Procedure Calls (ORPCs) to carry messages between components. Key features engineered into the DCOM architecture include language independence, transport neutrality, and static and dynamic object method invocation. Because DCOM supports Windows platforms, DCOM-based applications can take full advantage of existing Microsoft services such as security and transactions. Some vendors are migrating DCOM to other platforms such as UNIX, and DCOM eventually will become truly cross-platform.

2.4.5 Common Object Model Plus
The next generation of Microsoft's distributed object technology is COM+, which extends the DCOM services to other rich sets of services including event, queuing, security, and load-balancing services. COM+ is also tightly integrated with the Microsoft Transaction Server (MTS). All of the COM+ services are activated by "interceptors" that monitor components and determine which services are required.

2.5 DCOM/COM+/DNA and CORBA
Basically the major distributed technologies are CORBA and DCOM and COM+. Because Windows platforms are so widely used, DCOM and COM+ are becoming CORBA's major competitors.
CORBA uses a neutral definition language to describe the interface of an object. Compiling the definition of an interface you get proxy objects for client and server. The stub generated from the IDL is language dependent according to the OMG’s specification of the IDL mapping to that language. It is seen by the client as the real object, but it is actually only the interface of the object. The real implementation of the object’s interface is provided by the skeleton, which resides on the server. This environment allows the separation of the interface from the implementation. Thus you can have a stub written in Java and a skeleton written in C++. DCOM and COM+ use standardized interfaces and methodologies to pass data between components. As long as objects implement standard interfaces, they can communicate with each other no matter which language is used.

DCOM and COM+ and CORBA can be used over TCP/IP and thus work on the Internet.

CORBA is more mature than DCOM and COM+ as a cross-platform technology. It exists on virtually any platform, whereas DCOM and COM+ are tied to Windows platforms. Some vendors are planning to port DCOM and COM+ to several UNIX systems such as Solaris, and Linux, and to the Macintosh platform. IONA, a leading vendor of CORBA-compliant ORBs, is releasing a bridging product between DCOM and CORBA that enables the technologies to talk to each other. Netscape, IBM, and Oracle have embraced CORBA in their Web strategy, whereas Informix and Sybase use DCOM and CORBA. Obviously Microsoft is pushing DCOM and COM+ as its Web strategy.
Chapter 3. What Is CORBA?

There are many excellent reasons why you would choose to use a distributed object infrastructure for an enterprise application. In this chapter we focus on the types of activities that are performed in a distributed application and the specific way in which CORBA implements them.

The most common tasks that a distributed system performs are:

- Creating, locating, and sharing objects
- Querying object collections
- Authentication and authorization
- Managing shared access to objects through locking, managing deadlocks, race conditions, and performance tuning
- Servicing multiple clients in a threaded environment
- Providing a cache policy on a server or client
- Application management (including scalability, efficient use of existing resources, load balancing, and addition of new servers)

3.1 Object Management Group

The OMG was established in 1989 as an open consortium of more than 800 companies that work together to define open standards for an architectural framework for object computing. Some of the standards that the OMG has worked on are CORBA, IIOP, Object Services, Internet Facilities, and Domain Interface specifications. The first publication of the Object Management Architecture Guide (OMA) dates back to 1990.

CORBA 1.1, introduced in 1991, specified the IDL that defines the interaction between client and server objects in a specific implementation of an ORB. CORBA 2.0, adopted in December 1994, introduced true interoperability among ORBs from different vendors.

3.2 Components of a CORBA Implementation

A CORBA implementation has four discrete elements:

- The ORB, through which objects intercommunicate
• Object Services, which define system-level services that are added to the ORB. Examples of these services are naming, security, persistence, and transaction.

• Common Facilities, which define application-level services. Examples of these services are components, compound documents, and other vertical facilities.

• Application Objects, which capture real-world behavior, such as bank accounts, customers, and airplanes

Figure 4 depicts the relationships among these components.

![CORBA Components Diagram](image)

**3.2.1 Object Request Broker**

The ORB is the piece of middleware that establishes the client/server relationship between objects. It intercepts any requests that the client makes, is responsible for finding an object that can implement that request, passes the required parameters, invokes the method, and returns the result. The client does not need to be aware of the physical location of the object, its programming language, its operating system, or any other specific
information regarding its whereabouts. The ORB shields that information from the client.

The ORB is the only component of CORBA that must be present in order to claim CORBA compliancy. Many vendors ship implementations without CORBA services or CORBA facilities. You must either create these components or purchase them separately. The ORB, however, must be present in order for your distributed application to function.

Some of the responsibilities of an ORB are:

- Locating and instantiating objects on remote machines
- Marshalling method parameters in a consistent way between any combination of programming languages and operating systems
- Invoking methods on remote objects, using static invocation (compile-time resolution) or dynamic invocation (runtime resolution)
- Routing callback methods from the server to the appropriate client objects

The ORB provides all these services, and more, transparently. All you, as a developer, have to do is provide the appropriate initialization parameters to the ORB, and the rest is handled seamlessly by the ORB implementation.

### 3.2.2 Object Services

The OMG has published specifications for 15 different object services. In this section we briefly describe the behavior of each of the services.

- **Life Cycle**
  - Defines operations for creating, copying, moving, and deleting objects

- **Persistence**
  - Provides a single interface for storing objects persistently in a variety of storage servers that include RDBMSs, object databases management systems (ODBMSs), and flat files

- **Naming**
  - Allows the location of objects by name. The human-readable name is converted to an interoperable object reference (IOR) through what is known as name binding. This process also allows objects to be bound to other protocols such as X.500, DCE, Network Information System (NIS+), Netware Directory Services (NDS), and Lightweight Directory Access Protocol (LDAP).

- **Event**
  - Allows objects to register an interest in specific events that may be triggered by other objects. The Event
Service decouples the communication between event consumers and producers. In other words, the producers supply event data to the Event Service, which passes it on to the event consumers.

**Concurrency Control** Provides a means by which multiple clients can coordinate their access to the same resource. The Concurrency Control Service is designed primarily to be used in a transactional environment where locks can be acquired on behalf of the transaction.

**Transaction** Provides coordination for transactions when the transaction participants originate from different locations in a distributed system. This service is commonly referred to as the *Object Transaction Service* (OTS). OTS can begin a unit of work, coordinate the work done by each of the participants, and perform two-phase commit.

**Relationship** Allows relationships between objects to be explicitly defined. Type and cardinality rules can be expressed, with exceptions being raised if the rules are violated.

**Externalization** Defines a convention for reading and writing a serialized object to and from a data source; for example, writing an object to disk for offline processing. This service is closely aligned with the Relationship Service.

**Query** Allows users and objects to perform queries on collections of other objects. It permits the use of indexes and is based on existing standards for structured query language (SQL) and object query language (OQL).

**Licensing** Provides a mechanism for producers to control the use of their intellectual property. Some examples are controlling the number of concurrent users, time-bombed evaluation periods, and charge-back schemes.

**Properties** Provides operations that add named values (or properties) to an object. These properties are in addition to the static IDL-defined attributes.

**Time** Provides an interface for synchronizing time over a distributed network. The time service is often used to
determine the absolute order of events among distributed objects.

**Security**
Provides authentication, authorization, security auditing, nonrepudiation, and security policy administration

**Trader**
Provides a matchmaking service for objects. The Trader Service registers the availability of a service, and objects can query for objects that match certain criteria.

**Collection**
Supports a set of behaviors for a number of common collections, such as sets, queues, stacks, lists, and trees

### 3.2.3 Common Facilities
Common facilities are collections of services that are used at an application level to improve the consistency of distributed development. A common facility defines the way in which business components interact.

The common facilities that are under construction include mobile agents, workflow, business object frameworks, and internationalization. The OMG is constantly revising which facilities are deemed to be current, and so the number of facilities for which IDL is being produced changes on a regular basis.

### 3.2.4 Application Objects
Application objects encapsulate the behavior of the business. Each application object knows how to respond to requests, trigger events, and manage its relationships with other application objects. The underlying business model is implemented in application objects. Of course, the term *application* is used loosely, as an application object is really just any object that encapsulates and provides a set of discrete behaviors.

In order for clients to interact with application objects, the application objects must conform to some *rules of engagement*. These rules, as defined by CORBA, specify how an application object makes itself known to the outside world and which services it will publicly provide for clients.

Several companies are attempting to create a set of frameworks that address generic business logic issues. Two such frameworks are IBM’s San Francisco and the OMG’s Business Object Framework (BOF). In each framework, there are several domain-specific frameworks, such as Manufacturing, Healthcare,
Telecommunications, Transportation, and Finance, each of which is based on the root business object.

Not all frameworks have such lofty ideals though. Many distributed applications do not need the overhead of such a complex infrastructure and so define instead a set of specific business objects that are based on the root business object.

### 3.3 Internet Inter-ORB Protocol (IIOP)

In CORBA 1.1, the OMG did not specify the protocol that the ORBs would use to communicate with each other. As a result, the various vendors produced ORBs that used disparate communication transport layers and protocols. With CORBA 2.0, however, the OMG introduced a new communication protocol called, IIOP. In order for an ORB to be CORBA 2.0 compliant, it has to use IIOP as the communication protocol for inter-ORB interaction. IIOP is a TCP/IP specific implementation of the GIOP.

The GIOP specifies message formats and common data representation (CDR). As a result, the various ORB implementations can communicate across heterogeneous environments. The CDR takes care of any cross-platform data differences such as endian-ness and floating point representation.

The GIOP also defines the format for IORs. An ORB uses an IOR to uniquely identify, and subsequently locate, a specific object.
Chapter 4. Why Java?

If you are reading this book, we assume you are familiar with the Java language and the environment in which it runs. Rather than attempting to describe the Java language in its entirety, we discuss a few specific attributes that are worth reiterating with respect to the distributed programming environment.

We also cover the Enterprise JavaBeans specification and the implications it has with respect to CORBA.

4.1 Java Attributes

In this section, we review the attributes that make Java an excellent choice for distributed application development.

4.1.1 Object Oriented

Java is an object-oriented language. Therefore, as a developer, you can concentrate on the data and behavior of the business, rather than having to think in terms of processes. The nature of an object-oriented language inherently encourages code reuse, at both the project and enterprise level.

Sun has provided an extensive class library to augment the base function that comes with the JDK. Some of the features for which Java has defined specifications are database access, security, telephony, 3D graphics, speech recognition, mail, imaging, and help. With this rich set of functions you can spend less time developing code that has already been written by other vendors.

4.1.2 Architecture Neutral

Java achieves architecture (or platform) neutrality by using bytecodes. Bytecodes are not a new technology; however, for one reason or another the JVM has gained industry support very quickly. The JVM is a piece of software that is written for each specific operating system. It interprets bytecodes and performs the desired behavior for that operating system.

Consider the case where the bytecodes instruct the JVM to display a window. The actual end result is up to the implementation of the JVM. For example, the Win95 JVM knows how to display a Win95-style window, and the AIX JVM knows how to display an X-Window. In this way, new operating system vendors can write a JVM for their operating system and thus allow any and all
Java programs to execute on their platform with no changes necessary to the bytecodes.

4.1.3 Multithreaded

Traditionally, multithreaded development has been an art mastered by only a handful of expert programmers. Java, however, makes threading much easier, as it includes threading support as part of the language. There are Thread classes that objects can inherit from and the synchronized keyword that allows the developer to provide semaphorelike behavior for access to a variable, method, or class.

When developing a GUI-based application, it is easy to fall into the trap of having just one thread; that is, if the user begins an action that takes several seconds (or minutes) to perform, he or she is effectively locked out from any further work until that task is completed. However, Java makes it easy to spawn new worker threads to perform tasks, thus freeing the application for the user to perform other tasks in parallel.

A server application must be able to serve multiple requests from multiple clients in parallel. The use of threads, and the way in which Java provides them, makes writing a server application in Java a viable option.

4.1.4 Dynamic

Any Java class can be loaded into a JVM at any time. Instances of these dynamically loaded classes can then be instantiated. At any given time, you can query the Class object to determine its attributes and behavior, using a technique called reflection. Reflection enables you, for example, to dynamically invoke a method on an instance; construct an instance of an object, using a string that contains the name of the class; and alter values in a given instance.

4.1.5 Secure

Java has several controls to ensure a secure environment. Pointers cannot be used to directly access memory, and array overruns are detected and raised as exceptions. A bytecode verifier is run on every class as it is loaded to ensure that the bytecodes conform to a strict set of rules. Some of the things the verifier checks are whether the bytecodes are valid and that the stack will not overflow or underflow.

Java also has what is commonly called a sandbox. Much like a child’s sandbox, a Java sandbox limits the amount of mischief that a Java program can get into. The Java sandbox, as implemented on Web browsers, prevents...
the applet from reading from the hard drive, writing to the hard drive, connecting to other computers, and several other nefarious activities. As a corollary to the sandbox, Java also uses digital certificates to allow the applet to gain access to the user’s computer. A digital certificate guarantees that the providers of the applets are who they say they are. Assuming that the user trusts the applet provider and accepts the certificate, the sandbox restriction can be fully (or partially) lifted.

4.1.6 Portable

Portability is one of Java’s greatest strengths. Developers can write code in Java that they know will compile and run on any platform that supports Java (NT, OS/2, AIX, Solaris, HP, Linux, MVS, AS/400, DEC—to name but a few). Sun has announced a “100% Pure Java” initiative to encourage developers to ensure (and certify) that their code is portable. It is technically possible to write nonportable code in Java; however, Sun and others are encouraging developers to “Write Once, Run Anywhere.”

Portability is especially encouraging in an ORB environment, as it makes the interoperability issues a little easier to deal with. Although there will always be problems with communicating between disparate ORBs, having the language remain consistent among them is an excellent starting point.
Chapter 5. Why Enterprise JavaBeans?

Since the March 1998 announcement of the Enterprise JavaBeans (EJB) 1.0 specification, most of the companies already involved in CORBA, transaction monitors, or databases have announced that their products will implement the specification.

The Enterprise JavaBeans specification provides a solution for a clear separation of the business logic and the intricacies of dealing with persistency, transactions, and other middleware-related services.

Now you can create applications and concentrate exclusively on the business logic packaged as EJBs. The difficult part of dealing with all other services required by applications is provided by a container framework in which EJBs are deployed and executed.

Clients of EJBs are of course new EJB clients using Remote Method Invocation (RMI) and Java Naming and Directory Interface (JNDI) services. However, CORBA clients written in C++, as ActiveX, or in Java can be EJB clients as well. By using CORBA services, a CORBA client can have full access to EJBs.

5.1 Roles

The separation of the business logic from other tasks requires the definition of different roles.

5.1.1 Provider

The Enterprise JavaBeans provider is the developer who understands the business logic and knows how to translate it into Java code. The Enterprise JavaBeans provider needs to have an understanding of the interfaces and semantics of Enterprise JavaBeans.

The Enterprise JavaBeans provider is responsible for deciding, among other issues, whether the enterprise Bean is persistent or not. If the enterprise Bean is persistent, the enterprise Bean’s provider must provide a list of fields with persistent state. The enterprise Bean provider also must decide whether he or she is going to manage the persistency, using such support as files, databases, or CICS transactions, or let the container manage the persistency. Another decision the enterprise Bean’s provider must make concerns the behavior of the enterprise Bean when it is involved in the scope of a transaction. The Enterprise JavaBeans provider’s responsibility ends with the
packaging of the EJB and all associated files as a jar file, which is then given to the deployer.

5.1.2 Deployer
The deployer is responsible for installing the EJB classes in the EJB server. The deployer understands EJB and the runtime server environment and can map EJB developer requirements and configure them using EJB server tools. The deployer also must ensure that EJBs are accessible through JNDI.

5.1.3 Application Assembler
The application assembler writes applications that use EJBs. He or she has a client view of the EJB and can create new, composed EJBs and develop applets, servlets, or native CORBA applications that use EJBs.

5.1.4 Container Provider
The container provider focuses mainly on providing scalability, security, and transactional behavior for EJBs. The accompanying tools generate the “glue” code that creates the link between the business logic and the underlying services. The container intercepts client calls to the EJB and executes the appropriate services that ensure the transactional, persistent behavior expected for the EJB.

5.1.5 Server Provider
The server provider brings operating system and middleware services to the container. In most cases, the same vendor provides the server and container because there are as yet no specifications to define the interface between a container and a server.

5.1.6 System Administrator
The system administrator ensures that the system is working properly by using the monitoring and management tools provided by the server and container providers.

5.2 Components
In this section we present an overview of the major Enterprise JavaBeans components.
5.2.1 Container

A container is where an EJB lives. It can contain one or more EJB classes. These classes are identified by their home interface, which acts like a factory that allows clients to create, find, and remove instances of an EJB.

By using the Deployment Descriptor during EJB development, or later on by using deployment tools, it is possible to give a meaningful name to the home interface. The container places this name in the naming space that is accessible from clients using JNDI.

5.2.2 Session and Entity JavaBeans

There are two types of Enterprise JavaBeans:

- Session beans
- Entity beans

5.2.2.1 Session Beans

The life time of a session bean is typically that of its client. A session bean contains conversational states that are not persistent and will not survive a server failure. However, a session bean must indicate to its container a state management mode:

- STATELESS
- STATEFUL

In order for a container to manage efficiently a large number of bean instances, it can take an instance out of memory and store it in permanent storage. This is called passivation. When this bean is invoked again, the container creates a new instance and initializes it with the data saved during passivation. This is called activation.

Therefore, if the session bean contains a conversational state that must be preserved between method invocations, the session bean indicates STATEFUL management mode. Otherwise the container considers the session bean STATELESS and never uses passivation but instead can destroy it in case of memory resource shortage. Because all instances of stateless session beans are the same, the container can decide to use any available instance to satisfy a client request.

5.2.2.2 Entity Beans

Entity beans are used to represent persistent data. In the most common case, their fields are mapped into a relational database. But in more complex situations, the persistent data can result from the invocation of an application or the execution of an existing CICS transaction.
The persistent state of an entity bean is defined in two flavors:

1. **Bean-managed persistence (BMP)**
   With a BMP entity bean, the EJB provider manages the persistent state of the bean by coding database calls or any type of access to permanent storage. It is the provider responsibility to save and restore the state of the EJB when called by the container on such methods as: ejbFind, ejbLoad, and ejbStore.
   A BMP entity bean is inappropriate for large applications. This becomes obvious when you think about a large number of entity beans, each accessing a given database. For portability and scalability use a container-managed persistence entity bean.

2. **Container-managed persistence (CMP)**
   With container-managed persistence entity beans you do not have to know which source is used to provide the persistent state of the bean. You just have to specify which fields are persistent. Thus there is complete portability, and the EJB provider can focus on the business logic.
   The hard part is for the container. Each container provider also provides the tools to map EJB fields to databases or any existing application. In addition, it can efficiently manage database access by using a shared pool of connections and caching data. Vendors will differentiate their offerings by the number of containers they provide to access different sources, and the reliability and scalability of their implementation.
   When container-managed persistence is used, an Enterprise JavaBeans can be transparently moved from one container to another regardless of whether it is provided by the same vendor. The business logic remains untouched!

Every Enterprise JavaBeans entity bean has a unique identity within its home. Its identity is defined by a primary key. The primary key can be any class type. The only restriction is that the class must implement `serializable`, because it must be possible to pass the primary key between the client and the server.

The primary key is used by the client to create or find an instance of an EJB.
5.3 An Overall Picture

Without going into deep detail, let us look at how the major components relate to each other from a developer’s perspective.

Let us assume you want to develop the EmployeeID EJB. You first have to buy an EJB server with appropriate containers supporting the source of the persistency of your data. With the containers, the vendor must also provide tools to deploy your EJB packaged in jar files.

Assuming you have all this, you can start developing your EJBs.

For each EJB you need to decide whether it is a session or an entity bean. If it is a session bean, you have to inform its container about its management mode: STATEFUL or STATELESS. For an entity bean with maximum portability, scalability, and reliability, select a CMP bean. Otherwise implement it as a BMP. But don’t forget that mixing persistency state management with business logic is not good for portability.

For each bean, you have to provide an implementation, additional classes, and interfaces. Assuming that your bean name is EmployeeID, you have to:

1. Develop an EmployeeIDBean class
2. Develop an EmployeeID interface
3. Develop an EmployeeIDHome interface
4. If it is an entity bean, provide an EmployeeIDKey
5. Create a deployment descriptor file
6. Create a manifest file
7. Package all of these files inside a deployment jar that the container deployment tool uses to install and execute the EJBs.

The information set in the deployment descriptor contains the beanHomeName. This is the JNDI name you give to the EmployeeIDHome. It can be any name that makes sense in your organization, such as “applications/travelExpenses.” At deployment time, the deployer can still change it. When the container instantiates the EJB home, it registers in the naming space this name bound to EmployeeIDHome (see Figure 5 on page 32).

Later on, an EJB client uses the JNDI API to look up the home and then continues by creating, finding, or removing instances of the object managed by its home.
One of the major benefits of EJB comes from the separation of the business logic from the middleware supporting it. The specification also defines the contract with the container provider who provides such services as transactions, persistency, and security and the tools to generate the glue code that realizes the link between these services and the business logic.

The industry already has wide support for developing distributed objects from CORBA products. One of those products is Component Broker. When developing with Component Broker, you already benefit from a clear separation of the business logic from the middleware services. In addition, you are not limited to the Java language but can use C++ as well.
If you consider EJB too immature for developing your distributed application you can start with CORBA and in time expand it with EJBs. Component Broker has the best CORBA offering and supports EJBs.

To make the EJB offering even more attractive, VisualAge for Java 2.1 comes with complete EJB development support, including WebSphere 2.0 runtime. You can now develop your EmployeeIDBean and let the tool generate everything else including the container deployment classes. You can develop, debug, and test any type of EJBs: session beans, BMP entity beans, or CMP entity beans. VisualAge for Java also generates the jar file with all information required to deploy it in containers such as WebSphere, Component Broker, or any other vendor container.
Using VisualAge for Java Enterprise Version 2 to Develop CORBA and EJB Applications
Chapter 6. VisualAge for Java Enterprise

VisualAge for Java was designed for complete, end-to-end Java development. Using its tools, you can achieve productivity gains that no other tool suites can provide.

6.1 Dynamic and Iterative Development

The traditional edit-compile-debug development cycle can take too long in today’s competitive environment. By breaking down the sequential nature of this cycle, the VisualAge for Java paradigm makes the entire development cycle iterative.

6.1.1 Simplified Source Control

With VisualAge for Java, you no longer need to be concerned about file management and frequent backups. As program elements are created, they are automatically stored in the repository. When first created, the program element is also made available in the integrated development environment (IDE) workspace. Previously created program elements can be added to the workspace as needed. Whereas the workspace contains only a single edition of a program element (called the current edition), the repository contains all editions. In fact, all user source code is contained in the repository, including the current edition. When a program element is added to the workspace, the IDE compiles the source into Java bytecodes. Similarly, when a program element is created, the first “save” places the source in the repository, and compilation is automatically initiated.

As well as automatically creating editions, you can explicitly create versions of a class, package, or project. An open edition, denoted by time stamps, can be changed. Typically you work with an open edition until satisfied with a particular portion of code and then preserve it as a version. You can then use the version to create another edition. Thus it is possible to identify the completion of each stage of a project and ultimately to identify the content of a particular release.

The IDE also provides a powerful comparison tool for objects in the repository. This tool is especially useful for comparing different editions of the same objects. When editions of the same class are compared, any differences in the class definition are shown, as well as a list of differing methods and the type of difference: source changed, added, or deleted.
You can share code repositories that reside on a server, and team administrators can create new repositories based on existing repositories.

6.1.2 Interactive Execution

A development environment that allows programmers to enter and immediately run code snippets, without the need for batch processing and linking, is said to support interactive execution. Users of interpretive languages, such as BASIC and Smalltalk, often claim greater development productivity than users of compiled languages, such as C and C++, because they use development environments that support interactive execution.

Interactive execution allows a program to be developed and tested incrementally from the bottom up. The programmer creates new code and tests it interactively without having to develop special test programs. The productivity of users of compiled languages is constrained by the edit-compile-debug cycle. In contrast, users of interpreted languages simply enter and execute statements with no perceivable delay.

Java was designed to be interpreted, but its strong type system allows it to be effectively compiled.

In VisualAge for Java, interactive execution is supported in the Scrapbook window. The Scrapbook consists of one or more pages that can contain code snippets. Pages can be saved as text files and reloaded, allowing the programmer to build up a library of useful snippets that can be shared and reused.

To work with a code snippet, the programmer highlights its text and then pops up a menu that contains commands to run, display, or inspect the snippet.

VisualAge for Java emulates multiple virtual machine instances. Whenever an instance of a class is run, either through its main method or as an applet, a new virtual machine instance is created for it. Each virtual machine instance behaves as a separate virtual machine. The same considerations apply to code snippets that are executed from the Scrapbook window. Each snippet executes in its own virtual machine instance, providing a powerful and flexible environment.

6.1.3 Incremental Compilation and Linking

Compiled languages like C++ require that the physical layout of objects be known at compile time. This allows efficient code to be generated, but whenever a class definition is changed, any class that uses it must be recompiled. If a low-level class is changed, virtually the entire application may
have to be recompiled. In addition, linkers typically link an entire application as a batch operation; even though only a single object file may have changed, the entire application gets relinked. The compile and link time for a complex application can span several minutes or even hours. This time delay can seriously hamper development productivity because the effect of a small change to the source code cannot be quickly assessed.

Java was designed to avoid unnecessary recompilation. Java bytecode does not contain information about the physical layout of the other Java classes it uses. Instead, a Java bytecode file contains symbolic references to the methods and fields of other classes. These symbolic references are resolved at run time, when the class is loaded for execution. However, a change to a class may still affect any class that uses it. For example, if a method is deleted, any class that uses it becomes invalid. Therefore most Java development environments recompile a class whenever any class used by it has changed. In addition, most JVMs cannot incrementally load a changed class.

VisualAge for Java takes the inherent incremental capability of Java to its logical conclusion: rather than use a file-based compilation model, VisualAge for Java compiles source code incrementally whenever a field or a method of a class is edited and saved. The compiler checks the syntax of the source code and builds a list of methods and fields on which it depends. When these program elements are changed, the compiler determines the impact and marks any invalidated methods. The Unresolved Problems tab list all of the current problems in the code. The developer can navigate to the problem area and correct the problem.

Recall that VisualAge for Java emulates multiple virtual machine instances. Therefore code may be changed during execution. If a method on the call stack of any thread is changed, the stack is popped back to that method, and the new version is executed. If the fields of a class are changed, and any live instance of that class exists, they are modified appropriately. In many cases, execution can proceed in a reasonable way. Of course, some changes will be so drastic that execution must be halted and restarted to obtain meaningful behavior. However, many typical changes to methods and fields can be incorporated into executing code in a satisfactory manner. Thus VisualAge for Java is a powerful environment for developing long-running programs, such as servers.
6.1.4 Debugging

Debugging support is a critical function for any development environment. There are three categories of debugging technologies, each having its use in practice:

- **Invasive debugging**
  Inserting print statements in the code and directing the output to the console or log files
- **Remote debugging**
  Attaching a debugger to another, possibly remote, JVM process and stepping through the code
- **Local debugging**
  Debugging multiple virtual machine instances running within the IDE, and modifying the code during execution

VisualAge for Java supports local and remote debugging. However, Java application systems that use several virtual machines and that would normally require remote debugging support can be executed within VisualAge for Java because of its ability to emulate multiple virtual machine instances. Thus local debugging can emulate remote debugging.

The VisualAge for Java debugger can be activated:

- By inserting breakpoints into the code. The debugger is activated when the breakpoints are reached.
- By invoking the debugger programmatically, using a support library
- When an invalid method is invoked or an uncaught exception is thrown
- By interactively opening the debugger window and adding any available thread

Individual threads can be selected for examination. When a thread is selected, its call stack is displayed. You can then select any stack frame and view the source code for its method and its list of variables. While the source code for the method is displayed, you can edit it and save it, which causes the method to be automatically compiled and linked into the running program. You can step through the code as usual, resume execution, or kill the thread. You can also prune the call stack back to any method on the stack and restart execution from that point.

We now consider how these features come together in the important case of debugging servlets. Suppose that a servlet is being developed in a Web server that uses a standard JVM, and that, after many requests, a problem occurs. Because standard virtual machines cannot incrementally load a
modified class, the Web server must be stopped, restarted, and retested to recreate the conditions under which the error occurred, every time a code change is made. Moreover, you must work with the limited capability of a remote debugger or use invasive print statements.

Contrast the above scenario with development using VisualAge for Java. You:

1. Turn the development environment into a Web server by simply running the HTTPServer class that comes with the Java Servlet Development Kit (JSDK) from SUN.
2. Use a Web browser to create the conditions under which the error will occur and place a breakpoint in the servlet code that handles the HTTP request.
3. Generate another request, which activates the Debugger window when the breakpoint is reached.
4. Step through the source code, inspecting variables as required, until you find the problem.
5. Correct the code, save it (causing the program to be incrementally compiled and linked), and resume execution.

6.2 Construction-from-Parts Paradigm

Component modeling and visual programming complement one another. Parts created based on a component model are manipulated in a visual environment.

6.2.1 Visual Programming

VisualAge for Java offers a visual programming environment that is significantly different from that of other development tools. Most visual programming tools allow you to define the visual layout of the applet or application by selecting widgets and control mechanisms from a palette. Some tools assist in generating code to handle particular events and the relationships among the controls. This assistance typically takes the form of a wizard that generates code that is intended for a single use and is unable to handle arbitrary complex behavior. Once you customize the code, it is difficult to modify or extend it using the wizard. There is also a complete separation of the visual composition or layout of the user interface and the logic that drives its behavior; that is, support for integrating nonvisual parts on a visual surface is missing.
The VisualAge for Java Visual Composition Editor renders not only the visual elements of the user interface but also enables you to visually program the applet or application behavior. Using visible connections between both visual and nonvisual elements, you can define and integrate the logic for the user interface. Not only are visual elements displayed and manipulated; you can add beans with no visual elements on the canvas. These beans have a visual representation and can be manipulated like visual components.

The Visual Composition Editor uses the JavaBeans component model as the basis for composing and assembling beans. All connections are made between features of beans: properties, events, and methods. The JDK 1.1 event model is fundamental to the operation of connections and the code that is generated by the Visual Composition Editor to provide the behavior represented in the visual design. Thus the Visual Composition Editor is an ideal builder for creating and assembling beans.

6.2.2 JavaBeans Support

With the JavaBeans component model, you can construct whole applications from reusable parts, with potentially large savings on the development cycle. What separates JavaBeans from other components is their platform independence.

VisualAge for Java supports a variety of ways of generating and adding code to an application or bean. Through the Bean Info page, you work with code from a bean perspective. Properties, methods, and event sets can be added, and properties can be designated as indexed, bound, or constrained. If you have manually defined an event, you will certainly appreciate all the code automatically generated for you by the tool.

The standard for distributing JavaBeans is to package all related classes and resource files in a jar file. VisualAge for Java supports the importing and exporting of jar files from the IDE.

6.3 Code Generation

The use of SmartGuides increases the pace of the development cycle and can simplify difficult tasks. Coding middleware through SmartGuides makes accessing legacy resources a less daunting task.

6.3.1 SmartGuides

The VisualAge for Java IDE includes several SmartGuides that provide guidance and assist in executing particular tasks. They automatically create
or generate objects and code to meet particular requirements. SmartGuides are extensively used with the Enterprise Access Builders.

6.3.2 Code Generation for Enterprise Access

The Enterprise Access Builders that are included with VisualAge for Java Enterprise generate access code and thus enable you to quickly and easily access legacy applications, transactions, and data. Not only is the need to write middleware code eliminated; it is no longer necessary to rigorously test this high error-prone code.

VisualAge for Java Enterprise Version 1.0 includes the Data Access Builder, the RMI Access Builder, and the C++ Access Builder. VisualAge for Java Enterprise Version 2.0 provides new access builders for SAP R/3 and IDL development. With one programming interface, you can develop a single Java application that has access to diverse enterprise systems.

6.4 Additional Features

We finish this tour of VisualAge for Java by listing other important features included with Version 2.0:

• Team programming support. VisualAge for Java Enterprise provides an integrated team development environment that is based on a shared source code repository.

• Enterprise toolkits. These toolkits add scalability to your Java applications and include helpful tools for Java development. High-performance compilers maximize the execution speed of your server code. From your development workstation, you can use cross-platform debugging, testing, and performance analysis tools for applications that are targeted for OS/2, Windows NT, AIX, OS/400, and OS/390. AS/400 developers will find updates to the AS/400 feature of VisualAge for Java Version 1.01, including a new wizard for subfile support.

• Automated object to relational mapping. The Persistence Builder tool automates mapping the persistent state of Java objects to relational databases. It generates a layer of code that implements all JDBC access calls necessary to insert, update, or retrieve data for an object from an SQL database.

• CORBA applications in Java. The IDL Development Environment can be used to develop distributed object applications that run in IIOP and Component Broker environments or in third-party environments such as VisiBroker and Orbix.
• Servlet Builder. With VisualAge for Java Enterprise, you can now use visual programming techniques to create and test servlets. Custom and off-the-shelf business objects can be Web-enabled and used with the VisualAge for Java reusable parts library. By using the Servlet Builder along with the IBM WebSphere Application Server, you can test and debug Web sites built with pure HTML, compiled Java Server Pages (JSP), and visual servlets.

• Enterprise Access Builder. This tool, not to be confused with the Access Builders mentioned previously, enables you to create Java applications that use existing host applications and data to implement object behavior. The Enterprise Access Builder includes connectors for CICS and Encina; these beans are part of the same Common Connector Framework (CCF) that IBM e-Business connectors use.

• Wizards for building applications from IBM’s San Francisco Application components

• The AgentRunner tool, used for building Java-based business applications with Lotus eSuite components and for developing, debugging, and testing Lotus Notes Agents

• Tivoli beans, used for making Java applications that are “ready to manage” with Tivoli’s enterprise management software

• The Migration Assistant, which translates ActiveX components into Java beans

6.5 Conclusion

From both a technical and business perspective, strong market demand exists for powerful Java development tools. Software developers expect new tools to match existing tool suites in power and flexibility. Information technology managers require tools that exploit new information domains such as the Web, while at the same time leverage legacy enterprise data. IBM VisualAge for Java addresses these requirements, placing a premium on programmer productivity, ease of use, and robust code generation for middleware access.
Chapter 7. CORBA Environments

IBM provides CORBA environments with two products: WebSphere and the Component Broker.

WebSphere is a light implementation of the OMG specifications. The Component Broker is at the opposite end of the spectrum and provides one of the most complete implementations available in the industry.

For those of you who are not familiar with these two products, we present a brief description below.

7.1 WebSphere

IBM WebSphere is a set of products that includes WebSphere Application Server, WebSphere Performance Pack, and other products that help customers deploy and manage high-performance Web sites. It combines a runtime environment for Java servlets with connectors to common database formats; industry-standard ORBs; and enterprise middleware.

7.1.1 Servlet Runtime Environment

The Servlet Runtime Environment also known as ServletExpress, provides a fast engine for running server-side Java servlets and JavaBeans. Java servlets can be used to coordinate I/O between server-side JavaBeans and Web browser HTML clients. Servlets facilitate layering and separating the presentation logic, business logic, and back-end data access logic. Server-side JavaBeans are used for business logic and access to back-end databases, transactions and existing applications.

7.1.2 Java Server Pages

JSP enables server-side Java or JavaBeans to be integrated into HTML Web pages. JSP is analogous to how Active Server Pages enable the integration of server-side ActiveX into HTML Web pages. Without JSP, using servlets to server-side Java can be difficult because JavaBeans and HTML are mixed together inside the servlet. JSP makes it possible to keep server-side Java separate from the voluminous HTML used for the Web browser GUI. Here’s how WebSphere does it:

- JSP defines HTML tags that enable HTML pages to call server-side JavaBeans
- JSP defines a Java Servlet API that enables a servlet to call JSP HTML pages
For instance, when a servlet receives a request (for example, a button click) from a Web browser client, it calls one or more server-side JavaBeans. Once the response is formulated using a variety of database, transaction, and/or business logic JavaBeans:

- Servlets can pass the response to Web browsers by invoking a JSP HTML page
  - Without JSP, servlets need to format an HTML Web page dynamically inside the servlet
  - Hand coded and/or generated HTML inside servlets is hard to maintain or enhance over time
  - Competitive Web application servers generate non-JSP HTML and combine it into large servlets
- Accessing JavaBeans directly from JSP HTML pages is as easy as calling a Java method or property
  - JSP allows the Java developer to focus on writing the data, transactional, or business logic Java
  - JSP allows the GUI developer to focus on writing or generating HTML

### 7.1.3 CORBA Development and Runtime Support

In addition to server-to-server CORBA communication, VisualAge for Java can be used to develop applets that interact with server-side JavaBeans served by WebSphere and accessible through this CORBA support:

- CORBA Naming Service
- Server-to-client callbacks
- IDL codegen: object interfaces can be defined in Java or IDL.
- Automatic server object activation based on servlet requests from any Web browser
- Easy server-side object creation using distributed object factories: IHome and EJBHome
- Extensible generic object server: organizes groups of server objects into servlets
- Pass objects by value: surpass CORBA's limitation of primitive data types by passing objects
- Java exception support: provides transparency to exceptions that span distributed objects
7.1.4 WebSphere Performance Pack

The Performance Pack enhances Web application server scalability, availability, and load balancing. It combines load balancing, caching, proxy and filtering functions, file content management, and replication into a single Internet hosting infrastructure. It is the reason behind the records set for Web server performance at the Winter Olympics in Nagano, Japan in 1998 (103,000 hits per minute with normal response time).

Runtime platforms include Windows NT, AIX, and Sun Solaris. A similar range of function is available on OS/390 by using Domino Go Webserver V5.0. OS/390 Domino Go Webserver (DGW) V5.0 exploits the OS/390 Workload Manager (WLM).

The Performance Pack features:

• eNetwork Dispatcher
  • Dynamic load balancing of inbound requests
  • Web servers are clustered as one logical server
  • Transparent to clients and WebSphere servers
  • Real-time operation (no reboot or shutdown required)

• Andrew File System (AFS)
  • Web content is mirrored and synchronized among multiple servers for scalability
  • Data replication is automatic, nondisruptive, and realtime.
  • Files can be anywhere in the enterprise.
  • Load balancing criteria and status information are mirrored in high-availability configurations for:
    • “Hot-standby” WebSphere servers
    • Automatic and immediate transparent takeover

• WebTraffic Express
  • A caching proxy server complements load balancing.
  • A buffering facility reduces redundant outbound TCP/IP traffic.
  • Pages are cached locally to minimize outbound network traffic.
  • Can be used to cache “high hit ratio” pages.
7.1.5 WebSphere Developer Studio

- Optimizes VisualAge for Java, NetObjects Fusion, and NetObjects Script Builder for servlets
- Provides a team development environment based on NetObjects Team Fusion
- Wizards-based codegen of servlets, JSP, and HTML Web pages

7.1.6 Enterprise Java Server for EJBs

WebSphere 2.0 provides complete support for EJBs.

7.2 Component Broker

In this section, we provide an introduction to IBM’s new Component Broker. To develop a full understanding of Component Broker, we recommend that you read IBM Component Broker Connector Overview, SG24-2022. For those who have already read that redbook, you may find some of this text familiar, as it is simply a more concise version of the IBM CBConnector Overview redbook. Because, this time, we expect our audience to be developers, we will keep the introduction short and sweet. However, reuse of someone else’s work is a major paradigm in object technology. So, why not join the bandwagon? We start off by going into what Component Broker is; then we address the areas where we expect this new technology to be applied. Finally, we position the product with respect to related IBM packages.

7.2.1 What Is Component Broker?

IBM’s Component Broker is a new, integrated package that allows you to develop, deploy, run, and manage a new generation of distributed, multitier, object-oriented applications. Figure 6 on page 47 shows its major components.
7.2.2 Component Broker Dimensions

The three dimensions of Component Broker are:

- Runtime
- Development
- System management

7.2.2.1 Runtime

Component Broker connects to most of today’s popular client environments. It supports clients written as Java applets and applications, clients developed for Microsoft’s ActiveX platform, and clients written in VisualAge for C++. While you certainly can continue to run fat-client topologies, Component Broker is ideally suited for the thinner client variety. Component Broker does not restrict you in any way from using your favorite GUI frameworks or tools.
You code the user interface and any local business logic as usual. In addition, the developers of your Component Broker server code will provide you with the constructs needed to hook up to your server. What you get depends on your client platform. In any case, the appropriate source code usage bindings are available for all of the above environments. Java code can be downloaded or cached locally on the client, as usual. In the other environments, you have to install the necessary dynamic link libraries (DLLs) to build and run your application. All clients connect to the application server through a CORBA-compliant ORB using IIOP. However, the core of Component Broker lies in its middle-tier application server. The server allows you to separate the traditionally extremely “volatile” client software environment from the code that implements your business logic. It also isolates that business logic from back-end technology specifics, such as databases or transaction monitors. Thus it provides you more opportunity to concentrate on the demands of your business, leaving the idiosyncrasies of the other tiers to the specialists in those areas. The main components housed within the middle-tier application server are:

- **Application Adaptor and Managed Object Frameworks**
  These components provide the core infrastructure and scaffolding for your business logic to enable it to run in a mission-critical environment. They take care of such aspects as transactional behavior, connecting to back-end datastores and transaction monitors for persistence or legacy reuse, and interfacing to OMG object services. Application Adaptor and Managed Object frameworks “administer” what is called a *Managed Object Assembly*. Of that assembly, what is “visible” to a developer are the following objects:
  - **Business objects**
    These contain your actual business domain code, that is, your business logic.
  - **Data Objects**
    These isolate the business objects from the specifics of back-end databases and transaction monitors. The data objects themselves use the services of other, framework-provided objects to implement the connection to back-end data stores. They can make use of extensive caching services to provide the kind of performance you would expect in today’s online environments.
  - **Managed objects**
    These “wrap” business and data objects. The framework interfaces primarily with managed objects. When business logic operations are invoked on a managed object, the implementation is delegated to your business object. Component Broker-required methods are either implemented by the managed object itself or delegated to other parts of
the runtime environment. Examples of the latter are setting up the transactional environment and implementing object services such as the identity methods used by clients to compare object references.

There are a number of additional objects, such as:

- **Homes**
  In “object speak,” these are both factories and collections. They provide life-cycle functionality for objects of one particular type. They are also used to implement the query facility as well as iterability over the objects they contain.

- **Containers**
  These shield client programmers from hairy details, such as memory management for your business objects. They take care of activating and passivating objects according to policies you can specify through systems management facilities. In general, containers provide you with an administrative boundary for implementing policies on your business objects. These policies are set up at install and are then used at runtime to set up the “quality of service” required in the case at hand.

The frameworks are extensible, in that the implementation of framework interfaces can be adapted to your specific needs. One good example would be to extend them to run with your own application frameworks. Another might be the need to support a back-end database environment that is not yet supported by Component Broker per se.

- **Object Services**
  Object Services include:

  - **Naming Service**
    The Component Broker Naming Service is compliant with the OMG specification with some extensions to make working with the naming tree easier.

  - **Security Service**
    At the time of writing, an implementation based on the Open Software Foundation/Distributed Computing Environment (OSF/DCE) is available. It supports authentication.

  - **LifeCycle Service**
    Component Broker extends the OMG standard by giving you a measure of control over where you want to place your objects within the network of servers. This is accomplished through a location. A location embodies the notion of proximity, which may be interpreted in a geographical, structural, or even temporal sense.
• **Event Service**
The Component Broker Event Service at the Release 1 level implements OMG’s transient, nontyped events.

• **Externalization Service**
The Externalization Service has two main purposes:
  - Moving or copying the state of objects to different memory locations, between hosts, or simply creating a new object with the state of another existing object
  - Moving the state of objects in and out of a persistent store

• **Query Service**
You can access Component Broker server objects through keys. You can then follow links, using references to get to related objects. Component Broker also enables you to execute queries with the query engine and query evaluators it provides. Component Broker cooperates with back-end RDBMSs, in particular with DB2, to optimize query performance. In particular, it uses a query pushdown technology to evaluate as much as possible on the back-end. In this way, indexes present in the database can be used efficiently, and unnecessary data movement is avoided.

• **Transaction Service**
The Transaction Service is implemented mostly under the cover by Application Adapter Frameworks that support this requirement. Component Broker can act as a transaction coordinator, managing two-phase commit operations with back-end database systems. It works with the XA protocols, as defined in the X/Open Distributed Transaction Processing specification. It can be connected easily to other XA compliant resource managers.

• **Concurrency Service**
The Concurrency Service does the job you would expect of it when you want to serialize access to resources. All of the well-known types of locks are supported.

• **Object Server Runtime**
The Object Server manages the collaboration among most of the other parts of the Component Broker platform. With respect to user code, it coordinates system resources such as execution threads or context information related to transactions or security, for example.

• **Workload Management**
This essential scalability feature brings load balancing into the picture. It allows clients to talk to a single, logical server image. In reality, operations that clients invoke may well be dispatched to different physical server
processes. Introducing a new dimension in distributed object processing, Server Groups bring increased scalability and reliability to the picture.

- **Language Interoperability**
  Essentially, the Component Broker object model is based on CORBA 2.0. Moreover, Component Broker includes a set of runtime libraries that support inter-language calls between CORBA 2.0 objects within the same process. Interlanguage Object Model (IOM) is the language interoperability technology that enables objects written in C++ and objects written in Java to interact cooperatively within the same process.

  As you should expect from a distributed objects system worth its name, Component Broker client code does not have to be coded in the same language used for your server objects. At the time of writing, Component Broker supports Java, VisualAge for C++, and the languages supported within the Microsoft ActiveX platform.

### 7.2.2.2 Development
Of the tools within the Component Broker development environment (Figure 6 on page 47), the Object Builder is the major tool you use to define the structure and Component Broker-specific interfaces of your objects. Using a set of wizards, the Object Builder guides you through defining and building all of the constructs necessary in the Component Broker environment. The Object Builder generates and helps you build the scaffolding you need right up to creating a deployable install image for servers and clients. Thus, it saves you much of drudgery you would otherwise have to endure.

The Object Builder also has a bridging facility to outside analysis and design tools, so you can bring your analysis models into the Component Broker world and extend them for Component Broker as appropriate. In addition, you can import IDL files or RDBMS DDL files to begin reengineering your existing applications and databases.

### 7.2.3 Systems Management
You use Component Broker’s systems management facilities to define the network of host machines and their configurations of servers and application software and to parameterize the runtime environment.

At install time, you use the systems management facilities to introduce your applications and deploy them within those configurations, as needed. At runtime, you control the environment, bringing up servers, enabling or disabling applications, and so forth. Rather than being grafted on top as an afterthought, systems management was integrated into Component Broker
from the start, for example, by instrumenting the code with the appropriate hooks for controlling the system and collecting management information from it.

7.2.4 Component Broker Focus Areas

The time has come to talk about the “amorphous, elliptical blob” to the left of Figure 6 on page 47. Essentially, Component Broker is a platform for developing a new generation of flexible software built using distributed object technology. It is an ideal base for implementing thin-client topologies, cleanly separating the client-side concerns of ever-changing user interface technologies from the very different characteristics of the code needed to support business logic. The Internet is of course the most prominent example, and Component Broker is well suited to that dynamic environment.

One major focus area is the operational reuse of existing code and databases. One large customer in the insurance industry recently expressed enthusiasm about Component Broker’s potential for reusing legacy code. Specifically, the customer estimates that its investment in domain-specific, home-grown software is on the order of 10 times its present manpower capacity and believes it would be ludicrous to think about throwing all that away and starting any major system from scratch. IBM research shows that this situation is more the norm than the exception with Fortune 500 companies.

Another typical requirement today is that of integrating separately developed application subsystems, a situation aggravated by the recent surge in mergers and acquisitions. Through its middle-tier Business and Application Objects server, Component Broker offers an integrative platform that can bring together hardware and software systems.

7.2.5 Component Broker: Member of the Transaction Series Family

Component Broker does not stand alone but rather cooperates with existing transaction and resource managers. It functions as an object server by providing an application environment that lets clients work with mature back-end systems through object-oriented middleware. It cooperates fully with IBM’s industry-leading family of transaction servers.

Component Broker incorporates proven technology from other IBM middleware products such as CICS/ESA, Transaction Series (Distributed CICS and Encina), as well as OSF/DCE. Over time, IBM intends to add Component Broker technology to Transaction Series so that these existing, proven platforms will also benefit from the new computing model.
Future offerings will expand support for other enterprise systems (for instance, IMS) and clients (such as Smalltalk). Further integration with systems-management environments, such as the Tivoli Management Environment (TME), is planned. Extensions to Component Broker and corresponding extensions to the “Shareable Frameworks Project” will enable Component Broker and Java-AS/400 business frameworks to work together.

Component Broker development spans IBM teams and laboratories, ensuring that the product will take advantage of the diverse experiences and many disciplines required to build an integrated solution. Component Broker, a third-generation object-technology offering, is object-oriented and CORBA-compliant. The ultimate goal is for it to work seamlessly with other CORBA-compliant products in the marketplace.
Chapter 8. The Travel Expense Application

To demonstrate the new features of VisualAge for Java, we define a simple, yet typical, application that uses several CORBA services. The application is not a showcase of object technology; rather it highlights some of the benefits that VisualAge for Java provides when you develop a CORBA application.

In this chapter we outline the business domain for the application, introduce the use cases that are derived from the requirements, and discuss some nonfunctional requirements such as security and performance.

The requirements and use cases are not defined as thoroughly as they would be in a real project. Our intent is simply to give you an idea of the functionality the application can provide.

8.1 Application Overview

In today’s business environment, traveling interstate or overseas to perform work duties has become a commonplace activity. One of the more onerous tasks associated with traveling on business is submitting expense claims to recoup the costs of the business trip. The Travel Expense Application (TEA) is designed to make this task a little easier by enabling users to make online claims.

The following benefits will accrue from using TEA:

• Elimination of traditional paper forms
• Submittal of expense claims from any location that has a connection to the Internet
• Dramatic reduction in the amount of time spent entering claim details
• Eventual online processing of claims by Accounts Payable (outside the scope of this application)

TEA enables users to define a business trip and a collection of expenses associated with it. At any stage, a business trip can have zero or more expenses. A trip must have a justification, a start date, and an end date.

Once users have entered all of the expenses associated with their trips, they can submit the claim for approval. Once users have submitted their claims for approval, they cannot add any more expenses, unless Accounts Payable

1.For user interface screens, see Appendix B, “TEA User Interface” on page 219.
rejects the claim. The approval or rejection of a trip is outside the scope the application.

TEA supports three types of expenses:
1. Hotel expense - contains details about an overnight stay in a hotel
2. Rental car expense - contains details about rental car hire
3. Airline expense - contains details about flights

Only these three expenses are currently required, but the application should be designed to be extensible as the introduction of other types of expenses is foreseeable.

All expenses have a mandatory date, amount, description, payment type, and country where the expense occurred. Each type of expense has its own specific details as well. For example, the airline expense requires a flight name and number.

Users should be able to list all past expenses and the trips associated with those expenses. Having selected a previous expense, users should be able to view the details.

A separate, but related, part of TEA is a department monitoring application that allows online real-time viewing of a department’s expenses. This application is immediately notified of any change in the department’s expenses and displays the effect of such changes.

8.2 Use Cases

We use a very simple adaptation of the use case methodology to indicate the roles, actions, and responsibilities for the TEA. Each use case describes a discrete action or set of actions that can be performed within the system. The use cases are at the analysis level and thus do not contain details about the actual implementation. In a typical application design, you may want to document design use cases; however, for the purposes of our application, analysis use cases are sufficient.

8.2.1 UC01 - Log On

Users of TEA must be authenticated before they can use the application. If users cannot correctly identify themselves, they cannot use the application.

8.2.1.1 Preconditions

None
8.2.1.2 Steps
1. The system prompts the User to log in to the application.
2. The user provides a number and password to the system.
3. The system verifies that a user with that serial number exists and the password provided is valid (see Exception 1).

8.2.1.3 Postconditions
1. The user is logged in to the application.

8.2.1.4 Exceptions
1. The user is notified that the login attempt was unsuccessful.

8.2.2 UC02 - Create a Trip
Every expense in the application must belong to a trip. A trip has a start date and an end date that indicate the length of the trip. A trip also has a reason. These pieces of information are mandatory.

8.2.2.1 Preconditions
1. The user has logged in.

8.2.2.2 Steps
1. The system prompts the User for the mandatory trip reason, start date, and end date.
2. The user asks the system to create the trip.

8.2.2.3 Postconditions
1. A trip has been created, and the user can associate expenses with the new trip.

8.2.2.4 Exceptions
None

8.2.3 UC03 - Create or Modify an Expense
The user of the application can associate expenses with a trip.

8.2.3.1 Preconditions
1. The user must be logged in.
2. A trip with which this expense is associated must be selected.
8.2.3.2 Steps
1. If the user wants to create an expense, the system prompts the User for the type of expense to create.
2. The user can provide or modify the fields for the specific type of expense.

8.2.3.3 Postconditions
1. An expense has been modified, or created and associated with the trip.

8.2.3.4 Exceptions
None

8.2.4 UC04 - Submit Trip for Approval
After users have entered all expenses for a trip, they submit the trip for approval by Accounts Payable. Users can add expenses over a period of time until all expenses are entered. They do not have to enter all expenses at one time. Once users have submitted a trip for approval, however, they cannot add any more expenses.

8.2.4.1 Preconditions
1. The user must be logged in.
2. A trip must be selected.

8.2.4.2 Steps
1. The user asks the system to submit the trip for approval.
2. The system modifies the state of the trip to reflect submission.

8.2.4.3 Postconditions
1. The trip has been flagged as submitted.

8.2.4.4 Exceptions
None

8.2.5 UC05 - List Expenses
Users can list their trip expenses for review and modification.

8.2.5.1 Preconditions
1. The user must be logged in.
8.2.5.2 Steps
1. The system prompts the user for the trip for which to list the expenses.
2. The system displays the expenses for that trip.
3. If the trip has not been submitted for approval, the system allows the User to select an expense to review or modify (see UC03).

8.2.5.3 Postconditions
None

8.2.5.4 Exceptions
None

8.2.6 UC06 - Monitor Department Expenses
Managers of departments can view online in real-time the expenses incurred for their departments.

8.2.6.1 Preconditions
1. The user must be logged in.

8.2.6.2 Steps
1. The system prompts the user to select a department to monitor.
2. The system continually updates the information presented to the user to reflect the current expenses of the department.

8.2.6.3 Postconditions
None

8.2.6.4 Exceptions
None

8.3 TEA Object Model
We use the Rational Rose modeling tool to model the TEA business domain. Figure 7 on page 62 shows the business domain object model.
Figure 7. TEA Object Model
Chapter 9. Environment Configuration

Some exercises that we take you through during the course of this book assume that your environment will be set up in a particular way. In this chapter we document some assumptions about the location of various entities and walk you through the various steps to configure your environment.

9.1 Directory Locations

Most products let you choose where to install them, and the products that we cover in this book are no exception. We do not make any assumptions about where you install the various products. However, we document the locations where we installed the products so that we can have a consistent frame of reference throughout the book. Table 1 shows the directory locations of the various products. When we reference files from within the various product directories, we use the fully qualified file and directory name.

Table 1. Product Installation Directories

<table>
<thead>
<tr>
<th>Product</th>
<th>Directory</th>
</tr>
</thead>
<tbody>
<tr>
<td>VisualAge for Java</td>
<td>D:\IBMVJava</td>
</tr>
<tr>
<td>WebSphere</td>
<td>D:\WebSphere</td>
</tr>
<tr>
<td>Lotus Domino Go</td>
<td>D:\WWW</td>
</tr>
</tbody>
</table>

9.2 Client/Server Scenarios

Several different scenarios might arise as your development life cycle progresses. Typically, you will use VisualAge for Java to develop and debug both the client and server portions of the application. However, as time progresses you will want to test the various components of your application in an environment that more closely resembles your deployment environment. Fortunately, VisualAge for Java provides several facilities to help you in this transition.

The most typical scenario is when you move from testing both the client and server in VisualAge for Java to testing the client in a real Web browser but
keep development in VisualAge for Java. Table 2 summarizes the responsibilities of the client and server in several different scenarios.

Table 2. Summary of Browser-Web Server Scenarios

<table>
<thead>
<tr>
<th>Browser</th>
<th>Web Server</th>
<th>Description</th>
</tr>
</thead>
</table>
| VisualAge for Java | VisualAge for Java | - The client can use the VisualAge for Java classpath to find the classes.  
- The server can use the VisualAge for Java classpath to find the classes. |
| VisualAge for Java | Lotus Domino Go  | - The client must ignore the classes in VisualAge for Java and have the classes served to it by the server. The appletviewer must be invoked directly against the server URL.  
- The server must have all required classes in its classpath. |
| Netscape      | VisualAge for Java | - The client must have no classes in its classpath, and it must rely on the server to serve the classes.  
- The server must be able to serve classes as well as handle servlet requests. |
| Netscape      | Lotus Domino Go  | - The client must have no classes in its classpath, and it must rely on the server to serve the classes.  
- The server must have all required classes in its classpath. |

As you can see, you have to handle quite a few behaviors. Fortunately, VisualAge for Java, with some collaboration from WebSphere, can provide a solution for each of these scenarios.

9.3 Configuring VisualAge for Java and WebSphere

In this section we cover the steps necessary to enable VisualAge for Java to act as a Web server to both Netscape and VisualAge for Java clients. The steps mostly involve setting up VisualAge for Java, but you also have to configure several WebSphere and Lotus Domino Go files.

9.3.1 Adding VisualAge for Java Features

Decide which user you want to own the WebSphere projects and log on to VisualAge for Java, using that user. Press the F2 key to add new features to the VisualAge for Java workspace (see Figure 8).
Figure 8. Adding Features

Click on the OK button and select the features that you want to add. In this case, we want to add both IDL Development Environment 2.0 and IBM Servlet Builder 2.0 (see Figure 9).
It may take VisualAge for Java some time to import these features into the workspace. The length of time VisualAge for Java takes to import the features is dependent on the speed of your PC.

9.3.2 Importing WebSphere Classes

WebSphere provides a built-in Web server written in Java. In this section we describe how to import the classes you need to run the Web server.

Create a new project called *WebSphere Application Server Libraries* in the workspace. Import the following jar files into the newly created project:

- D:\WebSphere\AppServer\lib\jst.jar
- D:\WebSphere\AppServer\lib\ibmwebas.jar
You may notice (by clicking on the All Problems tab) that there are four compilation errors, but it is safe to ignore them. Do not version the project yet.

Create a new project called WebSphere Runtime Libraries in the workspace. Import the following jar file into the newly created project:

- D:\WebSphere\AppServer\web\class\ibmjbrt.jar

Again, do not version the project yet.

9.3.3 VisualAge for Java and WebSphere Workarounds

At the time of writing, there are a couple of known incompatibilities between VisualAge for Java and WebSphere 1.0 (should be fixed in WebSphere 2.0). The first is a bug that causes an exception to be thrown when trying to get a reference to the WebSphere naming context, and the second is one that prevents the Web server from starting correctly. To fix these problems, follow these steps:

1. Create a new edition of the WebSphere Runtime Libraries project and of the com.ibm.CORBA.iiop package within that project.

2. Import the ORB.class file that is located in the utilities\ directory from the sg245276.zip file (see Appendix E, “TEA Package Descriptions” on page 227) containing source and executable code (see Appendix D, “Workaround Details” on page 225 for details on creating this class manually).


4. Create a new edition of the Sun JSDK class libraries project and of the javax.servlet and javax.servlet.http packages within that project.


6. Create BeanInfo classes for the following classes within their respective projects:
   - javax.servlet.GenericServlet
   - javax.servlet.http.HttpServlet
   - com.ibm.servlet.service.SEServlet
7. Version and release the Sun JSDK class libraries project. (VisualAge for Java automatically versions and releases any packages and classes that have changed inside that project.)

8. Version and release the WebSphere Application Server Libraries project. (Again, VisualAge for Java automatically versions and releases any packages and classes that have changed inside that project.)

9.3.4 Importing the Travel Expense Application

The TEA (see Chapter 8, “The Travel Expense Application” on page 57) is located in the `code\tea.dat` directory (in the sg245276.zip file) in interchange file format. Import this file into your workspace.

9.3.5 Setting Up a Web Server in VisualAge for Java

WebSphere provides a Web server as part of its base distribution. The class that contains the main() method to invoke the Web server is called `com.ibm.servlet.SERunner`. In theory, you could run that class directly; however, we recommend that you create an empty subclass of it and use the subclass to start the Web server. In this way, you can have several Web server environments (by creating several subclasses) that have different operating environments.

For an example of what our subclassed Web server looks like, view the `tea.servlet.MyWebServer` class, or use the Java source code shown in Figure 10 on page 68.

```java
public class MyWebServer extends com.ibm.servlet.SERunner {
    {
    }
}
```

*Figure 10. Subclassing the WebSphere Web Server*

Edit the classpath properties of MyWebServer (see Figure 11 on page 69). Note, in particular, the values in the Command line arguments field and the Complete classpath field. The Command line arguments field tells the Web server where the WebSphere product is installed, and the Complete classpath field sets the execution-time classpath for the Web server.
Figure 11. Class Path Properties of Web Server

Usually, you can use the Compute Now function to generate the classpath. However, in this case, because our application’s servlet is instantiated and invoked dynamically (through Class.forName()), the Travel Expense Application project is not included in the list. Using the Edit function on Project path (see Figure 11), manually add the Travel Expense Application and WebSphere Runtime Libraries projects (see Figure 12 on page 70).
9.3.6 Configuring the Web Server for Clients

Using the Web server you just installed, VisualAge for Java can act as a server for both VisualAge for Java clients and Web browser clients. However, there is a subtle difference between these two scenarios:

- **Appletviewer** - The classes that are needed can be found in the VisualAge for Java workspace.
- **Web browser (Netscape/InternetExplorer)** - The browser looks for the classes in the classpath environment variable. If it does not find them, it asks the Web server to send them.

In the second scenario, VisualAge for Java cannot send the classes that it has in its workspace; rather it must fetch them from the file system. As a result, when you want to use a Web browser to connect to the VisualAge for Java server, you must export your classes to the file system.
Follow these steps to configure the Web server:

1. Edit D:\WebSphere\AppServer\properties\server\servlet\server.properties, and ensure that it contains the lines exactly as shown in Figure 13.

```
server.service.autostart=adminservice pluginservice servletservice
server.trace=ServerProcess ServiceManager
```

*Figure 13. server.properties File*

2. In the D:\WebSphere\AppServer\properties\server\servlet\pluginservice directory, edit the doc.properties file and ensure that it contains the line in Figure 14.

```
doc.root=d:\www\html
```

*Figure 14. doc.properties File*

3. The Web browser can find the WebSphere Runtime classes (which are located in D:\WebSphere\AppServer\web\classes\ibmjbrt.jar) in different ways:

   • You can place the jar file in the operating system classpath. In this case the browser automatically finds the classes with no transmission necessary. This is the recommended method.

   • You can unjar the jar file into the D:\WWW\HTML directory, and the browser will automatically request each class as it is referenced. This is an easy solution, but many uncompressed classes are transferred over the network. The good news is that you do not have to modify the HTML of your applet to use this solution.

   • You can place the jar file, intact, in the D:\WWW\HTML directory and change the HTML file to contain an `<ARCHIVE>` tag that points to ibmjbrt.jar. In this way, all WebSphere classes are transferred in one compressed batch. However, the user must wait for the entire jar file to be sent, and you have to change the generated HTML file from VisualAge for Java.

   • You can place the jar file into

     netscape\communicator\program\java\classes or winnt\java\classes directory if you use Internet Explorer.

You only have to perform these Web server configuration steps once.
9.3.7 Testing the Web Server

As a simple test to ensure that you have set up the Web server correctly, we recommend that you create a simple HTML file, and a correspondingly simple servlet. See Figure 15 for an example of an HTML file, and Figure 16 for an example of a servlet.

![HTML File Example](image)

Figure 15. A Simple HTML File: D:\WWW\HTML\Test.html

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

public class DummyServlet extends javax.servlet.http.HttpServlet
{
    public void doGet(HttpServletRequest request,
                        HttpServletResponse response) throws IOException
    {
        response.setContentType("text/html");
        PrintWriter pw = new PrintWriter(response.getOutputStream());

        pw.println("<TITLE>Test Servlet</TITLE>");
        pw.println("<BODY><H2>Lets see now... does this work?</H2>");
        pw.println("<P>Yes... I do believe it does!");
        pw.println("</BODY>");
        pw.flush();
        pw.close();
    }
}
```

Figure 16. A Simple Servlet: tea.servlet.DummyServlet

Place the HTML file in the D:\WWW\HTML directory and create the servlet inside the tea.servlet package in the Travel Expense Application package.

To start the Web server, select the MyWebServer class and run it. Check the VisualAge for Java console (see Figure 17) to confirm that the Web server has started correctly. Occasionally, the Web server fails to start correctly. (We have been unable to determine why.) If you see extra messages after the last
line shown in Figure 17, the Web server has not started correctly, and you should kill it, and restart.

![Console Showing Web Server Starting Successfully](image)

**Figure 17. Console Showing Web Server Starting Successfully**

As an extra precaution to ensure that the Web server has started correctly, we urge you to check the debugger (see Figure 18 on page 74) and ensure that there are two threads for the pluginservice and two threads for the servletservice.
Figure 18. Debugger Thread List Showing Web Server Starting Successfully
Chapter 10. CORBA Server-Side Development

In this chapter we discuss the steps required to design and implement a WebSphere CORBA server application. We cover some general concepts of WebSphere such as bootstrapping, life cycle, and naming services. To illustrate these concepts, we discuss our implementation of the TEA (see Chapter 8, “The Travel Expense Application” on page 57 for a detailed discussion of the application functionality).

10.1 WebSphere Overview

WebSphere consists of a CORBA 2.0 ORB and a collection of some of the CORBA services. The communication model for WebSphere is slightly different from most vendors ORBs in that it uses an applet-to-servlet paradigm. Each servlet hosts one or more server object instances in a container. A container can have several homes associated with it; each home can create, locate, and delete object instances.

CORBA does not support the notion of homes and containers as part of the specification, so if you choose to use a home to manage your objects' life cycle, be aware that it is nonportable code. Homes and containers, however, are an integral part of the EJB specification. WebSphere 1.0 contains a partial implementation of the EJB specification, and WebSphere 2.0 is expected to contain a full implementation.

10.1.1 IIOP versus HTTP

Most WebSphere interactions use IIOP to communicate with the ORB that is managing the object instance. However, several WebSphere interactions use HTTP to communicate with a WebSphere servlet. In this section we draw the distinction between the IIOP and HTTP interactions.

An IIOP interaction includes both raw IIOP and IIOP-over-HTTP (commonly referred to as tunneling). Strictly speaking, tunneling uses HTTP under the covers; however, the point is that the request is routed directly to an ORB-managed object, not to a Web-server-managed servlet.

An HTTP interaction is one where the client-side ORB makes an HTTP connection to the Web server and invokes a servlet. The servlet returns the desired result through the Web server.

IIOP is used in all cases except the following:
• orb.resolve_initial_references("NameService") uses HTTP to connect to the CosNamingServlet to get an IOR that identifies the actual naming context.

All method calls invoked on that naming context use IIOP as per normal CORBA operation; it is only the initial connection to get the IOR that uses HTTP.

• WebSphere-specific activities such as Utility.getIHome() and IHome.create() use HTTP to connect to the appropriate servlet. In the case of Utility.getIHome(), the servlet that it connects to is passed in as a constructor parameter.

10.2 Server Initialization

In a typical CORBA server application written in Java, you develop a class that contains a main() method, which the JVM invokes. However, a WebSphere server application does not have a main() method, as it operates in a servlet environment. Let us take a moment to discuss what “operating in a servlet environment” means.

The first time that a servlet is invoked (usually using the IHome methods), the Web server instantiates an instance of the servlet and invokes a method called init(). This method is where you would perform any once-off initialization. The Web server then invokes the servlet’s service() method. Subsequent invocations of the servlet see the Web server calling the service() method immediately (that is, init() is not called again).

WebSphere provides a servlet called GenericObjectServlet that contains enough implementation to act as a CORBA server with no additional development on your part. It is likely that you will want to provide some extra behavior that is specific to your application. By subclassing the GenericObjectServlet, you can reuse most of the WebSphere behavior yet still add your own. TEA’s subclass is called TEAServlet, of which two of the key methods are shown in Figure 19 on page 77.

The getImpls() method overrides the method found in GenericObjectServlet to return a list of the mapping between interface names and implementation class names. That is, if your application asks the IHome to create an Authenticator, WebSphere will call getImpls() to determine which concrete implementation to instantiate. This method is called only once, during the servlet initialization sequence.

The init() method overrides the method found in GenericObjectServlet. Note that it calls the init() method of the superclass before doing its own
processing. This is very important as this is how the ORB gets initialized. We then assign the orb variable (from the superclass) to a static variable so that we can get to the ORB later from other classes. We then call the initTEA() method (not shown), which creates some initial TEA objects. To see the complete class definition of TEAServlet, browse the tea.servlet.TEAServlet class in the Travel Expense Application project in VisualAge for Java.

```java
public Properties getImpls();
{
    Properties props = new Properties();
    props.put("Authenticator","tea.business.impl.AuthenticatorImpl");
    props.put("QueryService","tea.business.impl.QueryServiceImpl");
    return props;
}

public void init(ServletConfig config) throws ServletException
{
    super.init(config);
    _orb = orb;
    initTEA();
}
```

Figure 19. tea.servlet.TEAServlet Snippet

10.2.1 Initializing WebSphere Servlets

One of the disadvantages of using a servlet as a CORBA server is that the init() method is not invoked until a client uses an IHome to try and create or reference an ORB-managed object.

If your client application already has an IOR that gets converted to an object reference through string_to_object(), the client connects to the ORB directly and never has to call your servlet. If you are unsure why this is, please reread the section entitled “IIOP versus HTTP” on page 75.

The net result is that if you have initialization code that must be executed and you are not using an IHome, you must manually force your servlet's init() to be executed. To do this, you must add your servlet to the WebSphere Servlet Configuration panel (see Figure 20 on page 78) and specify Yes for the Load at Startup radio button (see Figure 21 on page 79). To get to these panels, point your browser to http://server:9090/ where server is the machine on which your Web server is installed.
Figure 20. Adding a Servlet to WebSphere Configuration
Strictly speaking, you do not have to add your servlet to the WebSphere configuration; however, we recommend that you do so.

The properties file that gets updated with the information that you specify in the configuration panel is called servlets.properties and is located in the D:/WebSphere/AppServer/properties/server/servlet/servletservice directory. Figure 22 shows an extract of the changed lines that result from your changes.
10.2.2 Setting WebSphere Classpath

When you run your application outside VisualAge for Java, the Web server needs to know where to find your servlet. This is typically done by using the classpath variable for the Web server. However, WebSphere intercepts requests for all servlets, so you must configure the WebSphere classpath. Figure 23 shows how you can set the classpath variable in WebSphere. For our application, we use the D:\WWW\HTML\java directory as the root directory. When you save the classpath variable, WebSphere updates the ncf.jvm.classpath variable in the jvm.properties file located in the D:\WebSphere\AppServer\properties\server\servlet\servletservice directory.

```
servlets.startup=invoker sam CosNamingServlet TEAServlet
servlet.TEAServlet.code=tea.servlet.TEAServlet
```
A fundamental part of application development is managing the life cycle of business objects. Such management is of particular interest in a distributed, object-oriented environment because:

- The creation of an object can be initiated from any number of clients.
- The object itself can reside on any number of servers (or perhaps even span multiple servers).
- The object can be moved among servers.
• The object can be "deleted" from many clients.

Each of these operations has associated with it issues that you need to address if you are to provide a robust distributed application. In this section we discuss how our application addresses each of those issues.

Note that in a large enterprise-scale application, the approach would more likely be based on a framework that encompasses persistence, life cycle, and transaction management. However, such a framework is beyond the scope of this book.

10.3.1 Creating Objects

The most common approach to the creation of objects is to use a factory. A factory, as you would imagine, is used to create objects. However, it does not just create any object; it creates instances of objects that conform to a given interface. The method that is used to instantiate the objects is shielded from the rest of the application. The applications can be guaranteed that they will get an object that conforms to the given interface, but they do not know which particular implementation will be instantiated (see [Gamma 1995] for a detailed explanation of the factory pattern).

Factories link particularly well with the way that CORBA operates, in that CORBA applications do not know about the implementation of objects; they merely use references to objects that conform to an interface.

10.3.1.1 Single User - Single Server

One version of the factory uses a technique known as the singleton pattern, which ensures that there is one, and only one, instance of a class. The key elements in a singleton pattern are a private instance variable, an instance() method, and a private constructor. See [Gamma 1995] for further information on the singleton pattern.

Figure 24 on page 83 shows an example of a very simple factory (as yet unaware of any CORBA semantics). Notice that although the factory creates a DepartmentImpl object, it is actually declared as returning an object that conforms to the Department interface. This is the cornerstone of the factory pattern.
Figure 24. Simple Factory

The simple factory in Figure 24 is of minimal value in a CORBA application, however, as the newly created object has not been registered with the ORB, and it is not a CORBA skeleton object. Accordingly, the object can be referenced only from within the local JVM that created it.

10.3.1.2 CORBA-Aware Implementation
Let us examine how we might go about moving to a CORBA-aware solution. Consider the code displayed in Figure 25 on page 84. It is a little more complex because we have introduced some infrastructure; however, it also gives us the starting point for our CORBA application.

```java
public interface Department {
    public String getName();
    public float getBudget();
}

public class DepartmentFactory {
    private static DepartmentFactory _instance = null;
    private DepartmentFactory() {
    }
    public DepartmentFactory instance() {
        if (_instance == null) _instance = new DepartmentFactory();
        return _instance;
    }

    public Department create(String name, float budget) {
        Department dept = new DepartmentImpl(name, budget);
        dept.name(name);
        dept.budget(budget);
        return dept;
    }
}
```
import org.omg.CORBA.Object;

public class BusinessObjectFactory
{
    protected Hashtable _allInstances = new Hashtable();
    public Hashtable allInstances()
    {
        return _allInstances;
    }
    public Object registerWithORB(Object object)
    {
        // first thing we do is connect the object to the ORB
        getORB().connect(object);
        // then we place the object in our cache of instances
        _allInstances.put(((BusinessObject)object).getId(), object);
        return object;
    }
}

public class DepartmentFactory extends BusinessObjectFactory
{
    private static DepartmentFactory _instance = null;
    private DepartmentFactory()
    {
    }
    public DepartmentFactory instance()
    {
        if (_instance == null) _instance = new DepartmentFactory();
        return _instance;
    }
    public Department create(String name, float budget)
    {
        Department dept = new DepartmentImpl(name, budget);
        dept.name(name);
        dept.budget(budget);
        return registerWithORB(new _DepartmentSkeleton(dept));
    }
}

Figure 25. CORBA-Aware Factory
The first item of interest is that our factory now has a superclass. We would like all of our factories to be able to display some common behavior, such as registering objects with an ORB and keeping a cache of the objects that have been instantiated.

The registerWithORB() method takes a CORBA object and connects it to the ORB. It then adds the object into the cache of instances, using a unique identifier that the object provides (through the BusinessObject interface).

The DepartmentFactory class has changed in only one way. Now, before returning the object, it registers it with the ORB. Notice that, because we are using the delegation (or tie) model, the factory class creates an encapsulating CORBA skeleton before passing the object to be registered.

You can implement the CORBA stub-skeleton model using one of two models:

- **Inheritance model**
  
The inheritance model is based on the assumption that the business object classes inherit from the org.omg.CORBA.portableSkeleton class. Thus the business object gains the functionality of a skeleton object for free. Extra coding is not required.

  The main disadvantage of this model is that it is often hard to modify existing business object hierarchies to fit into the CORBA hierarchy. For example, many persistence frameworks require that you inherit from a base class somewhere in their framework so they can control transactions. In this case, you must decide which is more important, the distributed nature of the application, or the persistence layer. Obviously, this decision will depend on many factors including whether the distributed transport layer might have to be changed transparently (that is, switch from CORBA to RMI to DCOM), how much functionality the persistence framework provides, and the cost of reworking the object model to fit into the new hierarchy.

- **Delegation model**
  
The delegation model is used when there is an existing hierarchy (or perhaps a set of business objects that have been designed independently of the communications layer) that has to be used in a CORBA context. In this case, the IDL compiler for Java generates a tie class (which inherits from the org.omg.CORBA.portableSkeleton) that will encapsulate the business object. Any public interface calls to the business object go through the tie wrapper first.
Using the delegation model, you can reuse existing business objects without having to change their hierarchy, or object model. However, an extra object has to be instantiated for each business object. In many applications, this is not an issue; however, in applications containing millions of business objects, this certainly becomes a consideration.

We chose to use the delegation model within WebSphere because we wanted to keep our business logic separate from the transport layer. Another consideration was the implementation of our persistence layer; our initial thoughts were that we were going to use the EJB model for persistence.

The BusinessObjectFactory not only provides the base behavior for all factories but also implements a very crude version of the CORBA Query Service. At the moment, this “query service” only supports returning all instances of a given class (it does not even allow for removal of objects from the cache). In the absence of a CORBA Query Service, the TEA uses this query service functionality to get a list of all Country objects.

**10.3.2 Copying and Moving Objects**

As your application increases in complexity, you will want to provide a clean, consistent way of moving your objects around and creating copies of them. Unfortunately, because of its small size, the TEA has no need to provide this functionality. However, a brief discussion of the issues involved is a worthwhile exercise.

In a truly distributed application, you should be able to create copies of any given object not only in the same server that the object exists in but also in any other server. Similarly, you could quite feasibly want to move an object from one server to another.

Because of WebSphere’s limited implementation of the CORBAservices (in particular, the absence of the Life Cycle Service), there is no framework support to help you in implementing the copying and moving. WebSphere also does not support copying objects between multiple WebSphere ORBs. If you decide that your application needs this sort of functionality, then we recommend the following approach:

1. Read the OMG Life Cycle Service specification [OMG CORBA Services 1996, Section 6-1] to gain an understanding of the CORBA life cycle model.
2. Create a Factory Finder for each of your factory classes.
3. Implement copy and move methods in your factories with the appropriate behavior.
This approach does not make your application CORBA-compliant, but it will facilitate the transition to another CORBA-compliant ORB.

### 10.3.3 Deleting Objects

Most applications must be able to delete their objects. However, deletion is an area that should be treated very carefully. Consider the case where you have two remote clients (A and B) that are both referencing an object. If client A deletes the object, then suddenly client B is left referencing an object that no longer exists. This is known as a dangling reference. A common way of alleviating this problem is to use a locking scheme to ensure that the clients cannot delete objects if another client is referencing them. However, this introduces many issues, most of which are outside the scope of this book.

As mentioned earlier, WebSphere does not implement the Life Cycle Service, so removal of the objects is entirely up to you as a developer. Again, we recommend that you read the OMG specifications to see how the CORBA life cycle model works, so that you can easily migrate to a conformant ORB. Specifically, this means implementing a method called remove().

### 10.4 Naming Services

An ORB uses an IOR to uniquely identify objects; however, an IOR is not very human-readable (see Figure 26). Therefore CORBA applications can use the Naming Service to easily find distributed objects, but instead using an intuitive naming convention. For example, an employee can be uniquely identified by a serial number, so when the application wants to find a specific employee, it can ask the Naming Service for an object that has a string ID of the serial number.

```
IOR:000000000000000000000002349444c3a7465612f627573696e6577332f41757468
656e746963617468f723a312e3000000000001000000000000000003800010100
0000000762617269756d000000d5c000000000035e438
4600000006000000000008000000000000000000
```

Figure 26. Sample IOR

This strategy is slightly flawed when it comes to a real-life implementation of a distributed application. If you were to register thousands (or even millions) of objects with your Naming Service, performance would quickly degenerate to the point of unusability. A more typical usage would be to register FactoryFinders with the Naming Service, and, once the FactoryFinder has
been found, the client application can ask it to find the object for which it is looking.

The WebSphere Naming Service is implemented with a servlet called CosNamingServlet. When a call to the resolve_initial_references() method is made to find the NamingService, the WebSphere ORB contacts the CosNamingServlet, using HTTP. In this manner, the IOR of the Naming Service is retrieved (also known as a naming context). Once this naming context has been retrieved, normal IIOP (or IIOP-over-HTTP if tunneling is used) operations are used to communicate with the Naming Service.

From an application development perspective, you do not have to be concerned that the Naming Service is implemented using a servlet. It is completely transparent from both a coding and an administration perspective.

Figure 27 shows a very simple example of how to use the Naming Service. Note that it conforms exactly to the CORBA specification. As far as we are aware, WebSphere offers no proprietary extensions to the OMG-defined specification.

```java
NameComponent nc1 = new NameComponent("TEAApplication");
NameComponent nc2 = new NameComponent("Employees");
NameComponent nc3 = new NameComponent("258780");
NameComponent[] context = {nc1, nc2, nc3};

Object object = rootNamingContext.resolve(context);
Employee emp = EmployeeHelper.narrow(object);
```

![Figure 27. Example of Using Naming Services](image)

This is a somewhat contrived example, as a large enterprise (such as IBM) would have hundreds of thousands of employees, and the application would never register all employees with the Naming Service. However, as you can see, the Naming Service does allow you to define object hierarchies to a varying degree.

### 10.5 Compiling IDL Files

VisualAge for Java provides, through the IDL Development Environment feature, the facilities necessary to edit, compile, and manage CORBA IDL files. When you install the IDL Development Environment feature (see Chapter 9, “Environment Configuration” on page 63), a new tab labeled IDL is added to the project and package browsers. By clicking on this tab, the IDL Development Environment is displayed (see Figure 28).
VisualAge for Java uses an **IDL Group** to maintain collections of related IDL files. An IDL Group has no special meaning other than a method that you can use to group your IDL files into related collections.

You can create IDL files manually by clicking on the IDL Group and using the **IDLS/Add IDL Object...** menu item. Alternatively, you can import the IDL files directly. We supply the IDL files for the TEA in the `\TEAapplication\Cb\EmployeeApplication` directory of the sg245276.zip file.
To generate the CORBA stubs and skeleton classes, you must first specify which IDL compiler you are using and which types of stubs and skeletons to produce. Right-mouse button click on the group, and select the **Change Compile Options...** menu option. The Change IDL-to-Java Compile Options window is displayed (see Figure 29).

![Change IDL-to-Java Compile Options](image)

**Figure 29. IDL Compile Options**

The Other Compiler option allows you to run an external program to generate the stubs and skeletons. VisualAge for Java does not reference the classes in its workspace when attempting to execute the external compiler. Therefore, you must specify the name of the compiler, and the classpath explicitly. Here is the exact text of the Command entry field:

```
java -classpath d:\ibmvjava\ide\program\lib\classes.zip;d:\websphere\appserver\web\classes \ibmjde.jar com.ibm.idl.toJava.Compile
```
The -fallTIE compile option indicates that we are using the tie- or delegation-based approach. Click on OK to save the options.

To actually compile the IDL files, you can use the IDLs/Generate Java/All Objects menu option. By selecting the IDL group, you can compile all IDL files, or if you choose, you can select individual IDL files for compilation. After the compilation, VisualAge for Java automatically imports any generated files.
Using VisualAge for Java Enterprise Version 2 to Develop CORBA and EJB Applications
Chapter 11. Java ORB Client Development

Java offers many advantages for developing distributed applications, not the least being the ability to create full-functioned user interfaces with a relatively small footprint. In this chapter we discuss how we used the Java Abstract Windowing Toolkit (AWT) to develop a 100% Java client applet for the TEA (see Figure 30). We also explain in detail some of the idiosyncracies of WebSphere when you use it in this environment.

11.1 Architectural Overview

The Java applet uses a Model-View-Controller (MVC) architecture to manage the interactions between the business objects and the user interface. If you take a look at the tea.applet and tea.controller packages in the TEA project in VisualAge for Java, you will see that for each user interface component (for example, tea.applet.ListTripsView), there is a corresponding controller (tea.controller.ListTripsController). In addition, for each controller, there are
some associated event listeners and their events
(tea.controller.ListTripsEvent and tea.controller.ListTripsListener). Figure 31 shows the general architecture of the Java client applet.

![Java Applet Client Architecture](image)

Each of these controllers is, in turn, managed by an instance of the
tea.controller.AppletController class (not shown in Figure 31). The TEAApplet is actually very simple, in that it merely provides a method called showPanel() that allows the AppletController to manage which panel is displayed at any given time.

### 11.2 Bootstrapping

The very first thing the TEAApplet does is to instantiate an instance of the AppletController and ask it to start:

```java
showStatus("Please wait. Connecting to server...");
_appletController = new AppletController(this);
_appletController.start();
```

The AppletController initializes the ORB by constructing an instance ofClientORBUtility. The ClientORBUtility is an abstract class that we use to help us transparently switch between ORB vendors. You will find that most ORB functionality is provided in the tea.util.WebSphereClientORBUtility class.

```java
if (_orb == null)
{
```

---

[94] Using VisualAge for Java Enterprise Version 2 to Develop CORBA and EJB Applications
Properties props = new Properties();
props.put("org.omg.CORBA.ORBClass", "com.ibm.CORBA.iiop.ORB");
_orb = (com.ibm.CORBA.iiop.ORB)ORB.init(_applet, props);
}

WebSphere does not really do anything special when initializing the ORB; the behavior is as you would expect from every other vendor.

11.3 Life Cycle

In a typical CORBA application you usually use factories on the server to instantiate objects, as this approach facilitates managing distribution, creation policies, persistence rules, and many other scalability concerns. However, having said all that, WebSphere does provide you with a utility class that gives you access to an IHome object. An IHome enables you to create server objects directly from the client.

In TEA we use an Authenticator class that takes a serial number and a password, authenticates them, and returns the Employee object that matches the serial number. Ideally, in our server application (TEAServlet) we would instantiate an Authenticator and register it with the Naming Service so that client applications can find it and use it to authenticate users.

However, in our initial development efforts, we were unable to get access to the naming context from the server side (due to a bug in WebSphere). As a result, we could not bind the authenticator to the Naming Service. To overcome this, we elected to instantiate the authenticator from the client, using an IHome. The naming context was accessible from the client, so we were able to bind the authenticator to the naming service.

As the development of TEA continued, we discovered a workaround on the server side that enabled us to obtain a reference to the naming context (see Chapter 16, “WebSphere and Component Broker Interoperability” on page 173). However, rather than change our application at the last minute, we decided to leave the code as is.

We create only two objects in the client applet, namely, the Authenticator and the QueryService, both of which use the same technique. First we see whether the Naming Service contains a reference to an object called “auth”:

```java
NameComponent[] authName = {new NameComponent("auth", "")};
NamingContext nc = getNamingContext();
_authenticator = AuthenticatorHelper.narrow(nc.resolve(authName));
```
If so, we narrow it to an Authenticator reference. If not, an exception is thrown, and we create an IHome so that we can create other objects:

```java
URL url;
String host = _applet.getCodeBase().getHost();
int port = _applet.getCodeBase().getPort();
url = new URL("http://"+host+":"+port+"/servlet/tea.servlet.TEAServlet");

IHome home = Utility.getIHome(getORB(), url);
Object object = home.create(tea.business.Authenticator");
_authenticator = AuthenticatorHelper.narrow(object);
```

The IHome is used to create the authenticator. It is important to note that the authenticator is created on the server side, not the client. We merely obtain a CORBA reference to the server implementation (as per normal CORBA operation). Once the authenticator has been created, we bind it to the naming context so that other clients (or this one) can find and reuse it:

```java
nc.bind(authName, _authenticator);
```

We use the same logic when we create the QueryService. To see the complete implementation, browse the tea.util.WebSphereClientORBUtility class.

### 11.3.1 Using the Correct Code Base

When developing your applet in VisualAge for Java, there is one particular quirk that you should be aware of: When you run an applet from within VisualAge for Java, a temporary file is used to contain the HTML file, and the appletviewer is invoked on that temporary file. In simple applets, this is not normally an issue, however, when you develop CORBA applications, it becomes more relevant.

The protocol the appletviewer uses to load the temporary file is not HTTP. Instead, it loads it from the file system, which leads to a code base of:

```java
file:/d:\IBM\Java\IDE\project_resources\Travel Expense Application/
```

Certain operations will cause WebSphere to establish a connection back to the Web server that served it (see Chapter 10, "CORBA Server-Side Development" on page 75). To determine the address of the Web server, WebSphere asks the applet for the code base and, using this code base, connects to the Web server. Obviously, this strategy will work only for applets that have been served by a Web server (that is, applets that contain an HTTP-based URL).
To trick WebSphere into thinking that it has been served from a Web server, you can override the getCodeBase() method on the applet. See Figure 32 for an example of how to do that or browse the tea.applet.TEAApplet class in VisualAge for Java.

```java
public URL getCodeBase()
{
    URL url = super.getCodeBase();

    if (url.getProtocol().equals("file"))
    {
        try
        {
            url = new URL("http://127.0.0.1:8080/");
        }
        catch(Exception e)
        {
            url = super.getCodeBase();
        }
    }

    return url;
}
```

*Figure 32. Overriding the getCodeBase() Method*

### 11.4 Naming Service

The Naming Service is where WebSphere differs slightly from most other vendors’ ORBs. When you invoke the orb.resolve_initial_reference() method, the WebSphere client ORB attempts to open an HTTP connection, using this URL:

http://server:port/servlet/CosNamingServlet?initref=true

where *server:port* is the address from which your applet has been served. The CosNamingServlet returns an IOR that the client ORB converts (using the string_to_object() method) to a naming context reference. If the client ORB is unsuccessful in getting the IOR from the CosNamingServlet, the ORB attempts to get the IOR, using a bootstrap host and port.

Once the client ORB has the naming context IOR, naming context operations occur as per normal (that is, using IIOP to communicate with the Naming Service). It is only in the fetching of the naming context IOR that WebSphere does something unique.
11.5 Security

The CORBA specification defines a Security Service that could be used to manage authentication and authorization. WebSphere, however, does not implement it, so we developed a minimal authentication scheme.

When the applet starts, the user is presented with a logon panel where he or she can enter a serial number and a password. The applet then asks the naming context for the authenticator and passes the serial number and password for authentication. On the server side, the authentication is trivial in that it merely checks that the password is the same as the serial number. Obviously, this is not a valid authentication scheme, but you could easily replace it with a more robust mechanism.

If the authentication is successful, an Employee object is returned to the applet; otherwise, the user is notified of the error and prompted to try again.

11.6 Query Service

The TEA can display a list of all countries in a drop-down list box when the user creates an expense. In a normal application, you would probably use the CORBA Query Service to get the list; however, WebSphere does not implement the Query Service, so an alternative method must be used. We chose to implement a pseudo-query service object that, although it does not exactly implement the CORBA Query Service interface, should make it easier to migrate to another ORB vendor or upgrade if WebSphere implements the Query Service in a future release.
Chapter 12. Pure HTML Client Development

The Java AWT enables you to create user interfaces with a small footprint. However, as small as that footprint might be, such technology restrictions as the following often make the use of a Java client applet infeasible:

- Specific application invocation performance requirements. For example, the application must take no more than 15 sec to load.
- Reuse of older technologies. Often a business cannot mandate that the users of the application must be using a certain level of software on the client. In this case, many clients may not have the required technology to support Java on the client.
- Security policies. Often enterprise network firewalls preclude client applications from operating interactively with the enterprise systems.

To overcome these restrictions, we provide a client application for the TEA that presents the user with 100% HTML markup (see Figure 33). In this chapter we describe how we developed the application, with detailed discussion on the more unusual aspects of using WebSphere in this way.

Figure 33. TEA: Adding Pure HTML Client
12.1 Architectural Overview

Like the Java applet client (see Chapter 11, “Java ORB Client Development” on page 93), the HTML application uses an MVC architecture to manage the interactions between the business objects and the user interface. We took the Java applet architecture diagram shown in Figure 31 and expanded it to include the HTML client application architecture (see Figure 34).

![Diagram of HTML Client Architecture]

Figure 34. HTML Client Architecture

The interactions among the various components in the HTML client application are simple enough, but they differ slightly from both traditional user interface and Internet development. Consider the simple case where the user logs on to the system, is presented with a list of trips, and elects to create a new trip. The following steps detail the various interactions:
1. The user points the Web browser to the HTML page that represents the home of the application.

2. The LogonController servlet is invoked on the Web server. The LogonController knows that the user has not logged on yet, so it passes control to the LogonServlet. The LogonServlet knows how to generate the HTML required to let the user log on. This HTML is passed back to the browser.

3. The user types in a serial number and password and clicks on the Logon button.

4. The LogonController, after authentication, passes control to the ListTripsServlet. The ListTripsServlet knows how to generate the HTML to list the trips, and the HTML is passed back to the browser.

5. The user clicks on the New Trip button.

6. The ListTripsController passes control to the NewTripServlet. The NewTripServlet knows how to generate the HTML to create a new trip, and the HTML is passed back to the browser.

7. The user enters the appropriate information for a new trip and clicks on the Save button.

8. The NewTripController creates a new trip and passes control to the ListTripsServlet.

The general logic flow for a given screen, say XX, is that the XX’s servlet generates the appropriate HTML (which contains a pointer to XX’s controller). XX’s controller determines which is the next screen that needs to be displayed and passes control to its servlet.

There is a slight flaw in this architecture in that, unlike the Java client applet, there is no controller controlling the controllers. Therefore the application flow logic has to be duplicated in both the AppletController (in the Java client applet) and the various controllers for the HTML application. Ideally, the AppletController would be redesigned slightly to be able to control both the applet’s controllers and the servlet controllers. This is left as an exercise for the reader.

It is important to note that the servlets that we developed for TEA are not WebSphere-derived. We developed them using the Servlet Builder in VisualAge for Java, and they bear no relationship to the WebSphere servlets other than the fact that, as part of the application logic, they use the WebSphere CORBA libraries, which in turn communicate with the WebSphere servlets.
12.2 VisualAge for Java Servlet Builder

VisualAge for Java Enterprise provides a Servlet Builder tool that enables you to easily build applications that are accessible through the Web. With the Servlet Builder, you use the Visual Composition Editor to build dynamic Web pages using HTML components, much as you would use java.awt components to build windows for an ordinary graphical user interface. The Web pages you build provide the user interface for your application, and you implement program logic by making connections and writing scripts using Java. The Servlet Builder provides a set of visual and nonvisual beans designed for building Web applications. The beans are in the Servlets category.

The visual beans in the Servlets category represent the various HTML elements that can be displayed by a Web browser. Because you are building an HTML page, you cannot use java.awt and other VisualAge for Java visual beans in the user interface. The full power of Java programming, including nonvisual beans such as the VisualAge for Java database access beans, are still available to you when creating your Servlet Builder application.

Servlet Builder beans have the ability to render themselves as HTML elements. At run time, instead of creating windows and user-interface controls on your screen, the beans generate HTML elements that re-create your user interface in a Web browser. Incoming requests from Web browsers trigger the generation of HTML code by the application.

Most of the Servlet Builder bean properties correspond to attributes of HTML elements. To find out how the property of a Servlet Builder HTML bean is used, refer to an HTML reference.

When you build a Web page, you will find that the editing behavior of the Visual Composition Editor automatically flows the HTML elements so that they appear as they will when viewed in a Web browser. This differs somewhat from building a VisualAge for Java application, again because the architecture of HTML does not support as many options for customization and layout as java.awt components do. The Servlet Builder supports only what HTML supports, and it will not let you build a user interface that will not work on the Web.

We used VisualAge for Java’s Servlet Builder to create all the servlets that render HTML pages for the TEA application Figure 35 shows an example taken from the Visual Composition Editor.
Figure 35. Using Servlet Builder from the Visual Composition Editor

Figure 36 shows the corresponding screen as seen from the Web browser.
12.3 Bootstrapping

When the LogonController servlet is invoked by the Web browser, it asks the LogonManager object to authenticate the user. The LogonManager makes the appropriate calls to initialize the ORB and then obtains a reference to the authenticator object. The Java applet goes through these very same steps when it starts; however, there is a further complication.

In a normal server-side ORB initialization, the signature of the init() method contains an array of strings (compared to the applet client which contains a reference to an applet). If you initialize the ORB by passing an array of strings, however, it will throw a COMM_FAILURE exception when you attempt to resolve the Naming Service (see “Naming Services” on page 105).
To work around this, we developed a class called KludgeApplet that you can pass to the init() method. In this way, your application "tricks" WebSphere into thinking that it is running as an applet:

```java
Properties props = new Properties();
props.put("org.omg.CORBA.ORBClass", "com.ibm.CORBA.iiop.ORB");
ORB orb = ORB.init(new KludgeApplet(), props);
```

The Kludge Applet class is defined as an inner class in the tea.proxy.ServletController class.

### 12.4 Life Cycle

The pure HTML client solution uses no new life cycle techniques other than those discussed in the Java client applet (see “Life Cycle” on page 95).

### 12.5 Naming Services

The pure HTML client solution uses no Naming Service techniques other than those discussed in the Java client applet (see “Naming Services” on page 105).

### 12.6 Security

Security is the one area that deserves particular focus when developing servlet-based applications. Because HTTP is a stateless protocol, the user must be either reauthenticated every time, or some state information must be kept by the application to ensure that users are who they say they are.

For example, the pure HTML client application has many servlets that are used to present the user interface and to control the application logic. Each of these servlets can be invoked explicitly by a malicious user, so you must ensure that controls are in place to prevent a user from connecting directly to the ListTripsServlet.

The Servlet Builder in VisualAge for Java provides a component called SessionDataWrapper that conveniently manages information that is kept between invocations of servlets. When the user logs in to the TEA, the application stores the serial number and password in these session data variables. If a servlet is invoked and there is no stored serial number and password, the user is considered to be unauthorized and, instead of being presented with the requested servlet, he or she is presented with a logon screen.
To introduce Component Broker in the application, we modify the TEA with a new object (see Figure 37).

Figure 37. TEA: Adding Component Broker Object

The object, running under Component Broker runtime, acts as an extension of the Authenticator object used to identity users of the TEA application. Therefore, if we find a record corresponding to this user in this table, we consider this user authorized to use the TEA.

Component Broker has been designed to accommodate all types of development situations. If you are developing a brand new application and are free to create new databases, you use a top-down development. If you start from existing applications and databases and want to create a new application, reusing existing legacy data and applications, you use a bottom-up development process. Sometimes, you may have started developing your application and want to reuse, let us say, an existing database. In this case you use a meet-in-the-middle development process. Component Broker provides the tools to help you handle these various combinations. We use the
meet-in-the-middle process because we are going to extend the TEA and reuse the DB2 SAMPLE database.

In this chapter we describe the process of using Component Broker to develop a distributed application. To use all of the services and features Component Broker provides, you must accurately follow several steps in the development process, using Component Broker-specific development tools that enable objects to be managed by the Component Broker’s frameworks.

If you are already familiar with the concepts of containers in Enterprise JavaBeans, you will find similarities in developing Component Broker. The difference is that in Component Broker there is no separation between the business logic developer and deployer tools.

With Component Broker, you can implement your object in C++ and Java. CB supports C++, Active X, and Java technology. In this chapter you use a Java implementation and Component Broker Version 1.3.

13.1 Component Broker Core

Component Broker is middleware that provides a complete middle-tier application environment. The core infrastructure of Component Broker consists of components that interoperate with each other to achieve back-end data stores, transactional systems, and legacy system interfaces. To allow your objects to use all system services, Component Broker provides an environment where all the objects are managed.

Within this environment, services such as transaction, persistence, and security are implemented by the environment itself (Application Adaptor) or delegated to other components. An Application Adaptor is a base framework. It describes the basic interaction between business objects and Application Adaptor implementations. By customizing this framework, you can adapt it to your specific requirements.

For example, Component Broker already supports Application Adaptor for DB2 and Oracle databases or CICS. An instance of an Application Adaptor is also called a container.

You can think of a container as a collection of managed objects using a configured Application Adaptor. You could have, for example, an instance of a DB2 Application Adaptor accessing the database X and another instance of a DB2 Application Adaptor accessing database Y. All objects within an instance are managed according to the configuration of the Application Adaptor instance.
A container also interacts with Homes, which are the birthplace of the objects or so-called factories and interfaces with the ORB through the Basic Object Adapter (BOA). An object must be manageable by the system to live within a container. Figure 38 shows the core of the Component Broker.

![Component Broker Core Diagram](image)

Figure 38. Component Broker Core

From a developer’s point of view, you do not have to know all of the low-level details of the Component Broker components. By adapting your objects to the system environment, Component Broker shields all low-level details.

### 13.2 Development Phases

When you develop a distributed CORBA-compliant application, first you design your object-oriented model, using a CASE tool. During this phase, you define the interfaces you are going to implement, the interactions between your objects and the hierarchy of the model. With the logic view of the
classes, you generate the interfaces, using IDL to allow your objects to be used across a distributed CORBA-compliant environment.

The implementation of the objects consists of the business, data, and presentation logic. But often you need to reuse existing code in your application or interface legacy systems. You may provide back-end data stores or transactional monitors. Sometimes you have to wrap old technologies to be reusable in your application.

The whole development process takes a long time, and you would like to focus mainly on your business logic instead of implementing the persistence of the objects, providing security services, rewriting old applications, or using transactional monitors in your business implementation.

By using a Component Broker framework, you can focus on the business logic, delegating persistence, transaction monitor, security and all system services to the Component Broker environment. While you design your model, you need to use the Component Broker’s framework to add all supported functions to your application and more precisely to your objects. Following the model and the hierarchy of the framework, you make your objects manageable through the Component Broker environment, which interfaces system resources with manageable objects. According to the configuration of the Application Adaptor, the Component Broker runtime manages all the objects and provides all services needed to shield the low-level details from the developer.

Basically, the Component Broker development process consists of the following steps:

1. Design the model
2. Use Object Builder tool
3. Create Component Broker client application support
4. Create a server application containing the business objects

13.2.1 Design the Model

Analysis and design constitute the first phase of development. Component Broker provides a bridge to the Rational Rose tool to allow you to use the tool to design and import the model into Object Builder’s development environment. The Component Broker frameworks are available in a cat format, thus it can be imported into Rational Rose. You can use all the classes used by Component Broker frameworks while you design the model. Once you have finished the model, using the bridge to Rational Rose you can
export it to Object Builder. Figure 39 shows the logical view of the Component Broker’s managed frameworks imported within Rational Rose.

Once the bridge is installed, in the Rose menu you have the export Object Builder XML model item. When you have completed the model, you can export the model, using the menu->Object Builder XML model. Figure 40 shows how the new object is positioned in the Component Broker object architecture.
As you can see, you have to model your design, using the Component Broker frameworks.

### 13.2.2 Object Builder Tool.

The Component Broker frameworks and Component Broker toolkit provide the development environment to adapt your objects to the Component Broker runtime. Object Builder is the main development tool in Component Broker.

With Object Builder, you can create the interfaces, IDL files, implementations, and all language-specific files for running your application on the client and the server. You can define the persistence policy of your objects, mapping their essential state to data stores, which can be DB2 or other databases. All the class and method implementations of the Component Broker frameworks needed to manage your objects by the system are included and defined within Object Builder. When you have defined the interfaces of your application, you can generate the language-specific source files and all files needed by the Component Broker runtime.
Using Object Builder, you define the methods and the attributes of your interface. Once you have implemented the interface's methods, with some additional information provided by you, Object Builder generates all the files for managing the persistence of the object, such as Data Object (DO) and Persistent Object (PO), the schema mapping, the implementation of the Key and Copy helpers, and all of the C++ code needed to wrap your Java object.

With Object Builder, you also define the behavior of the container in which all instances of your classes live. By packaging all generated code in shared libraries and Java jar files, Object Builder makes your object manageable by the Component Broker runtime and thus it can be used remotely by a client application.

Using the Object Builder user interface, you generate the files simply by selecting the Generate option in the opened pop-up menu from the User-Defined Business Objects folder. After that, you can build the application by selecting Generate All -> Java Default Target in the opened pop-up menu from the Build folder. Object Builder generates all the files and builds the required DLLs and jar files, hiding all details about the way it does that. Thus you can focus on the main problem: the business logic.

Object Builder stores all of the generated files in a directory named with the same name of your project. In our sample, we have a project named employeeApplication, so Object Builder creates the employeeApplication directory, and two subdirectories, working and model. In the model subdirectory, Object Builder stores the UNI model. In the working\nt subdirectory (if you develop on the NT platform) are all of the generated files needed to build the DLLs, jar files, and configuration files. Some of these files are used by the System Manager to load the application.

In your EmployeeID sample, for your business logic the Object Builder generates the files listed in the Figure 41 on page 115 in the working directory.

<table>
<thead>
<tr>
<th>_EmployeeIDBOBase.java</th>
</tr>
</thead>
<tbody>
<tr>
<td>EmployeeIDKeyHelper.java</td>
</tr>
<tr>
<td>EmployeeIDKeyImpl.java</td>
</tr>
<tr>
<td>EmployeeIDCopyHelper.java</td>
</tr>
<tr>
<td>_EmployeeIDCopyImpl.java</td>
</tr>
</tbody>
</table>

*Figure 41. Object Builder Generated Java Files*
_EmployeeIDBOBase is the main class of your development and contains your business logic implementation. It inherits from IManagedClient._IManageableBase and extends the EmployeeIDBO interface where you have defined the methods of your EmployeeID object.

The _EmployeeIDBOBase class also includes the relationship with the data object, which holds the essential state of your object. With the initializeState(), syncToDataObject(), syncFromDataObject() and other methods, you delegate the persistence of the essential state of the object to the data object (in our case, EmployeeIDDO). You must implement the interface methods. Object Builder implements the methods related to the data object and the methods used internally.

The EmployeeIDKeyHelper.java and _EmployeeIDKeyImpl.java classes are necessary to uniquely identify the instances of a class. You can decide which attributes of the classes to use to find or create the instances. The classes are created and implemented by Object Builder.

The EmployeeIDCopyHelper.java and _EmployeeIDCopyImpl.java classes are optional. When you have a class with several attributes, rather than creating an instance on the server and then setting each attribute that results in a method invocation through the network, Copy Helper improves performance by allowing the client to set all attributes and then send the information within the request to create a new instance of the object. These classes are generated and implemented by Object Builder.

Unfortunately, tasks done with Object Builder cannot be done with a different tool. Any modification of your interface must be synchronized with the Object Builder’s UNI model, which keeps track of all information about the application you are developing.

All of the files shown in Figure 41 on page 115 can be generated only by Object Builder. VisualAge for Java or another development tool cannot be used to generate them.

Because Object Builder holds all the information about the application in a UNI model file, Object Builder must be notified of all changes made to the generated files so that it can update the related UNI file. For example, you can edit the Java source files outside the Object Builder environment and modify the implementation. But you have to synchronize the changes with the Object Builder’s UNI model. Figure 42 shows how to import files modified with an external development tool.
Although you can use an external development tool to modify these files, there is one limitation: Any changes must be made within the interface methods. You cannot add a new method or a new class variable because after they are imported, they get lost.

As a workaround to this limitation, rename the file you have modified (in our case to _EmployeeIDBOBase.java) in Object Builder’s working directory Object Builder will pick up your file instead of the original file during compile time. Keep in mind, however, that this workaround works only if you manually process the makefile generated by Object Builder. If you edit your modified file within Object Builder, you still lose all your changes because Object Builder synchronizes this file with the file in the UNI model.
Figure 43 shows how to specify to Object Builder which files you have modified.

Let us suppose that you have modified the `serialnumber()` method. Figure 44 shows the shell window where you can see which files Object Builder has updated.
The only way to develop outside Object Builder is to code the methods in different files and with Object Builder link each method of the interface with a specific file. This is not very productive specially in a team environment. Figure 45 shows the Object Builder panel where you can link a method’s code to an external file.
When you develop Component Broker objects, you have two different development processes: client and server. Component Broker’s ORB is written in C++, but there is a Java ORB for Java clients. Server objects have to communicate with an ORB written in C++. You have to produce different bindings for client-side and server-side applications. On the server side, to allow an object written in Java to talk to another object written in C++ and vice versa within the same process, you have to use the interobject modeling (IOM) technology, which uses cross-language calls to interoperate between C++ and Java. IOM stubs are different from ORB stubs, so they are separately emitted and packaged. Component Broker also requires some C++ code to wrap Java objects, implement the data object class, and package the application. Therefore, you have to produce several classes with
different bindings that are specific to either the client or server side. Object Builder takes care of all the low-level details about the differences between client and server bindings and packages in deploying the application.

13.2.3 Create Client Application

To develop the client application, you have to generate the proxies and all of the helpers of the Component Broker objects in order to use the remote objects. For our EmployeeID sample, Object Builder generates the IDL files shown in Figure 46.

```
employeeFileKey.idl
employeeFileCopy.idl
employeeFile.idl
```

*Figure 46. Object Builder IDL Generated Files*

With the help of two internal utilities, Object Builder creates all the code for the client application.

All of the generated and compiled C++ code is used to build a client DLL. The generated Java code is packaged in two different jar files with the same name but in different directories. By default the jar file used by the client is in the JCB subdirectory of the working directory. The client DLL and the jar file with the server bindings are placed in the working directory. Object Builder builds the client DLL, Java client and server bindings, and Java jar files when you select **Build->all Targets** from the Build folder. (This option also generates the server side. See “Creating Server Files” on page 123.) That is all you have to do to create the client application support with Object Builder.

Object Builder uses the main class, `com.ibm.idl.toJava.Compile` (IBM IDL-to-Java compiler), contained in the somojij.zip file to produce the client proxies. The IBM IDL-to-Java compiler accepts an IDL file as a parameter and produces the related Java source files. In our sample, we have the two IDL files generated by Object Builder (Figure 46 on page 121). `EmployeeFileCopy` and `employeeFileKey` are files for the IDL description of Copy and Key helpers, and `employeeFile` is the description of our `EmployeeID` interface. Given these classes as parameters, the IBM IDL-to-Java compiler generates the related Java files that are used by the client application as shown in the Figure 47 on page 122.
Object Builder uses another utility called IDLC (Java Emitter) that creates the implementation bindings for the interfaces described in IDL files. The implementation of the Java Emitter is written in Java and the classes are included in the somshcl.zip file.

You can invoke the program directly from the command line or start a JVM with `com.ibm.som.util.Sc` as the class to execute. When Object Builder uses the Java emitter to produce client code, it specifies the -euj option to produce the cross-language Java usage bindings. For the complete syntax of the Java Emitter, refer to the Component Broker documentation.

Object Builder uses a makefile to produce all of the Java and C++ code. During this process, the Java Emitter also generates some C++ code needed for the C++ usage bindings, a skeleton for the Basic Object Adapter, local implementations needed by the C++ usage bindings, and Java usage bindings to access the object from the client and from the server.

You can verify that two stubs files are generated for EmployeeIDStub class: one file located in the `\working\nt\Jcb` directory for the Java client where EmployeeIDStub extends `com.ibm.CORBA.portable.ObjectImpl` class, and another file located in the `workin\nt` directory for the server where EmployeeIDStub extends `com.ibm.som.corba.rt.StubBase` class.
13.2.4 Creating Server Files

The development process on the server is more complex. You have to generate code to wrap your Java classes, implement the data object of your interfaces, and produce the Manageable Object code to adapt your object to the Component Broker environment. The C++ code generated in the client development process is linked to the code generated in the server development process. Once again, Object Builder simplifies this operation. All you have to do is select in the opened pop-up menu of the Build folder the **Build->all Targets** option. All the C++ code generated and compiled in this stage and the client DLL are used to build a server DLL. All Java classes are packaged in jar files. Both the server DLL and jar files are placed in the working directory. The Java classes are placed in the **working\javacls** directory in the related subdirectory. Component Broker uses all the jar files and DLLs built in the processes to run your objects.

At this stage Object Builder does not use the IDL-to-Java compiler because it has created the proxies and helpers during the client process. It submits the following files to the IDLC Java Emitter to generate the related C++ code:

- `employeeFileMO.idl`
- `employeeFileBO.idl`
- `employeeFileDO.idl`
- `employeeIDBOKeyAssistant.idl`
- `employeeFileDOImpl.idl`

By invoking the Java Emitter with the `-euj` and `-ebj` options on the file `employeeFileBO.idl` file and the `-euj` option on the `employeeFileDO.idl`, it generates all server-side Java files. Figure 48 lists all of the Java classes packaged into a server jar file.
Figure 48. Contents of Server jar File

You have the data object files that implement the persistence of your business object and the _EmployeeIDBOImpl and _EmployeeIDBOWrapper files which wrap your Java business object.

13.3 Object Builder and VisualAge for Java Interoperability

Object Builder is a very powerful tool for adapting your objects to the Component Broker environment. It generates some Component Broker-specific and implementation files. To create Component Broker objects, you must use Object Builder.

When it comes to implementing business logic, however, there is no comparison between Object Builder and VisualAge for Java. VisualAge for Java is a complete development environment, with a powerful debugger, a visual development environment, and an IDE. You can work in team, sharing the same repository with different versions of your classes. You can create several projects, check syntax errors, add and delete methods and classes easily, and use a powerful editor and a smart search engine and several facilities to develop enterprise Java applications. Unfortunately, the current version of Object Builder does not have any way of interfacing and cooperating with VisualAge for Java, making the use of a RAD tool like VisualAge for Java almost impossible.
Fortunately, the Component Broker’s Java ORB enables you to develop Java clients under the full power of VisualAge for Java.

13.3.1 **VisualAge for Java Client-Side Application Development**

To use VisualAge for Java to develop client applications, you need files generated by the IBM IDL-to-Java compiler and two of the files generated by Object Builder, such as `_employeeIDKeyImpl.java` and `_employeeIDCopyImpl.java`, which are the implementation of the Copy and Key helpers. Figure 49 shows all of the Java source files to import in VisualAge for Java to develop a client application. As a client application example, we developed the Runner class, which executes all the steps required to initialize the ORB; find the Naming Service, the Factory-Finder, and the Factory; and then instantiate EmployeeID.

![Image of Workbench](image)

*Figure 49. Client-Related Java Classes Generated by Object Builder*
You also need to import into VisualAge for Java the Component Broker Java ORB. The corresponding Java classes are included in a single file named somojor.zip in the CBroker\lib directory. After this file is imported, you can start developing, using all of the advanced features of VisualAge for Java.

### 13.3.2 VisualAge for Java Server-Side Application Development

Developing the server application is quite different from developing the client-side application.

As you have seen, the whole server development process is very tightly coupled with the Object Builder and Component Broker environment. Object Builder plays a fundamental role in generating and implementing all files required by Component Broker run time to run your objects. Your Java objects must be wrapped by C++ code to communicate with the server C++ ORB.

We made an attempt to use VisualAge for Java to implement the business logic of our interface by importing all of the Component Broker frameworks and server Java files into VisualAge for Java. We had to import in VisualAge for Java the files listed in the Figure 50.

```plaintext
ibmcbjs.zip
somshcl.zip
somshor.zip
```

*Figure 50. Jar Files Required for Server Development*

These files are located in the *CBroker\lib* directory. Once we imported those files, we could import the Java source code generated by Object Builder for the server in the *working\int* directory. Figure 51 shows all of the files needed for the implementation of our interface, employeeID.
Component Broker Development Process

Figure 51. Component Broker List of Java Files for EmployeeID Implementation

After the server files were imported, we had in the VisualAge for Java workspace the files shown in Figure 52.
Keep in mind that when you want to upgrade your implementation to Object Builder, you have to use the Object Builder import changes option.

VisualAge for Java’s IDE facilitates the implementation of the business logic of your interface. You can version implementations, work in teams, and share changes with other developers. Type declarations and syntax error checking are available.

**Restrictions and Limitations**

There are some restrictions and limitations in using VisualAge for Java.
After you have defined the IDL interface and added an implementation, Object Builder can generate the corresponding file (_EmployeeIDBOBase.java) for you to add your business logic. However, if you import this file in VisualAge for Java, you get errors. The reason for these errors is due to a reference to EmployeeIDDO type, which is not yet defined. Object Builder defines this type at the very end of the development process when building the DLLs. If you want to use VisualAge for Java, you must first complete the Object Builder tasks, even those that are not directly related to your business logic but instead deal with the persistence state.

Once Object Builder generates the necessary files and you develop your business logic in VisualAge for Java, you cannot test your application and use VisualAge for Java’s integrated debugger. The Component Broker server is an exe file that uses the Sun JVM to interpret the Java code. Because you cannot configure the Component Broker to use the VisualAge for Java’s JVM, you cannot run and debug your application inside VisualAge for Java.

13.3.3 Conclusion

Using VisualAge for Java to develop Component Broker Java business objects is not yet a solution.

Object Builder is written in Java, and IBM teams are already working on providing a nice integration with VisualAge for Java. This activity will provide server-side development with the same powerful IDE you have today for Component Broker Java client development.

However, if Enterprise JavaBeans really take off, VisualAge for Java 2.1 will become again the ultimate Java development environment, and with Component Broker already supporting Enterprise JavaBeans with appropriate deployment tools, you can just consider developing Component Broker Java business objects as an ephemeral solution.
Chapter 14. Component Broker Application Development

In this chapter we explain how to use Component Broker Version 1.3 to develop a server application. Our server application consists of one Component Broker object, EmployeeID, which is implemented in Java. We use an existing DB2 database, SAMPLE, to store and retrieve the essential state of the object. We cover all of the details needed of building the EmployeeID object and all steps to follow to install the application in Component Broker with the Component Broker System Management application.

To design a model we usually use Rational Rose, ObjectTeam, or other CASE tools. In this example, we do not use Rational Rose to design the EmployeeID object because we have only one object. Instead, we simply show how the new Component Broker object is integrated within the existing model.

In the TEA model, the Authenticator object uses a Component Broker object to get some information about an employee through the employee serial number. After we imported all files needed to use the Component Broker frameworks, our model looked like that in Figure 53.
Figure 53. EmployeeID Object Added to TEA Rational Rose Model

Note that in Figure 53 we only show the new Component Broker object inside the existing model. All objects managed by Component Broker must inherit from IManageable, which is a class from a Component Broker Managed Object framework.

14.1 Define the EmployeeID Object

You create the EmployeeID object with Object Builder:

1. Go to Object Builder, select the User-Defined Business Objects folder and open the pop-up menu. Select Add File and type EmployeeFile in the name field. You do not have to supply any other information. Click Finish.
2. To define a business interface, select **EmployeeIDFile**, open the pop-up menu, select **Add interface**, and type **EmployeeID** in the name field. Continue to click **Next** until you reach the Attributes page.

Now you have to define the attributes of the EmployeeID object. In this sample you use the EMPLOYEE table provided by the DB2 SAMPLE database. You define all attributes in your object that match the database columns of the EMPLOYEE table. You work with an already existing database, and you map your object's attributes with the types defined in the corresponding columns. You also need to take into account the database design, which specifies some columns as nonnullable. That is why we add extra attributes, which are not really necessary in your case. Figure 54 on page 134 shows the design of the SAMPLE database and details of the EMPLOYEE table. Two rectangles surround the columns we
use, and the selected row identifies the primary key for the object.

Figure 54. Design and Details of the EMPLOYEE Table

Thus we define the following attributes:

- `serialnumber` of type `String`
- `firstname` of type `String`
- `md` of type `Char`
- `lastname` of type `String`
- `edlevel` of type `short`

3. For each attribute, click **Add Another** on the Attributes page and fill in the corresponding information:
   - In the Attribute Name field, enter `serialnumber`.
   - In the Type field, select `String`. 
• In the Size field, enter 6.

4. Repeat the actions in step 3 for all attributes: firstname, md, lastname, and edlevel and click Finish.
You do not define methods. You only need the setters and getters that Object Builder generates automatically.

5. Once you have defined the EmployeeID interface, you have to add the Key and the Copy Helper objects. The Key object defines the attributes needed to find a particular instance of the component on the server. Add the Key by selecting EmployeeID interface and open the pop-up menu. Then select Add Key. Select the serialnumber attribute from the Business Object Attributes list and click Finish.

6. In this sample we add a Copy object, but this is an optional operation. Select the EmployeeID interface and Add Copy Helper from the pop-up menu. Click All to move all attributes from the Business Object Attributes list to the Copy Helper Attributes list and click Finish.

7. Now you have an EmployeeID interface and the Copy and Key helpers. You need to define the business object implementation. To add a business object implementation, select the EmployeeID interface and select Add Implementation from the pop-up menu. Select the Caching, Use lazy evaluation, and Create a new one now radio buttons in the Data Object Interface. Click Next and check that the IManagedClient::IManageable is selected as a parent. Click Next and select Java as the implementation language. Continue to click Next until you reach the Data Object Interface page. Click All to move all attributes from the Business Object Attributes list to the State Data list and click Finish.

You have added a data object interface to the business object implementation. The next step is to provide a data object implementation.

14.2 Data Object Implementation and Persistent Object Mapping

Because you reuse the DB2 sample database, you have to create an SQL file that you later import inside Object Builder. If you do not know how to create the file, use this command:
Drive:\yourProjectDirectory>d:\sqlib\misc\db2look -d sample -u piconr3 -t employee -e -o TEAEmployee

1. From the left-hand panel of the Object Builder main screen, select DBA-Defined Schemas and right click on it. From the pop-up menu select Import SQL.... In the Import SQL dialog, click the Find button and navigate to Drive:\yourProjectDirectory. Select TEAEmployee.sql and
click **Open**. Change the name of the database from **Database** to **Sample**. Click **Finish**.

2. Expand **DBA-Defined Schemas->TEA->sample."PICONR3"."EMPLOYEE**. Right click on **sample."PICONR3"."EMPLOYEE** and select **Add Persistent Object**.... Accept all the defaults and click **Finish**. At this stage, your Object Builder tree should have the objects shown in Figure 55.


4. On the Name and Behavior page:
   - In the Environment group, select **BOIM with any Key**.
   - In the Form of Persistent Behavior and Implementation group, select **Embedded SQL**.
   - In the Data Access Pattern, select **Local copy**.
• In the Handle for Storing Pointers group, select **Home name and key**. Click **Next**.

5. On the Implementation Inheritance page, select **IRDBIMExtLocalToServer::IDataObject** for the parent class. Click **Next**.

6. On the Attributes page, click **Next**.

7. On the Methods page, click **Next**.

8. On the Key and Copy Helper page, click **Next**.

9. On the Associate Persistent Objects page, click **Add Another** and then **Next**.

10. On the Attributes Mapping page, you will see a panel with a folder entitled **Attributes**. Select the first attribute, **serialnumber**, then right click and select **Primitive**. The display on the right changes and a combo box labeled **Persistent Object Attribute** is displayed. Select from the combo box the PO attribute that matches the data object you have selected (in this case, iPO.EMPNO). Then select the next attribute, **name**, from the panel on the left. Right click and select **Primitive** again. Select iPO.name from the combo box on the right. Repeat for each attribute in the list (that is, for firstname, md, lastname, and edlevel) and click **Next**.

11. On the Methods Mapping page, you will see a panel with a folder entitled **Special Framework Methods**. Select the first method, **insert**. Right click and select **Add Mapping**. Select the appropriate mapping from the Persistent Object Method combo box on the right (in this case, it is iPO.insert). Repeat for each of the methods (update, retrieve, del, setConnection). Click **Finish**. You should get the Object Builder tree shown in Figure 56.
12. Select the EmployeeIDBO object and in the pop-up menu select Add Managed Object. Click Next and Finish. Figure 57 shows the inheritance hierarchy of the EmployeeIDBO interface.
14.3 Create the EmployeeID Application

To create the EmployeeID application, you have to generate all of the C++, Java, and database-specific files:

1. Select EmployeeFile, open the pop-up menu, and select Generate -> All.

2. To define the client DLL, select the Build Configuration folder and select Add client DLL in the pop-up menu. Type employeeIDc in the name field. Click Next to reach the Client Source Files. Click All>> to move the client Employee object to the Items chosen list and click Finish.

3. To define the server DLL, select the Build Configuration folder and select Add server DLL. Type EmployeeIDs in the name field. Type IVB_TRACE_DEBUG=1 in the Make Options field to enable debugging and tracing. Click Next to reach the Server Source Files page and click All>> to move the EmployeeID server object to the Items chosen list. Click Next and on the Libraries to Link With page select EmployeeIDc. Click All>> to move the DLL file to the Items chosen list and click Finish.
4. To build the DLLs, select the **Build Configuration** folder, open the pop-up menu, and select **Generate -> All -> Java Default Targets**. (Remember that even if you generate files for the Java target, Object Builder creates few C++ files.) On the same menu select **Build -> All Targets**.

5. To package the application you first have to create an application family. Select the **Application Configuration** folder and select **Add Application Family** on the pop-up menu. Type `employeeApplicationFamily` in the Name field and click **Finish**. Add a server application by selecting `employeeApplicationFamily`, opening the pop-up menu, and selecting **Add Application**. Type `employeeAppServer` in the Name field. Click **Next** to reach the Additional Executables page and, by browsing the file system, pick up the `teamemployeeID.sql` and `EMPLOYEEPO.bnd` and click **OK** and **Finish**. Add a client application by selecting **Add Application** again and type `employeeApplClient`. Click **Finish** to end the operation.

6. To define the container, select the **Container Definition** folder and select **Add Container Instance** in the pop-up menu. Type `employeePersistentContainer` in the Name field and click **Next** twice. From the Services page, set the **Use RDB Transaction Services** check box and click **Next**. On the Services Details page, set the **Throw an exception and abandon the call** radio button, click **Next**, set the **Caching** and **Local copy** radio buttons on the Data Access Patterns page, and click **Finish**.

7. Configure the Managed Object by selecting `EmployeeAppServer` in the Application Configuration folder and **Add Managed Object** in the pop-up menu. Click **Next** and click **Add Another** button on the Data Object Implementations page and click **Finish**.

8. Expand the **Application Configuration** folder, select `employeeApplicationFamily` in the pop-up menu, and select **Generate**. This generates the `employeeApplicationFamily` DDL file.

9. To bind the application to the SAMPLE database, open the command shell in the working directory of the Employee application and use the following command:

   obdatapr EMPLOYEEPO.bnd sample bind

10. Change the directory to the CBroker\bin in the drive where you have installed the Component Broker and execute these commands:

    obdatapr db2cntcs.bnd sample bind
    obdatapr db2cntrr.bnd sample bind
14.4 Install the EmployeeID Application

Follow these steps to load the application into the Component Broker System Manager:

1. Start the Component Broker System Manager User Interface.
2. Set the User Level to Expert (from the menu select View->User Level->Expert).
3. Expand Host Images and select your host.
4. Right click on your host to open the pop up menu. Select Load Application.
5. Browse for and select EmployeeApplicationFamily.ddl, which is located in your projectDirectory\Working\NT\employeeApplicationfamily directory.
6. Click OK twice.

Follow these steps configure each application with a management zone:

1. If necessary, start the Component Broker System Manager User Interface.
2. Create a new management zone. Select Management Zone, right click on it, and select Insert.
3. In the Management Zone name field enter TEAzone and click OK.
4. Select Management Zone->TEAzone, right click on it, and in on the pop-up menu select New->configuration.
5. In Configuration name field enter TEAconfiguration and click OK.
6. Expand Available Applications, select employeeAppServer, right click on it to open the pop-up menu, and select Drag.
7. Expand the Management Zones->TEAzone->Configurations->TEAconfiguration menu item and select TEAconfiguration.
8. Open the pop-up menu for TEAconfiguration and select Add Application.
9. Repeat steps 6, 7, and 8 for iDB2IMServices.
10. Select TEAconfiguration, open the pop-up menu, and select New->Server(free standing).
11. In the Server name field, enter TEAserver.
13. Select **TEAconfiguration->Servers->TEAserver** and **Configure Application**.

14. Repeat steps 12 and 13 for iDB2IMServices.

15. Select **TEAserver**, open the pop-up menu, and select **Drag**.

16. Expand the **Hosts** folder, select your hostname, open the pop-up menu, and select **Configure Server (free standing)**.

17. Select **TEAzone->Configurations->TEAconfiguration**, open the pop-up menu, and select **Activate**.

18. Verify that your server is running by expanding **HostImages->yourHostname->Server Images->TEAserver**. On the status line you should read: **TEAserver:run on request running 77 excellent**. If this is not the case, verify that you did not skip any of the above steps. Compare what you have on your System Manager user interface with Figure 58 through Figure 61.
Figure 58. System Manager User Interface: Available Applications
Figure 59. System Manager User Interface: Management Zones
Figure 60. System Manager User Interface: Hosts
We have completed the server-side Component Broker development.

The client of a Component Broker Business Object can be another Business Object connected to a server ORB or an application using a Java client ORB.

The TEA application gets connected to a client Java ORB. In Chapter 16, “WebSphere and Component Broker Interoperability” on page 173, we describe the steps required for a client application to connect to the ORB.
access the Naming Service, find a factory Finder, and obtain a Factory that
implements the IDL interface we want to use.

All these steps have been described by the detail in different documents. You
will find a step by step description of what a client application must perform
as well as how to integrate it with a GUI user interface taking advantage of
swing classes. Rather than repeating this same information, we rather advice
you to refer to the following books:

• *IBM Component Broker Programming Guide*, G04L-2376
• *IBM CBConnector Cookbook Collection: First Steps*, SG24-2033
• *IBM CBConnector Cookbook Collection: CBConnector Bank
  Implementation*, SG24-2033
• *Application Development with VisualAge for Java Enterprise*, SG24-5081
Chapter 15. Remote Debugging

Debugging is one of the most important aspects of the development process. Memory leaks, null pointers, wrong variable values, loop cycles, if conditions, out of boundary arrays, and other bugs within your code can be very hard to find. Printing out to the console the value of your variables and the sequence of method invocations is not enough to find out where the bugs are in your code. The more complex your program is, the more time you need to debug it. Just think about how many classes and methods an enterprise application uses. A wrong value passed to a method as a parameter could change the entire flow of your program. A null pointer or an out of boundary array could cause an uncaught exception to be thrown and block the program execution. The application could run correctly in some conditions and could fail in others.

15.1 Development Environments

Once you have understood the importance of the debugging, you have to consider the type of application you would like to debug. Stand-alone applications are simple to debug because all runtime and code resides on the same machine, so you can develop and debug your application on the same machine.

When you develop distributed applications, your code runs on different machines, and several objects are distributed across the network. On the client, the presentation logic uses the interfaces of the objects but the real implementation of the objects runs on remote servers. You can debug your client easily because you have all the runtime environment you need to run the client. Remote objects run in different servers, on different operating systems located in different places. One of the goals of CORBA application development is to allow a client to use an interface of an object, no matter in which language the object is implemented or which operating system is used. When you have a small distributed application using only one server, you can reproduce the server environment on your development machine. However, when you have hundreds of servers using different ORBs on different operating systems with several databases and several technologies such as TP monitors (CICS) and other legacy systems, or old technologies, it is not possible to simulate all the server runtimes in a single machine.

Java and Visual Age for Java can help you do remote debugging in distributed environments. Because Java claims to be "write once, run
everywhere," theoretically, once you have a compiled class in bytecode, you
can run the class on any platform to which the JVM has been ported.

You might think that if you can execute the code everywhere, you do not have
to debug your remote objects. You can simply run them on your machine and
debug them. For very small applications, this approach could be viable. But
enterprise applications include and use different technologies, and the
subsystems that participate in the runtime environment are several and
heterogeneous. Remote objects can use databases, transaction services,
native code, or shared libraries and wrap old applications in heterogeneous
runtimes.

Using a standard JVM, you can debug a running bytecode remotely.
According to the debugging specification defined by Sun Microsystems, a
client debugger can be connected to a remote JVM through TCP/IP, get
debugging information in a well-known format, and interpret the information
as well. With the Java Development Kit, you can use a jdb program to
remotely debug Java bytecode. But jdb is a very primitive debugger with no
user interface or visual environment. Visual Age for Java brings to us a
complete set of tools for debugging from inside the IDE.

Unfortunately Visual Age for Java does not use a standard JVM to execute
the bytecode. Thus you cannot use the VisualAge for Java internal debugger
to debug code running outside the VisualAge for Java environment. A kind of
plug-in to allow a VisualAge for Java JVM to be invoked externally, outside
the IDE, would be a great feature enabling you to use the powerful VisualAge
for Java integrated debugger. To debug Java code remotely you can use
external tools provided by VisualAge for Java. You can debug a JVM remotely
in a different manner. Figure 62 shows how you can debug a remote Java
object in a distributed application scenario.
Remote Debugging

Figure 62. Debugging Java Code Remotely

A debug daemon written in Java attaches one JVM and passes the debugging information to a client debugger. A daemon runs in the same machine or in a different machine. You can now perform all operations typically available with a GUI debugger, such as analyze all the objects, set breakpoints, and execute instructions step by step.

15.2 Debugging Java Virtual Machine Objects

To debug your business objects remotely, you must follow certain steps. Some are standard steps and apply no matter what distributed environment you use; others are specific to the ORB and the runtime environment you are using. The following steps are required to debug a business object written in Java:

1. Include debugging information in the compiled code.
2. Set up a remote ORB to allow debugging.
3. Run the JVM in debugging mode.
4. Start the VisualAge for Java debugging daemon on the remote machine.
5. Start the user interface debugger on the client machine.

15.2.1 Include Debugging Information in the Compiled Code

When your source code is ready to be compiled, you have to tell the compiler that you want to include debugging information in the code.

15.2.1.1 Using the JDK
Using a JDK javac compiler, specify the -g flag, and the compiler provides all of the debugging information.

15.2.1.2 Using VisualAge for Java
With Visual Age for Java, you specify that you want the debugging information by setting a flag in the SmartGuide Export to a jar file window (Figure 63).
15.2.1.3 Using Component Broker

You instruct Object Builder to include debugging information in the compiled code by setting the IVB_TRACE_DEBUG flag equal to 1 in the Make Options field of the server DLL Build properties.

15.2.2 Set Up a Remote ORB to Allow Debugging

Once you have compiled your code with debugging information, you have to set up the runtime remote ORB’s environment to allow debugging.

In addition to debugging Java Business Objects, Component Broker provides different ways of tracing. It can trace all services and store the resulting trace log files in the directory where you installed your application files. There is
also a general logging file, activity.log, located in the cbroker\services
directory. Use showlog.exe (showlog activity.log) to convert the information to
a readable format.

If you need to understand the information exchanged among the different
Component Broker processes and a client, you can trace these messages. A
lower level debugging is available with Component Broker that supports
GIOP-IIOP tracing.

With Component Broker, you can have many servers on a specific host. First
you have to decide which server you want to debug. Your decision depends
on the objects you use. Each server supports a given set of interfaces that a
client can invoke. Let us suppose you want to debug the EmployeeID object.
First you must stop the server you are going to debug. Using the System
Manager User Interface, stop the TEAserver, edit it, and in the Main page set
debug enabled to yes and run control to run on request. Now you can open
a shell window and start the server manually. Figure 64 shows how to enable
debugging for the TEAserver.
You also have to set these environment variables within the shell session window:

- Set SOMDGETENV=1
- Set SOMDCOMMDEBUGFLAG=257
- Set hostname=<fully qualified server hostname>
- Set SOMCBENV=S:<Component Broker server name>

Let us assume you would like to debug on a ganges.almaden.ibm.com machine, the TEAserver implementing the EmployeeID interface. The environment variables settings should be:

- Set SOMDGETENV=1
• Set SOMDCOMDEBUGFLAG=257
• Set hostname=ganges.almaden.ibm.com
• Set SOMCBENV=S:TEAserver

Now you can start the server by running the somsrsms.exe daemon in the command shell. If you have set everything correctly, you see all GIOP-IIOP messages generated by Component Broker during the startup procedure of this server. Figure 65 shows the end of the Component Broker start up procedure indicating that you can now debug it.

Now that you have activated different traces, you can debug a Java Business Object.

15.2.3 Run the Java Virtual Machine in Debugging Mode

You must start the JVM in debug mode. When you invoke the JVM, specify the -debug flag. For security reasons, a client cannot connect to the remote JVM unless it gets an agent password. The JVM creates the agent password when the -debug flag has been specified. Once you have the agent password, the authorized client can attach to the JVM, and debugging can start. With Component Broker you cannot control the JVM with the -debug flag. However, you obtain the same result by setting IVB_TRACE_DEBUG=1 in Object Builder. You have to remember that the server process is executing C++ code and uses IOM to run Java in that same process. You can see the agent password from the shell where you started the server daemon (see Figure 66).
Remote Debugging

15.2.4 Debugging Daemon on the Remote Machine

The VisualAge for Java remote debugging daemon is written in Java, and all code is packaged in a file called jdebug.jar. You can find this file in the VisualAge for Java directory (drive:\IBM\Java\eab\runtime20). To invoke the daemon, issue this command:

```
java com.ibm.debug.engine.Jde -host=<JVM_host> -password=<agent password> -qport=<listening port>
```

where:
host is a hostname on which the JVM is running.

password is the agent password printed out by the remote JVM.

port is the TCP/IP port where the daemon listens for incoming requests from clients. If you do not specify the TCP port, by default the daemon listens on port 8000. In our example, you start the daemon on port 10000 as shown in Figure 67. If you have supplied all the required parameters correctly, you should see the daemon message “waiting for connection on port” followed by the TCP port you specified.

```
C:\Java\com\IBM\debug\engine\Jde -qhost=ganjies -qpassword=3k5r8 -qport=10000
IBM Remote Debugger
Version 1.1.0 - Licensed Material - Property of IBM
(c) Copyright IBM Corp 1997, 1998 All Rights Reserved
US Government Users Restricted Rights - Use, duplication or disclosure restricted by GSR 46.5 Schedule Contract
with IBM Corp.
Waiting for connection on port: 10000
```

Figure 67. VisualAge for Java Daemon Waiting for Connection on Port 10000

Be careful to select an unused TCP port. If you selected an already bound port, the daemon will not be able to create the socket and will display the “unable to create server socket. The port may be in use” message. We suggest taking a look at the services file to see which ports your system uses.

For all of options of the remote debugging daemon, refer to the documentation provided by VisualAge for Java.

15.2.5 Start the User Interface Debugger on the Client Machine

To connect the client to the remote daemon, issue this command:

```
jdebug -qhost=<remote Host> -qport=<daemon port>
```

where:

qhost is the hostname where your daemon runs.

qport is the TCP/IP port where you can connect to the daemon.

We executed this command to connect to the daemon shown in Figure 67 and started the client user interface:
jdebug -qhost=ganges -qport=10000

By following all the steps you have described so far, you should see your debugger up and running. You also should see the JVM that is running on the remote server (Figure 68).

![Process List](image)

Click on Remote JVM in the Process List window and click on Attach to start debugging your application. Now you can see all of the objects that your business object uses, the value of the variables, the source code, and the flow of your code. Figure 69 shows the Java debugger jdb shell, and Figure 70 shows the VisualAge for Java remote debugger. Note what you have to do in jdb to see the methods of a class. First you have to get a class ID and then, using a methods command, you can list the class methods.
Figure 69. Jdb Shell: List of Methods for Employee Class
Remote Debugging

15.3 Debugging the Client

Before you start debugging the client, you have to set up VisualAge for Java.
15.3.1 Setting up VisualAge for Java

To develop and debug the client part, you have to import the somojor.zip file located in the cbroker\lib directory. This file contains all of the classes of the Java ORB.

15.3.2 Debugging a CORBA Client inside VisualAge for Java

Debugging the client is much more simple than debugging server objects. You can run your code, using the VisualAge for Java JVM, and thus you can use the integrated VisualAge for Java debugger. In VisualAge for Java you can set breakpoints in your source code, suspend and resume a thread, see the value of a variable, and modify that value during the debugging. As indicated in Figure 71, the VisualAge for Java integrated debugger is a complete and powerful tool for debugging your application.

![Figure 71. The Visual Age Debugger](image)

Once you have stopped code execution by setting a breakpoint, you can inspect any variable, regardless of whether it is an integer or an instance of
Remote Debugging

The user interface of the debugger is very intuitive and friendly. You can set a breakpoint by just clicking on a line. Controlling the flow of the program with the tool bar is simple as well. Figure 72 shows how to inspect a variable during debugging.

![Image of debugger interface](image)

Figure 72. Inspecting a Variable with the VisualAge for Java Integrated Debugger

VisualAge for Java Version 2.0 adds some new interesting features to the debugger (see Figure 72). You can set a breakpoint outside the workspace in a class or you can trigger the VisualAge for Java debugger when a specific exception is thrown.
Sometimes you do not know exactly where your code has failed. In a complex application, debugging the program from the beginning takes too much time. One of the most common errors is a thrown exception. A thrown exception indicates that something has gone wrong somewhere in your program. You know what kind of exception has been thrown but you do not know where in your code. A program can throw an exception whenever and wherever. With the VisualAge for Java integrated debugger you can start debugging an application just after the program has thrown a specific exception. For example, say you want to control where the org.omg.CORBA::BAD_PARAM is thrown. When you specify the exception you want to monitor, as soon as that exception is thrown, the debugger is triggered and automatically invoked. Figure 74 shows the window where you can select the exceptions you would like the VisualAge for Java debugger to catch.
Sometimes you run an application in the VisualAge for Java workspace that uses external Java classes or archive files as jar or zip files. You can set a breakpoint to a method of a class outside the workspace without importing it inside Visual Age. Once you have specified the jar or zip file, you can select the classes and the methods where you want to set a breakpoint. Figure 75 shows how to set a breakpoint within an external class. During the execution of the code, when the application invokes a method of an external class, the VisualAge for Java debugger starts the debugging.
Figure 75. Setting a Breakpoint in an External Class outside the Workspace

When an application is started it is useful to know the sequence of the classes that your code loads and initializes. With VisualAge for Java you can trace the initialization of all classes. You have to set a "Trace class initialization for running programs" flag in the Coding->Debugging section of the Options window (see Figure 76). You can view the entire class initialization history by going to the Debugger window and selecting the program.
To view the entire class initialization history, open the Debugger window, and click on the running thread (see Figure 77).
Figure 77. Class Initialization History View
As you can see, there is no comparison between the debugging facilities available inside the VisualAge for Java IDE and the elementary debugging features provided by the JDK.
Chapter 16. WebSphere and Component Broker Interoperability

Heterogeneous computing environments are becoming more and more common as the level of technology rapidly increases. Businesses are often forced into maintaining existing, legacy applications while deploying new solutions on different platforms using different technology. Although CORBA is not a panacea for all problems, it provides the building blocks for distributing individual, or multiple, applications over a diverse range of platforms and development environments.

Even with the features that CORBA has that facilitate this distribution, there are still issues with regard to communicating among the various ORB vendors. Some of these issues are:

- Proprietary extensions to CORBA functionality. Many vendors add their own features to enhance and complement the base CORBA function.
- Not all ORBs are equal; that is, some provide fewer CORBA services than others.
- Conformance to CORBA specifications. Although many ORBs claim CORBA compliance, the reality is that this is not always true.

In this chapter we briefly touch on how we combined Component Broker function with the existing WebSphere application.

16.1 Overview of New Function

The new function provided by Component Broker is quite trivial. Using the EMPLOYEE table provided with DB2 Universal Database (UDB) 5.0, we now move the authentication function from a WebSphere object to a Component Broker object.

On receiving a request for an Employee object, Component Broker instantiates an EmployeeID object using the key that is passed (see Chapter 14, “Component Broker Application Development” on page 131). If the employee does not exist in the database, the client is notified accordingly. If the employee does exist in the database, the authenticator asks the EmployeeID object to authenticate itself using the password that the user entered.
16.2 Changing the Existing Application

The class that was originally used for authentication for the TEA was tea.business.impl.AuthenticatorImpl. To use the new authentication model, we created the tea.business.cb.AuthenticatorImpl class. Both of these classes implement the tea.business.Authenticator interface:

```java
public interface Authenticator extends BusinessObject
{
    Employee logon(String serialNumber, String password);
}
```

The difference between these two classes is quite marked. The original authenticator merely checks that the serial number is equal to the password and returns the employee if true, null otherwise.

The new authenticator instantiates an ORB that is connected to Component Broker and looks for an EmployeeID that contains a key equal to the serial number. If it finds one, it requests that the password be authenticated. If everything is successful, the employee is returned, otherwise null.

Here is an explanation of the steps you have to execute before you can look for the EmployeeID object (refer to step numbers indicated as comment lines in the source code):

1. As the CORBA specification does not specify a bootstrapping mechanism to find an entry point into a CORBA network, each vendor has implemented the mechanism in its own way. With Component Broker, during the ORB.init, you have to provide the host machine that establishes the link with the CORBA servers. It is called the BootstrapHost; in our example, the host name is atlantic.almaden.ibm.com. Every Component Broker host has a daemon listening on a well-known port. In addition to the BootstrapHost, you have to provide BootstrapPort 900.

2. This ORB.init implementation always expects an applet parameter. Because the server code does not run as an applet but as a server-side object without any user interface, we had to create a dummy applet, KludgeApplet, to connect to the ORB.

3. The Component Broker EmployeeID object has been implemented as an object to be executed inside the scope of a transaction. The transaction can be implicitly started by the server when the object is invoked or started by the client. In the latter case, a client trying to use the object without first starting a transaction will be thrown an exception. A transaction can be started from a client in different ways. A simple way is to request from the ORB the Current object and use it to begin the
scope of the transaction. (This action could have been moved closer to the actions related to this object.)

4. After we get an ORB object reference, we are almost in the situation of someone having a telephone handset in the hand. He is connected to the public phone network, but he still needs to know which phone number to dial. For that, he uses the phone book directory; in the CORBA world, the Naming Service plays the same role. By asking the ORB object to resolve_initial_references("Name Service"), we get back a naming context.

5. The CORBA specifications define structures for names and compound names. Given the complexity of creating such compound names, IBM has developed an extension to the Naming Service that allows you to simply use a string instead of creating a compound name. In that way, you can see the Component Broker naming tree as a file system and access any file by prefixing it with the corresponding path.

In addition, IBM and other ORB vendors have agreed on the definition of a structure for the naming tree. This has resulted in a proposal to the OMG. Therefore, in this step, when we call resolve_with_string method on the naming context with the "host/resources/factory-finders/host-scope" string, this corresponds to a well-defined location that could be found in any naming tree provided by these ORB vendors. At that place in the naming tree, we expect to find a FactoryFinder. This FactoryFinder knows about existing factories and can locate any factory implementing a given interface.

6. Finding a factory is accomplished by asking the FactoryFinder to find a factory that implements an "EmployeeID" IDL interface.

7. Before we can ask the Factory to retrieve a given EmployeeID instance, we have to create a PrimaryKey. In Component Broker a factory or home creates, deletes, and manages instances. Each instance is identified by its PrimaryKey. We create an instance of a PrimaryKey by using a KeyHelper generated by Component Broker’s Object Builder tool.

8. We use the serialNumber entered by the user of the TEA application to set the PrimaryKey.

9. Now, we can ask the home to find an EmployeeID object instance identified by the PrimaryKey.

10. Once the EmployeeID instance is found, we can close the scope of the transaction and return the result. Otherwise, if an instance does not exist, we get a "NoObjectWithKey" exception and roll back the transaction.
public Employee logon(String serialNumber, String password) {
    com.ibm.CORBA.iiop.ORB myOrb = null;
    org.omg.CORBA.Object obj = null;
    com.ibm.IExtendedNaming.NamingContext nc = null;
    com.ibm.IExtendedLifeCycle.FactoryFinder finder = null;
    com.ibm.IManagedClient.IHome home = null;
    org.omg.CORBA.Current cur = null;

    // step 01
    Properties props = new java.util.Properties();
    props.put("org.omg.CORBA.ORBClass", "com.ibm.CORBA.iiop.ORB");
    props.put("com.ibm.CORBA.BootstrapHost", "atlantic.almaden.ibm.com");
    props.put("com.ibm.CORBA.BootstrapPort", "900");
    props.put("com.ibm.CORBA.EnableDCESecurity", "no");
    try {
        // step 02
        myOrb = (com.ibm.CORBA.iiop.ORB) com.ibm.CORBA.iiop.ORB.init(new tea.util.KludgeApplet(), props);
        // step 03
        obj = myOrb.get_current("CosTransactions::Current");
        current = org.omg.CosTransactions.CurrentHelper.narrow(obj);
        current.begin();
        // step 04
        obj = myOrb.resolve_initial_references("NameService");
        nc = com.ibm.IExtendedNaming.NamingContextHelper.narrow(obj);
        // step 05
        obj = nc.resolve_with_string("host/resources/factory-finders/host-scope");
        finder = com.ibm.IExtendedLifeCycle.FactoryFinderHelper.narrow(obj);
        // step 06
        obj = finder.find_factory_from_string("EmployeeID.object interface");
        home = com.ibm.IManagedClient.IHomeHelper.narrow(obj);
        // step 07
        EmployeeIDKey empKey = EmployeeIDKeyHelper._create();
        // step 08
        empKey.serialnumber(serialNumber);
// step 09
obj = home.findByPrimaryKeyString(empKey._toString());
EmployeeID employeeID = EmployeeIDHelper.narrow(obj);

// step 10
current.commit(true);

// if we get this far, then the employee must exist on the
// component broker system. Job done!
return
(Employee)EmployeeFactory.instance().getAllInstances().get(""+Integer.parseInt(serialNumber));
}
catch (Exception e)
{
// if we get to here, then we probably haven’t been able
// to find the employee on the Component Broker system.
// Error checking should be much more robust though
current.rollback
return null;
}

As yet, we have only added code; we have not changed any existing code. In
order for our application to instantiate the new authenticator, we change the
getImpls() method of the tea.servlet.TEAServlet class:

Properties props = new Properties();

// hard coded for now, but could be read from a config file
String auth = "CB";

if (auth.equals("CB"))
    props.put("tea.business.Authenticator",
    "tea.business.cb.AuthenticatorImpl");
else
    props.put("tea.business.Authenticator",
    "tea.business.impl.AuthenticatorImpl");

props.put("tea.business.QueryService",
    "tea.business.impl.QueryServiceImpl");
return props;
So, now when the client asks the IHome to create the authenticator, the new implementation of the authenticator will be instantiated. In terms of code changes, that is all that needs to be done.

16.3 Migrating to the New Application

Three projects are affected by the migration to the new application:

- **Travel Expense Application**
  
  Import Version 2.6 from the tea.dat file. This version contains the new authenticator code as well as the Component Broker EmployeeID stubs.

- **WebSphere Runtime Libraries**
  
  This project is no longer required as it is superseded by the class in the Component Broker project. Delete it from your workspace.

- **Component Broker Runtime Libraries**
  
  This project contains the Component Broker ORB classes. Fortunately for us, Component Broker and WebSphere have a common code base for their ORB implementation, so many of the classes can be reused from one ORB to the other. The Component Broker libraries are a superset of the WebSphere libraries with a few small exceptions. We copied the following packages and classes from the WebSphere implementation into this project:

  - `com.ibm.jbroker` package
  - `com.ibm.jbroker.ejb` package
  - `com.ibm.CosNaming._NamingContextImplBase` class
  - `com.ibm.CosNaming.NamingContextDataStore` class
  - `com.ibm.CosNaming.NamingContextImpl` class
  - `com.ibm.CosNaming.TransientNameService` class
  - `com.ibm.CosNaming.TransientNamingContext` class

  We also replaced the `com.ibm.CORBA.iiop.ORB` class with a patched version that fixes a bug in the initialization of the WebSphere Naming Service (see Appendix D.1, “CosNamingServlet Not Starting” on page 225).

Start the Component Broker server, the Web server, and run your application.
Chapter 17. Moving to Enterprise JavaBeans

VisualAge for Java Enterprise V2.1 introduces a new installable feature called IBM Enterprise Java Server (EJS) reference release. This reference release contains:

- An implementation of the Java Web Server and Java Server Toolkit that provides JSP and Java servlet support.
- An implementation of an Enterprise Java Server (EJS) and a set of Enterprise JavaBeans (EJB) containers that allow EJBs to access relational databases.
- Support services such as the JNDI, CORBA CosNaming services, and administration and deployment tools to operate the server and manage EJBs, servlets, and JSPs.
- A conforming implementation of the EJS Reference Release architecture.

In this chapter we discuss how you can use VisualAge for Java with Enterprise Java Server to develop and deploy EJBs. In particular, we focus on providing the same functionality that we provide in the Component Broker (see Chapter 16, “WebSphere and Component Broker Interoperability” on page 173) by adding a new implementation of the EmployeeID object as a bean-managed persistence EJB (see Figure 78).
17.1 Installing the Enterprise Java Server

To use the EJB development environment, you must install IBM WebSphere Runtime Support VisualAge for Java 2.1beta, which contains the WebSphere integration and EJB runtime support components.

To load the required feature:

1. Start VisualAge for Java.
2. On the Workbench Projects page, select File - Quick Start. The Quick start dialog appears.
3. In the left pane, select Features. In the right pane, select Add Feature and click OK. The Selection Required dialog appears.
4. In the Selection Required dialog, select IBM WebSphere Runtime Support VisualAge for Java 2.1beta and click OK. The dialog closes and the IBM WebSphere Runtime Support project is retrieved from the repository and displayed in the Projects page.
To run and test entity beans in VisualAge for Java, you must add a JDBC driver to the classpath. Follow these steps to add the DB2 JDBC driver:

1. On the Workbench Projects page, select **Windows - Options**. The Options dialog appears.
2. In the tree view, select **Resources**. The Resources page appears in the dialog.
3. Beside the Workspace classpath field, click the **Edit** button to open a Classpath dialog.
4. Click the **Add Jar/Zip** button and navigate to the drive:`\SQLLIB\java\` directory.
5. Select the JDBC driver file `db2java.zip` and click **Open**.
6. In the Classpath dialog, click **OK**.
7. In the Options dialog, click **OK**.

After installation of EJB support, your workspace will contain only one visible change. There will be an extra tab on the browser window labeled *EJBs*. You use the pane associated with this tab when developing EJBs. The EJB generation process will work correctly at this stage; however, not all necessary classes will have been imported into the workspace. This will cause compilation errors for the classes that have been generated. VisualAge for Java provides a feature that you can import into your workspace so that all the necessary classes are loaded.

EJS does, however, contain some classes that are already provided by either Component Broker, WebSphere, or Sun’s Java Servlet Development Kit (JSDK). If you import the EJS feature into your workspace while any of these projects are loaded, you will get an error. By removing the aforementioned projects from your workspace before importing the feature, however, you can successfully import. The projects that have been created during your progress during this book that need to be temporarily removed are:

- Component Broker Runtime Libraries
- WebSphere Runtime Libraries
- Sun JSDK Class Libraries

Once these projects have been removed, you can import the EJS feature. The EJS feature is called IBM WebSphere Runtime Support VAJava 2.1beta (see Figure 79).
Figure 79. Importing the EJS Feature

After the import has completed, you will have three new projects added to your workspace:

- IBM EJB Runtime Support 1.0
- IBM WebSphere Runtime Support 1.1
- JSP Page Compile Generated Code 1.0

Of these, the first contains the EJS support code. You can safely delete the other two projects (remember that you can reload them from the repository at any time if you so choose).

You are now ready to reimport the requisite projects for the TEA. As a reminder, you should have the following projects in your workspace:

- Travel Expense Application
- IBM EJB Runtime Support
Notice at this point that you have some errors in the tea.business.cb package because you do not have any of the Component Broker libraries loaded. We have been unable to find any successful method of having the EJB and Component Broker libraries coexist in VisualAge for Java. This is sure to change as they are derived from the same code base. However, in the versions that we used, they were incompatible.

There are also errors in several other TEA packages because of some missing WebSphere function. To get around that, we will just import the missing packages and classes from WebSphere into the IBM EJB Runtime Support project. Copy the following packages and classes from the WebSphere implementation into the IBM EJB Runtime Support project:

- com.ibm.jbroker package
- com.ibm.jbroker.ejb package
- com.ibm.CosNaming.TransientNameService class

If you are unsure of how to copy the packages and classes, consult the VisualAge for Java help, or import Version 1.3 of the IBM EJB Runtime Support from the repository supplied with the sg245276.zip file.

17.2 Changing the Existing Application

This time, to introduce the use of EJBs, we are going to change the application by replacing the EmployeeID Component Broker object developed in Chapter 16, “WebSphere and Component Broker Interoperability” on page 173 with an equivalent EJB. This EJB provides the exact same functionality. It just covers our purpose of showing how to integrate EJBs in our application. Among the different possible EJBs, we choose to implement a bean-managed persistence bean.

17.3 Creating an EJB

Follow these steps to create an EJB:

1. Write a remote interface.
2. Write a home interface.
3. Write a bean.
4. Write a bean key.
5. Write a deployment descriptor.
6. Write a manifest file.
7. Package all these files in a jar file.

Fortunately, we can use VisualAge for Java and reduce our effort to develop the bean class only. All other tasks are handled by the tool, which generates all additional classes and even provides the final jar file ready for deployment. And that is not all. Because WebSphere is packaged with VisualAge for Java, we also have the runtime environment to deploy and test our EJB.

So, let us start from the VisualAge for Java Workbench and select the EJB tab.

VisualAge for Java gathers EJBs into groups. To create a group, select the **EJBs->Add->EJB Group**... menu item and fill in the fields as shown in Figure 80.
An EJB consists of a number of Java classes and interfaces, such as the bean class, remote interface, home interface, and a set of deployed classes. You must manage all these classes as a single entity, so that they can exist in synchronization. For example, if you need to delete a method from the bean class, you want to ensure that the corresponding method in the remote interface is deleted automatically. To manage all Java classes and interfaces that belong to a bean as a single object, VisualAge for Java allows you to create EJBs as first-class objects. In the Create Enterprise JavaBean SmartGuide, you can define a name for the EJB and then define the individual classes and interfaces that belong to that EJB. You can then

---

**Figure 80. Creating a New EJB Group**

Select **TEAGroup** before adding an EJB.

<table>
<thead>
<tr>
<th>Available group names</th>
<th>Available editions</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

---

*Moving to Enterprise JavaBeans* 185
version the EJB as a single entity. For example, when you version an EJB, it will version all of the corresponding Java classes and interfaces and save their signatures into the EJB object. When a specific version of the EJB is loaded, the correct version of all the EJB bean classes and remote and home interfaces are loaded.

Follow these steps to add EJBs:

1. In the EJBs pane, select an EJB group to contain your EJB and click mouse button 2.

2. From the pop-up menu, select Add - Enterprise JavaBean. The Create Enterprise JavaBean SmartGuide appears (see Figure 81).

3. Click Finish, and VisualAge for Java will create four classes for you:
   - EmployeeID interface
   - EmployeeIDBean class
   - EmployeeIDHome class
   - EmployeeIDKey class

Figure 81. Creating an Enterprise Java Bean
17.3.1 Adding Behavior to Your EJB

When you create an EJB in the EJB development environment, a set of required methods for the bean is automatically created. You can modify these methods and add new remote (business) and home methods of your own.

Adding methods in the EJB development environment is similar to adding methods elsewhere in VisualAge for Java. Once you have created remote (business) methods or home methods for your classes and interfaces, you can promote the methods to the remote and home interfaces from the Bean class without editing the remote and home interfaces directly.

Follow these steps to add a new method:

1. In the Types pane of the EJBs page, select the class or interface to which you want to add a method.
2. Click mouse button 2, then select Add Method from the pop-up menu. The Add Method SmartGuide appears.
3. Follow the SmartGuide instructions on creating a new method.
4. Click Finish to create and compile the new method.

If you want to remove a remote method from the remote interface, select the method and then select Methods-> Remove From ->EJB Remote Interface.

If you want to remove a home method from the home interface, select the method and then select Methods-> Remove From ->EJB Home Interface.

Before we add methods to our EJB, we are going to define some properties that are used later on. We add the following properties to the EmployeeIDBean class:

```java
private Connection _connection;
private static String _jdbcURL = "jdbc:db2:sample";
private static String _jdbcDriverName = "COM.ibm.db2.jdbc.app.DB2Driver";
public String userid;
private Statement _statement;
```

and an import statement:

```java
import java.sql.*;
```

In our EmployeeID bean, we are going to add two methods:

- getConnection()

This method loads the JDBC driver and returns a connection
private Connection getConnection()
{
    System.out.println("getConnection ");
    if (_connection == null)
    {
        try
        {
            Class.forName(_jdbcDriverName).newInstance();
            _connection = DriverManager.getConnection(_jdbcURL);
        }
        catch(Exception e)
        {
            System.out.println("getConnection error ");
            e.printStackTrace();
        }
    }
    return _connection;
}

• authenticate(String)

This method compares the user entered password against the first name found in the database table corresponding to the serial number. If they match, the user has been authenticated.

public boolean authenticate(String password) throws java.rmi.RemoteException
{
    return password.equals(userFirstname);
}

If you want to promote a remote method to the remote interface, select the method and then select Methods->Add to->EJB Remote Interface.

If you want to promote a home method to the home interface, select the method and then select Methods->Add to->EJB Home Interface.

Typically, to add a method in a development environment that uses interfaces and implementations, you would add the method first to the interface, and then to the implementation class. However, the EJB development model is slightly different. If you browse the EmployeeID bean, you will notice that it does not implement the EmployeeID interface. The EmployeeID interface is a generated class, and if you add method signatures to it, they will be overwritten.
So, add the method to the interface, using the **Add To->EJB Remote Interface** menu items (see Figure 82).

The EmployeeID interface now contains an authenticate() method.

Now we need to modify some of the generated methods to add additional behavior:

1. ejbActivate()

   We just request the connection by calling the added method getConnection.

   ```java
   public void ejbActivate() throws java.rmi.RemoteException {
       getConnection();
   }
   ```

2. ejbCreate(EmployeeIDKey)

   ```java
   public /**@EJBHome*/ EmployeeIDKey ejbCreate(EmployeeIDKey key) throws javax.ejb.CreateException {
   ```
3. ejbFindByPrimaryKey(EmployeeIDKey)

```java
public /**@EJBHome*/ EmployeeIDKey ejbFindByPrimaryKey (EmployeeIDKey key) throws javax.ejb.FinderException, RemoteException {
    Statement statement = null;
    ResultSet rs = null;
    System.out.println("ejbFindByPrimaryKey ");
    try {
        ejbActivate();
        String sql = "SELECT FIRSTNME FROM EMPLOYEE WHERE EMPNO = "+key.primaryKey+" ";
        statement = getConnection().createStatement();
        rs = statement.executeQuery(sql);
        if (rs.next()){
            //System.out.println("firstname is: " + rs.getString("FIRSTNME").trim());
            userFirstname = rs.getString("FIRSTNME").trim();
            return new EmployeeIDKey(key.primaryKey);
        }
        else
            throw new FinderException("Employee ID not found: "+key.primaryKey);
    }
    catch(SQLException e)
    {
        throw new RemoteException(e.getMessage(), e);
    }
    finally
    {
        try {
            statement.close();
        } catch(Exception e){
        }
    }
}
```

4. ejbPassivate

```java
public void ejbPassivate() throws java.rmi.RemoteException {
    System.out.println("ejbPassivate ");
    try {
        getConnection().close();
    }
    catch(Exception e){
    }
    finally
    {
        _connection = null;
    }
}
```
17.3.2 Setting Deployment and Control Descriptors

You can also change some parameters for your individual bean. In particular, we want to change the transactional semantics of EmployeeID. Select the EmployeeID bean in the EJBs pane, and select the EJBs->Properties menu item. Change the Transaction Attribute to TX_BEAN_MANAGED and click on OK.

![EJB Properties](image)

Figure 83. EJB Properties

17.4 Generating EJB Deployment Classes

Once you have set the deployment and control descriptors for an EJB, you can use the Deployment processor to generate the required EJB deployment classes.

To generate EJB deployment classes:
1. In the EJBs pane of the EJBs page, select an EJB group or an EJB.
2. Click mouse button 2, then do one of the following:
• If you want to generate the implementation classes for home or remote
to generate the implementation classes for home or remote interfaces and persist or finder classes for CMP beans, select
Generate->EJS Implementations.

• If you want to generate the Java ORB stubs, skeleton, and helper classes for home or remote interfaces, select Generate -> Stub and Skeleton.

• If you want to generate both EJS implementation classes and the stubs and skeletons, select Generate - All.

Once the classes have been generated, they are displayed in the Types pane of the EJBs page.

3. If you want to hide the generated classes in the Types pane, click the Show Generated Classes toggle button on the tool bar.

If you are deploying through another EJB server, such as Component Broker, you can use the EJBs->Export EJB Jar... menu item to create a jar file that can be used by the EJB server (see “Exporting EJBs to a Jar File” on page 197). VisualAge for Java, however, can act as an EJB server without you exporting the jar file. It finds the EJB from the classpath. But you still need to generate EJS implementations.

In addition, if you are going to use VisualAge for Java as an EJB client (which the TEA does), you have to create the stubs and skeletons. Select the EmployeeID EJB in the EJBs pane, and use the EJBs->Generate All menu item to generate the necessary classes. Once you have created the EJB, you can begin the necessary steps for deployment. But first you have to create an AuthenticatorImpl class to use the new EmployeeID EJB.

17.5 Writing a New AuthenticatorImpl Class

Just as in Chapter 16, “WebSphere and Component Broker Interoperability” on page 173, we create a new instance of the AuthenticatorImpl class (see the tea.business.ejb.AuthenticatorImpl class) that performs an authentication role. However, this time, instead of connecting to a Component Broker ORB, we connect to an EJB server.

Hashtable env = new Hashtable(5);
env.put("java.naming.factory.initial",
   "com.ibm.jndi.CosNaming.CNInitialContextFactory");

InitialContext initContext = null;
initContext = new InitialContext(env);

// Get the home interface from JNDI using the first parm
// passed in on the command line as the name
EmployeeIDHome employeeHome = null;

Object o = initContext.lookup("EmployeeIDBean");

if (o instanceof org.omg.CORBA.Object)
    employeeHome = EmployeeIDHomeHelper.narrow((org.omg.CORBA.Object) o);

// Find a customer by primary key
EmployeeIDKey key = new EmployeeIDKey(serialNumber);
EmployeeID employee = (EmployeeID) employeeHome.findByPrimaryKey(key);
employee.authenticate(password);

Notice that the connection is made to a server called TEAServer and a container called TEAContainer. The reasoning behind these values will become apparent when we discuss how to start the EJB servers.

17.6 Creating an EJB Server for Your EJB Objects

Once you have implemented the new AuthenticatorImpl class and have generated the deployment classes for an EJB, you can use the Unit Test tools to test its operation.

Follow these steps to test an EJB:

1. If the EJB that you want to test is an entity bean, start your database servers. If you want to run and test entity beans in the EJB Development Environment, you need to start the DB2 database servers.
   To start the DB2 servers:
   • On the Windows desktop, select Start->Programs->DB2 for Windows NT/95->Command Window to open a DB2 command window.
   • Issue the following two commands to ensure that the DB2 database server and the DB2 administration server are running:
     db2start
     db2admin start
   If the two servers are not running, the commands start the servers. (If the servers are already running, you are simply advised of this by a message.)

2. In the EJBs pane of the EJBs page, select one or more EJB groups that contain the EJBs that you want to test, then click mouse button 2 and select Add to - Server Configuration. The Enterprise Java Server
Configurations window appears (Figure 84).

![Enterprise Java Server Configurations Window](image)

In the Servers pane of the Enterprise Java Server Configurations window, entries appear for the Location Service Daemon and the Persistent Name Server. These name servers provide services that enable your client to locate your EJB.

The Servers pane also contains an EJB Server (server1) entry. This EJB Server will be used to test your EJB.

3. If you need to set properties for one of the name servers, select the server in the Servers pane of the Enterprise Java Server Configurations window, then click mouse button 2 and select **Properties** from the pop-up. In most cases, the default properties currently set for the two name servers should be sufficient. However, if you do change the properties, ensure that the LSD Port value for the Persistent Name Server matches the ORB Listener Port value for the Location Service Daemon.

### 17.7 Starting the EJB Servers

In the Servers pane, press and hold the Ctrl key and select both the **Location Service Daemon** and the **Persistent Name Server**, then click mouse button 2 and select **Start Server**. The Console window (Figure 87) appears in the
background and monitors the two servers.

For each of your EJB groups, you must start a server process to manage the objects in the group. The EJB servers are configured and started from the Enterprise Java Server Configurations window (Figure 84). If the window is not open, select the EJBs->Open To->Server Configuration menu option.

Select the EJB server, and select Properties from the pop-up menu to show the properties for the TEAServer (see Figure 86). The Database URL is used by the EJB server to manage the persistence of the EJB primary key information. It is not necessarily the database that the EJB itself uses when instantiating from, or saving to, persistent storage.

Note that the Deployed Server Name and the Deployed Entity Bean Container Name match those that we mentioned in “Writing a New AuthenticatorImpl Class” on page 192. After you click OK, your server name changes to EJB server(TEAServer).
Follow these steps to create and start your server:

1. In the Enterprise Java Server Configurations window, select the entry **EJB Server (TEAServer)** in the Servers pane. Beside the Servers pane in the EJBs pane, the name of the EJB group you selected earlier appears.

2. Click mouse button 2 and select **Start Server**. The Console window monitors the EJB server. When the EJB server is started, the EJB contained in the EJB group is automatically deployed to the EJB server, and the EJBs are automatically registered with the name servers under an assigned name.

This creates a new EJB server and adds it to the list of servers.

It also opens the Console window (Figure 87) where the corresponding server has been added. On the Standard Out pane, you see the trace generated by the server as well as your own, if any, inserted in EmployeeIDBean implementation. Once the server is initialized, you should check that the last two lines of the trace indicate:

```
bindings:
(EmployeeIDBean=tea.business.ejb._EmployeeIDHomeSkeleton@2f47)
```

If the last line contains only empty braces, something went wrong.
Once you have tested your EJB, you can export it to either an EJB jar file or an EJS jar file. An EJS jar file contains both the user-written code and the deployed code.

To export an EJB to a jar file:

1. In the EJBs pane of the EJBs page, select one or more EJB groups or EJBs.

2. Click mouse button 2, then do one of the following:
   - If you want to export your EJBs to an EJS jar file and you want to export generated types in addition to the usual EJB types, then select Export EJS JAR from the pop-up menu.
   - If you want to export your EJBs to an EJB jar file and you want to export the usual EJB types while excluding the generated types, select Export EJB JAR from the pop-up menu.

The Export to a jar File SmartGuide appears (Figure 88).
3. In the jar file field, type in the name of the jar file that you want to create (ex: toComponentBroker.jar). Alternatively, you can click the **Browse** button and browse to an existing jar file that you want to overwrite. The SmartGuide marks all of the class files associated with the selected EJB groups or individual EJBs for exportation to the jar file.
4. Beside the beans check box, click the Details button to open a dialog that lists the beans that are marked for inclusion in the jar file. These beans represent the selected EJB groups or individual EJBs.

5. If you want to add or remove beans, class files, java files, or resource files from the exportation list, you can click the appropriate Details button and add or remove them in the resulting dialog. Note, however, that the SmartGuide does not automatically add or remove the .class files associated with added or removed beans.

6. To ensure that all required class files are automatically exported for any types that are referenced (directly or indirectly) by the selected types, click the Select referenced types and resources button.

Once the jar file has been created, you can use the jar utility to check its contents:

S:\VajavaEJB>jar xvf toComponentBroker.jar

extracted: tea\business\ejb\_EmployeeIDHomeSkeleton.class
extracted: tea\business\ejb\_EmployeeIDHomeStub.class
extracted: tea\business\ejb\_EmployeeIDSkeleton.class
extracted: tea\business\ejb\_EmployeeIDStub.class
extracted: tea\business\ejb\EJSRemoteEmployeeID.class
extracted: tea\business\ejb\EmployeeID.class
extracted: tea\business\ejb\EmployeeID.ser
extracted: tea\business\ejb\EmployeeIDBean.class
extracted: tea\business\ejb\EmployeeIDHelper.class
extracted: tea\business\ejb\EmployeeIDHolder.class
extracted: tea\business\ejb\EmployeeIDHome.class
extracted: tea\business\ejb\EmployeeIDHomeHelper.class
extracted: tea\business\ejb\EmployeeIDHomeHolder.class
extracted: tea\business\ejb\EmployeeIDKey.class
extracted: README
extracted: META-INF\MANIFEST.MF

The README gives some examples of how to use the JNDI API, and the MANIFEST.MF file provides a description of the jar file contents.

If you need to explore the deployment descriptor, you can use the following code and the VisualAge for Java debugger to inspect it:

```java
package Descriptor;

/**
 * This type was created in VisualAge.
 */
```
import java.io.*;
import java.util.*;
import javax.ejb.deployment.*;

public class ReadDescriptorFile {
/**
 * ReadDescriptorFile constructor comment.
 */
public ReadDescriptorFile() {
  super();
}
/**
 * This method was created in VisualAge.
 * @param args java.lang.String[]
 */
public static void main(String args[]) {
  String filename = args[0];
  EntityDescriptor myDescriptor = null;

  try {
    FileInputStream fin = new FileInputStream(filename);
    ObjectInputStream oin = new ObjectInputStream(fin);
    myDescriptor = (EntityDescriptor) oin.readObject();
    java.lang.reflect.Field[] containerFields =
      myDescriptor.getContainerManagedFields();
    System.out.println("object = " + myDescriptor.toString() );
    oin.close();
    fin.close();
  } catch (ClassNotFoundException e) {System.out.println("ERROR ClassNotFound");}
  catch (IOException e ) {System.out.println("ERROR "); } }
}

You can set a breakpoint after the descriptor object has been read from the serialized file and use the debugger to inspect it (see Figure 89).
Figure 89. Inspecting the Deployment Descriptor Object
You can also use a tool like jet, which is provided with WebSphere and Component Broker. Then you can import the deployment jar file and analyze and change it (see Figure 90).

Figure 90. Analyzing and Modifying Your EJB jar File with Jet
You can use this newly created jar file and deploy it on any EJB container. A good candidate is Component Broker, which has supported EJB since Version 1.3.

17.9 Importing EJBs from a Jar File

Before you import an EJB jar file, you have to create an EJB group from the EJBs page. Then you can add EJBs to the EJB groups in one of the following ways:

1. In the EJBs pane of the EJBs page, select the EJB group that you want to contain the imported EJBs.

2. Click mouse button 2, then select Import EJB JAR from the pop-up menu. The Import from a jar/Zip File SmartGuide appears (see Figure 91).
3. Beside the Filename field, click the **Browse** button. A dialog appears.

4. In the dialog, navigate to the jar file and then select it and click **OK**. Alternatively, you can type the name of the jar file in the Filename field. The SmartGuide marks for importation all of the types and beans found in the jar file.

5. Beside the .ser check box, click the **Details** button. A dialog appears that displays a list of the serialized deployment descriptors in the jar file. Each deployment descriptor maps to an EJB in the file. If you want, you can use
the dialog to add or remove .ser files. However, the SmartGuide does not add or remove any class files associated with the added or removed .ser files.

Note that you must select the associated .class files for the beans you want to import. The Import from a jar/Zip File SmartGuide cannot determine the appropriate set of .class files based on the beans selected in the Details page. If you do not select a .class file required by a bean that is selected, the import will fail. If you select .class files that are not used by any bean selected, the import will succeed, but only selected beans will be created in the group. However, all .class files selected will be imported into the workspace.

This is the end of our EJB tour.
Appendix A. IDL to Java Mapping

CORBA uses a neutral definition language, the Interface Definition Language (IDL), to describe the services it is implementing. Because proxy objects are written in a dependent language, you have to define a way to map IDL to Java, keeping the conversion as natural as possible.

The names of the IDL definition are mapped to Java classes of the same name. If the IDL compiler detects name collisions, it prefixes an underscore (_) to the mapped name. Because an IDL definition can produce more than one Java construct, the IDL compiler adds a descriptive suffix to the mapped name. For example, for an IDL interface named “customer,” you have “customer” Java interface, “customerHolder,” and “customerHelper” classes. When you describe an interface, you design your model in IDL, and then an IDL compiler generates language-specific files. This way works fine when you design a model from scratch, or when you have an existing model and want to implement the IDL interfaces with Java. Sometimes, however, you have a model already implemented in Java and would like to produce the related IDL to use your interfaces within a CORBA-compliant distributed environment. The Java to IDL compiler makes sense in this case, because you already have the Java classes that have implemented your business logic, and what you need is to describe the interface in a neutral descriptive language.

A.1 IDL Module

In IDL, the module is a name-scoping mechanism. In Java you have packages as well. Each IDL module is mapped to a Java package with the same name:

```idl
// Security.idl
module Security {
const boolean activated = FALSE;
// Security.activated.java
package Security;
public final class activated {
public static final boolean value = false;
}
```

Any IDL declarations not included within the module are mapped to the (unnamed) Java global scope.
A.2 Constant Value

IDL constant values outside the interface are mapped to the public interface with the same name. The value of the constant is mapped to the public static final named value. Within an interface a constant value is mapped to the public static final named value of that interface:

```idl
module Awk {
  const long time = 1000;
}
```

```java
package Awk
public interface time{
  public static final int value = 1000;
}
```

To access the constant time, you use Awt.time.value.

A.3 Basic Types

In this section we describe the mapping of the basic IDL types to Java types. Some Java types are larger than the corresponding IDL types. Java does not support unsigned types. Table 3 on page 208 shows the IDL to Java basic type mapping.

<table>
<thead>
<tr>
<th>IDL Type</th>
<th>Java Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean</td>
<td>boolean</td>
</tr>
<tr>
<td>char</td>
<td>char</td>
</tr>
<tr>
<td>wchar</td>
<td>char</td>
</tr>
<tr>
<td>octer</td>
<td>byte</td>
</tr>
<tr>
<td>string</td>
<td>java.lang.String</td>
</tr>
<tr>
<td>wstring</td>
<td>java.lang.String</td>
</tr>
<tr>
<td>short</td>
<td>short</td>
</tr>
<tr>
<td>unsigned short</td>
<td>short</td>
</tr>
<tr>
<td>long</td>
<td>int</td>
</tr>
<tr>
<td>unsigned long</td>
<td>int</td>
</tr>
<tr>
<td>long long</td>
<td>long</td>
</tr>
</tbody>
</table>
A few considerations about mappings of basic types: IDL chars use the ISO 8859-1 character set, having at most 255 different values. Java chars use the Unicode character set, having 65536 different values. Because Java chars are larger than the IDL chars, you must check to ensure that the Java chars are not outside the ISO 8859-1 character set. The same applies for string mapping. IDL supports unsigned numbers, whereas Java supports only signed numbers. The IDL numbers and chars are mapped to the correspondingly sized Java types. Thus you are responsible for ensuring that large unsigned IDL type values are handled correctly as negative numbers in Java.

### A.4 Constructed Types

Constructed types are enums, structs, unions, sequences, and arrays.

#### A.4.1 Enumeration

An IDL enum is mapped to a final class with the same name that defines a static final int field for each enum member. It also includes a method for checking the range of the enum.

Example:

```java
// NYStore.idl
module NYStore{
 enum Cars { Ferrari, Lamborghini, Jaguar};
 // NYStore/Car
 package NYStore;
 final public class Cars {
     final public static int _Ferrari = 0;
     final public static int _Lamborghini = 1;
     final public static int _Jaguar = 2;
     final public static NYStore.Cars Ferrari = new NYStore.Cars(_Ferrari);
     final public static NYStore.Cars Lamborghini = new NYStore.Cars(_Lamborghini);
     final public static NYStore.Cars Jaguar = new NYStore.Cars(_Jaguar);
     private int __value;
     private Cars(int value) {
```

<table>
<thead>
<tr>
<th>IDL Type</th>
<th>Java Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>unsigned long long</td>
<td>long</td>
</tr>
<tr>
<td>float</td>
<td>float</td>
</tr>
<tr>
<td>double</td>
<td>double</td>
</tr>
</tbody>
</table>
this.__value = value;
}
public int value() {
    return __value;
}
public static NYStore.Cars from_int(int $value) {
    switch($value) {
        case _Ferrari:
            return Ferrari;
        case _Lamborghini:
            return Lamborghini;
        case _Jaguar:
            return Jaguar;
        default:
            throw new org.omg.CORBA.BAD_PARAM("Enum out of range:
                [0.." + (3 - 1) + "]: " + $value);
    }
}
public java.lang.String toString() {
    org.omg.CORBA.Any any = org.omg.CORBA.ORB.init().create_any();
    NYStore.CarsHelper.insert(any, this);
    return any.toString();
}

The IDL compiler also generates a holder class and a helper class called,
respectively, CarsHolder.java and CarsHelper.java.

### A.4.2 Struct

An IDL struct is mapped to a final Java class with the same name. Each field
in the struct is mapped to an attribute of the generated class. A holder class is
also generated.

**Example**

// Block.idl
module Block{
    struct Blockid {
        long id;
        char type;
        string name;
    // Block/Blockld.java
    package Block;
    final public class Blockid {
        public int id;
        public char type;


210 Using VisualAge for Java Enterprise Version 2 to Develop CORBA and EJB Applications
A.4.3 Union

An IDL union is mapped to a final Java class with the same name that has a default constructor, an accessor method for the discriminator, and an accessor and modifier method for each of the branches. The accessor method name consists of "get" followed by the name of the branch. The modifier method name consists of "set" followed by the name of the branch. A Holder class is also generated.

Example:

// Example.idl
module Example
{
    enum MicroChip{Intel, Sparc, Motorola, AMD};
    union platform switch(MicroChip)
    {
        case Intel: string window;
        case Sparc: long bit;
        default: char none;
    };
};
// Example/platform.java
package Example;
final public class platform {
    private java.lang.Object _object;
    private Example.MicroChip _disc;
    private Example.MicroChip _defdisc = Example.MicroChip.AMD;
    public platform() { }
    public Example.MicroChip discriminator() {
        return _disc;
public java.lang.String window() {
    if (_disc.value() != (int) Example.MicroChip.Intel.value() && true) {
        throw new org.omg.CORBA.BAD_OPERATION("window");
    }
    return (java.lang.String)_object;
}

public int bit() {
    if (true) {
        throw new org.omg.CORBA.BAD_OPERATION("bit");
    }
    return ((java.lang.Integer)_object).intValue();
}

public char none() {
    if (_disc.value() != (int) Example.MicroChip.AMD.value() && true) {
        throw new org.omg.CORBA.BAD_OPERATION("none");
    }
    return ((java.lang.Character)_object).charValue();
}

public void window(java.lang.String value) {
    _object = value;
}

public void bit(int value) {
    _object = new java.lang.Integer(value);
}

public void none(char value) {
    _disc = _defdisc;
    _object = new java.lang.Character(value);
}

public java.lang.String toString() {
    org.omg.CORBA.Any any = org.omg.CORBA.ORB.init().create_any();
    Example.platformHelper.insert(any, this);
    return any.toString();
}

A.4.4 Sequence

An IDL sequence is mapped to a Java array. Bounds checking is done on the bounded sequences. A Holder class and a Helper class are generated. The name of the sequence is appended with the Holder name.

Example:

// Attribute.idl
module Attribute {
    typedef sequence <char, 10> bounded;
    // boundedHolder.java
    package Attribute;
}
final public class boundedHolder implements org.omg.CORBA.portable.Streamable {
    public char[] value;
    public boundedHolder() {
    }
    public boundedHolder(char[] value) {
        this.value = value;
    }
    public void _read(org.omg.CORBA.portable.InputStream input) {
        value = boundedHelper.read(input);
    }
    public void _write(org.omg.CORBA.portable.OutputStream output) {
        boundedHelper.write(output, value);
    }
    public org.omg.CORBA.TypeCode _type() {
        return boundedHelper.type();
    }
}

A.4.5 Array

An IDL array is mapped in the same way as an IDL bounded sequence. The name of the array is appended with the Holder name.

Example:

// Example.idl
module Example {
typedef long larray[10];
// larrayHolder.java
package Example;
final public class larrayHolder implements org.omg.CORBA.portable.Streamable {
    public int[] value;
    public larrayHolder() {
    }
    public larrayHolder(int[] value) {
        this.value = value;
    }
    public void _read(org.omg.CORBA.portable.InputStream input) {
        value = larrayHelper.read(input);
    }
    public void _write(org.omg.CORBA.portable.OutputStream output) {
        larrayHelper.write(output, value);
    }
    public org.omg.CORBA.TypeCode _type() {
        return larrayHelper.type();
    }
}
A.5 Interface

An IDL interface is mapped to a Java interface with the same name. Two additional classes are generated, the Holder class and the Helper class. The Java interface extends the base interface, org.omg.CORBA.Object. The interface contains the signature of the methods. Methods are invoked on the object reference to the interface. The attributes of the IDL interface are mapped to the modifier and selector methods of the Java interface. The read-only attribute of the interface has no modifier method in the Java interface.

Example:

```java
// Satellite.idl
module Satellite {
    interface Monitor {
        long getID( in string name );
        attribute char alfa;
        readonly attribute long size;
    };
};
// Satellite/Monitor.java
package Satellite;
public interface Monitor extends org.omg.CORBA.Object {
    public int getID( java.lang.String name );
    public void alfa();
    public char alfa();
    public long size();
}
```

We have to say a few words about the parameter passing modes. IDL supports call-by-value (in parameters), call-by-result (out parameters), and call-by-value/result (inout parameters) semantics. Java supports only call-by-value. To map all of the parameter types, you need an additional class, the Holder class, to implement the parameter modes. The Holder class is passed by value, but before the invocation of the method the input value must be set in the Holder class that is the out/inout parameter. When the method returns, you can use the value of the Holder class to access the out/inout parameters.

Example:

```java
// Joke.idl
module Joke
```
{ interface Clown{
    long start( out string ok, in long age )
};
// Joke/Clown.java
package Joke;
public interface Clown extends org.omg.CORBA.Object {
    public int start( org.omg.CORBA.StringHolder ok, int age);
}

You can set the value of the out-inout parameters, creating an instance of the relative Holder class:

IntHolder intHolderParam = new IntHolder(1234);
Retrieve the value:
myInt = intHolderParam.value;

To access the value of the in-inout parameter, use StringHolder.value. The IDL compiler generates for each basic type and user-defined type a Holder class to support in-inout parameters.

A.6 Exception

An IDL exception is mapped to a Java class. There are two kinds of exceptions in CORBA: user-defined exceptions and system exceptions. A user-defined exception is mapped to the final Java classes that inherit from org.omg.CORBA.UserException, which extends the java.lang.Exception. A Helper class and a Holder class are also generated.

Example of user-defined exception:

// errors.idl
module Errors
{
    exception error1 {string why; };
// Errors/error1.java
package Errors;
final public class error1 extends org.omg.CORBA.UserException {
    public java.lang.String why;
    public error1() {}
    public error1(java.lang.String why) {
        this.why = why;
    }
    public java.lang.String toString() {
        org.omg.CORBA.Any any = org.omg.CORBA.ORB.init().create_any();
        Errors.error1 Helper.insert(any, this);
        return any.toString();
    }
}
System exceptions are mapped to the final Java classes that extend `java.lang.RuntimeException`. Table 4 on page 216 shows the mapping of the system IDL exceptions to Java classes.

<table>
<thead>
<tr>
<th>IDL Exception</th>
<th>Java Class Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>CORBA::UNKNOWN</td>
<td>org.omg.CORBA.UNKNOWN</td>
</tr>
<tr>
<td>CORBA::BAD_PARAM</td>
<td>org.omg.CORBA.BAD_PARAM</td>
</tr>
<tr>
<td>CORBA::NO_MEMORY</td>
<td>org.omg.CORBA.NO_MEMORY</td>
</tr>
<tr>
<td>CCORBA::IMP_LIMIT</td>
<td>org.omg.CORBA.IMP_LIMIT</td>
</tr>
<tr>
<td>CORBA::COMM_FAILURE</td>
<td>org.omg.CORBA.COMM_FAILURE</td>
</tr>
<tr>
<td>CORBA::INV_OBJREF</td>
<td>org.omg.CORBA.INV_OBJREF</td>
</tr>
<tr>
<td>CORBA::NO_PERMISSION</td>
<td>org.omg.CORBA.NO_PERMISSION</td>
</tr>
<tr>
<td>CORBA::INTERNAL</td>
<td>org.omg.CORBA.INTERNAL</td>
</tr>
<tr>
<td>CORBA::MARSHAL</td>
<td>org.omg.CORBA.MARSHAL</td>
</tr>
<tr>
<td>CORBA::NO_IMPLEMENT</td>
<td>org.omg.CORBA.NO_IMPLEMENT</td>
</tr>
<tr>
<td>CORBA::BAD_TYPECODE</td>
<td>org.omg.CORBA.BAD_TYPECODE</td>
</tr>
<tr>
<td>CORBA::BAD_OPERATION</td>
<td>org.omg.CORBA.BAD_OPERATION</td>
</tr>
<tr>
<td>CORBA::NO_RESOURCES</td>
<td>org.omg.CORBA.NO_RESOURCES</td>
</tr>
<tr>
<td>CORBA::NO_RESPONSE</td>
<td>org.omg.CORBA.NO_RESPONSE</td>
</tr>
<tr>
<td>CORBA::PERSIST_STORE</td>
<td>org.omg.CORBA.PERSIST_STORE</td>
</tr>
<tr>
<td>CORBA::BAD_INV_ORDER</td>
<td>org.omg.CORBA.BAD_INV_ORDER</td>
</tr>
<tr>
<td>CORBA::TRANSIENT</td>
<td>org.omg.CORBA.TRANSIENT</td>
</tr>
<tr>
<td>CORBA::FREE_MEM</td>
<td>org.omg.CORBA.FREE_MEM</td>
</tr>
<tr>
<td>CORBA::INV_IDENT</td>
<td>org.omg.CORBA.INV_IDENT</td>
</tr>
<tr>
<td>CORBA::NV_FLAG</td>
<td>org.omg.CORBA.INV_FLAG</td>
</tr>
<tr>
<td>CORBA::INV_IDENT</td>
<td>org.omg.CORBA.INV_IDENT</td>
</tr>
</tbody>
</table>
A.7 Any Type

The IDL Any type is mapped to the org.omg.CORBA.Any class in Java. This class has all of the methods required to extract and insert values of all primitive IDL types and other two other methods to handle all user-defined types in a portable way.

A.8 Typedef

An IDL typedef is mapped to the original mapped type wherever the typedef type appears.
Using VisualAge for Java Enterprise Version 2 to Develop CORBA and EJB Applications
Appendix B. TEA User Interface

In this appendix we provide a few screen captures for the applet and HTML versions of the TEA client.

A typical navigation scenario consists of:

1. Logging on and creating a trip with the Java IIOP client (see Figure 92) and the pure HTML client (see Figure 93)
2. On a selected trip, adding new expenses or listing existing expenses with the Java IIOP client (see Figure 94).
Figure 92. Applet User Interface: From Logging on to Creating a Trip
Figure 93. HTML User Interface: Logging on and Creating a Trip
Figure 94. Applet User Interface: Creating or Listing Expenses Related to a Trip
Appendix C. Troubleshooting Guide

In this appendix we describe some common problems that you might encounter during your development process and explain how you might overcome them.

C.1 COMM_FAILURE Exception - Minor Code 3

The interface name that you specified in the interface-to-implementation mapping probably does not exist. Check the getImpls() method in your servlet (in the case of TEA, this is tea.servlet.TEAServlet), or if you are not overriding this method, the ObjectImpl.properties file.

C.2 COMM_FAILURE Exception - Minor Code 5

The IHome was unable to find an implementation class for the interface name that was passed. Be sure that your getImpls() method contains a valid, fully qualified class name for the implementation. The interface name must also be fully qualified.

C.3 COMM_FAILURE Exception - Minor Code 6

Usually a communications error when trying to connect to the ORB. Ensure that:

• The Web server is up and running.
• The address in the IHome URL is valid.
• The address in the IHome URL is the same as the Web server that served the page. The sandbox rules for Java applets mandate that connections are permitted only to the same address from which the applet was served. Note that in this context IP address and hostname are not considered equivalent. For example, the browser does not consider 127.0.0.1 to be the same as localhost.

C.4 COMM_FAILURE Exception - Minor Code 10

The IHome was unable to create an instance of your implementation class. Be sure that the implementation class is in the classpath and that it has a public constructor.
C.5 COMM_FAILURE Exception - Minor Code 15

The IHome was unable to create an instance of the skeleton class for your implementation class. This problem occurs only when you use the tie- or delegation-based approach. Be sure that your skeleton class exists and belongs to the same package as the implementation class.

C.6 Starting the Web Server Throws ClassNotFound Exception

WebSphere and VisualAge for Java have a known incompatibility that causes this exception to be thrown sporadically upon Web server initiation. The problem is caused by the absence of the BeanInfo classes for several servlets. See Chapter 9, “Environment Configuration” on page 63 for more details on how to fix the problem.

C.7 OutOfMemory Exception

When you run WebSphere, Lotus Go seems to have a problem shutting down properly. If you get this error when trying to start the Web server in VisualAge for Java, you most likely still have an instance of the Lotus Go Web server still running. From the NT Task Manager, kill the wHttpg.exe task.

C.8 Empty Screen in Browser When Starting HTML Client

This symptom is sometimes exhibited when you have omitted the pluginservice from the server.properties file. See “Configuring the Web Server for Clients” on page 70.
Appendix D. Workaround Details

As with many products that are implementing the latest technologies, both WebSphere and VisualAge for Java exhibit several incompatibilities and bugs. In this appendix we cover a few of these incompatibilities and explain the solutions we used to overcome them.

D.1 CosNamingServlet Not Starting

When the Web server receives a request to initialize the CosNamingServlet (in order to get the IOR of the Naming Service), for some reason WebSphere mangles the servlet parameters. A sample invocation of the CosNamingServlet looks like this:

http://localhost/servlet/CosNamingServlet?initref=true

However, because of to this bug, by the time the CosNaming servlet receives the request, the request line has changed to:

http://localhost/servlet/CosNamingServlet?initref=true?initref=true

The CosNamingServlet, in its initialization routine, makes a call to the getParameter() method looking for the value of "initref." Instead of getting back the value of "true," it gets back a value of "true?initref=true." Obviously, this is not desired behavior.

To work around this, we decompiled the com.ibm.CORBA.iiop.ORB class to make the client send a request that looks like this:

http://localhost/servlet/CosNamingServlet?initref=true&blah=xx

The addition of the extra parameter does not cure the mangling, but instead of the initref parameter getting mangled, the unused blah parameter is modified.

D.2 Unable to Get Naming Context on the Server

WebSphere has a restriction that results in the naming context only being accessible from applets. Rephrased, this means that a Java application cannot use the naming context. An application in this sense includes both clients that have been written as applications (rather than applets), Java CORBA application servers, and WebSphere servlets. Obviously, this is quite a severe restriction. The only way that we found to overcome this problem is to "trick" the WebSphere ORB into thinking that it was running in an applet environment. In order to do this, we created a class that subclassed Applet
and used it to initialize the ORB. Using this class, you can initialize your ORB in the following manner:

```java
ORB orb = ORB.init(new KludgeApplet(), properties);
```

This appears to work in all instances of application-based CORBA development with WebSphere.

D.3 BeanInfo ClassNotFound Exceptions

WebSphere and VisualAge for Java have a known incompatibility that causes ClassNotFound exceptions to be thrown when the Web server is started. To work around this problem, we created BeanInfo classes for the servlets in question (see Chapter 9, “Environment Configuration” on page 63 for further details).
Appendix E. TEA Package Descriptions

In this appendix we describe the contents of the various packages in the TEA, the behavior of the classes, and how they were created. The corresponding source and executable code are available in the sg245276.zip file and can be downloaded from the ITSO home page on the Internet:


**tea.applet** - Contains the various panels that comprise the user interface. The authors manually created these classes.

**tea.applet.controller** - Contains the controllers in the MVS architecture that manage the population of and navigation among the panels in the tea.applet package. The authors manually created these classes.

**tea.business** - Contains the generated interfaces, stubs, and skeletons as a result of the IDL compilation. The IDL compiler generated these classes.

**tea.business.cb** - Contains the Component Broker generated EmployeeID interfaces and classes and the implementation of the AuthenticatorImpl class that is used when Component Broker is the authentication mechanism.

**tea.business.DepartmentPackage** - Contains generated helper classes for the department interface. The IDL compiler generated these classes.

**tea.business.ejb** - Contains the EJB generated EmployeeID interfaces and classes and the implementation of the AuthenticatorImpl class that is used when EJB is the authentication mechanism.

**tea.business.EmployeePackage** - Contains generated helper classes for the employee interface. The IDL compiler generated these classes.

**tea.business.impl** - Contains the business logic of the application. The authors manually created these classes.

**tea.QueryServicePackage** - Contains generated helper classes for the query service interface. The IDL compiler generated these classes.

**tea.TripPackage** - Contains generated helper classes for the trip interface. The IDL compiler generated these classes.

**tea.proxy** - Contains the servlets that are used for the pure HTML client. The authors manually created these servlets, using the ServletBuilder tool in VisualAge for Java.
**tea.servlet** - Contains the Web server that is used when developing within VisualAge for Java and the TEAServlet class that is used by WebSphere as an IHome. The authors wrote these classes.

**tea.util** - Contains some utility classes written by the authors.

**tea.TestGroupEJBReserved** - Generated package used by the EJB tool.

**tea.Travel$Expense$ApplicationEABPMReserved** - Generated package used by the IDL tool.
Appendix F. Special Notices

This publication is intended to help application developers create CORBA applications using VisualAge for Java. The information in this publication is not intended as the specification of any programming interfaces that are provided by VisualAge for Java Enterprise. See the PUBLICATIONS section of the IBM Programming Annoucement for VisualAge for Java Enterprise for more information about what publications are considered to be product documentation.

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

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

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

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

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

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

Any pointers in this publication to external Web sites are provided for convenience only and do not in any manner serve as an endorsement of these Web sites.

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

IBM CICS
DB2 MVS
S/390 OS/390
ThinkPad VisualAge

The following terms are trademarks of other companies:

C-bus is a trademark of Corollary, Inc.

Java and HotJava are trademarks of Sun Microsystems, Incorporated.

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

PC Direct is a trademark of Ziff Communications Company and is used by IBM Corporation under license.

Pentium, MMX, ProShare, LANDesk, and ActionMedia are trademarks or registered trademarks of Intel Corporation in the U.S. and other countries.

UNIX is a registered trademark in the United States and other countries licensed exclusively through X/Open Company Limited.

Other company, product, and service names may be trademarks or service marks of others.
Appendix G. Related Publications

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

G.1 International Technical Support Organization Publications

For information on ordering these ITSO publications see “How to Get ITSO Redbooks” on page 233.

- IBM Component Broker Connector Overview, SG24-2022
- IBM CBConnector Cookbook Collection: First Steps, SG24-2033
- IBM CBConnector Cookbook Collection: CBConnector Bank Implementation, SG24-5119
- Application Development with VisualAge for Java Enterprise, SG24-5081
- VisualAge for Java Enterprise Version 2 Team Support, SG24-5245
- Unlimited Enterprise Access with Java and VisualAge Generator, SG24-5246
- Factoring JavaBeans in the Enterprise, SG24-5051
- From Client/Server to Network Computing, A Migration to Java, SG24-2247
- JavaBeans by Example: Cooking with Beans in the Enterprise, SG24-2035
- VisualAge for Java Enterprise Version 2
  DataBeans - Servlets - CICS Connector, SG24-5265

G.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>
</tbody>
</table>

© Copyright IBM Corp. 1998
G.3 Other Publications

These publications are also relevant as further information sources:


<table>
<thead>
<tr>
<th>CD-ROM Title</th>
<th>Subscription Number</th>
<th>Collection Kit Number</th>
</tr>
</thead>
<tbody>
<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>
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 EHOME4 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>Country</th>
<th>IBMMAIL</th>
<th>Internet</th>
</tr>
</thead>
<tbody>
<tr>
<td>In United States</td>
<td>usib6fpl at ibmmail</td>
<td><a href="mailto:usib6fpl@ibmmail.com">usib6fpl@ibmmail.com</a></td>
</tr>
<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**
  
<table>
<thead>
<tr>
<th>Country</th>
<th>Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>United States (toll free)</td>
<td>1-800-879-2755</td>
</tr>
<tr>
<td>Canada (toll free)</td>
<td>1-800-IBM-4YOU</td>
</tr>
<tr>
<td>Outside North America</td>
<td>(long distance charges apply)</td>
</tr>
<tr>
<td>(+45) 4810-1320 - Danish</td>
<td>(+45) 4810-1020 - German</td>
</tr>
<tr>
<td>(+45) 4810-1420 - Dutch</td>
<td>(+45) 4810-1620 - Italian</td>
</tr>
<tr>
<td>(+45) 4810-1540 - English</td>
<td>(+45) 4810-1270 - Norwegian</td>
</tr>
<tr>
<td>(+45) 4810-1670 - Finnish</td>
<td>(+45) 4810-1120 - Spanish</td>
</tr>
<tr>
<td>(+45) 4810-1220 - French</td>
<td>(+45) 4810-1170 - Swedish</td>
</tr>
</tbody>
</table>

- **Mail Orders** – send orders to:
  
<table>
<thead>
<tr>
<th>Country</th>
<th>IBM Publications</th>
<th>IBM Direct Services</th>
</tr>
</thead>
<tbody>
<tr>
<td>IBM Publications Customer Support</td>
<td>144-4th Avenue, S.W.</td>
<td>Sortemosevej 21</td>
</tr>
<tr>
<td>P.O. Box 29570</td>
<td>Calgary, Alberta T2P 3N5</td>
<td>DK-3450 Allerød</td>
</tr>
<tr>
<td>Raleigh, NC 27626-0570 USA</td>
<td>Canada</td>
<td>Denmark</td>
</tr>
</tbody>
</table>

- **Fax** – send orders to:
  
<table>
<thead>
<tr>
<th>Country</th>
<th>Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>United States (toll free)</td>
<td>1-800-445-9269</td>
</tr>
<tr>
<td>Canada</td>
<td>1-800-267-4455</td>
</tr>
<tr>
<td>Outside North America</td>
<td>(+45) 48 14 2207 (long distance charge)</td>
</tr>
</tbody>
</table>

- **1-800-IBM-4FAX (United States) or (+1) 408 256 5422 (Outside USA)** – ask for:
  
<table>
<thead>
<tr>
<th>Index #</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>4421</td>
<td>Abstracts of new redbooks</td>
</tr>
<tr>
<td>4422</td>
<td>IBM redbooks</td>
</tr>
<tr>
<td>4420</td>
<td>Redbooks for last six months</td>
</tr>
</tbody>
</table>

- **On the World Wide Web**
  
<table>
<thead>
<tr>
<th>Website</th>
<th>URL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Redbooks Web Site</td>
<td><a href="http://www.redbooks.ibm.com">http://www.redbooks.ibm.com</a></td>
</tr>
</tbody>
</table>

---

**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>
<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.
Glossary

This glossary defines terms and abbreviations that are used in this book. If you do not find the term you are looking for, refer to the IBM Dictionary of Computing, New York: McGraw-Hill, 1994.

This glossary includes terms and definitions from the American National Standard Dictionary for Information Systems, ANSI X3.172-1990, copyright 1990 by the American National Standards Institute (ANSI). Copies may be purchased from the American National Standards Institute, 1430 Broadway, New York, New York 10018.

A

abstract class. A class that provides common behavior across a set of subclasses but is not itself designed to have instances that work. An abstract class represents a concept; classes derived from it represent implementations of the concept. See also base class.

access application. Generated by the Data Access Builder for each schema mapping, an executable GUI that provides access to the database using the other classes generated for the mapping.

accessor methods. Methods that an object provides to define the interface to its instance variables. The accessor method to return the value of an instance variable is called a get method or getter method, and the accessor method to assign a value to an instance variable is called a set method or setter method.

applet. A Java program designed to run within a Web browser. Contrast with application.

application. In Java programming, a self-contained, stand-alone Java program that includes a main() method. Contrast with applet.

argument. A data element, or value, included as a bean in a method call. Arguments provide additional information that the called method can use to perform the requested operation.

attribute. A specification of a property of a bean. For example, a customer bean could have a name attribute and an address attribute. An attribute can itself be a bean with its own behavior and attributes. In the Data Access Builder, the aspect of a schema mapping that represents a column in a database table.

B

base class. A class from which other classes or beans are derived. A base class may itself be derived from another base class. See also abstract class.

bean. A definition or instance of a JavaBeans component. See also JavaBeans.

BeanInfo. (1) A companion class for a bean that defines a set of methods that can be accessed to retrieve information on the bean’s properties, events, and methods. (2) In the VisualAge for Java IDE, a page in the class browser that provides bean information.

beans palette. In the Visual Composition Editor, a two-column pane that contains prefabricated beans that you can select and manipulate to create programs. The left column contains categories of beans, and the right column contains beans for the selected category. The default set of beans generally represents JDK AWT components. You can add your own categories and beans to the beans palette.

break point. A point in a computer program where the execution can be halted.

browser. (1) In VisualAge for Java, a window that provides information on program elements. There are browsers for projects, packages, classes, methods, and interfaces.
(2) An Internet-based tool that lets users browse Web sites.

C

C++ Access Builder. A VisualAge for Java Enterprise tool that generates beans to access C and C++ DLLs.

category. In the Visual Composition Editor, a selectable grouping of beans represented by an icon in the left-most column. Selecting a category displays the beans belonging to that category in the next column. See also beans palette.

CICS Access Builder. A VisualAge for Java Enterprise tool that generates beans to access CICS transactions through the CICS Gateway for Java and CICS Client.

CICS Client. A server program that processes CICS ECI calls, forwarding transaction requests to a CICS program running on a host.

CICS ECI. An API that provides C and C++ programs with procedural access to transactions.

CICS Gateway for Java. A server program that processes Java ECI calls and forwards CICS ECI calls to the CICS Client.

class. An aggregate that defines properties, operations, and behavior for all instances of that aggregate.

class hierarchy. The relationships between classes that share a single inheritance. All Java classes inherit from the Object class.

class library. A collection of classes.

class method. See method.

CLASSPATH. In your deployment environment, the environment variable that specifies the directories in which to look for class and resource files.

client/server. The model of interaction in distributed data processing where a program at one location sends a request to a program at another location and awaits a response. The requesting program is called a client, and the answering program is called a server.

client-side server proxy. Generated by the RMI Access Builder, a local representative of a remote bean. This proxy provides access to the operations of the server bean, allowing a Java client to work with it as if it were the server bean. See also proxy bean and server-side server proxy.

Class Browser. In the VisualAge for Java IDE, a tool used to browse the classes loaded in the workspace.

collection. A set of features in which each feature is an object.

commit. The operation that ends a unit of work and updates the database such that other processes can access any changes made.

Common Object Request Broker Architecture (CORBA). A middleware specification which defines a software bus—the Object Request Broker (ORB)—that provides the infrastructure.

communications area (COMMAREA). In a CICS transaction program, a group of records that describes both the format and volume of data used.

component model. An architecture and an API that allows developers to define reusable segments of code that can be combined to create a program. VisualAge for Java uses the JavaBeans component model.

composite bean. A bean that is composed of a bean and one or more subbeans. A composite bean can contain visual beans, nonvisual beans, or both. See also nonvisual bean, bean, and visual bean.

concrete class. A subclass of an abstract class that is a specialization of the abstract class.

connection. In the Visual Composition Editor, a visual link between two components that represents the relationship between the components. Each connection has a source, a target, and other properties. See also event-to-method connection, event-to-property connection, parameter connection, property-to-method connection, and property-to-property connection.
console. In VisualAge for Java, the window that acts as the standard input (System.in) and standard output (System.out) device for programs running in the VisualAge for Java IDE.

collection from parts. A software development technology in which applications are assembled from existing and reusable software components, known as parts. In VisualAge for Java, parts are called beans.

constructor. A special class method that has the same name as the class and is used to construct and possibly initialize objects of its class type.

container. A component that can hold other components. In Java, examples of containers include applets, frames, and dialogs. In the Visual Composition Editor, containers can be graphically represented and generated.

current edition. The edition of a program element that is currently in the workspace. See also open edition.

cursor. A database control structure used by the Data Access Builder to point to a specific row within some ordered set of rows and to retrieve rows from a set, possibly making updates or deletions.

D

data abstraction. A data type with a private representation and a public set of operations. The Java language uses the concept of classes to implement data abstraction.

Data Access Builder. A VisualAge for Java Enterprise tool that generates beans to access and manipulate the content of JDBC/ODBC-compliant relational databases.

DB2 for MVS/ESA. An IBM relational database management system for the MVS operating system.

double-byte character set (DBCS). A set of characters in which each character is represented by 2 bytes. Languages such as Japanese, Chinese, and Korean, which contain more symbols than can be represented by 256 code points, require double-byte character sets. Compare with single-byte character set.

Distributed Computing Environment (DCE). Adopted by the computer industry as a de facto standard for distributed computing. DCE allows computers from a variety of vendors to communicate transparently and share resources such as computing power, files, printers, and other objects in the network.

Distributed Component Object Model (DCOM). A protocol that enables software components to communicate directly over a network in a reliable, secure, and efficient manner. Previously called "Network OLE," DCOM is designed for use across multiple network transports, including Internet protocols such as HTTP. DCOM is based on the Open Software Foundation's DCE-RPC specification and works with both Java applets and ActiveX components through its use of the Component Object Model (COM).

dynamic link library (DLL). A file containing executable code and data bound to a program at run time rather than at link time. The C++ Access Builder generates beans and C++ wrappers that let your Java programs access C++ DLLs.

E


encapsulation. The hiding of a software object's internal representation. The object provides an interface that queries and manipulates the data without exposing its underlying structure.

enterprise access builders. In VisualAge for Java Enterprise, a set of code-generation tools. See also C++ Access Builder, CICS Access Builder, Data Access Builder, and RMI Access Builder.

event. An action by a user program, or a specification of a notification that may trigger specific behavior. In JDK 1.1, events notify the relevant listener classes to take appropriate actions.
**event-to-method connection.** A connection from an event generated by a bean to a method of another bean. When the connected event occurs, the method is executed. See also connection.

**event-to-property connection.** A connection that changes the value of a property when a certain event occurs. See also connection.

**feature.** (1) A major component of a software product that can be installed separately. (2) In VisualAge for Java, a method, field, or event that is available from a bean’s interface and to which other beans can connect.

**field.** A data object in a class. For example, a customer class could have a name field and an address field. A field can itself be an object with its own behavior and fields. By default, a field, in contrast to a property, does not support event notification.

**free-form surface.** The large open area of the Visual Composition Editor where you can work with visual and nonvisual beans. You add, remove, and connect beans on the free-form surface.

**framework.** A set of cooperative classes with strong connections that provide a template for development.

**garbage collection.** A Smalltalk process for periodically identifying unreferenced objects and deallocating their memory.

**gateway.** A host computer that connects networks that communicate in different languages. For example, a gateway connects a company’s LAN to the Internet.

**graphical user interface (GUI).** A type of interface that enables users to communicate with a program by manipulating graphical features, rather than by entering commands. Typically, a graphical user interface includes a combination of graphics, pointing devices, menu bars and other menus, overlapping windows, and icons.

**H**

**hypertext.** Text in a document that contains a hidden link to other text. You can click a mouse on a hypertext word and it will take you to the text designated in the link. Hypertext is used in Windows help programs and CD encyclopedias to jump to related references elsewhere within the same document. Hypertext can link—using HTTP over the Web—to any Web document in the world, with only a single mouse click.

**Hypertext Markup Language (HTML).** The basic language that is used to build hypertext documents on the World Wide Web. It is used in basic, plain ASCII-text documents, but when those documents are interpreted (rendered) by a Web browser such as Netscape, the document can display formatted text, color, a variety of fonts, graphics images, special effects, hypertext jumps to other Internet locations, and information forms.

**Hypertext Transfer Protocol (HTTP).** The protocol for moving hypertext files across the Internet. Requires an HTTP client program on one end, and an HTTP server program on the other end.

**inheritance.** (1) A mechanism by which an object class can use the attributes, relationships, and methods defined in more abstract classes related to it (its base classes). (2) An object-oriented programming technique that allows you to use existing classes as bases for creating other classes.

**instance.** Synonym for object, a particular instantiation of a data type.

**Integrated Development Environment (IDE).** In VisualAge for Java, the set of windows that provide the user with access to development tools. The primary windows are Workbench, Log, Console, Debugger, and Repository Explorer.
interchange file. A file that you can export from VisualAge for Java that contains information about selected projects or packages. This file can then be imported into any VisualAge for Java session.

interface. A set of methods that can be accessed by any class in the class hierarchy. The Interface page in the Workbench lists all interfaces in the workspace.

Internet. The vast collection of interconnected networks that use TCP/IP and evolved from the ARPANET of the late 1960s and early 1970s.

intranet. A private network, inside a company or organization, that uses the same kinds of software that you would find on the public Internet. Many of the tools used on the Internet are being used in private networks; for example, many companies have Web servers that are available only to employees.


IP number. An Internet address that is a unique number consisting of four parts separated by dots, sometimes called a dotted quad (for example: 198.204.112.1). Every Internet computer has an IP number, and most computers also have one or more domain names that are plain language substitutes for the dotted quad.

Java. A programming language invented by Sun Microsystems that is specifically designed for writing programs that can be safely downloaded to your computer through the Internet and immediately run without fear of viruses or other harm to your computer or files. Using small Java programs (called applets), Web pages can include functions such as animation, calculators, and other fancy tricks. We can expect to see a huge variety of features added to the Web through Java, because you can write a Java program to do almost anything a regular computer program can do and then include that Java program in a Web page.

Java archive (JAR). A platform-independent file format that groups many files into one. JAR files are used for compression, reduced download time, and security. Because the JAR format is written in Java, JAR files are fully extensible.

JavaBeans. In JDK 1.1, the specification that defines the platform-neutral component model used to represent parts. Instances of JavaBeans (often called beans) may have methods, properties, and events.

Java Database Connectivity (JDBC). In JDK 1.1, the specification that defines an API that enables programs to access databases that comply with this standard.

Java Native Interface (JNI). In JDK 1.1, the specification that defines a standard naming and calling convention so that the Java virtual machine can locate and invoke methods written in a language different from Java. See also native method.

K

keyword. A predefined word, reserved for Java, that cannot be used as an identifier.

L

legacy code. Existing code that a user might have. Legacy applications often have character-based, nongraphical user interfaces. Usually they are written in a non-object-oriented language, such as C or COBOL.

listener. In JDK 1.1, a class that receives and handles events.

local area network (LAN). A computer network located on a user’s establishment within a limited geographical area. A LAN typically consists of one or more server machines providing services to a number of client workstations.

log. In VisualAge for Java, the window that displays messages and warnings during development.
mapping. See schema mapping.

member. (1) A data object in a structure or a union. (2) In Java, classes and structures can also contain functions and types as members.

method. A fragment of Java code within a class that can be invoked and passed a set of parameters to perform a specific task.

method call. A communication from one object to another that requests the receiving object to execute a method. A method call consists of a method name that indicates the requested method and the arguments to be used in executing the method. The method call always returns some object to the requesting object as the result of performing the method. Synonym for message.

message. A request from one object that the receiving object implement a method. Because data is encapsulated and not directly accessible, a message is the only way to send data from one object to another. Each message specifies the name of the receiving object, the method to be implemented, and any arguments the method needs for implementation. Synonym for method call.

model. A nonvisual bean that represents the state and behavior of an object, such as a customer or an account. Contrast with view.

native method. Method written in a language other than Java that can be called by a Java object through the JNI specification.

named package. In the VisualAge for Java IDE, a package that has been explicitly named and created.

nonvisual bean. In the Visual Composition Editor, a bean that has no visual representation at run time. A nonvisual bean typically represents some real-world object that exists in the business environment. Compare with model. Contrast with view and visual bean.

notification framework. In JDK 1.1, a set of classes that implement the notifier/listener protocol. The notification framework is the base of the construction from beans technology (Visual Composition Editor).

object. (1) A computer representation of something that a user can work with to perform a task. An object can appear as text or an icon. (2) A collection of data and methods that operate on that data, which together represent a logical entity in the system. In object-oriented programming, objects are grouped into classes that share common data definitions and methods. Each object in the class is said to be an instance of the class. (3) An instance of an object class consisting of attributes, a data structure, and operational methods. It can represent a person, place, thing, event, or concept. Each instance has the same properties, attributes, and methods as other instances of the object class, although it has unique values assigned to its attributes.

object class. A template for defining the attributes and methods of an object. An object class can contain other object classes. An individual representation of an object class is called an object.

object factory. A nonvisual bean capable of dynamically creating new instances of a specified bean. For example, during the execution of an application, an object factory can create instances of a new class to collect the data being generated.

object-oriented programming (OOP). A programming approach based on the concepts of data abstraction and inheritance. Unlike procedural programming techniques, object-oriented programming concentrates on those data objects that constitute the problem and how they are manipulated, not on how something is accomplished.

Object Request Broker (ORB). A CORBA term designating the means by which objects transparently make requests and receive
responses from objects, whether they are local or remote.

**ODBC driver.** A DLL that implements ODBC function calls and interacts with a data source.

**Open Database Connectivity (ODBC).** A Microsoft developed C database API that allows access to database management systems calling callable SQL, which does not require the use of an SQL preprocessor. In addition, ODBC provides an architecture that allows users to add modules (database drivers) that link the application to their choice of database management systems at run time. Applications no longer need to be directly linked to the modules of all the database management systems that are supported.

**open edition.** An edition of a program element that can still be modified; that is, the edition has not been versioned. An open edition may reside in the workspace as well as in the repository.

**operation.** A method or service that can be requested of an object.

**overloading.** An object-oriented programming technique that allows redefinition of methods when the methods are used with class types.

**package.** A program element that contains related classes and interfaces.

**palette.** See beans palette.

**parameter connection.** A connection that satisfies a parameter of an action or method by supplying either a property’s value or the return value of an action, method, or script. The parameter is always the source of the connection. See also connection.

**parent class.** The class from which another bean or class inherits data, methods, or both.

**part.** An existing, reusable software component. In VisualAge for Java, all parts created with the Visual Composition Editor conform to the JavaBeans component model and are referred to as beans. See also nonvisual bean and visual bean. Compare with Class Editor and Composition Editor.

**primitive bean.** A basic building block of other beans. A primitive bean can be relatively complex in terms of the function it provides.

**private.** In Java, an access modifier associated with a class member. It allows only the class itself to access the member.

**process.** A collection of code, data, and other system resources, including at least one thread of execution, that performs a data processing task.

**program.** In VisualAge for Java, a term that refers to both Java applets and applications.

**project.** In VisualAge for Java, the topmost kind of program element. A project contains Java packages.

**promote features.** Make features of a subbean available to be used for making connections. This applies to subbeans that are to be included in other beans, for example, a subbean consisting of three push buttons on a panel. If this sample subbean is placed in a frame, the features of the push buttons would have to be promoted to make them available from within the frame.

**property.** An initial setting or characteristic of a bean; for example, a name, font, text, or positional characteristic.

**property sheet.** In the Visual Composition Editor, a set of name-value pairs that specify the initial appearance and other bean characteristics. A bean’s property sheet can be viewed from the Properties secondary window.

**property-to-method connection.** A connection that calls a method whenever a property’s value changes. It is similar to an event-to-method connection because the property’s event ID is used to notify the method when the value of the property changes. See also connection.

**property-to-property connection.** A connection from a property of one bean to a property of another bean. When one property is updated, the other property is updated automatically. See also connection.
**property-to-method connection.** A connection from a property of a bean to a method. When the property undergoes a state change, the method is called. See also connection.

**protected.** In Java, an access modifier associated with a class member. It allows the class itself, subclasses, and all classes in the same package to access the member.

**protocol.** (1) The set of all messages to which an object will respond. (2) Specification of the structure and meaning (the semantics) of messages that are exchanged between a client and a server. (3) Computer rules that provide uniform specifications so that computer hardware and operating systems can communicate. It is similar to the way that mail, in countries around the world, is addressed in the same basic format so that postal workers know where to find the recipient’s address, the sender’s return address, and the postage stamp. Regardless of the underlying language, the basic protocols remain the same.

**prototype.** A method declaration or definition that includes both the return type of the method and the types of its arguments.

**proxy bean.** A group of client-side and server-side objects that represent a remote server bean. The top-level class that implements the proxy bean is the client-side server proxy. See also client-side server proxy and server-side server proxy.

**R**

**Remote Method Invocation (RMI).** In JDK 1.1, the API that enables you to write distributed Java programs, allowing methods of remote Java objects to be accessed from other Java virtual machines.

**remote object instance manager.** Creates and manages instances of RMI server beans through their associated server-side server proxies.

**repository.** In VisualAge for Java, the storage area, separate from the workspace, that contains all editions (both open and versioned) of all program elements that have ever been in the workspace, including the current editions that are in the workspace. You can add editions of program elements to the workspace from the repository.

**Repository Explorer.** In VisualAge for Java, the window from which you can view and compare editions of program elements that are in the repository.

**resource file.** A noncode file that can be referred to from your Java program in VisualAge for Java. Examples include graphics and audio files.

**RMI Access Builder.** A VisualAge for Java Enterprise tool that generates proxy beans and associated classes and interfaces so you can distribute code for remote access, enabling Java-to-Java solutions.

**RMI compiler.** The compiler that generates stub and skeleton files that facilitate RMI communication. This compiler can be automatically invoked by the RMI Access Builder or from the Tools menu item.

**RMI registry.** A server program that allows remote clients to get a reference to a server bean.

**roll back.** The process of restoring data changed by SQL statements to the state at its last commit point.

**S**

**schema.** In the Data Access Builder, the representation of the database that will be mapped.

**schema mapping.** In the Data Access Builder, a set of definitions for all attributes matching all columns for your database table, view, or SQL statement. The mapping contains the information required by the Data Access Builder to generate Java classes.

**Scrapbook.** In VisualAge for Java, the window from which you can write and test fragments of code, without having to define an encompassing class or method.

**server.** A computer that provides services to multiple users or workstations in a network; for
example, a file server, a print server, or a mail server.

**server bean.** The bean that is distributed using RMI services and deployed on a server.

**server-side server proxy.** Generated by the RMI Access Builder, a companion class to the client-side server proxy, facilitating client-side server proxy communication over RMI. See also **client-side server proxy** and proxy bean.

**service.** A specific behavior that an object is responsible for exhibiting.

**single-byte character set.** A set of characters in which each character is represented by a 1-byte code.

**SmartGuide.** In IBM software products, an interface that guides you through performing common tasks.

**SQL predicate.** The conditional part of an SQL statement.

**sticky.** In the Visual Composition Editor, the mode that enables an application developer to add multiple beans of the same class (for example, three push buttons) without going back and forth between the beans palette and the free-form surface.

**stored procedure.** A procedure that is part of a relational database. The Data Access Builder can generate Java code that accesses stored procedures.

**superclass.** See **abstract class** and **base class**.

**T**

**Transmission Control Protocol/Internet Protocol (TCP/IP).** The basic programming foundation that carries computer messages around the globe through the Internet. The suite of protocols that defines the Internet. Originally designed for the UNIX operating system, TCP/IP software is now available for every major kind of computer operating system. To be truly on the Internet, your computer must have TCP/IP software.

**tear-off property.** A property that a developer has exposed to work with as though it were a stand-alone bean.

**thread.** A unit of execution within a process.

**tool bar.** The strip of icons along the top of the free-form surface. The tool bar contains tools to help an application developer construct composite beans.

**transaction.** In a CICS program, an event that queries or modifies a database that resides on a CICS server.

**type.** In VisualAge for Java, a generic term for a class or interface.

**U**

**Unicode.** A character coding system designed to support the interchange, processing, and display of the written texts of the diverse languages of the modern world. Unicode characters are normally encoded using 16-bit integral unsigned numbers.

**uniform resource locator (URL).** A standard identifier for a resource on the World Wide Web, used by Web browsers to initiate a connection. The URL includes the communications protocol to use, the name of the server, and path information identifying the objects to be retrieved on the server. A URL looks like this:

http://www.matisse.net/seminars.html
or telnet://well.sf.ca.us.br
or news:new.newusers.question.br

**user interface (UI).** (1) The hardware, software, or both that enables a user to interact with a computer. (2) The visual presentation and its underlying software with which a user interacts.

**V**

**variable.** (1) A storage place within an object for a data feature. The data feature is an object, such as number or date, stored as an attribute of the containing object. (2) A bean that receives an
identity at run time. A variable by itself contains no data or program logic; it must be connected such that it receives run-time identity from a bean elsewhere in the application.

**versioned edition.** An edition that has been versioned and can no longer be modified.

**versioning.** The act of making an open edition a versioned edition; that is, making the edition read-only.

**view.** (1) A visual bean, such as a window, push button, or entry field. (2) A visual representation that can display and change the underlying model objects of an application. Views are both the end result of developing an application and the basic unit of composition of user interfaces. Compare with visual bean. Contrast with model.

**visual bean.** In the Visual Composition Editor, a bean that is visible to the end user in the graphical user interface. Compare with view. Contrast with nonvisual bean.

**visual programming tool.** A tool that provides a means for specifying programs graphically. Application programmers write applications by manipulating graphical representations of components.

**Visual Composition Editor.** In VisualAge for Java, the tool where you can create graphical user interfaces from prefabricated beans and define relationships (connections) between both visual and nonvisual beans. The Visual Composition Editor is a page in the class browser.

W

**Workbench.** In VisualAge for Java, the main window from which you can manage the workspace, create and modify code, and open browsers and other tools.

**workspace.** The work area that contains all the code you are currently working on (that is, current editions). The workspace also contains the standard Java class libraries and other class libraries.
<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ANSI</td>
<td>American National Standards Institute</td>
</tr>
<tr>
<td>API</td>
<td>application programming interface</td>
</tr>
<tr>
<td>ATM</td>
<td>automated teller machine</td>
</tr>
<tr>
<td>AWT</td>
<td>Abstract Windowing Toolkit</td>
</tr>
<tr>
<td>CAE</td>
<td>Client Access Enabler</td>
</tr>
<tr>
<td>CB</td>
<td>Component Broker</td>
</tr>
<tr>
<td>CICS</td>
<td>Customer Information Control System</td>
</tr>
<tr>
<td>CLI</td>
<td>call level interface</td>
</tr>
<tr>
<td>COM</td>
<td>Component Object Model</td>
</tr>
<tr>
<td>CORBA</td>
<td>Common Object Request Broker Architecture</td>
</tr>
<tr>
<td>COS</td>
<td>CORBA object services</td>
</tr>
<tr>
<td>DB2</td>
<td>DATABASE 2</td>
</tr>
<tr>
<td>DBCS</td>
<td>double-byte character set</td>
</tr>
<tr>
<td>DBMS</td>
<td>database management system</td>
</tr>
<tr>
<td>DL/I</td>
<td>Data Language/I</td>
</tr>
<tr>
<td>DLL</td>
<td>dynamic link library</td>
</tr>
<tr>
<td>DNS</td>
<td>domain name server</td>
</tr>
<tr>
<td>DRDA</td>
<td>Distributed Relational Database Architecture</td>
</tr>
<tr>
<td>ECD</td>
<td>edit-compile-debug</td>
</tr>
<tr>
<td>ECI</td>
<td>external call interface</td>
</tr>
<tr>
<td>FTP</td>
<td>File Transfer Protocol</td>
</tr>
<tr>
<td>GUI</td>
<td>graphical user interface</td>
</tr>
<tr>
<td>HTML</td>
<td>Hypertext Markup Language</td>
</tr>
<tr>
<td>HTTP</td>
<td>Hypertext Transfer Protocol</td>
</tr>
<tr>
<td>IBM</td>
<td>International Business Machines Corporation</td>
</tr>
<tr>
<td>IDE</td>
<td>integrated development environment</td>
</tr>
<tr>
<td>IDL</td>
<td>interface definition language</td>
</tr>
<tr>
<td>IIOP</td>
<td>Internet Inter-ORB Protocol</td>
</tr>
<tr>
<td>IMS</td>
<td>Information Management System</td>
</tr>
<tr>
<td>IOR</td>
<td>interoperable object reference</td>
</tr>
<tr>
<td>ITSO</td>
<td>International Technical Support Organization</td>
</tr>
<tr>
<td>JAR</td>
<td>Java archive</td>
</tr>
<tr>
<td>JDBC</td>
<td>Java Database Connectivity</td>
</tr>
<tr>
<td>JDK</td>
<td>Java Developer's Kit</td>
</tr>
<tr>
<td>JNI</td>
<td>Java Native Interface</td>
</tr>
<tr>
<td>JVM</td>
<td>Java Virtual Machine</td>
</tr>
<tr>
<td>LAN</td>
<td>local area network</td>
</tr>
<tr>
<td>MOFW</td>
<td>managed object framework</td>
</tr>
<tr>
<td>MVS</td>
<td>Multiple Virtual Storage</td>
</tr>
<tr>
<td>NLS</td>
<td>National Language Support</td>
</tr>
<tr>
<td>NT</td>
<td>new technology</td>
</tr>
<tr>
<td>ODBC</td>
<td>Open Database Connectivity</td>
</tr>
<tr>
<td>OMG</td>
<td>Object Management Group</td>
</tr>
<tr>
<td>OMT</td>
<td>object modeling technique</td>
</tr>
<tr>
<td>OO</td>
<td>object-oriented</td>
</tr>
<tr>
<td>OOA</td>
<td>object-oriented analysis</td>
</tr>
<tr>
<td>OOD</td>
<td>object-oriented design</td>
</tr>
<tr>
<td>ORB</td>
<td>Object Request Broker</td>
</tr>
<tr>
<td>OS/2</td>
<td>Operating System/2</td>
</tr>
<tr>
<td>OTS</td>
<td>object transaction service</td>
</tr>
<tr>
<td>PIN</td>
<td>personal identification number</td>
</tr>
<tr>
<td>RAD</td>
<td>rapid application development</td>
</tr>
<tr>
<td>RDBMS</td>
<td>relational database management system</td>
</tr>
<tr>
<td>RMI</td>
<td>Remote Method Invocation</td>
</tr>
<tr>
<td>SBCS</td>
<td>single-byte character set</td>
</tr>
<tr>
<td>SDK</td>
<td>Software Developer's Kit</td>
</tr>
<tr>
<td>SQL</td>
<td>structured query language</td>
</tr>
<tr>
<td>TCP/IP</td>
<td>Transmission Control Protocol/Internet Protocol</td>
</tr>
<tr>
<td>TP</td>
<td>transaction processing</td>
</tr>
<tr>
<td>UOW</td>
<td>unit of work</td>
</tr>
<tr>
<td>URL</td>
<td>uniform resource locator</td>
</tr>
<tr>
<td>WWW</td>
<td>World Wide Web</td>
</tr>
</tbody>
</table>
Using VisualAge for Java Enterprise Version 2 to Develop CORBA and EJB Applications
Index

A
agent password
   debugging 156
authenticator
   Component Broker implementation 174
   AuthenticatorImpl 192

B
BootstrapHost
   Component Broker 174
BootstrapPort 174

C
classes
   deployment 191
   implementation 192
client
   debugger 150
   EJB 192
COM+
   DCOM extension 14
Component Broker
   adding EmployeeID object 6
   and Tivoli 53
   and Transaction Series 52
   Application Adaptor 48
   BootstrapHost 174
   Business objects 48
   containers 49
   CORBA environment 4
   data objects 48
   development 51
   EJB container 192
   Homes 49
   Java ORB 5
   language interoperability 51
   managed objects 48
   object server runtime 50
Object Services 49
   compound name 175
   Concurrency Control 50
   Event Service 50
   Externalization Control 50
   LifeCycle Service 49
   Naming Service 49
   Naming Service extension 175
   Query Service 50
   resolve_with_string 175
   Security Service 49
   Transaction Service 50
PrimaryKey 175
   runtime 47
   systems management 51
   understanding 46
Workload Management 50
container
tools 6
CORBA
   application objects 18
   bootstrapping 94
   client 93
   Common Facilities 18
   delegation model 85
   goal 5
   inheritance model 85
   IOR 87
   object services
      Collection 21
      components 18
      Concurrency Control 20
      Event 19
      Externalization Control 20
      Licensing 20
      LifeCycle Control 19
      Naming 19
      Persistence 19
      Properties 20
      Query 20
      Relationship 20
      Security 21
      Time 20
      Trader 21
      Transaction 20
      TEA objects 4
      versus COM+ 14

D
collection
   technology 12
DCOM 14
debug
daemon 151
Using VisualAge for Java Enterprise Version 2 to Develop CORBA and EJB Applications

EJB
VisualAge for Java
debugger
client
debugging
activity.log
agent password
allowing
and Object Builder
breakpoint outside the workspace
breakpoints
client
Component Broker showlog
Component Broker client
convert information
debug enabled
-debug flag
different machine
distributed applications
environment variables
exception thrown
external tools
flag
GIOP-IIOP tracing
heterogeneous environment
host
include information for compiler
information
invoking the daemon
IVB_TRACE_DEBUG
javac
jdebug.jar
ORB
password
port
remote
remote ORB
remote servers
run control
run JVM
running bytecode remotely
selecting servers
sequence of loaded classes
SmartGuide Export to a jar
specific exception
stand-alone application
step by step
User Interface Debugger

VisualAge for Java daemon
deployment
classes
descriptor
role
descriptor
control
deployment
develop
EJB
development
object-oriented application
Distributed Component Object Model
see DCOM
Distributed Computing Environment
see DCE
distributed technologies
objects
review

E
EJB
announcement
assembler
bean-managed persistence
client
components
container
container-managed
table
entity beans
persistence
CORBA and Java
deployer
Deployment Descriptor
entity beans
export JAR
importing
installation
list of servers
location service daemon
Persistent Name Server
PrimaryKey
provider
required methods
server provider
session beans
system administrator

250
<table>
<thead>
<tr>
<th>T</th>
<th>TEA and Component Broker 5</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>application overview 57</td>
</tr>
<tr>
<td></td>
<td>use case 58</td>
</tr>
<tr>
<td>test 7</td>
<td>tracing</td>
</tr>
<tr>
<td></td>
<td>Component Broker business objects 153</td>
</tr>
<tr>
<td></td>
<td>transaction</td>
</tr>
<tr>
<td></td>
<td>behavior 6</td>
</tr>
<tr>
<td></td>
<td>exception 174</td>
</tr>
<tr>
<td></td>
<td>started by the client 174</td>
</tr>
<tr>
<td></td>
<td>started by the server 174</td>
</tr>
<tr>
<td>Travel Expense Application</td>
<td>see TEA</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>V</th>
<th>versioning</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>EJB 7</td>
</tr>
</tbody>
</table>

| Visual Composition Editor | palette 238 |
| VisualAge for Java | Access Builders 41 |
|                       | ActiveX 42 |
|                       | configuration 64 |
|                       | Debugging 38 |
|                       | Enterprise Access Builder 42 |
|                       | Enterprise toolkits 41 |
|                       | IDE 35 |
|                       | IDL Development Environment 4 |
|                       | importing WebSphere classes 66 |
|                       | Lotus eSuite 42 |
|                       | Persistence Builder tool 41 |
|                       | programmer productivity 42 |
|                       | Servlet Builder 5, 42 |
|                       | setting a Web server 68 |
|                       | team programming 41 |
|                       | Visual Composition Editor 40 |
|                       | visual programming 39 |

<table>
<thead>
<tr>
<th>W</th>
<th>WebSphere</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>AFS 45</td>
</tr>
<tr>
<td></td>
<td>and Component Broker 4</td>
</tr>
<tr>
<td></td>
<td>configuration 64</td>
</tr>
<tr>
<td></td>
<td>CORBA application 75</td>
</tr>
<tr>
<td></td>
<td>CORBA objects 6</td>
</tr>
<tr>
<td></td>
<td>Developer Studio 46</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>N</th>
<th>Name binding</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>JNDI 196</td>
</tr>
<tr>
<td></td>
<td>naming tree 175</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>O</th>
<th>Object Request Broker</th>
</tr>
</thead>
<tbody>
<tr>
<td>see ORB</td>
<td></td>
</tr>
</tbody>
</table>

| ORB | compliance 19  |
|     | init 174       |
|     | portability 25 |
| WebSphere initialization 77 |

| P | persistence 6  |
|   | Persistent Name Server 194 |

| Q | quality of service 3 |

| R | RAD 5  |
|   | Rapid Application Development |
| see RAD |
| remote debugger 5 |
| remote interfaces |
| generation 192 |
| Remote Method Invocation |
| see RMI |
| resolve_with_string 175 |
| RMI distributed objects 13 |

| S | security 6   |
|   | servlets    |
|   | servlet builder 5 |
|   | skeletons 4  |
|   | stubs 4      |

252 Using VisualAge for Java Enterprise Version 2 to Develop CORBA and EJB Applications
EJB support  46
eNetwork Dispatcher  45
JSP  43
packaged with VisualAge for Java  184
Performance Pack  45
Servlet Runtime Environment  43
servlet.properties file  79
setting classpath  80
Version 2.0  6
wrapping  11
ITSO Redbook Evaluation

Using VisualAge for Java Enterprise Version 2 to Develop CORBA and EJB Applications
SG24-5276-00

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