BEA WebLogic Enterprise

Creating Client Applications

WebLogic Enterprise 5.0
Document Edition 5.0
December 1999

Creating Client Applications

Document Edition	Date	Software Version
5.0	December 1999	BEA WebLogic Enterprise 5.0

Contents

About This Document

2. Creating CORBA Client Applications

3. Creating ActiveX Client Applications

4. Using Security

5. Using Transactions

6. Using the Dynamic Invocation Interface

7. Handling Exceptions

Index

About This Document

This document describes how to create CORBA C++, CORBA Java, and ActiveX client applications for the BEA WebLogic (WLE) software. This document introduces important product concepts, provides step-by-step instructions for creating client applications, and includes code examples to illustrate the development process.

This document covers the following topics:

- Chapter 1, "Client Application Development Concepts," introduces the concepts you need to know to develop client applications for the WLE software.
- Chapter 2, "Creating CORBA Client Applications," provides instructions for creating CORBA C++ and CORBA Java client applications.
- Chapter 3, "Creating ActiveX Client Applications," provides instructions for creating ActiveX client applications.
- Chapter 4, "Using Security," describes using security in CORBA C++, CORBA Java, and ActiveX client applications.
- Chapter 5, "Using Transactions," describes using transactions in CORBA C+, CORBA Java, and ActiveX client applications.
- Chapter 6, "Using the Dynamic Invocation Interface," explains how to use the Dynamic Invocation Interface (DII) from CORBA C++ and CORBA Java client applications.
- Chapter 7, "Handling Exceptions," explains how CORBA C++, CORBA Java, and ActiveX client applications handle CORBA exceptions.

What You Need to Know

This document is intended for programmers who want to develop client applications for the WLE software.

e-docs Web Site

The BEA WebLogic Enterprise product documentation is available on the BEA corporate Web site. From the BEA Home page, click the Product Documentation button or go directly to the “e-docs” Product Documentation page at http://e-docs.beasys.com.

How to Print the Document

You can print a copy of this document from a Web browser, one file at a time, by using the File—>Print option on your Web browser.

A PDF version of this document is available on the WebLogic Enterprise documentation Home page on the e-docs Web site (and also on the documentation CD). You can open the PDF in Adobe Acrobat Reader and print the entire document (or a portion of it) in book format. To access the PDFs, open the WebLogic Enterprise documentation Home page, click the PDF Files button, and select the document you want to print.

If you do not have the Adobe Acrobat Reader, you can get it for free from the Adobe Web site at http://www.adobe.com/.

Related Information

For more information about CORBA, Java 2 Enterprise Edition (J2EE), BEA TUXEDO, distributed object computing, transaction processing, C++ programming, and Java programming, see the WLE Bibliography in the WebLogic Enterprise online documentation.

Contact Us!

Your feedback on the BEA WebLogic Enterprise documentation is important to us. Send us e-mail at **docsupport@beasys.com** if you have questions or comments. Your comments will be reviewed directly by the BEA professionals who create and update the WebLogic Enterprise documentation.

In your e-mail message, please indicate that you are using the documentation for the BEA WebLogic Enterprise 5.0 release.

If you have any questions about this version of BEA WebLogic Enterprise, or if you have problems installing and running BEA WebLogic Enterprise, contact BEA Customer Support through BEA WebSupport at www.beasys.com. You can also contact Customer Support by using the contact information provided on the Customer Support Card, which is included in the product package.

When contacting Customer Support, be prepared to provide the following information:

- Your name, e-mail address, phone number, and fax number
- Your company name and company address
- Your machine type and authorization codes
- The name and version of the product you are using
- A description of the problem and the content of pertinent error messages

Documentation Conventions

The following documentation conventions are used throughout this document.

Convention	Item
boldface text	Indicates terms defined in the glossary.
Ctrl+Tab	Indicates that you must press two or more keys simultaneously.
italics	Indicates emphasis or book titles.

`monospace text`	Indicates code samples, commands and their options, data structures and their members, data types, directories, and file names and their extensions. Monospace text also indicates text that you must enter from the keyboard.
	Examples:
	`#include <iostream.h> void main ( ) the pointer psz`
	`chmod u+w *`
	`\tux\data\ap`
	`.doc`
	`tux.doc`
	`BITMAP`
	`float`

`monospace boldface text`	Identifies significant words in code. *Example*: `void` **`commit`** `( )`
`monospace italic text`	Identifies variables in code. *Example*: `String` *`expr`*
UPPERCASE TEXT	Indicates device names, environment variables, and logical operators. *Examples*: LPT1 SIGNON OR

<table>
<tr><th>Convention</th><th>Item</th></tr>
<tr><td><code>{ }</code></td><td>Indicates a set of choices in a syntax line. The braces themselves should never be typed.</td></tr>
<tr><td><code>[ ]</code></td><td>Indicates optional items in a syntax line. The brackets themselves should never be typed.
Example:
<code>buildobjclient [-v] [-o name] [-f file-list]...
[-l file-list]...</code></td></tr>
<tr><td><code>|</code></td><td>Separates mutually exclusive choices in a syntax line. The symbol itself should never be typed.</td></tr>
<tr><td><code>...</code></td><td>Indicates one of the following in a command line:
■ That an argument can be repeated several times in a command line
■ That the statement omits additional optional arguments
■ That you can enter additional parameters, values, or other information
The ellipsis itself should never be typed.
Example:
<code>buildobjclient [-v] [-o name] [-f file-list]...
[-l file-list]...</code></td></tr>
<tr><td>.
.
.</td><td>Indicates the omission of items from a code example or from a syntax line. The vertical ellipsis itself should never be typed.</td></tr>
</table>

1 Client Application Development Concepts

This topic reviews the types of client applications supported by the **BEA WLE software** and introduces the following concepts that you need to understand before you develop client applications for the WLE software:

- OMG IDL
- Static and dynamic invocation
- Client stubs
- Interface Repository
- Domains
- Environmental objects
- ActiveX
- Views
- Bindings
- Naming conventions for ActiveX views

Overview of Client Applications

The WLE software supports the following types of client applications:

- **CORBA C++**

  This type of client application uses the C++ environmental objects to access the **CORBA object**s in an **WLE domain**, and the WLE **Object Request Broker (ORB)** to process requests to CORBA objects. Use the WLE development commands to build CORBA C++ client applications.

- **CORBA Java**

  This type of client application uses the Java environmental objects to access CORBA objects in an WLE domain. However, these client applications use an ORB product other than the WLE ORB to process requests to CORBA objects. CORBA Java client applications are built using the ORB product's Java development tools. The WLE software supports interoperability with the following:

  - Netscape Enterprise Server Version 3.0 and Javasoft JDK Version 1.1.5.
  - JDK Version 1.2

- **ActiveX**

  This type of client application uses the Automation environmental objects to access CORBA objects in an WLE domain, and the BEA ActiveX Client to process requests to CORBA objects. Use the Application Builder to select the CORBA interfaces that are available to ActiveX client applications, to create ActiveX views of the CORBA interfaces, and to create packages for deploying ActiveX views of CORBA interfaces to client machines. These client applications are built using an automation development tool such as Visual Basic or PowerBuilder.

OMG IDL

With any distributed application, the client/server application needs some basic information to communicate. For example, the client application needs to know which operations it can request, and the arguments to the operations.

You use the **Object Management Group (OMG) Interface Definition Language (IDL)** to describe available CORBA interfaces to client applications. An interface definition written in OMG IDL completely defines the CORBA interface and fully specifies each operation's arguments. OMG IDL is a purely declarative language. This means that it contains no implementation details. Operations specified in OMG IDL can be written in and invoked from any language that provides CORBA bindings. C++ and Java are two of the supported languages.

Generally, the application designer provides the OMG IDL files for the available CORBA interfaces and operations to the programmer who creates the client applications.

OMG IDL to C++ Mapping

The WLE software conforms to *The Common Object Request Broker:Architecture and Specification, Version 2.2*. For complete information about the OMG IDL to C++ mapping, see *The Common Object Request Broker:Architecture and Specification, Version 2.2,* Revised: February, 1998.

OMG IDL to Java Mapping

The WLE software conforms to *The Common Object Request Broker:Architecture and Specification, Version 2.2*. For complete information about the OMG IDL to Java mapping, see *The Common Object Request Broker:Architecture and Specification, Version 2.2*, Revised: February, 1998.

OMG IDL to COM Mapping

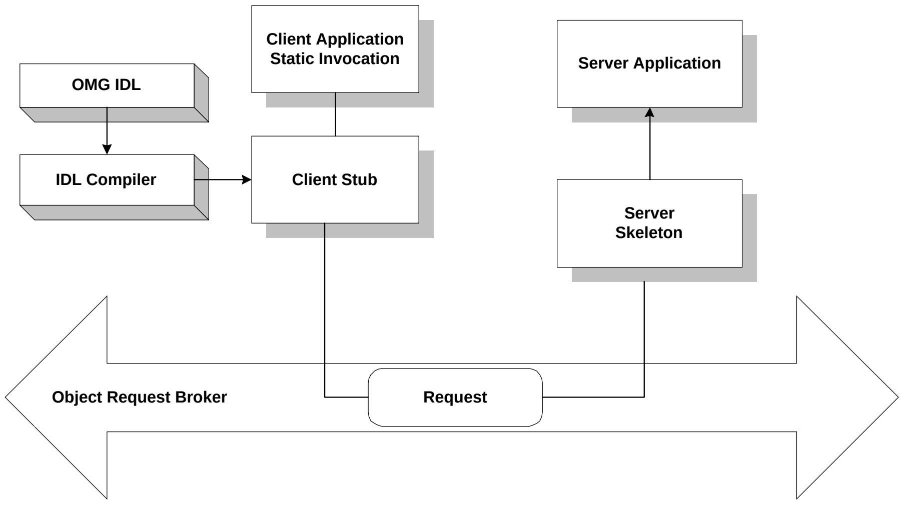
The WLE software conforms to the *OMG COM/CORBA Internetworking Specification, Version 1.1*. The mapping of COM data types is included in the *OMG COM/CORBA Internetworking Specification, Version 1.1*.

Static and Dynamic Invocation

The WLE Object Request Broker (ORB) supports two types of client/server invocations: static and dynamic. In both cases, the client application performs a request by gaining access to an object reference for a server application and invoking the operation that satisfies the request. The server application cannot tell the difference between static and dynamic invocations.

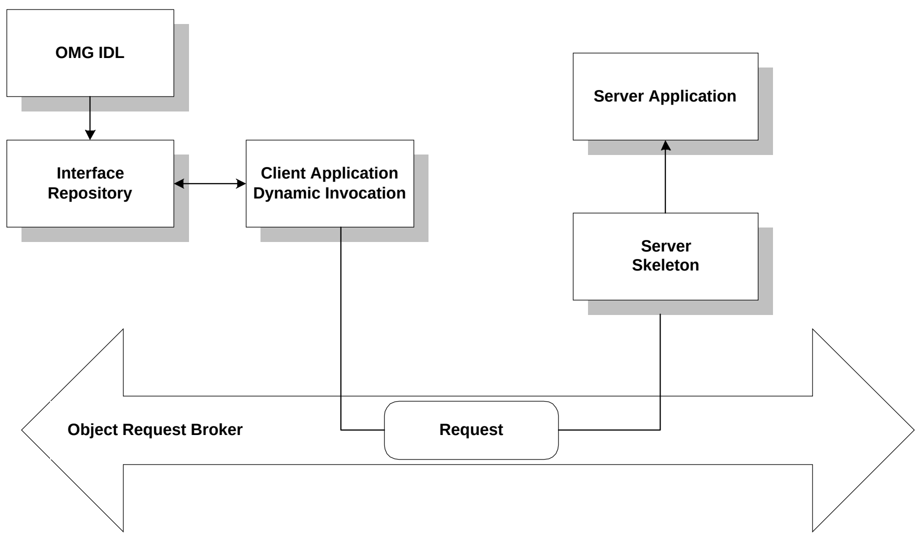
When using static invocation, the client application invokes operations directly on the **client stubs**. Static invocation is the easiest, most common type of invocation. The stubs are generated by the IDL compiler. Static invocation is recommended for applications that know at compile time the particulars of the operations they need to invoke and can process within the synchronous nature of the invocation. Figure 1-1 illustrates static invocation.

Figure 1-1 Static Invocation



While dynamic invocation is more complicated it enables your client application to invoke operations on any CORBA object without having to know the CORBA object's interfaces at compile time. Figure 1-2 illustrates dynamic invocation.

Figure 1-2 Dynamic Invocation



When using dynamic invocation, the client application can dynamically build operation requests for a CORBA object interface that has been stored in the Interface Repository. Server applications do not require any special design to be able to receive and handle dynamic invocation requests. Dynamic invocation is generally used when the client application requires deferred synchronous communication, or by dynamic client applications when the nature of the interaction is undefined. For more information about using dynamic invocation, see Chapter 6, “Using the Dynamic Invocation Interface.”

Client Stubs

Client stubs provide the programming interface to operations that a CORBA object can perform. A client stub is a local proxy for the CORBA object. Client stubs provide a mechanism for performing a synchronous invocation on an object reference for a CORBA object. The client application does not need special code to deal with the CORBA object or its arguments; the client application simply treats the stub as a local object.

A client application must have a stub for each interface it plans to use. You use the `idl` command (or your Java product's equivalent command) to generate a client stub from the OMG IDL definition of the CORBA interface. The command generates a stub file and a header file that describe everything that you need if you want to use the client stub from a programming language, such as C++ or Java. You simply invoke a **method** from within your client application to request an operation in the server application.

Interface Repository

The Interface Repository contains descriptions of a CORBA **object**'s interfaces and operations. The information stored in the Interface Repository is equivalent to the information defined in an OMG IDL file, but the information is accessible programmatically at run time. Client applications use the Interface Repository for the following reasons:

- CORBA client applications that use static invocation do not access the Interface Repository at run time. The information about the CORBA object's interfaces is included in the client stub.
- CORBA client applications that use dynamic invocation use the Interface Repository to learn about a CORBA object's interfaces, and to invoke operations on the object.
- ActiveX client applications are not aware that they are using the Interface Repository. The BEA ActiveX Client uses CORBA operations to obtain information about CORBA objects from the Interface Repository.

You use the following WLE development commands to manage the Interface Repository:

- The `idl2ir` command populates the Interface Repository with CORBA interfaces. This command creates an Interface Repository if an Interface Repository does not exist. Also use this command to update the CORBA interfaces in the Interface Repository.
- The `ir2idl` command creates an OMG IDL file from the contents of the Interface Repository.
- The `irdel` command deletes CORBA interfaces from the Interface Repository.

For a description of the development commands for the Interface Repository, see the *C++ Programming Reference.*

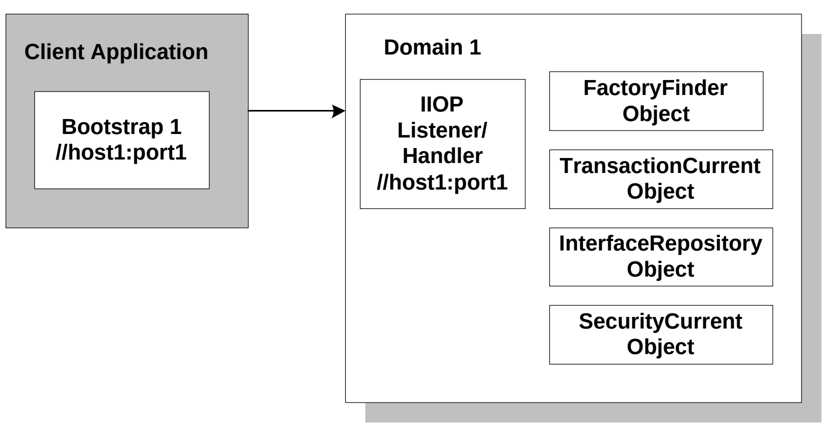
Domains

A domain is a way of grouping objects and services together as a management entity. An WLE domain has at least one **IIOP Listener/Handler** and is identified by a name. One client application can connect to multiple WLE domains using different Bootstrap objects. For each WLE domain, a client application can get a FactoryFinder object, an InterfaceRepository object, a SecurityCurrent object, and a TransactionCurrent object, which correspond to the services offered within the WLE domain. For a description of the Bootstrap object, the FactoryFinder object, the InterfaceRepository object, the SecurityCurrent object, and the TransactionCurrent object, see "Environmental Objects" in this topic.

Note: Only one TransactionCurrent object and one SecurityCurrent object can exist at the same time, and they must be associated with the same Bootstrap object.

Figure 1-3 illustrates how an WLE domain works.

Figure 1-3 How an WLE Domain Works



Environmental Objects

The WLE software provides a set of **environmental objects** that set up communication between client applications and server applications in a particular WLE domain. The WLE software provides the following environmental objects:

- **Bootstrap**

  This object establishes communication between a client application and an WLE domain. It also obtains **object references** for the other environmental objects in the WLE domain.

- **FactoryFinder**

  This CORBA object locates a **factory**, which in turn can create object references for CORBA objects.

- **SecurityCurrent**

  Thisobject can be used to log a client application into an WLE domain with the proper security. The WLE software provides an implementation of the **CORBAservices Security Service**.

- **TransactionCurrent**

  This object allows a client application to participate in a transaction. The WLE software provides an implementation of the **CORBAservices Object Transaction Service (OTS)**.

- **UserTransaction**

  This object allows a client application to participate in a transaction. The WLE software provides an implementation of the Sun Microsystems, Inc. Java Transaction Application Programming Interface (JTA API). This object is supported with Java client and server applications only.

- **InterfaceRepository**

  This CORBA object contains interface definitions for all the available CORBA interfaces and the factories used to create object references to the CORBA interfaces.

The WLE software provides environmental objects for the following programming environments:

- C++
- Java
- Automation

Bootstrap Object

The client application creates a Bootstrap object. A list of IIOP Listener/Handlers can be supplied either as a parameter or via the `TOBJADDR` environmental variable or Java property. A single IIOP Listener/Handler is specified as follows:

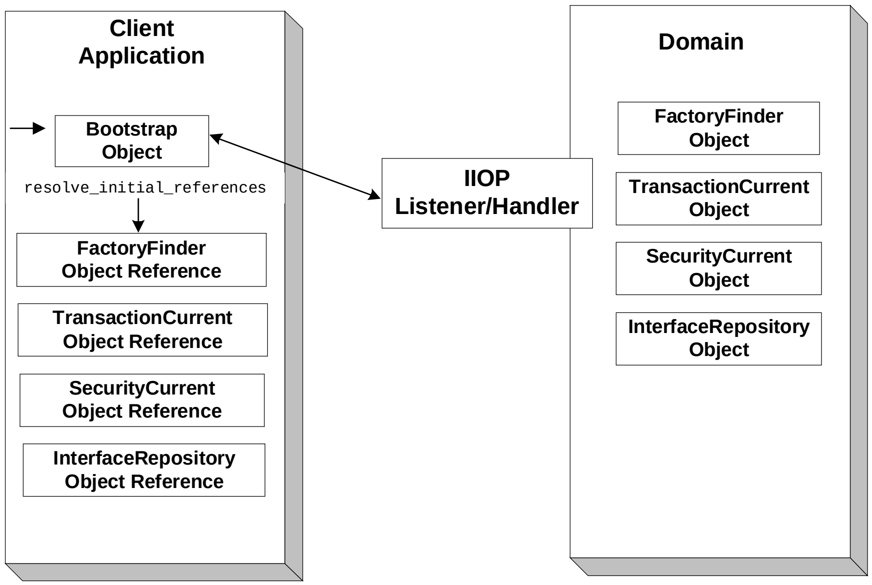
`//`*`host`*`:`*`port`*

For example, //`myserver:4000`

Once the Bootstrap object is instantiated, the `resolve_initial_references` method is invoked, passing in a string id, to obtain a reference to an available object. The valid values for the string id are FactoryFinder, TransactionCurrent, SecurityCurrent, and InterfaceRepository.

Figure 1-4 illustrates how the Bootstrap object works in an WLE domain.

Figure 1-4 How the Bootstrap Object Works



Factories and the FactoryFinder Object

Client applications get object references to CORBA objects from a factory. A factory is any CORBA object that returns an object reference to another CORBA object and registers itself with the FactoryFinder object.

To use a CORBA object, the client application must be able to locate the factory that creates an object reference for the CORBA object. The WLE software offers the FactoryFinder object for this purpose. The factories available to client applications are those that are registered with the FactoryFinder object by WLE server applications at startup.

The client application uses the following sequence of steps to obtain a reference to a CORBA object:

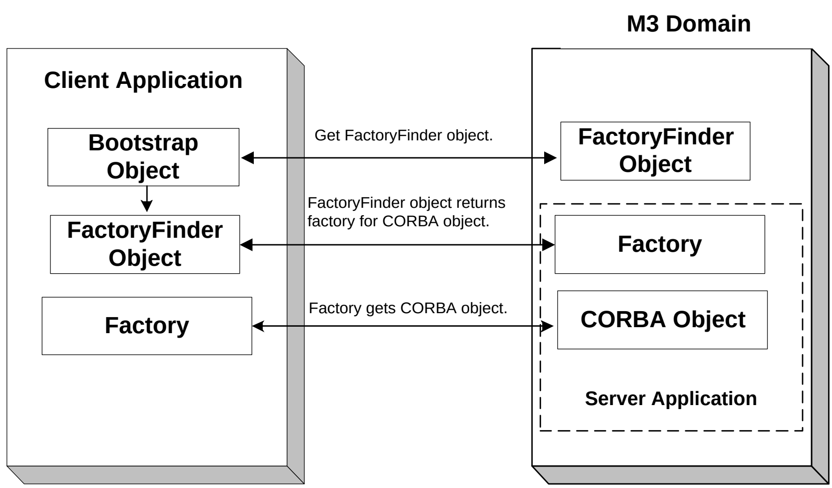
1. Once the Bootstrap object is created, the `resolve_initial_references` method is invoked to obtain the reference to the FactoryFinder object.

2. Client applications query the FactoryFinder object for object references to the desired factory.

3. Client applications call the factory to obtain an object reference to the CORBA object.

Figure 1-5 illustrates the client application interaction with the FactoryFinder object.

Figure 1-5 How Client Applications Use the FactoryFinder Object



Naming Conventions and WLE Extensions to the FactoryFinder Object

The factories available to client applications are those that are registered with the FactoryFinder object by the WLE server applications at startup. Factories are registered using a key consisting of the following fields:

- The Interface Repository Id of the factory’s interface
- An object reference to the factory

The FactoryFinder object used by the WLE software is defined in the CORBAservices LifeCycle Service. The WLE software implements extensions to the `COS::LifeCycle::FactoryFinder` interface that make it easier for client applications to locate a factory using the FactoryFinder object.

The CORBAservices Life Cycle Service specifies the use of names as defined in the CORBAservices Naming Service to locate factories with the `COS::LifeCycle::FactoryFinder` interface. These names consist of a sequence of *`NameComponent`* structures, which consist of `ID` and `kind` fields.

The use of CORBA names to locate factories is cumbersome for client applications; it involves many calls to build the appropriate name structures and assemble the Naming Service name that must be passed to the `find_factories` method of the `COS::LifeCycle::FactoryFinder` interface. Also, since the method can return more than one factory, client applications must manage the selection of an appropriate factory and the disposal of unwanted object references.

The FactoryFinder object is designed to make it easier for client applications to locate factories by extending the interface with simpler method calls.

The extensions are intended to provide the following simplifications for the client application:

- Let the client application locate factories by id, using a simple string parameter for the id field. This reduces the work needed by the client application to build name structures.
- Permit the FactoryFinder object to implement a load balancing scheme by choosing from a pool of available factories.
- Provide methods that return one object reference to a factory, instead of a sequence of object references. This eliminates the need for client applications to provide code to handle the selection of a single factory from a sequence, and then dispose of the unneeded references.

The most straightforward application design can be achieved by using the `Tobj::FactoryFinder::find_one_factory_by_id` method in client applications. This method accepts a simple string for factory id as input and returns one factory to the client application. The client application is freed from the necessity of manipulating name components and selecting among many factories.

To use the `Tobj::FactoryFinder::find_one_factory_by_id` method, the application designer must establish a naming convention for factories that client applications can use to easily locate factories for specific CORBA object interfaces.

Ideally, this convention should establish some mnemonic types for factories that supply object references for certain types of CORBA object interfaces. Factories are then registered using these conventions. For example, a factory that returns an object reference for Student objects might be called StudentFactory. For more information about registering factories with the FactoryFinder object, see *Creating C++ Server Applications* and *Creating Java Server Applications*.

It is recommended that you either use the actual interface ID of the factory in the OMG IDL file, or specify the the factory ID as a constant in the OMG IDL file. This technique ensures naming consistency between the client application and the server application.

SecurityCurrent Object

The SecurityCurrent object is an WLE implementation of the CORBAservices Security Service. The WLE security model is based on authentication. You use the SecurityCurrent object to specify the appropriate level of security. The following levels of authentication are provided:

- TOBJ_NOAUTH

  No authentication is needed; however, the client application may still authenticate itself, and may specify a user name and a client application name, but no password.

- TOBJ_SYSAUTH

  The client application must authenticate itself to the WLE domain and must specify a user name, client application name, and application password.

- TOBJ_APPAUTH

  In addition to the TOBJ_SYSAUTH information, the client application must provide application-specific information. If the default WLE authentication service is used in the application configuration, the client application must provide a user password; otherwise, the client application provides authentication data that is interpreted by the custom authentication service in the application.

Note: If a client application is not authenticated and the security level is `TOBJ_NOAUTH`, the IIOP Listener/Handler of the WLE domain registers the client application with the user name and client application name sent to the IIOP Listener/Handler.

In the WLE software, only the PrincipalAuthenticator and Credentials properties on the SecurityCurrent object are supported. For information about using the SecurityCurrent object in client applications, see Chapter 4, "Using Security." For a description of the `SecurityLevel1::Current` and `SecurityLevel2::Current` interfaces, refer to the *C++ Programming Reference* or the *WLE Javadoc*.

TransactionCurrent Object

The TransactionCurrent object is an WLE implementation of the CORBAservices Object Transaction Service. The TransactionCurrent object maintains a transactional context for the current session between the client application and the server application. Using the TransactionCurrent object, the client application can perform transactional operations, such as initiating and terminating a transaction and getting the status of a transaction.

Transactions are used on a per-interface basis. During design, the application designer decides which interfaces within an WLE application will handle transactions. A transaction policy for each interface is then defined in an Implementation Configuration File (ICF). The transaction policies are:

- Never

  The interface is not transactional. Objects created for this interface can never be involved in a transaction. The WLE software generates an exception (`INVALID_TRANSACTION`) if an interface with this policy is involved in a transaction.

- Optional

  The interface may be transactional. Objects can be involved in a transaction if the request is transactional.

- Always

  The interface must always be part of a transaction. If the interface is not part of a transaction, a transaction will be automatically started by the TP framework.

- Ignore

  The interface is not transactional. The interface can be included in a transaction, however, the AUTOTRAN policy specified for this interface in the `UBBCONFIG` file is ignored.

For information about using the TransactionCurrent object in client applications, see Chapter 5, “Using Transactions.” For a description of the TransactionCurrent object, see the *C++ Programming Reference* or the *WLE Javadoc*.

InterfaceRepository Object

The InterfaceRepository object returns information about the **Interface Repository** in a specific WLE domain. The InterfaceRepository object is based on the CORBA definition of an Interface Repository. It offers the proper set of CORBA interfaces as defined by the *Common Request Broker Architecture and Specification, Version 2.2*.

CORBA client applications that use the Dynamic Invocation Interface (DII) need to access the Interface Repository programmatically. The exact steps taken to access the Interface Repository depend on whether the client application is seeking information about a specific CORBA interface or browsing the Interface Repository to find an interface. In either case, the client application can only *read* to the Interface Repository, it cannot *write* to the Interface Repository.

Before a CORBA client application using DII can browse the Interface Repository in an WLE domain, the client application needs to obtain an object reference for the InterfaceRepository object in that domain. CORBA client applications using DII use the Bootstrap object to obtain the object reference.

ActiveX client applications are not aware they are using the Interface Repository object. Like CORBA client applications, ActiveX client applications use the Bootstrap object to obtain a reference to the Interface Repository object.

For information about using the Interface Repository object in CORBA client applications that use DII, see Chapter 6, “Using the Dynamic Invocation Interface.” For a description of the Interface Repository object, see the *C++ Programming Reference*.

Concepts for ActiveX Client Applications

The following sections describe concepts that are specific to ActiveX client applications.

What is ActiveX?

ActiveX is a set of technologies from Microsoft that enables software components to interact with one another in a networked environment, regardless of the language in which the components were created. ActiveX is built on the **Component Object Model (COM)** and integrates with **Object Linking and Embedding (OLE)**. OLE provides an architecture for document embedding. Automation is the part of COM that allows applications such as Visual Basic, Delphi, and PowerBuilder to manipulate Automation objects, ActiveX controls, and ActiveX documents.

The **BEA ActiveX Client** provides interoperability between the WLE and COM **object systems**. The ActiveX Client transforms the interfaces of CORBA objects in an WLE domain into methods on Automation objects.

Views and Bindings

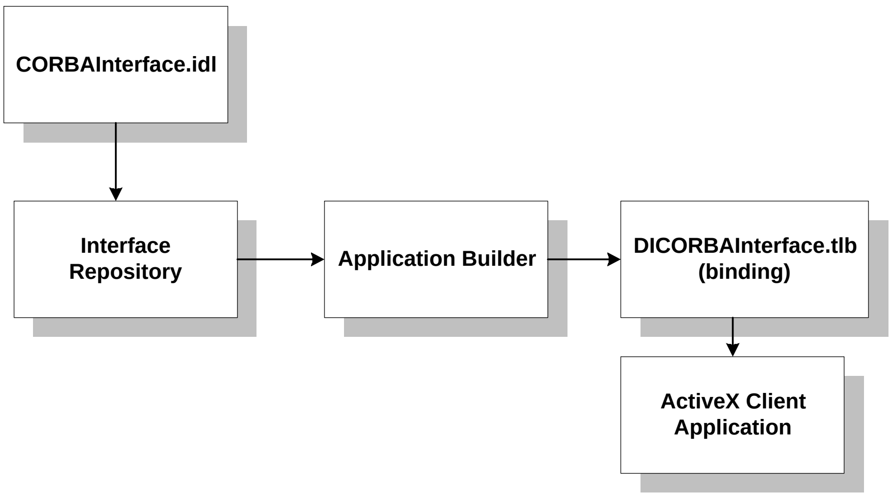
ActiveX client applications use views of CORBA interfaces. **Views** represent the CORBA interfaces in an WLE domain locally as Automation objects. To use an ActiveX view of a CORBA object (referred to as an ActiveX view), you need to create a **binding** for ActiveX. The binding describes the interface of a CORBA object to ActiveX. The interfaces of the CORBA objects are loaded into the Interface Repository. You then use the **BEA Application Builder** to create Automation bindings for the interfaces.

The Application Builder is a development tool that you use along with a client development tool (such as Visual Basic) to select which CORBA objects in an WLE domain you want your ActiveX client application to interact with. For a description of the Application Builder and how it works, see the online help that is integrated into the Application Builder graphical user interface (GUI).

The combination of the ActiveX client application and the generated binding creates the ActiveX view of the object.

Figure 1-6 illustrates how the ActiveX Client works.

Figure 1-6 How the ActiveX Client Works



Naming Conventions for ActiveX Views

Naming conventions describe an algorithm for mapping CORBA interfaces to ActiveX to avoid type and variable name conflicts. Naming conventions also indicate how to use a given object. The names of all ActiveX methods begin with `DI`.

The ActiveX Client observes this naming convention when it creates Automation bindings for CORBA interfaces. If a CORBA interface has the name `Account`, the Automation binding for that interface has the name `DIAccount`.

CORBA interface names are often scoped within nested levels known as modules; however, in ActiveX, there is no scoping. To avoid name conflicts, the ActiveX Client exposes a CORBA interface into ActiveX with the name of the different scopes prepended to the name of the interface.

For example, a CORBA interface named `Account` is defined in the OMG IDL file as:

```
module University
{
   module Student
     {
       interface Account
           {//Operations and attributes of the Account interface
           };
     };

};
```

In CORBA, this interface is named `University::Student::Account`. The ActiveX Client translates this name to `DIUniversity_Student_Account` for ActiveX.

CHAPTER

2 Creating CORBA Client Applications

This topic includes the following sections:

- The development process for CORBA C++ client applications
- The development process for CORBA Java client applications
- Obtaining the Object Management Group (OMG) Interface Definition Language (IDL) file
- Selecting the invocation type
- Compiling the OMG IDL file
- Writing the CORBA client application
- Building the CORBA client application
- Server applications acting as client applications
- Using Java2 Applets

Summary of the Development Process for CORBA C++ Client Applications

The steps for creating a **CORBA** C++ **client application** are as follows:

Step	Description
1	Obtain the **OMG IDL** file for the CORBA interfaces used by the CORBA C++ client application.
2	Select the invocation type.
3	Use the IDL compiler to compile the OMG IDL file. The client stubs are generated as a result of compiling the OMG IDL.
4	Write the CORBA C++ client application. This topic describes creating a basic client application. You can also implement security and transactions in your CORBA C++ client applications. ■ For information about implementing security in your CORBA Java client application, see Chapter 4, “Using Security.” ■ For information about using transactions in your CORBA Java client application, see Chapter 5, “Using Transactions.”
5	Build the CORBA C++ client application.

Each step in the process is explained in detail in the following sections.

The WLE development environment for CORBA C++ client applications includes the following:

- The `idl` command, which compiles the OMG IDL file and generates the client stubs required for the CORBA interface.
- The `buildobjclient` command, which constructs a CORBA C++ client application executable.

- The C++ environmental objects, which provide access to CORBA objects in an WLE domain and to the services provided by the CORBA objects.

Summary of the Development Process for CORBA Java Client Applications

The **BEA** WLE **software** supports interoperability with the following products:

- Netscape Enterprise Server Version 3.0 and Java Development Kit (JDK) Version 1.1.5
- JDK Version 1.2

The steps for creating a CORBA Java client application are as follows:

Step	Description
1	Obtain the OMG IDL file for the CORBA interfaces used by the CORBA Java client application.
2	Select the invocation type.
3	Use the development tools provided by your CORBA Java **Object Request Broker** (ORB) to compile the OMG IDL file and generate client stubs.
4	Write the CORBA Java client application. This topic describes creating a basic client application. You can also implement security and transactions in your CORBA Java client applications. ■ For information about implementing security in your CORBA Java client application, see Chapter 4, “Using Security.” ■ For information about using transactions in your CORBA Java client application, see Chapter 5, “Using Transactions.”
5	Build the CORBA Java client application.

Each step in the process is explained in detail in the following sections.

You need to use the development tools provided by your CORBA Java ORB product to compile the OMG IDL file, generate the client stubs, and build the CORBA Java client application executable. You use the Java environmental objects, which provide access to CORBA objects in an WLE **domain** and to the services provided by the CORBA objects.

Step 1: Obtaining the OMG IDL File

Generally, the OMG IDL files for the available interfaces and operations are provided to the client programmer by the application designer. This section contains the OMG IDL for the Basic sample application. Listing 2-1 shows the `univb.idl` file, which defines the following interfaces:

Interface	Description	Operations
`Registrar`	Obtains course information from the course database	`get_courses_synopsis()` `get_courses_details()`
`RegistrarFactory`	Creates object references to the Registrar object	`find_registrar()`
`CourseSynopsisEnumerator`	Gets a subset of the information from the course database, and iteratively returns portions of that subset to the client application	`get_next_n()` `destroy()`

Listing 2-1 OMG IDL File for the Basic Sample Application

```
#pragma prefix "beasys.com"

module UniversityB

{

   typedef unsigned long CourseNumber;
   typedef sequence<CourseNumber> CourseNumberList;

   struct CourseSynopsis
     {
        CourseNumber   course_number;
        string         title;
     };

   typedef sequence<CourseSynopsis> CourseSynopsisList;
   interface CourseSynopsisEnumerator
   {
        CourseSynopsisList get_next_n(
            in  unsigned long number_to_get,
            out unsigned long number_remaining
   };
         void destroy();
   };

   typedef unsigned short Days;
   const Days MONDAY    =  1;
   const Days TUESDAY   =  2;
   const Days WEDNESDAY =  4;
   const Days THURSDAY  =  8;
   const Days FRIDAY    = 16;

   struct ClassSchedule
   {
       Days           class_days; // bitmask of days
       unsigned short start_hour; // whole hours in military time
       unsigned short duration;   // minutes
   };

   struct CourseDetails
   {
       CourseNumber   course_number;
       double         cost;
       unsigned short number_of_credits;
       ClassSchedule  class_schedule;
       unsigned short number_of_seats;
       string         title;
```

```
        string         professor;
        string         description;
     };

    typedef sequence<CourseDetails> CourseDetailsList;

    interface Registrar
    {
        CourseSynopsisList
        get_courses_synopsis(
        in string                   search_criteria,
        in unsigned long            number_to_get, // 0 = all
        out unsigned long           number_remaining,
        out CourseSynopsisEnumerator rest
     );

        CourseDetailsList get_courses_details(in CourseNumberList
         courses);

    interface RegistrarFactory
    {
         Registrar find_registrar(
     );

   };

};
```

Step 2: Selecting the Invocation Type

Select the invocation type (static or dynamic) that you will use in the requests in the client application. You can use both types of invocation in a client application.

For an overview of static and dynamic invocation, see Chapter 1, “Client Application Development Concepts.”

The remainder of this topic assumes that you chose to use static invocation in your CORBA client application. If you chose to use dynamic invocation, see Chapter 6, “Using the Dynamic Invocation Interface.”

Step 3: Compiling the OMG IDL File

When creating CORBA C++ client applications, use the `idl` command to compile the OMG IDL file and generate the files required for the interface. The following is the syntax of the `idl` command:

`idl` *`idlfilename(s)`*

The IDL compiler generates a client stub (*`idlfilename`*`_c.cpp`) and a header file (*`idlfilename`*`_c.h`) that describe everything you need to have to use the client stub from the C++ programming language. You need to link these files into your client application.

In addition, the IDL compiler generates skeletons that contain the signatures of the CORBA object's operations. The generated skeleton information is placed in the *`idlfilename`*`_s.cpp` and *`idlfilename`*`_s.h` files. During development of the client application, it can be useful to look at the server header files and skeleton file.

Note: Do not modify the generated client stub or the skeleton.

For a complete description of the `idl` command and options, see the *C++ Programming Reference*.

When creating CORBA Java client applications:

- If you are using JDK Version 1.2, you can use the `idltojava` command to compile the OMG IDL file. For more information about the `idltojava` command, see the documentation for the JDK Version 1.2.
- If you are using Netscape Version 3.0 and Java Development Kit (JDK) Version 1.1.5, you need to use that product's IDL compiler to compile the OMG IDL.

The `idltojava` command or the IDL compiler generates the following:

- The client stubs for each interface (`_`*`interface`*`Stub.java`)
- The CORBA helper class (*`interface`*`Helper.java`) and the CORBA holder class (*`interface`*`Holder.java`) that describe everything you need to use the client stub from the Java programming language.

Note that each OMG IDL defined exception defines an exception class and its helper and holder classes. The compiled `.class` files must be in the `CLASSPATH` of your client application.

In addition, the `idltojava` command or the IDL compiler generates skeletons that contain the signatures of the operations of the CORBA object. The generated skeleton information is placed in the `_`*`interface`*`ImplBase` file.

Step 4: Writing the CORBA Client Application

To participate in a session with an WLE server application, an WLE client application must be able to get an object reference for a CORBA object and invoke operations on the object. To accomplish this, the CORBA client application code must do the following:

1. Initialize the WLE ORB.
2. Establish communication with the WLE domain.
3. Resolve initial references to the FactoryFinder object.
4. Use a **factory** to get an object reference for the desired CORBA object.
5. Invoke operations on the CORBA object.

The following sections use portions of the client applications in the Basic sample application to illustrate the steps. For information about the Basic sample application, see the *Guide to University Sample Applications*. The Basic sample application is located in the following directory on the WLE software kit:

`drive:``\M3dir\samples\corba\university\basic`

Initializing the ORB

All CORBA client applications must first initialize the ORB.

Use the following code to initialize the ORB from a CORBA C++ client application:

C++

```
CORBA::ORB_var orb=CORBA::ORB_init(argc, argv, ORBid);
```

Typically, no ORBid is specified and the default ORBid specified during installation is used. However, when a client application is running on a machine that also has server applications running and the client application wants to access server applications in another WLE domain, you need to override the default ORBid. This can be done by hard coding the ORBid as `BEA_IIOP` or by passing the ORBid in the command line as `_ORBid BEA_IIOP`.

Use the following code to initialize the ORB from a CORBA Java client application:

Java Application

```
org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init (args,props);
```

Use the following code to initialize the ORB from a CORBA Java client applet:

Java Applet

```
org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init (this,null);
```

where *`this`* is the name of the Java applet

Establishing Communication with the WLE Domain

The client application creates a Bootstrap object. A list of IIOP Listener/Handlers can be supplied either as a parameter, via the `TOBJADDR` Java property or applet property. A single IIOP Listener/Handler is specified as follows:

```
//host:port
```

When the IIOP Listerner/Handler is provided via `TOBJADDR`, the second argument of the constructor can be null.

The host and port combination for the IIOP Listener/Handler is defined in the **`UBBCONFIG` file**. The host and port combination that is specified for the Bootstrap object must exactly match the ISL parameter in the WLE domain's `UBBCONFIG` file. The format of the host and port combination, as well as the capitalization, must match. If the addresses do not match, the call to the Bootstrap object will fail and the following message appears in the log file:

```
Error: Unofficial connection from client at <tcp/ip
adress>/<portnumber>
```

For example, if the network address is specified as `//TRIXIE::3500` in the ISL parameter in the `UBBCONFIG` file, specifying either `//192.12.4.6.:3500` or `//trixie:3500` in the Bootstrap object will cause the connection attempt to fail.

On UNIX systems, use the `uname -n` command on the host system to determine the capitalization used. On Window NT, use the Network Control Panel to determine the capitalization.

The following C++ and Java examples show how to use the **Bootstrap** object:

C++

```
Tobj_Bootstrap* bootstrap = new Tobj_Bootstrap(orb, "//host:port");
```

Java

Use the following commands to get the valid IIOP Listener/Handlers for the client application:

Native client applications

```
Properties prop = Tobj_Bootstrap.getNativeProperties();
```

Remote client applications

```
Properties prop = Tobj_Bootstrap.getRemoteProperties();
```

Use the IIOP Listerner/Handler in the following command:

```
Tobj_Bootstrap bootstrap = new Tobj_Bootstrap(orb, "//host:port");
```

Java Applet

```
Tobj_Bootstrap bootstrap = new Tobj_Bootstrap(orb, "//host:port", this);
```

where *`this`* is the name of the Java applet

An WLE domain can have multiple IIOP Listener/Handlers. If you are accessing an WLE domain with multiple IIOP Listener/Handlers, you supply a list of `Host:Port` combinations to the Bootstrap object. If the second parameter of the Bootstrap command is an empty string, the Bootstrap object walks through the list until it connects to an WLE domain. The list of IIOP Listener/Handlers can also be specified in `TOBJADDR`.

If you want to access multiple WLE domains, you must create a Bootstrap object for each WLE domain you want to access.

Resolving Initial References to the FactoryFinder Object

The client application must obtain initial references to the environmental objects that provide services for the application. The Bootstrap object's `resolve_initial_references` operation can be called to obtain references to the **FactoryFinder**, InterfaceRepository, SecurityCurrent, and TransactionCurrent environmental objects. The argument passed to the operation is a string containing the name of the desired object reference. You need to get initial references only for the environmental objects you plan to use in your client application.

The following C++ and Java examples show how to use the Bootstrap object to resolve initial references to the FactoryFinder object:

C++

```
//Resolve Factory Finder
CORBA::Object_var var_factory_finder_oref = bootstrap.resolve_initial_references
       (“FactoryFinder”);
Tobj::FactoryFinder_var var_factory_finder_ref = Tobj::FactoryFinder::_narrow
      (factory_finder_oref.in());
```

Java

```
//Resolve Factory Finder
org.omg.CORBA.Object off = bootstrap.resolve_initial_references
      (“FactoryFinder”);
FactoryFinder ff=FactoryFinderHelper.narrow(off);
```

For information about using security in client applications, see Chapter 4, “Using Security.” For information about transactions in client applications, see Chapter 5, “Using Transactions.”

Using the FactoryFinder Object to Get a Factory

CORBA client applications get object references to CORBA objects from factories. A factory is any CORBA object that returns an object reference to another CORBA object and registers itself as a factory. The client application invokes anoperation on a factory to obtain an object reference to a CORBA object of a specific type. To use factories, the client application must be able to locate the factory it needs. The FactoryFinder object serves this purpose. For information about the function of the FactoryFinder object, see Chapter 1, “Client Application Development Concepts.”

The FactoryFinder object has the following methods:

- `find_factories()`

  Returns a sequence of factories that match the input key exactly.

- `find_one_factory()`

  Returns one factory that matches the input key exactly.

- `find_factories_by_id()`

  Returns a sequence of factories whose id field in the name component matches the input argument.

- `find_one_factory_by_id()`

  Returns one factory whose id field in the factory’s CORBA name component matches the input argument.

The following C++ and Java examples show how to use the FactoryFinder `find_one_factory_by_id` **method** to get a factory for the Registrar object used in the client application for the Basic sample applications:

C++

```
CORBA::Object_var var_registrar_factory_oref = var_factory_finder_ref->
     find_one_factory_by_id(UniversityB::_tc_RegistrarFactory->id()
);
UniversityB::RegistrarFactory_var var_RegistrarFactory_ref =
     UniversityB::RegistrarFactory::_narrow(
          var_RegistrarFactory_oref.in()
     );
```

Java

```
org.omg.CORBA.Object of = FactoryFinder.find_one_factory_by_id
    (UniversityB.RegistrarFactoryHelper.id());
UniversityB.RegistrarFactory F = UniversityB.RegistrarFactoryHelper.narrow(of);
```

Using a Factory to Get a CORBA Object

Client applications call the factory to get an object reference to a CORBA object. The client applications then invoke operations on the CORBA object by passing it a pointer to the factory and any arguments that the **operation** requires.

The following C++ and Java examples illustrate getting the factory for the Registrar **object** and then invoking the `get_courses_details()` method on the Registrar object:

C++

```
UniversityB::Registrar_var var_Registrar = var_RegistrarFactory->
             find_Registrar();
UniversityB::CourseDetailsList_var course_details_list = Registrar_oref->
             get_course_details(CourseNumberList);
```

Java

```
UniversityB.Registrar gRegistrarObjRef = F.find_registrar();
gRegistrarObjRef.get_course_details(selected_course_numbers);
```

Step 5: Building the CORBA Client Application

The final step in the development of the CORBA client application is to produce the executable client application. To do this, you need to compile the code and link against the **client stub**.

When creating CORBA C++ client applications, use the `buildobjclient` command to construct an WLE client application executable. The command combines the client stubs for interfaces that use static invocation, and the associated header files with the standard WLE libraries to form a client executable. For the syntax of the `buildobjclient` command, see the *C++ Programming Reference*.

When compiling CORBA Java client applications, you need to include the Java Archive (JAR) file that contains the Java classes for the WLE environmental objects in your `CLASSPATH`. If you are using JDK Version 1.2, the `m3envobj.jar` file is located in the following directory:

```
m3dir/udataobj/java/jdk
```

Server Applications Acting as Client Applications

To process a **request** from a client application, the server application may need to request processing from another server application. In this situation, the server application is acting as a client application.

To act as a client application, the server application must obtain a Bootstrap object for the current WLE domain. The Bootstrap object for the server application is already available via `TP::Bootstrap` (for CORBA C++ client applications) or `TP.Bootstrap` (for CORBA Java client applications). The server application then uses the FactoryFinder object to locate a factory for the CORBA object that can satisfy the request from the client application.

Using Java2 Applets

BEA WLE supports Java2 applets. To run Java2 applets, you need to install the Java Plug-In product from Sun Microsystems, Inc. The Java Plug-in runs Java applets in an HTML page using Sun's Java Virtual Machine (JVM).

Before downloading the Java Plug-in kit from the Sun web site, verify whether or not the Java Plug-In is already installed on your machine.

Netscape Navigator

In Netscape Navigator, choose the About Plug-Ins option from the Help menu in the browser window. The following will appear if the Java Plug-In is installed:

```
application/x-java-applet;version 1.2
```

Internet Explorer

From the Start menu in Windows NT Version 4.0, select the Programs option. If the Java Plug-In is installed, a Java Plug-In Control Panel option will appear.

If the Java Plug-In is not installed, you need to download and install the JDK1.2 plug-in (`jre12-win32.exe`) and the HTML Converter tool (`htmlconv12.zip`). You can obtain both these products from `java.sun.com/products/plugin`.

You also need to read the *Java Plug-In HTML Specification* located at `java.sun.com/products/plugin/1.2/docs`. This specification explains the changes Web page authors need to make to their existing HTML code to have existing JDK 1.2 applets run using the Java Plug-In rather that the brower's default Java run-time environment.

Write your Java applet. Use the following command to intialize the ORB from the Java applet:

```
org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init (this,null);
```

To automatically launch the Java Plug-In when Internet Explorer or Netscape Navigator browses the HTML page for your applet, use the `OBJECT` tag and the `EMBED` tag in the HTML specification. If you use the HTML Converter tool to convert your applet to HTML, these tags are automatically inserted. For more information about using the `OBJECT` and `EMBED` tags, see `java.sun.com/products/plugin/1.2/docs/tags.html`.

3 Creating ActiveX Client Applications

This topic includes the following sections:

- The development process for ActiveX client applications
- An overview of the BEA Application Builder
- Starting the Interface Repository server application
- Loading the Automation environmental objects into the Interface Repository
- Loading the CORBA interfaces into the Interface Repository
- Creating ActiveX bindings for CORBA interfaces
- Loading the type library for the ActiveX bindings
- Writing the ActiveX client application
- Creating a deployment package for the ActiveX client application

For a description of the concepts you need to understand before developing an ActiveX client application, see Chapter 1, “Client Application Development Concepts.”

Summary of the Development Process for ActiveX Client Applications

The steps for creating an **ActiveX client application** are as follows:

Step	Description
1	Load the Automation **environmental objects** into the Interface Repository.
2	Verify that the CORBA interfaces you want to access from your ActiveX client application are loaded in the **Interface Repository**. If necessary, load the **Object Management Group (OMG) Interface Definition Language (IDL)** definitions for the CORBA interfaces into the Interface Repository.
3	Start the **server application** process for the Interface Repository.
4	Use the **BEA Application Builder** to create ActiveX **bindings** for the interfaces of the CORBA object.
5	Load the type library for the ActiveX binding in your development tool.
6	Write the ActiveX client application. This topic describes creating a basic client application. You can also implement security and transactions in your ActiveX client applications. ■ For information about implementing security in your ActiveX client application, see Chapter 4, “Using Security.” ■ For information about using transactions in your ActiveX client application, see Chapter 5, “Using Transactions.”
7	Create a deployment package for the ActiveX client application.

Each step in the process is explained in detail in the following sections.

The WLE development environment for ActiveX client applications includes the following:

- The `idl2ir` command, which loads interface definitions defined in OMG IDL into the Interface Repository
- The Application Builder, which creates ActiveX bindings for the interfaces of CORBA objects and creates deployment packages for the interfaces
- The Automation environmental objects, which provide access to ActiveX views of CORBA objects (referred to as ActiveX views) in an WLE domain and the services provided by the ActiveX views

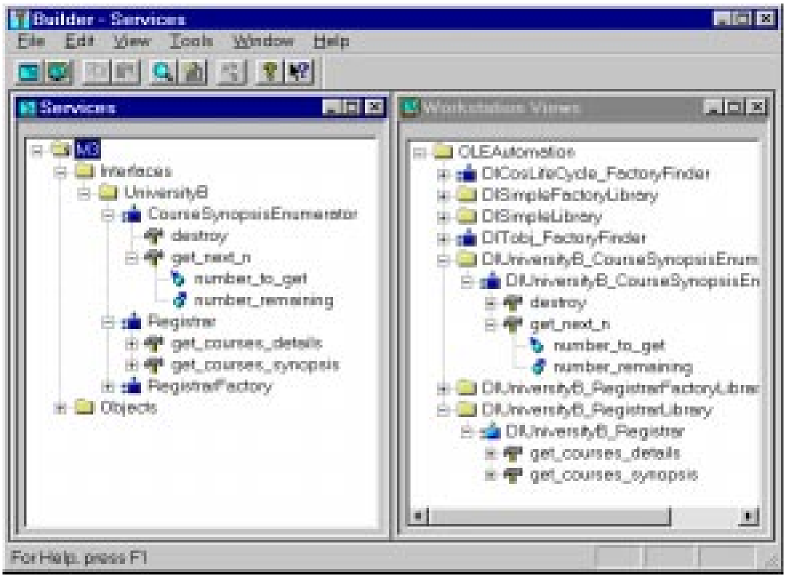
The BEA Application Builder

The Application Builder is the development tool that creates ActiveX views of CORBA objects. The Application Builder is the primary user interface to the BEA ActiveX Client. It can be used to select which CORBA objects are available to desktop applications, to create ActiveX views of the CORBA objects, and to create packages for deploying ActiveX views of CORBA objects to client machines.

To use an ActiveX view, you load the interfaces of the CORBA objects into the Interface Repository. You then create an ActiveX binding for the CORBA interface. The binding describes the interface of a CORBA object to ActiveX. The combination of the ActiveX client application and the generated **binding** creates the view of the object.

As shown in Figure 3-1, the Application Builder main window is partitioned into two parts: the Services window and the Workstation Views window.

Figure 3-1 Application Builder Main Window



The Services window presents all the CORBA modules, interfaces, and operations contained in the Interface Repository in the local WLE domain. You can create bindings for all the interfaces in the Interface Repository.

At the top of the Services window are entries for each **object system** that is available from the WLE domain. The ActiveX Client supports only the WLE object system. The objects are displayed in the same hierarchical format used in the Interface Repository, that is, as modules, interfaces, operations, and the parameters contained in operations. The [+] symbol indicates an object that can be expanded to display the other objects.

The Workstation Views window presents all the ActiveX bindings that have been created for CORBA interfaces. To create a binding for a CORBA interface, you drag an entry from the Services window and into the Workstation Views window.

For a description of the Application Builder and how it works, see the online help, which is integrated into the product graphical user interface (GUI).

Step 1: Loading the Automation Environmental Objects into the Interface Repository

Load the Automation environmental objects into the **Interface Repository** so that the interface definitions for the objects are available to ActiveX client applications. From the MS-DOS prompt, enter the following command to load the OMG IDL file (`TOBJIN.idl`) into the Interface Repository:

```
prompt> idl2ir -D _TOBJ -I drive:\m3dir\include drive:\m3dir\include\tobjin.idl
```

Step 2: Loading the CORBA Interfaces into the Interface Repository

Before you can create an ActiveX view for a CORBA object, the interfaces of the CORBA object need to be loaded into the Interface Repository. If the interfaces of a CORBA object are not loaded in the Interface Repository, they do not appear in the Services window of the Application Builder. If a desired CORBA interface does not appear in the Services window, use the `idl2ir` command to load the OMG IDL that defines the CORBA into the Interface Repository. The syntax for the `idl2ir` command is as follows:

```
idl2ir [repositoryfile.idl] file.idl
```

Option	Description
`repositoryfile`	Directs the command to load the OMG IDL files for the CORBA interface into the specified Interface Repository. Specify the name of the Interface Repository in the WLE domain that the ActiveX client application will access.

Option	Description
`file.idl`	Specifies the OMG IDL file containing definitions for the CORBA interface.

For a complete description of the `idl2ir` command, refer to the *C++ Programming Reference*.

Chapter 2, "Creating CORBA Client Applications," provides a sample OMG IDL file that is the starting point for all the WLE University sample applications. Based on this OMG IDL file, the following CORBA interfaces should appear in the Application Builder window:

- `RegistrarFactory`
- `Registrar`
- `CourseSynopsisEnumerator`

For a complete description of the University sample applications, see the *Guide to the University Sample Applications*.

Step 3: Starting the Interface Repository Server Application

ActiveX client applications read the interface definitions for CORBA objects from the Interface Repository dynamically at run time and translate them to Automation objects. Therefore, the server application for the Interface Repository needs to be started so that the interface definitions are available. Use the **`UBBCONFIG` file** to start the server application process for the Interface Repository.

Note: In some cases, the system administrator may have performed this step.

In the `UBBCONFIG` file for the WLE domain, check that `TMIFRSVR`, the server application for the Interface Repository, is started. The following entry should appear in the `UBBCONFIG` file:

```
TMIFRSVR
     SRVGRP = SYS_GRP
     SRVID = 6
     RESTART = Y
     MAXGEN = 5
     GRACE = 3600
```

In addition, make sure that the `ISL` parameter to start the **IIOP Listener/Handler** is specified. The following entry should appear in the `UBBCONFIG` file:

```
ISL
      SRVGRP = SYS_GRP
      SRVID  = 5
      CLOPT  = "-A -- -n //TRIXIE:2500"
```

where `TRIXIE` is the name of the host (server) system and `2500` is the port number.

For more information about starting server applications and specifying the ISL parameter, see *Administration Guide*.

Step 4: Creating ActiveX Bindings for the CORBA Interfaces

For an ActiveX client application to access a CORBA object, you must generate ActiveX bindings for the interfaces of the CORBA object. You use the Application Builder to create the ActiveX bindings for CORBA interfaces.

To create an ActiveX binding for a CORBA interface:

1. Click the BEA Application Builder icon in the BEA WLE System v2.1 program group.

   The Domain logon window appears.

2. Enter the host name and port number that you specified in the `ISL` parameter in the `UBBCONFIG` file in the logon window. You must match exactly the capitalization used in the `UBBCONFIG` file.

   The Application Builder logon window appears.

3. Highlight the desired CORBA interface in the Services window and drag it to the Workstation Views window, or cut the CORBA interface from the Services window and paste it into the Workstation Views window.

   The Application Builder:

   - Creates a type library. By default, the type library is placed in `\M3dir\TypeLibraries.`

     The type library file is named: `DI`*`modulename_interfacename`*`.tlb`

   - Creates a Windows system registry entry, including unique Program IDs for each object type, for the CORBA interface.

You can now use the ActiveX view from an ActiveX client application.

For a complete description of the features of the Application Builder, see the online help that is integrated into the Application Builder graphical user interface (GUI).

Step 5: Loading the Type Library for the ActiveX Bindings

Before you start writing your ActiveX client application, you need to load the type library that describes the ActiveX binding for the CORBA interface in your development tool. Follow your development product's instructions for loading type libraries.

For example, in Visual Basic Version 5.0, you use the References option on the Project menu to get a list of available type libraries. You then select the desired type libraries from the list.

By default, the Application Builder places all generated type libraries in `\M3dir\TypeLibraries`. The type library for the ActiveX binding of the CORBA interface has the following format:

```
DImodulename_interfacename.tlb
```

Step 6: Writing the ActiveX Client Application

The ActiveX client application must do the following:

1. Include declarations for the Automation environmental objects, the factory for the ActiveX view, and the ActiveX view.
2. Establish communication with the WLE domain.
3. Use the Bootstrap object to obtain a reference to the FactoryFinder object.
4. Use a factory to obtain an object reference to an ActiveX view.
5. Invoke operations on the ActiveX view.
6. Deploy the ActiveX client application.

The following sections use portions of the ActiveX client applications in the Basic sample application to illustrate the steps. For information about the Basic sample application, see the *Guide to the University Sample Applications*. The Basic sample application is located in the following directory on the **BEA WLE software** kit:

`drive:\M3dir\samples\corba\university\basic`

Including Declarations for the Automation Environmental Objects, Factories, and ActiveX Views of CORBA Objects

To prevent errors at run time, you need to declare the object types of:

- The Automation environmental objects
- The factories that create the ActiveX views of the CORBA objects
- The ActiveX views

The following example is Visual Basic code that declares the Bootstrap and FactoryFinder objects, the factory for the ActiveX view of the Registrar object, and the ActiveX view of the Registrar object:

```
\\Declare Bootstrap object\\
    Public objBootstrap As DITobj_Bootstrap
\\Declare FactoryFinder object\\
    Public objFactoryFinder As DITobj_FactoryFinder
\\Declare factory object for Registrar Object\\
    Public objRegistrarFactory As DIUniversityB_RegistrarFactory
\\Declare the ActiveX view of the Registrar object\\
    Public objRegistrar As DIUniversityB_Registrar
```

Establishing Communication with the WLE Domain

When writing an ActiveX client application, there are two steps to establishing communication with the WLE domain:

1. Create the Bootstrap object.
2. Initialize the Bootstrap object.

The following Visual Basic example illustrates using the `CreateObject` operation to create a Bootstrap object:

```
Set objBootstrap = CreateObject(“Tobj.Bootstrap”)
```

You then initialize the Bootstrap object. When you initialize the Bootstrap object, you supply the host and port of the IIOP Listener/Handler of the desired WLE domain, as follows:

```
objBootstrap.Initialize “//host:port”
```

The host and port combination for the IIOP Listener/Handler is defined in the `ISL` parameter of the `UBBCONFIG` file. The host and port combination that is specified for the Bootstrap object must exactly match the `ISL` parameter. The format of the host and port combination, as well as the capitalization, must match. If the addresses do not match, the call to the Bootstrap object will fail and the following message appears in the log file:

```
Error: Unofficial connection from client at <tcp/ip adress/<portnumber>
```

For example, if the network address is specified as `//TRIXIE::3500` in the ISL parameter in the `UBBCONFIG` file, specifying either `//192.12.4.6.:3500` or `//trixie:3500` in the Bootstrap object will cause the connection attempt to fail.

An WLE domain can have multiple IIOP Listener/Handlers. If you are accessing an WLE domain with multiple IIOP Listener/Handlers, you supply a list of *`host:port`* combinations to the Bootstrap object. The Bootstrap object walks through the list until it connects to an WLE domain. The list of IIOP Listener/Handlers can also be specified in the `TOBJADDR` environmental variable.

If you want to access multiple WLE domains, you must create a Bootstrap object for each WLE **domain** you want to access.

Obtaining References to the FactoryFinder Object

The client application must obtain initial references to the objects that provide services for the application. The Bootstrap object is used to obtain references to the FactoryFinder object, SecurityCurrent object, and TransactionCurrent object. The argument passed to the operation is a string containing the `progid` of the desired object. You have to get references only for the objects that you plan to use in your ActiveX client application.

The following Visual Basic example shows how to use the **Bootstrap** object to obtain a reference to the FactoryFinder object:

```
Set objFactoryFinder = objBootstrap.CreateObject(“Tobj.FactoryFinder”)
```

Using a Factory to Get an ActiveX View

ActiveX client applications get interface pointers to ActiveX views of CORBA objects from factories. A factory is any CORBA object that returns an object reference to another CORBA object. The ActiveX client application invokes an operation on a factory to obtain an object reference to a CORBA object of a specific type. To use factories, the ActiveX client application must be able to locate the factory it needs. The FactoryFinder object serves this purpose. For information about the function of the FactoryFinder object, see Chapter 1, “Client Application Development Concepts.”

Use the `CreateObject` function to create the FactoryFinder object, and then use one of the FactoryFinder object **method**s to find a factory. The FactoryFinder object has the following methods:

- `find_factories()`

  Returns a sequence of factories that match the input key exactly.

- `find_one_factory()`

  Returns one factory that matches the input key exactly.

- `find_factories_by_id()`

  Returns a sequence of factories whose ID field in the name component matches the input argument.

- `find_one_factory_by_id()`

  Returns one factory whose ID field in the factory's CORBA name component matches the input argument.

The following Visual Basic example shows how to use the **FactoryFinder** `find_one_factory_by_id()` method to get a factory for the Registrar object used in the client application for the Basic sample applications:

```
Set objRegistrarFactory = objBsFactoryFinder.find_one_factory_by_id
    (“RegistrarFactory”)
Set objRegistrar = RegistrarFactory.find_registrar
```

Invoking Operations on the ActiveX View

Invoke **operation**s on the ActiveX view by passing it a pointer to the **factory** and any arguments that the operation requires.

The following Visual Basic example shows how to invoke operations on an ActiveX view:

```
‘Get course details from the Registrar object’
aryCourseDetails = objRegistrar.get_course_details(aryCourseNumbers)
```

Step 7: Deploying the ActiveX Client Application

To distribute ActiveX client applications to other client machines, you need to create a deployment package. A deployment package contains all the data needed by the client application to use ActiveX views of CORBA objects, including the bindings, the type libraries, and the registration information. The deployment package is a self-registering ActiveX control with the file extension `.ocx`.

To create a deployment package for an ActiveX view:

1. Select an ActiveX view from the Workstation Views window.
2. Click Tools->Deploy Modules, or click the right mouse button on the desired view and choose the Deploy Modules option from the menu.

   A confirmation window is displayed.
3. Click Create to create the deployment package.

   By default, the deployment package is placed in `\M3dir\Packages.`

4 Using Security

This topic describes how to use security in CORBA C++, CORBA Java, and ActiveX client applications for the BEA WLE software.

For an example of how security is implemented in working client applications, see the description of the Security sample application in the *Guide to the University Sample Applications*.

For an overview of the SecurityCurrent object, see Chapter 1, “Client Application Development Concepts.”

Overview of WLE Security

CORBA C++, CORBA Java, and **ActiveX** client applications use security to authenticate themselves to the WLE **domain**. Authentication is the process of verifying the identity of a client application. By entering the correct logon information, the **client application** authenticates itself to the WLE domain. The WLE **software** uses authentication as defined in the **CORBAservices Security Service** and provides extensions for ease of use.

A client application must provide security information according to the security level defined in the desired WLE domain. This information is defined by the WLE system administrator in the **`UBBCONFIG` file** for the WLE domain. When creating client applications, you must work with the WLE system administrator to obtain the correct security information (such as the user name and user password) for the WLE domain you want to access from the client application.

Summary of the Development Process for Security

The steps for adding security to a client application are as follows:

Step	Description
1	Use the **Bootstrap** object to obtain a reference to the SecurityCurrent object in the specified WLE domain.
2	Get the PrincipalAuthenticator object from the SecurityCurrent object.
3	Use the `get_auth_type` operation of the PrincipalAuthenticator object to return the type of authentication expected by the WLE domain.
4	Log on to the WLE domain using the required security information.
5	Log off the WLE domain.

The following sections describe these steps and use portions of the client applications in the Security sample application to illustrate the steps. For information about the Security sample application, see the *Guide to the University Sample Applications*. The Security sample application is located in the following directory on the WLE software kit:

```
drive:\M3dir\samples\corba\university\security
```

Step 1: Using the Bootstrap Object to Obtain the SecurityCurrent Object

Use the Bootstrap object to obtain an object reference to the SecurityCurrent object for the specified WLE domain. The SecurityCurrent object is a `SecurityLevel2::Current` object as defined by the CORBAservices Security Service. For a complete description of the SecurityCurrent object, see *Using Security*.

The following C++ , Java, and Visual Basic examples illustrate how the Bootstrap object is used to return the SecurityCurrent object:

C++

```
CORBA::Object_var var_security_current_oref =
     bootstrap.resolve_initial_references(“SecurityCurrent”);
SecurityLevel2::Current_var var_security_current_ref =
     SecurityLevel2::Current::_narrow(var_security_current_oref.in());
```

Java

```
org.omg.CORBA.Object SecurityCurrentObj =
      gBootstrapObjRef.resolve_initial_references(“SecurityCurrent”);
org.omg.SecurityLevel2.Current secCur =
     org.omg.SecurityLevel2.CurrentHelper.narrow(secCurObj);
```

Visual Basic

```
Set objSecurityCurrent = objBootstrap.CreateObject(“Tobj.SecurityCurrent”)
```

Step 2: Getting the PrincipalAuthenticator Object from the SecurityCurrent Object

The SecurityCurrent object returns a reference to the PrincipalAuthenticator for the WLE domain. The PrincipalAuthenticator is used to get the authentication level required for an WLE domain.

The following C++, Java, and Visual Basic examples illustrate how to obtain the PrincipalAuthenticator for an WLE domain:

C++

```
//Get the PrincipalAuthenticator
SecurityLevel2::PrincipalAuthenticator_var var_principal_authenticator_oref =
     var_security_current_oref->principal_authenticator();
//Narrow the PrincipalAuthenticator
Tobj::PrincipalAuthenticator_var var_bea_principal_authenticator =
     Tobj::PrincipalAuthenticator::_narrow
                                    var_principal_authenticator_oref.in());
```

Java

```
//Get the PrincipalAuthenticator
org.omg.SecurityLevel2.PrincipalAuthenticator authlevel2 =
     secCur.principal_authenticator();
//Narrow the PrincipalAuthenticator
com.beasys.Tobj.PrincipalAuthenticatorObjRef gPrinAuthObjRef =
     (com.beasys.Tobj.PrincipalAuthenticator)
     org.omg.SecurityLevel2.PrincipalAuthenticatorHelper.narrow(authlevel2);
```

Visual Basic

```
Set objPrincAuth = objSecurityCurrent.principal_authenticator
```

Step 3: Obtaining the Authentication Level

Use the `Tobj::PrincipalAuthenticator::get_auth_type()` method to get the level of authentication required by the WLE domain.

For a complete description of the `Tobj::PrincipalAuthenticator` methods, see the *C++ Programming Reference* available from the WLE online information set.

The following C++, Java, and Visual Basic examples illustrate how to obtain the PrincipalAuthenticator for an WLE domain:

C++

```
//Determine the security level
Tobj::AuthType auth_type = var_bea_principal_authenticator->get_auth_type();
```

Java

```
//Determine the security level
com.beasys.Tobj.Authtype authType = gPrinAuthObjRef.get_auth_type();
```

Visual Basic

```
AuthorityType = objPrinAuth.get_auth_type
```

Step 4: Logging on to the WLE Domain with Proper Authentication

Use the `Tobj::PrincipalAuthenticator::logon()` method to log your client application into the desired WLE domain. The method requires the following arguments:

- *`user_name`*

  The WLE user name. This information is required for TOBJ_SYSAUTH and TOBJ_APPAUTH authentication levels. This information may be supplied for the TOBJ_NOAUTH authentication level; however, it is not required. The system designer decides this name at design time.

- *`client_name`*

  The WLE client application name. This information is required for TOBJ_SYSAUTH and TOBJ_APPAUTH authentication levels. This information may be supplied for the TOBJ_NOAUTH authentication level; however, it is not required. Obtain this information from the system administrator.

- *`system_password`*

  The WLE password. This information is required for TOBJ_SYSAUTH and TOBJ_APPAUTH authentication levels. Obtain this information from the system administrator.

- *`user_password`*

  The user password for the WLE authentication service. This information is required for the TOBJ_APPAUTH authentication level.

- *user_data*

  Application-specific data for authentication. This information is required when the WLE domain the client application is accessing is not using the authentication service provided with the WLE software.

The *user_password* and *user_data* arguments are mutually exclusive, depending on the authentication service used in the configuration of the WLE software. If you are using an authentication service other than an authentication service provided by the WLE software, provide the information required for logon in the *user_data* argument. The `Tobj::PrincipalAuthenticator::logon()` method raises a `CORBA::BAD_PARAM` exception if both *user_password* and *user_data* are set.

If an WLE domain has a TOBJ_NOAUTH authentication level, the client application is not required to supply a *user_name* or *client_name* when logging on to the WLE domain. If the client application does not logon with a *user_name* and *client_name*, the IIOP Listener/Handler of the WLE domain registers the client application with the *user_name* and the *client_name* set for the IIOP Listener/Handler in the `UBBCONFIG` file. However, the client application can log on with any *user_name* and *client_name*.

The `logon()` method returns one of the following:

- `Security::AuthenticationStatus::SecAuthSuccess` if the authentication succeeded

- `Security::AuthenticationStatus::SecAuthFailure` if the authentication failed or if the client application was already authenticated and did not log off the WLE domain

The following C++, Java, and Visual Basic examples illustrate how to use the `Tobj::PrincipalAuthenticator::logon()` method:

C++

```
Tobj::AuthType auth_type = var_bea_principal_authenticator->get_auth_type();
Security::AuthenticationStatus status = var_bea_principalauthenticator->logon(
                                                    user_name,
                                                    client_name,
                                                    system_password,
                                                    user_password,
                                                    0);
```

Java

```
org.omg.Security.AuthenticationStatus status = gPrinAuthObjRef.logon
     (gUserName, ClientName, gSystemPassword, gUserPassword,0);
```

Visual Basic

```
If AuthorityType = TOBJ_APPAUTH Then logonStatus = oPrincAuth.Logon(
                           UserName,_
                           ClientName,_
                           SystemPassword,_
                           UserPassword,_
                           UserData)
End If
```

Step 5: Logging off the WLE Domain

The client application must log off the current WLE domain before it can log on as another user in the same WLE domain. Use the `Tobj::PrincipalAuthenticator::logoff()` method to discard the WLE current authentication context and credentials. This method does not close the network connections to the WLE domain. After logging off the WLE domain, calls using the existing authentication fail if the authentication type is not TP_NOAUTH.

5 Using Transactions

This topic describes how to use transactions in CORBA C++, CORBA Java, and ActiveX client applications for the WLE software.

For an example of how transactions are implemented in working client applications, see the descrption of the Transactions sample application in the *Guide to the University Sample Applications*.

For an overview of the TransactionCurrent object, see Chapter 1, “Client Application Development Concepts.”

Overview of WLE Transactions

Client applications use transaction processing to ensure that data remains correct, consistent, and persistent. The transactions in the WLE software allow client applications to begin and terminate transactions and to get the status of transactions. The WLE software uses transactions as defined in the **CORBAservices Object Transaction Service**, with extensions for ease of use.

Transactions are defined on **interfaces**. The application designer decides which interfaces within an WLE client/server application will handle transactions. Transaction policies are defined in the Implementation Configuration File (ICF) for C++ server applications, or in the Server Description file (XML) for Java server applications. Generally, the ICF file or the Server Description file for the available interfaces is provided to the client programmer by the application designer.

If you prefer, you can use the Transaction application programming interface (API) defined in the `javax.transaction` package that is shipped with the BEA WLE Java V2.2 software.

Summary of the Development Process for Transactions

The steps for adding transactions to a client application are as follows:

Step	Description
1	Use the **Bootstrap object** to obtain a reference to the TransactionCurrent object in the specified WLE domain.
2	Use the methods of the TransactionCurrent object to include the interface of a CORBA object in a transaction operation.

The following sections describe these steps and use portions of the client applications in the Transactions sample application to illustrate the steps. For information about the Transactions sample application, see the *Guide to the University Sample Applications*. The Transactions sample application is located in the following directory on the WLE software kit:

`drive:\M3dir\samples\corba\university\transactions`

Step 1: Using the Bootstrap Object to Obtain the TransactionCurrent Object

Use the Bootstrap object to obtain an object reference to the TransactionCurrent object for the specified WLE domain. For a complete description of the TransactionCurrent object, see *Using Transactions.*

The following C++, Java, and Visual Basic examples illustrate how the Bootstrap object is used to return the TransactionCurrent object:

C++

```
CORBA::Object_var var_transaction_current_oref =
     Bootstrap.resolve_initial_references("TransactionCurrent");
CosTransactions::Current_var transaction_current_oref=
     CosTransactions::Current::_narrow(var_transaction_current_oref.in());
```

Java

```
org.omg.CORBA.Object transCurObj = gBootstrapObjRef.resolve_initial_references
     ("TransactionCurrent");
org.omg.CosTransactions.Current gTransCur=
     org.omg.CosTransactions.CurrentHelper.narrow(transCurObj);
```

Visual Basic

```
Set objTransactionCurrent = objBootstrap.CreateObject("Tobj.TransactionCurrent")
```

Step 2: Using the TransactionCurrent Methods

The TransactionCurrent object has **method**s that allow a client application to manage transactions. These methods can be used to begin and end transactions and to obtain information about the current transaction. The TransactionCurrent object provides the following methods:

- `begin()`

  Creates a new transaction. Future **operation**s take place within the scope of this transaction. When a client application begins a transaction, the default transaction timeout is 300 seconds. You can change this default, using the `set_timeout` method.

- `commit()`

  Ends the transaction successfully. Indicates that all operations on this client application have completed successfully.

- `rollback()`

  Forces the transaction to roll back.

- `rollback_only ()`

Marks the transaction so that the only possible action is to roll back. Generally, this method is used only in server applications.

- `suspend()`

  Suspends participation in the current transaction. This method returns an object that identifies the transaction and allows the client application to resume the transaction later.

- `resume()`

  Resumes participation in the specified transaction.

- `get_status()`

  Returns the status of a transaction with a client application.

- `get_transaction_name()`

  Returns a printable string describing the transaction.

- `set_timeout()`

  Modifies the timeout period associated with transactions. The default transaction timeout value is 300 seconds. If a transaction is automatically started instead of explicity started with the begin() method, the timeout value is determined by the value of the TRANTIME parameter in the UBBCONFIG file. For more information about setting the TRANTIME parameter, see *Administration Guide* available from the WLE online information set.

- `get_control()`

  Returns a control object that represents the transaction.

A basic transaction works in the following way:

1. A client application begins a transaction using the `Tobj::TransactionCurrent::begin()` method. This method does not return a value.

2. The operations on the CORBA interface execute within the scope of a transaction. If a call to any of these operations raises an exception (either explicitly or as a result of a communications failure), the exception can be caught and the transaction can be rolled back.

3. Use the `Tobj::TransactionCurrent:commit()` method to commit the current transaction. This method ends the transaction and starts the processing of the operation. The transaction is committed only if all of the participants in the transaction agree to commit.

   The association between the transaction and the client application ends when the client application calls the `Tobj::TransactionCurrent:commit()`method or the `Tobj::TransactionCurrent:rollback()` method.The following C++, Java, and Visual Basic examples illustrate using a transaction to encapsulate the operation of a student registering for a class:

C++

```
//Begin the transaction
var_transaction_current_oref->begin();
try {
//Perform the operation inside the transaction
     pointer_Registar_ref->register_for_courses(student_id, course_number_list);
     ...
//If operation executes with no errors, commit the transaction:
     CORBA::Boolean report_heuristics = CORBA_TRUE;
     var_transaction_current_ref->commit(report_heuristics);
    }
catch (...) {
//If the operation has problems executing, rollback the
//transaction. Then throw the original exception again.
//If the rollback fails,ignore the exception and throw the
//original exception again.
try {
     var_transaction_current_ref->rollback();
    }
catch (...) {
            TP::userlog("rollback failed");
            }
throw;
}
```

Java

```
try{
   gTransCur.begin();
   //Perform the operation inside the transaction
   not_registered =
      gRegistrarObjRef.register_for_courses(student_id,selected_course_numbers);


   if (not_registered != null)
```

```
        //If operation executes with no errors, commit the transaction
        boolean report_heuristics = true;
        gTransCur.commit(report_heuristics);

    } else gTransCur.rollback();


} catch(org.omg.CosTransactions.NoTransaction nte) {
    System.err.println("NoTransaction: " + nte);
    System.exit(1);
} catch(org.omg.CosTransactions.SubtransactionsUnavailable e) {
    System.err.println("Subtransactions Unavailable: " + e);
    System.exit(1);
} catch(org.omg.CosTransactions.HeuristicHazard e) {
    System.err.println("HeuristicHazard: " + e);
    System.exit(1);
} catch(org.omg.CosTransactions.HeuristicMixed e) {
    System.err.println("HeuristicMixed: " + e);
    System.exit(1);
}
```

Visual Basic

```
' Begin the transaction
'
objTransactionCurrent.begin
'
' Try to register for courses
'
NotRegisteredList = objRegistrar.register_for_courses(mStudentID,
     CourseList, exception)
'
If exception.EX_majorCode = NO_EXCEPTION then
     ' Request succeeded, commit the transaction
     '
     Dim report_heuristics As Boolean
     report_heuristics = True
     objTransactionCurrent.commit report_heuristics
Else
     ' Request failed, Roll back the transaction
     '
     objTransactionCurrent.rollback
          MsgBox "Transaction Rolled Back"
End If
```

CHAPTER

6 Using the Dynamic Invocation Interface

This topic includes the following sections:

- When to use the Dynamic Invocation Interface (DII)
- DII concepts
- The development process for client applications using DII
- Loading the CORBA interfaces into the Interface Repository
- Obtaining a generic object reference
- Creating a request
- Sending DII requests and retrieving the results
- Deleting the request
- Using the Interface Repository with DII

The information in this topic applies to CORBA C++ and CORBA Java client applications. DII is not supported in ActiveX client applications.

For an overview of the invocation types and DII, see Chapter 1, “Client Application Development Concepts.”

For a complete description of the CORBA member functions mentioned in this topic, see the *C++ Programming Reference* or the JDK 1.2 documentation for Java mappings of the CORBA member functions.

When to Use DII

There are good reasons to use either static or dynamic invocation to send requests from the client application. You may find you want to use both invocation types in the same client applications. To choose an invocation type, you need to understand the advantages and disadvantages of DII.

One of the major differences between static invocation and dynamic invocation is that, while both support synchronous and one-way communication, only dynamic invocation supports deferred synchronous communication.

In synchronous communication, the **client application** sends a request and waits until a response is retrieved; the client application cannot do any other work while it is waiting for the response. In deferred synchronous communication, the client application sends the request and is free to do other work. Periodically, the client application checks to see if the request has completed; when the request has completed, the client application makes use of the result of that request.

In addition, DII enables a client application to invoke a method on a **CORBA object** whose type was unknown at the time the client application was written. This contrasts with static invocation, which requires that the client application include a **client stub** for each type of CORBA object the client application intends to invoke. However, DII is more difficult to program (your code has to do the work of a client stub).

A client application can use DII to obtain better performance. For example, the client application can send multiple deferred synchronous requests at the same time and can handle the completions as they occur. If the requests go to different **server application**s, this work can be done in parallel. You cannot do this when you are using synchronous client stubs.

Note: The client stubs have optimizations, that allow the client stubs to achieve quicker response time than is achieved with DII when sending a single request and immediately blocking to get the response for that request.

DII is purely an interface to the client application; static and dynamic invocations are identical from a server application's point of view.

DII Concepts

DII frequently offers more than one way to accomplish a task, the trade-off being programming simplicity versus performance. This section describes the high-level concepts you need to understand to use DII. Details, including code examples, are provided later in this topic.

The concepts presented in this section are as follows:

- Request objects
- Request sending options
- Reply receiving options

Request Objects

A request object represents one invocation on one method of a CORBA object. If you want to make two invocations on the same method, you need to create two request objects.

To invoke a method, you need an object reference to the CORBA object that contains the method. You use the object reference to create a request object, populate the request object with arguments, send the request, wait for the reply, and obtain the result from the request.

You can create a request object in the following ways:

- Use the `CORBA::Object::_request` member function.

  Use the `CORBA::Object::_request` member function to create an empty request object specifying only the interface name you intend to invoke in the request (for example, `get_course_details`). Once the request object is created, the arguments, if there are any, must be added before the request can be sent to the server application. You invoke the `CORBA::NVList::add_value` member function for each argument required by the method you intend to invoke.

  You must also specify the type of the request's result using the `CORBA::Request::result` member function. For performance reasons, the messages exchanged between Object Request Brokers (ORBs) do not contain type information. By specifying a place holder for the result type, you give the ORB the information it needs to properly extract the result from the reply. Similarly, if the method you are invoking can raise user exceptions, you must add a place holder for exceptions before sending the request object.

- Use the `CORBA::Object::_create_request` member function.

  When you use the `CORBA::Object:: _create_request` member function to create a request object, you pass all the arguments required to make the request and to specify the types of the result and user exceptions, if there are any, that the request may return. Using this member function, you create an empty NVList, add arguments to the NVList one at a time, and create the request object, passing the completed NVList as an argument to the request. The potential advantage of the `CORBA::Object::_create_request` member function is performance. You can reuse the arguments in multiple `CORBA::ORB::_create_request` calls if you invoke the same method on multiple target objects.

For a complete description of the CORBA member functions, see the *C++ Programming Reference* available from the WLE online information set.

Options for Sending Requests

Once you have created and populated a request object with arguments, a result type, and exception types, you send the request to the CORBA object. There are several ways to send a request:

- The simplest way is to call the `CORBA::Request::invoke` member function, which blocks until the reply message is retrieved.
- More complex, but not blocking, is to use the `CORBA::Request::send_deferred` member function.
- If you want to invoke multiple CORBA requests in parallel, use the `CORBA::ORB::send_multiple_requests_deferred` member function. It takes a sequence of request objects.
- Use the `CORBA::Request::send_oneway` member function if, and only if, the CORBA method has been defined as oneway in the OMG IDL file.

- You can invoke multiple oneway methods in parallel with the ORB's `CORBA::ORB::send_multiple_requests_oneway` member function.

Note: When using the `CORBA::Request::send_deferred` member function, the invocation on the request object acts synchronously when the target object is in the same address space as the client application issuing the invocation. As a result of this behavior, calling the `CosTransaction::Current::suspend` operation does not raise the `CORBA::BAD_IN_ORDER` exception, because the transaction has completed.

For a complete description of the CORBA member functions, see the *C++ Programming Reference* or the JDK 1.2 documentation for the Java mappings of the CORBA member functions.

Options for Receiving the Results of Requests

If you send a request using the invoke method, there is only one way to get the result: use the request object's `CORBA::Request::env` member function to test for an exception; and if there is not exception, extract the NVList from the request object using the `CORBA::Request::result` member function.

If you send a request using the deferred synchronous method, you can use any of the following member functions to get the result:

- `CORBA::ORB::poll_response`

  This member function determines whether a request has completed and is ready to be processed. This member function does not block. If the request is ready, the client application has to use the `get_response()` or `get_next_response()` member functions to process the response. Use this member function when you don't care about the order in which reponses are processed, you want the client application to process other requests while waiting for a specific response, or you want to impose a timeout.

- `CORBA::ORB::poll_next_response`

  This member function indicates whether a response for any outstanding request is ready to be processed. If the request is ready, the client application has to use the `get_response()` or `get_next_response()` member functions to process the response. Use this member function when the order in which requests are processed is not important and you want the client application to process other requests while waiting for a specific response.

- `CORBA::ORB::get_reponse`

  This member function blocks until the reponse for the specific request is completed and processed. Use this member function when you want to process the requests for outstanding requests in a particular order.

- `CORBA::ORB::get_next_response`

  This member function blocks until a reponse for any outstanding requests are completed and processed. Use this member function when the order in which requests are processed is not important.

If you used the `CORBA::Request::send_oneway` member function, there is no result.

For a complete description of the CORBA member functions, see the *C++ Programming Reference*.

Summary of the Development Process for DII

The steps for using DII in client applications are as follows:

Step	Description
1	Load the CORBA interfaces into the Interface Repository.
2	Obtain an object reference for the CORBA object on which you want to invoke methods.
3	Create a request object for the CORBA object.
4	Send the DII request and retrieve the results.
5	Delete the request.
6	Use the Interface Repository with DII.

The following sections describe these steps in detail and provide C++ code examples.

Step 1: Loading the CORBA Interfaces into the Interface Repository

Before you can create CORBA client applications that use DII, the interfaces of the CORBA object need to be loaded into the Interface Repository. If the interfaces of a CORBA object are not loaded in the Interface Repository, they do not appear in the Services window of the Application Builder. If a desired CORBA interface does not appear in the Services window, use the `idl2ir` command to load the OMG IDL that defines the CORBA into the Interface Repository. The syntax for the `idl2ir` command is as follows:

```
idl2ir [-f repositoryfile.idl] file.idl
```

Option	Description
`-f` *`repositoryfile`*	Directs the command to load the OMG IDL files for the CORBA interface into the specified Interface Repository. Specify the name of the Interface Repository in the WLE domain that the ActiveX client application will access.
`file.idl`	Specifies the OMG IDL file containing definitions for the CORBA interface.

For a complete description of the `idl2ir` command, see the *C++ Programming Reference*.

Step 2: Obtaining the Object Reference for the CORBA Object

Use the Bootstrap object to get a FactoryFinder object. Then use the FactoryFinder object to get a factory for the CORBA object you want to access from the DII request. For an example of using the Boostrap and FactoryFinder objects to get a factory, see Chapter 2, "Creating CORBA Client Applications."

Step 3: Creating a Request Object

When your client application invokes a method on a CORBA object, you create a request for the method invocation. The request is written to a buffer and sent to the server application. When your client application uses client stubs, this processing occurs transparently. Client applications that want to use DII must create a request object and must send the request.

Using the CORBA::Object::_request Member Function

The following C++ code example illustrates how to use the `CORBA::Object::_request` member function:

```
Boolean          aResult;
CORBA::ULong     long1 = 42;
CORBA::Any       in_arg1;
CORBA::Any       &in_arg1_ref = in_arg1;

in_arg1 <<= long1;

// Create the request using the short form
Request_ptr reqp = anObj->_request("anOp");

// Use the argument manipulation helper functions
reqp->add_in_arg() <<= in_arg1_ref;
```

```
// We want a boolean result
reqp->set_return_type(_tc_boolean);

// Provide some place for the result
CORBA::Any::from_boolean boolean_return_in(aResult);
reqp->return_value() <<= boolean_return_in;

// Do the invoke
reqp->invoke();

// No error, so get the return value
CORBA::Any::to_boolean boolean_return_out(aResult);
reqp->return_value() >>= boolean_return_out;
```

Using the CORBA::Object::create_request Member Function

When you use the `CORBA::Object::create_request` member function to create a request object, you create an empty NVList and you add arguments to the NVList one at a time. You create the request object, passing the completed NVList as an argument to the request.

Setting Arguments for the Request Object

The arguments for a request object are represented with an NVList object that stores named/value objects. Methods are provided for adding, removing, and querying the objects in the list. For a complete description of `CORBA::NVList`, see the *C++ Programming Reference*.

Setting Input and Output Arguments with the CORBA::NamedValue Member Function

The `CORBA::NamedValue` member function specifies a named/value object that can be used to represent both input and output arguments for a request. The named/value objects are used as arguments to the request object. The `CORBA::NamedValue` pair is also used to represent the result of a request that is returned to the client application. The name property of a named/value object is a character string, and the value property of a named/value object is represented by a CORBA `Any`.

For a complete description of the `CORBA::NamedValue` member function, see the *C++ Programming Reference* available from the WLE online information set.

Example of Using CORBA::Object::create_request Member Function

The following C++ code example illustrates how to use the `CORBA::Object::create_request` member function:

```
CORBA::Request_ptr         reqp;
CORBA::Context_ptr         ctx;
CORBA::NamedValue_ptr      boolean_resultp = 0;
Boolean                    boolean_result;
CORBA::Any             boolean_result_any(CORBA::_tc_boolean, & boolean_result);
CORBA::NVList_ptr          arg_list = 0;
CORBA::Any                 arg;

// Get the default context
orbp->get_default_context(ctx);

// Create the named value pair for the result
(void) orbp->create_named_value(boolean_resultp);
CORBA::Any *tmpany = boolean_resultp->value();
*tmpany = boolean_result_any;

arg.replace(CORBA::_tc_long, &long_arg, CORBA_FALSE)

// Create the NVList
orbp->create_list(1, arg_list);

// Add an IN argument to the list
arg_list->add_value(“arg1”, arg, CORBA::ARG_IN);

// Create the request using the long form
anObj->_create_request (ctx,
                        “anOp”,
                        arg_list,
                        boolean_resultp,
                        reqp,
                        CORBA::VALIDATE_REQUEST );
// Do the invoke
reqp->invoke();

CORBA::NamedValue_ptrresult_namedvalue;
Boolean aResult;
CORBA::Any *result_any;
// Get the result
result_namedvalue = reqp->result();
result_any = result_namedvalue->value();
```

```
// Extract the Boolean from the any
*result_any >>= aResult;
```

Step 4: Sending a DII Request and Retrieving the Results

You can invoke a request in several ways, depending on what kind of communication type you want to use. This section describes how the CORBA member functions are used to send requests and retrieve the results.

Synchronous Requests

If you want synchronous communication, the `CORBA::Request::invoke` member function sends the request and waits for a response before it returns to the client application. Use the `CORBA::Request::result` member function to return a reference to a named/value object that represents the return value. Once the results are retrieved, you read the values from the NVList stored in the request.

Deferred Synchronous Requests

The nonblocking member function, `CORBA::Request::send_deferred`, is also provided for sending requests. It allows the client application to send a request and then use the `CORBA::Request::poll_response` member function to determine when the response is available. The `CORBA::Request::get_response` member function blocks until a response is available.

The following code example illustrates how to use the `CORBA::Request::send_deferred`, `CORBA::Request::poll_response`, and `CORBA::Request::get_response` member functions:

```
request->send_deferred ();

if (poll)
```

```
{
     for ( int k = 0 ; k < 10 ; k++ )
     {
         CORBA::Boolean done = request->poll_response();
         if ( done )
           break;

     }
}
request->get_response();
```

Oneway Requests

Use the `CORBA::Request::send_oneway` member function to send a oneway request. Oneway requests do not involve a response from the server application. For a complete description of the `CORBA::Request::send_oneway` member function, see the *C++ Programming Reference* available from the WLE online information set.

The following code example illustrates how to use the `CORBA::Request::send_oneway` member function:

```
request->send_oneway();
```

Multiple Requests

When a sequence of request objects is sent using the `CORBA::Request::send_multiple_requests_deferred` member function, the `CORBA::ORB::poll_response`, `CORBA::ORB::poll_next_response`, `CORBA::ORB::get_response`, and `CORBA::ORB::get_next_response` member functions can be used to retrieve the response the server application sends for each request.

The `CORBA::ORB::poll_response` and `CORBA::ORB::poll_next_response` member functions can be used to determine if a response has been retrieved from the server application. These member functions return a 1 if there is at least one response available, and a zero if there are no responses available.

The `CORBA::ORB::get_response` and `CORBA::ORB::get_next_response` member functions can be used to retrieve a response. If no response is available, these member functions block until a response is retrieved. If you do not want your client application to block, use the `CORBA::ORB::poll_next_response` member function to first determine when a response is available, and then use the `CORBA::ORB::get_next_response` method to retrieve the result.

You can also send multiple oneway requests by using the `CORBA::Request::send_multiple_requests_oneway` member function.

The following code example illustrates how to use the `CORBA::Request::send_multiple_requests_deferred`, `CORBA::Request::poll_next_response`, and `CORBA::Request::get_next_response` member functions:

```
CORBA::Context_ptr          ctx;
CORBA::Request_ptr          requests[2];
CORBA::Request_ptr          request;
CORBA::NVList_ptr           arg_list1, arg_list2;
CORBA::ULong                i, nreq;
CORBA::Long                 arg1 = 1;
Boolean                     aResult1 = CORBA_FALSE;
Boolean                     expected_aResult1 = CORBA_TRUE;
CORBA::Long                 arg2 = 3;
Boolean                     aResult2 = CORBA_FALSE;
Boolean                     expected_aResult2 = CORBA_TRUE

try
{
     orbp->get_default_context(ctx);

     populate_arg_list ( &arg_list1, &arg1, &aResult1 );

     nreq = 0;

     anObj->_create_request ( ctx,
                            “Multiply”,
                             arg_list1,
                             0,
                             requests[nreq++],
                             0);

     populate_arg_list ( &arg_list2, &arg2, &aResult2 );

     anObj->_create_request ( ctx,
                              “Multiply”,
                              arg_list2,
                              0,
                              requests[nreq++],
                              0 );
```

```
// Declare a request sequence variable...
CORBA::ORB::RequestSeq rseq ( nreq, nreq, requests, CORBA_FALSE );

orbp->send_multiple_requests_deferred ( rseq );
for ( i = 0 ; i < nreq ; i++ )

{
      requests[i]->get_response();
}

// Now check the results

if ( aResult1 != expected_aResult1 )
{
      cout << "aResult1=" << aResult1 << " different than expected: " <<
expected_aResult1;
}

if ( aResult2 != expected_aResult2 )
{
      cout << "aResult2=" << aResult2 << " different than expected: " <<
expected_aResult2;
}

aResult1 = CORBA_FALSE;
aResult2 = CORBA_FALSE;

// Using the same argument lists, multiply the numbers again.
// This time we intend to poll for response...

orbp->send_multiple_requests_deferred ( rseq );

// Now poll for response...

for ( i = 0 ; i < nreq ; i++ )
{

// We will randomly poll maximum 10 times...
     for ( int j = 0 ; j < 10 ; j++ )
     {
      CORBA::Boolean done = requests[i]->poll_response();

      if ( done ) break;
      }
}
// Now actually get the response...
for ( i = 0 ; i < nreq ; i++ )
{
     requests[i]->get_response();
}
```

```
// Now check the results
if ( aResult1 != expected_aResult1 )
{
      cout << “aResult1=” << aResult1 << “ different than expected: “ <<
expected_aResult1
}
if ( aResult2 != expected_aResult2 )
{
       cout << “aResult2=” << aResult2 << “ different than expected: “ <<
expected_aResult2;
}

aResult1 = CORBA_FALSE;
aResult2 = CORBA_FALSE;

// Using the same argument lists, multiply the numbers again.
// Call get_next_response, and WAIT for a response.
orbp->send_multiple_requests_deferred ( rseq );

// Poll until we get a response and then use get_next_response get it...
for ( i = 0 ; i < nreq ; i++ )
     {
           CORBA::Boolean res = 0;

           while ( ! res )
           {
             res = orbp->poll_next_response();
           }
           orbp->get_next_response(request);
           CORBA::release(request);
     }
// Now check the results
if ( aResult1 != expected_aResult1 )
{
      cout << “aResult1=” << aResult1 << “ different than expected: “ <<
expected_aResult1;
}
if ( aResult2 != expected_aResult2 )
{
      cout << “aResult2=” << aResult2 << “ different than expected: “ <<
expected_aResult2;
}



static void populate_arg_list (
CORBA::NVList_ptr           ArgList,
CORBA::Long                 * Arg1,
CORBA::Long                 * Result )
{
```

```
CORBA::Any                    any_arg1;
CORBA::Any                    any_result;

(* ArgList) = 0;
orbp->create_list(3, *ArgList);

any_arg1 <<= *Arg1;
any_result.replace(CORBA::_tc_boolean, Result, CORBA_FALSE);

(*ArgList)->add_value(“arg1”, any_arg1, CORBA::ARG_IN);
(*ArgList)->add_value(“result”, any_result, CORBA::ARG_OUT);

return;

}
```

Step 5: Deleting the Request

Once you have been notified that the request has successfully completed, you need to decide if you want to delete the existing request, or reuse portions of the request in the next invocation.

To delete the entire request, use the `CORBA::Release(request)` member function on the request to be deleted. This operation releases all memory associated with the request. Deleting a request that was issued using the deferred synchronous communication type causes that request to be canceled if it has not completed.

Step 6: Using the Interface Repository with DII

A client application can create, populate, and send requests for objects that were not known to the client application when it was built. To do this, the client application uses the Interface Repository to retrieve information needed to create and populate the requests. The client application uses DII to send the requests, since it does not have client stubs for the interfaces.

Although this technique is useful for invoking operations on a CORBA object whose type is unknown, performance becomes an issue because of the overhead interaction with the Interface Repository. You might consider using this type of DII request when creating a client application that browses for objects, or when creating a client application that is an administration tool.

The steps for using the Interface Repository in a DII request are as follows:

1. Set `ORB_INCLUDE_REPOSITORY` in `CORBA.h` to the location of the Interface Repository file in your WLE system.
2. Use the Bootstrap object to obtain the InterfaceRepository object, which contains a reference to the Interface Repository in a particular WLE domain. Once the reference to the Interface Repository is obtained, you can navigate the Interface Repository from the root.
3. Use the `CORBA::Object::_get_interface` member function to communicate with the server application that implements the desired CORBA object.
4. Use `CORBA::InterfaceDef_ptr` to get the definition of the CORBA interface that is stored in the Interface Repository.
5. Locate the `OperationDescription` for the desired operation in the `FullInterfaceDescription` operations.
6. Retrieve the repository ID from the `OperationDescription`.
7. Call `CORBA::Repository::lookup_id` using the repository ID returned in the `OperationDescription` to look up the `OperationDef` in the Interface Repository. This call returns the contained object.
8. Narrow the contained object to an `OperationDef`.
9. Use the `CORBA::ORB::create_operation_list` member function, using the `OperationDef` argument, to build an argument list for the operation.
10. Set the argument value within the operation list.
11. Send the request and retrieve the results as you would any other request. You can use any of the options described in this topic to send a request and to retrieve the results.

7 Handling Exceptions

This topic describes how CORBA C++, CORBA Java, and ActiveX client applications handle CORBA exceptions.

CORBA Exception Handling Concepts

CORBA defines the following types of exceptions:

- **System exceptions,** which are general errors, such as running out of memory and communication failures. System exceptions include exceptions raised by the object request broker (**ORB**). The CORBA specification defines a set of system exceptions that can be raised when errors occur in the processing of a request from a client application.
- **User exceptions,** which are exceptions triggered by an object, where the exception contains user-defined data. When you define your CORBA object's interface in **OMG IDL**, you can specify the user exceptions that the object may raise.

The following sections describe how each type of client application handles exceptions.

CORBA System Exceptions

Table 7-1 lists the CORBA system exceptions.

Table 7-1 CORBA System Exceptions

Exception Name	Description
BAD_CONTEXT	An error occured while processing context objects.
BAD_INV_ORDER	Routine invocations are out of order.
BAD_OPERATION	Invalid operation.
BAD_PARAM	An invalid parameter was passed.
BAD_TYPECODE	Invalid typecode.
COMM_FAILURE	Communication failure.
DATA_CONVERSION	Data conversion error.
FREE_MEM	Unable to free memory.
IMP_LIMIT	Implementation limit violated.
INITIALIZE	ORB initialization failure.
INTERNAL	ORB internal error.
INTF_REPOS	An error occurred while accessing the Interface Repository.
INV_FLAG	Invalid flag was specified.
INV_IDENT	Invalid identifier syntax.
INV_OBJREF	Invalid object reference was specified.
MARSHAL	Error marshaling parameter or result.
NO_IMPLEMENT	Operation implementation not available.
NO_MEMORY	Dynamic memory allocation failure.
NO_PERMISSION	No permission for attempted operation.
NO_RESOURCES	Insufficient resources to process request.
NO_RESPONSE	Response to request not yet available.

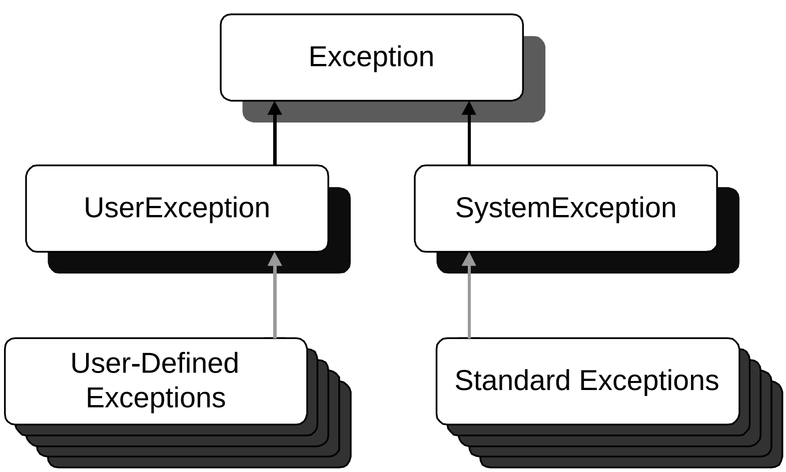
Table 7-1 CORBA System Exceptions

Exception Name	Description
`OBJ_ADAPTER`	Failure detected by object adapter.
`OBJECT_NOT_EXIST`	Object is not available.
`PERSIST_STORE`	Persistent storage failure.
`TRANSIENT`	Transient failure.
`UNKNOWN`	Unknown result.

CORBA C++ Client Applications

Since both system and user exceptions require similar functionality, the `SystemException` and `UserException` classes are derived from the common `Exception` class. When an exception is raised, your client application can narrow from the `Exception` class to a specific `SystemException` or `UserException` class. The C++ Exception inheritance hierarchy is shown in Figure 7-1.

Figure 7-1 C++ Exception Inheritance Hierarchy



The `Exception` class provides the following public operations:

- `copy constructor`
- `destructor`
- `_narrow`

The `copy constructor` and `destructor` operations automatically manage the storage associated with the exception.

The `_narrow` operation allows your client application to catch any type of exception and then determine its type. The `exception` argument passed to the `_narrow` operation is a pointer to the base class `Exception`. The `_narrow` operation accepts a pointer to any Exception object. If the pointer is of type `SystemException`, the `narrow()` operation returns a pointer to the exception. If the pointer is not of type `SystemException`, the `narrow()` operation returns a `Null` pointer.

Unlike the `_narrow` operation on object references, the `_narrow` operation on exceptions returns a suitably typed pointer to the same exception argument, not a pointer to a new exception. Therefore, you do not free a pointer returned by the `_narrow` operation. If the original exception goes out of scope or is destroyed, the pointer returned by the `_narrow` operation is no longer valid.

Note: The WLE sample applications do not use the `_narrow` operation.

Handling System Exceptions

The CORBA C++ client applications in the WLE sample applications use the standard C++ try-catch exception handling mechanism to raise and catch exceptions when error conditions occur, rather than testing status values to detect errors. This exception-handling mechanism is also used to integrate CORBA exceptions into WLE client applications. In C++, `catch` clauses are attempted in the order specified, and the first matching handler is called.

The following example from the C++ client application in the Basic sample application shows printing an exception using the << operator.

Note: Throughout this topic, bold text is used to highlight the exception code within the example.

```
try{

//Initialize the ORB
CORBA::ORB* orb=CORBA::ORB_init(argc, argv, ORBid);

//Get a Bootstrap Object
Tobj_Bootstrap* bs= new Tobj_Bootstrap(orb, "//host:port");

//Resolve Factory Finder
CORBA::Object_var var_factory_finder_oref = bs->
     resolve_initial_reference("FactoryFinder");
Tobj::FactoryFinder_var var_factory_finder_ref = Tobj::FactoryFinder::_narrow
     (var_factory_finder_oref.in());

catch(CORBA::Exception& e) {
     cerr <<e.get_id() <<end1;
}
```

User Exceptions

User exceptions are generated from the user-written OMG IDL file in which they are defined. When handling exceptions, the code should first check for system exceptions. System exceptions are predefined by CORBA, and often the application cannot recover from a system exception. For example, system exceptions may signal problems in the network transport or signal internal problems in the ORB. Once you have tested for the system exceptions, test for specific user exceptions.

The following C++ example shows the OMG IDL file that declares the `TooManyCredits` user exception inside the `Registar` interface. Note that exceptions can be declared either within a module or within an interface.

```
exception TooManyCredits
{
    unsigned short maximum_credits;
};

interface Registrar

NotRegisteredList register_for_courses(
    in StudentId                    student,
    in CourseNumberList             courses
) raises (
    TooManyCredits
);
```

The following C++ code example shows how a `TooManyCredits` user exception would work within the scope of a transaction for registering for classes:

```
//Register a student for some course

try {
     pointer_registrar_reference->register_for_courses
         (student_id, course_number_list);

catch (UniversityT::TooManyCredits& e) {
    cout <<"You cannot register for more than"<< e.maximum_credits
    <<"credits."<<end1;
}
```

CORBA Java Client Applications

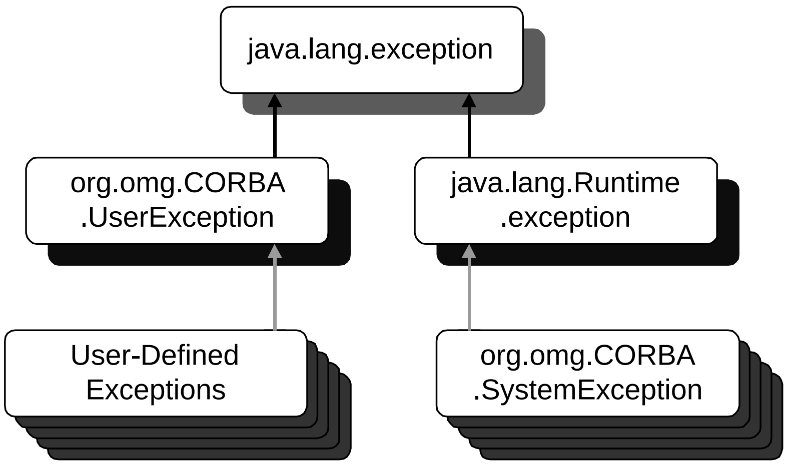
Note: The information in this section is based on the OMG *IDL/Java Language Mapping Specification, orbos/97-03-01.* Revised: March 19, 1997.

Java client applications handle exceptions in a similar way to C++ client applications:

- System exceptions inherit from `java.lang.RuntimeException`.
- User-defined exceptions inherit from `java.lang.Exception`.

Figure 7-2 shows the inheritance hierarchy of the Java Exception classes.

Figure 7-2 Java Exception Inheritance Hierarchy



System Exceptions

The standard OMG IDL system exceptions are mapped to final Java classes that extend `org.omg.CORBA.SystemException` and provide access to the OMG IDL major and minor exception code, as well as a string describing the reason for the exception.

Note: There are no public constructors for `org.omg.CORBA.SystemException`; only classes that extend it can be instantiated.

The Java class name for each standard OMG IDL exception is the same as its OMG IDL name and is declared to be in the `org.omg.CORBA` package. For example, the CORBA-defined system exception `BAD_CONTEXT` maps to Java as `org.omg.CORBA.BAD_CONTEXT`. The default constructor supplies zero for the minor code, `COMPLETED_NO` for the completion code, and `""` for the reason string. There is also a constructor that takes the reason and uses defaults for the other fields, as well as a constructor that requires all three parameters to be specified.

The following Java code example illustrates how to use system exceptions:

```
try
 {
  //Resolve FactoryFinder
  org.omg.CORBA.Object off = bs.resolve_initial_references
     (“FactoryFinder”);
  FactoryFinder ff=FactoryFinderHelper.narrow(off);

  org.omg.CORBA.Object of = FactoryFinder.find_one_factory_by_id
    (UniversityT.RegistrarFactoryHelper.id());
  UniversityT.RegistrarFactory F =
     UniversityT.RegistrarFactoryHelper.narrow(of);

catch (org.omg.CORBA.SystemException e)
   {
        System.err.println("System exception " + e);
        System.exit(1);
}
```

User Exceptions

User exceptions are mapped to final Java classes that extend `org.omg.CORBA.UserException` and are otherwise mapped like the OMG IDL `struct` data type, including the generation of Helper and Holder classes.

If the exception is defined within a nested OMG IDL scope (essentially within an interface), its Java class name is defined within a special scope. Otherwise, its Java class name is defined within the scope of the Java package that corresponds to the exceptions’s enclosing OMG IDL module.

The following is an example of a user exception:

```
//Register for Courses
try{
   gRegistrarObjRef.register_for_courses(student_id, selected_course_numbers);

catch(UniversityT.TooManyCredits e)
{
       System.err.println("TooManyCredits: " +e);
       System.exit(1);
}
```

ActiveX Client Applications

ActiveX client applications use the Visual Basic error handling model, which allows you to perform special actions when an error occurs, such as jumping to a particular error handling routine. When an exception occurs in an ActiveX client application, the standard Visual Basic error handling works as expected; however, the amount of error information that Visual Basic returns for any exceptions is very limited.

Visual Basic provides additional information about the exception that occurred through the `description` property of the Visual Basic built-in `Error` object. When an error occurs, the description string is set to indicate what type of error occurred. The object returns a predefined data type for the exceptions. User exceptions are named to distinguish between them.

When using the Visual Basic error handling model, the description string describes the following:

- Whether the exception was a user or a system exception
- The name of the exception
- Whether or not the operation completed before the exception occurred

The Visual Basic error handling model cannot return exception-specific information, such as the user data of a user exception.

To compensate for this shortcoming, ActiveX views of CORBA objects have an additional optional exception return parameter that returns a user exception. When you supply the optional exception object, no Visual Basic exception is triggered. Instead, the return parameter returns the exception information.

If an exception occurs, the return parameter contains an object to get the data from the exception. This object is similar to a structure pseudo-object, with properties for each value. To determine the type of exception, use the exception object properties `EX_majorCode` or `EX_minorCode`. The `EX_majorCode` object property has three possible values:

- 0 when no exception occurred
- 1 when a system exception occurred
- 2 when a user exception occurred

The following is an example of Visual Basic code that handles exceptions:

```
Dim exceptType As ExceptionType
Dim exceptInfo As DIForeignException

Set exceptInfo = Exc
exceptType = exceptInfo.EX_majorCode

Select Case exceptType

     Case NO_EXCEPTION

            msg = "No Exception" & vbCrLf

            MsgBox msg

     Case SYSTEM_EXCEPTION

            'For a system exception, the returned variant supports the
            'minorCode and completionStatus properties.

            Dim minorCode As Long
            Dim completionStatus As CORBA_CompletionStatus
            Dim completionMsg As String

            minorCode = exceptInfo.EX_minorCode
            completionStatus = exceptInfo.EX_completionStatus
            Select Case completionStatus
                Case COMPLETION_NO
                     completionMsg = "No"
                Case COMPLETION_YES
                     completionMsg = "Yes"
                Case COMPLETION_MAYBE
                     completionMsg = "Maybe"
            End Select

            msg = "System Exception" & vbCrLf
            msg = msg & "    Minor Code = " & minorCode & vbCrLf
            msg = msg & "    Completion Status = " & completionMsg & vbCrLf

            MsgBox msg
```

```
Case USER_EXCEPTION

    'If it is a user exception, the returned variant supports
    'the properties for the defined user exceptions.

    msg = "User Exception" & vbCrLf
    msg = msg & "    Exception: " & exceptInfo.INSTANCE_repositoryId &
    vbCrLf
    MsgBox msg

End Select
```

Index

C

D

E

F

H

I

J

M

N

O

P

R

S

T

U

V

W