BEA Tuxedo®

Creating CORBA Client Applications

Version 10.0
Document Released: August 28, 2007

Contents

3. Using the Dynamic Invocation Interface

4. Handling Exceptions

CHAPTER 1

CORBA Client Application Development Concepts

This topic reviews the types of client applications supported by the CORBA environment in the BEA Tuxedo product and introduces the concepts that you need to understand before you develop CORBA client applications.

Notes: The BEA Tuxedo CORBA Java client and BEA Tuxedo CORBA Java client ORB were deprecated in Tuxedo 8.1 and are no longer supported. All BEA Tuxedo CORBA Java client and BEA Tuxedo CORBA Java client ORB text references, associated code samples, should only be used to help implement/run third party Java ORB libraries, and for programmer reference only.

Technical support for third party CORBA Java ORBs should be provided by their respective vendors. BEA Tuxedo does not provide any technical support or documentation for third party CORBA Java ORBs.

This topic includes the following sections:

Overview of Client Applications

The BEA Tuxedo software supports the following types of client applications:

- CORBA C++

  This type of client application uses C++ environmental objects to access the CORBA objects in a BEA Tuxedo domain and the CORBA C++ Object Request Broker (ORB) to process requests to CORBA objects. Use the BEA Tuxedo development commands to build CORBA C++ client applications.CORBA C++ client applications now support object by value and the CORBA Interoperable Naming Service (INS).

  Note: See *Installing the BEA Tuxedo System* for the specific versions of supported software.

OMG IDL

With any distributed application, the client/server application needs some basic information to communicate. For example, the CORBA 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 BEA Tuxedo software conforms to The Common Object Request Broker:Architecture and Specification, Version 2.3. For complete information about the OMG IDL-to-C++ mapping, see *The Common Object Request Broker:Architecture and Specification*, Version 2.3.

OMG IDL-to-Java Mapping

The BEA Tuxedo 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.

OMG IDL-to-COM Mapping

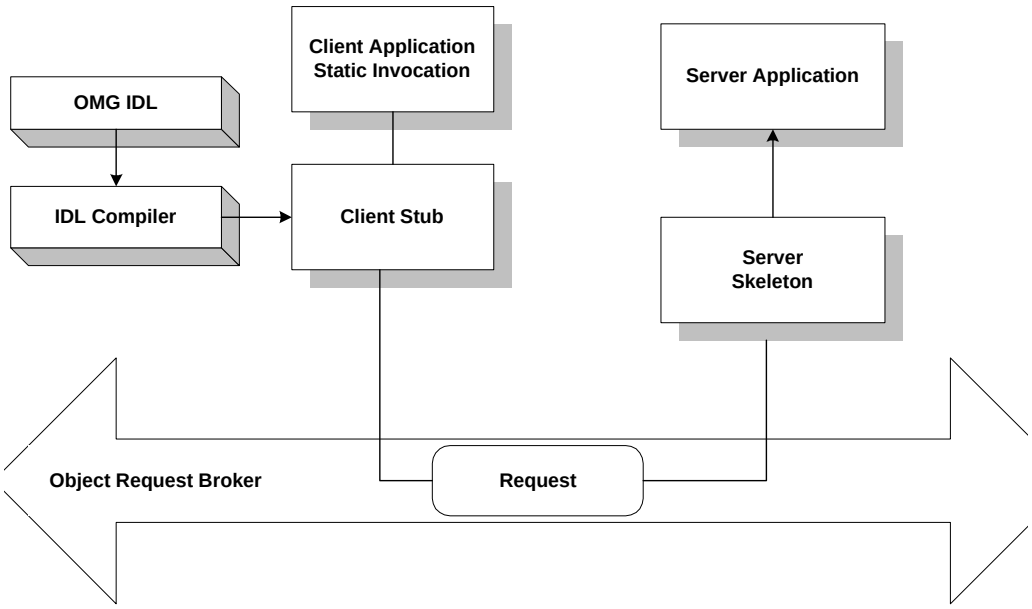
The BEA Tuxedo software conforms to the OMG IDL to COM mapping as defined in the Common Object Request Broker:Architecture and Specification, Version 2.3. For complete information about the OMG IDL to COM mapping, see *The Common Object Request Broker:Architecture and Specification*, Version 2.3.

Static and Dynamic Invocation

The CORBA ORB in the BEA Tuxedo product supports two types of client/server invocations: static and dynamic. In both cases, the CORBA client application performs a request by gaining access to a reference for a CORBA object and invoking the operation that satisfies the request. The CORBA server application cannot tell the difference between static and dynamic invocations.

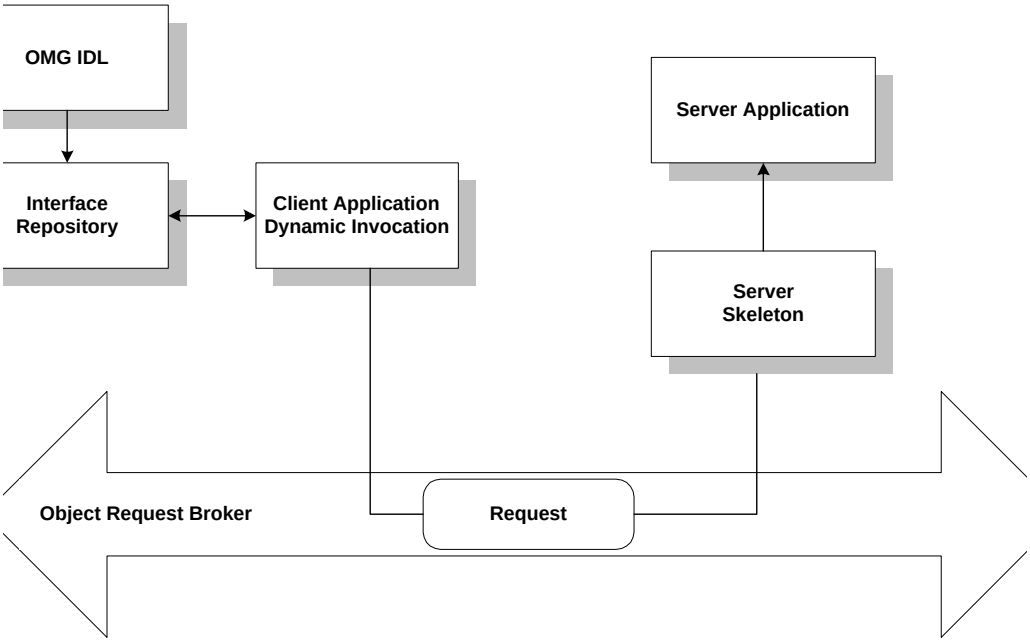
When using static invocation, the CORBA client application invokes operations directly on the client stubs. Static invocation is the easiest, most common type of invocation. The stubs are generated by the IDL compiler. Static invocation is recommended for applications that know at compile time the particulars 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 CORBA 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 CORBA client application can dynamically build operation requests for a CORBA object interface that has been stored in the Interface Repository. CORBA 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 CORBA 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 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 CORBA 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 CORBA client application must have a stub for each interface it plans to use. You use the `idl` command (or your Java ORB 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 CORBA client application to request an operation on the CORBA object.

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. CORBA client applications use the Interface Repository for the following reasons:

- 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.

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.

You use the following BEA Tuxedo 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 *BEA Tuxedo Command Reference*.

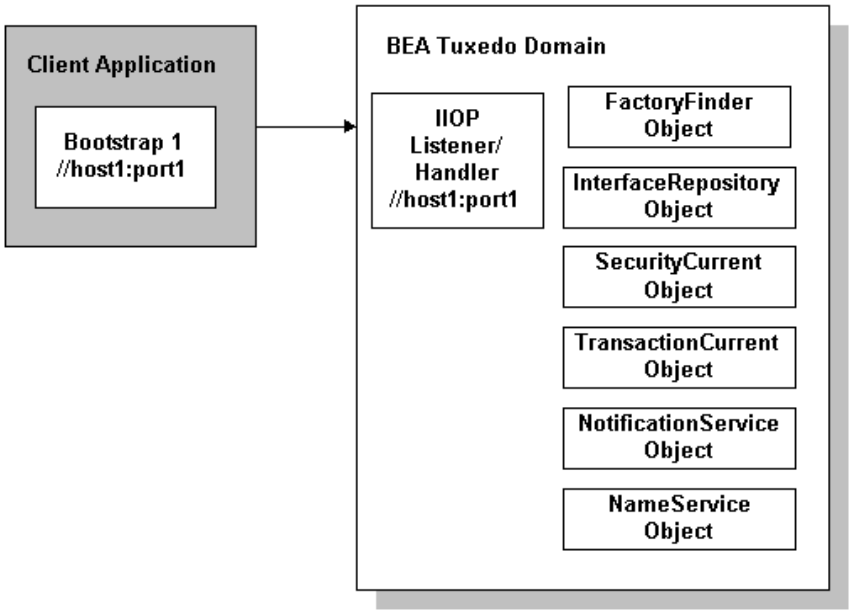
Domains

A domain is a way of grouping objects and services together as a management entity. A BEA Tuxedo domain has at least one IIOP Listener/Handler and is identified by a name. One CORBA client application can connect to multiple BEA Tuxedo domains using different Bootstrap objects. For each BEA Tuxedo domain, a CORBA client application can get objects which correspond to the services (for example, transactions, security, naming, events) offered within the BEA Tuxedo domain. For a description of the Bootstrap object and the CORBA services available in a BEA Tuxedo domain, see Environmental Objects.

Note: Only one environmental object per service can exist at the same time and the environmental objects must be associated with the same Bootstrap object.

Figure 1-3 illustrates how a BEA Tuxedo domain works.

Figure 1-3 How a BEA Tuxedo Domain Works



Environmental Objects

The BEA Tuxedo software provides a set of environmental objects that set up communication between CORBA client and server applications in a BEA Tuxedo domain and provide access to the CORBA services provided by the domain. The BEA Tuxedo software provides the following environmental objects:

- Bootstrap

  This object establishes communication between a CORBA client application and a BEA Tuxedo domain. It also obtains object references for the other environmental objects in the BEA Tuxedo domain.

Note: Third-party client ORBs can also use the CORBA Interoperable Naming Service (INS) to access the services within a BEA Tuxedo domain. For more information, see the "CORBA Bootstrap Object Programming Reference" topic in the *CORBA Programming Reference*.

- FactoryFinder

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

- 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.

- SecurityCurrent

  This BEA-proprietary object is used to log a CORBA client application into a BEA Tuxedo domain with the proper security credentials. The BEA Tuxedo software provides an implementation of the CORBAservices Security Service.

- TransactionCurrent

  This BEA-proprietary object allows a CORBA client application to participate in a transaction. The TransactionCurrent object provides an implementation of the CORBAservices Object Transaction Service (OTS).

- NotificationService

  This CORBA object allows a CORBA client application to obtain a reference to the event channel factory (`CosNotifyChannelAdmin::EventChannelFactory`) in the CosNotification Service. In turn, the EventChannelFactory is used to locate the Notification Service channel.

  In addition, a `Tobj_SimpleEventsService` object is provided. This BEA-proprietary object allows a CORBA client application to obtain a reference to a BEA-proprietary events interface. The events interface passes standard, structured events as defined by the CosNotification Service, however, the API has been simplified for easier use.

- NameService

  This CORBA object allows a CORBA client application to use a namespace to resolve object references. The BEA Tuxedo software provides an implementation of the CORBAservices Name Service.

The BEA Tuxedo software provides environmental objects for the following programming environments:

- C++
- Java
- Automation

Bootstrap Object

A CORBA client application creates a Bootstrap object which defines the address of an IIOP Listener/Handler. The IIOP Listener/Handler is the access point to a BEA Tuxedo domain and the CORBA services provided by the domain. A list of IIOP Listener/Handlers can be supplied either as a parameter or via the `TOBJADDR` environmental variable or a Java property. A single IIOP Listener/Handler is specified as follows:

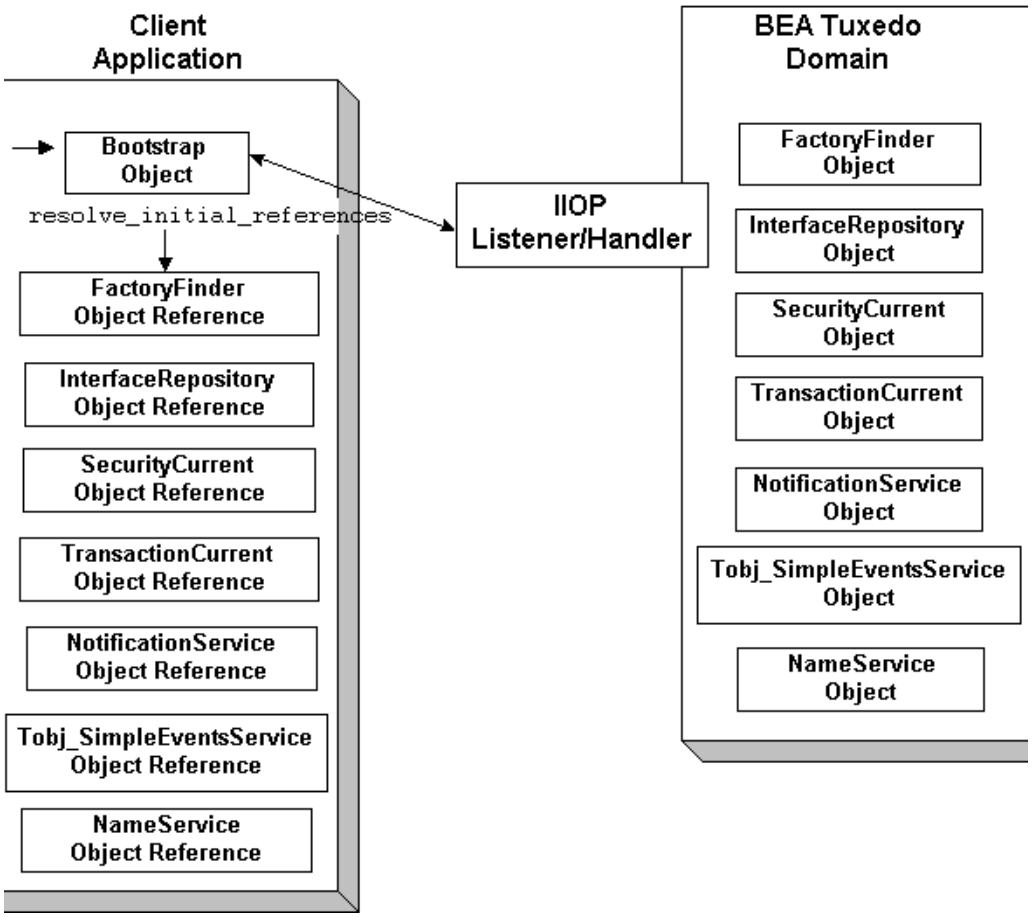
`//host:port`

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, Interface Repository, SecurityCurrent, TransactionCurrent, NotificationService, TObj_SimpleEventsService, and NameService.

Figure 1-4 illustrates how the Bootstrap object works in a BEA Tuxedo domain.

Figure 1-4 How the Bootstrap Object Works



Third-party client ORBs can also use the CORBA Interoperable Naming Service (INS) mechanism to gain access to a BEA Tuxedo domain and its services. The Interoperable Naming Service allows third-party client ORBs to use their ORB's `resolve_initial_references()` function to access CORBA services provided by the BEA Tuxedo domain and use stubs generated from standard OMG IDL to act on the instances returned from the domain. For more information about using the Interoperable Naming Service, see the *CORBA Programming Reference*.

Factories and the FactoryFinder Object

CORBA 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.

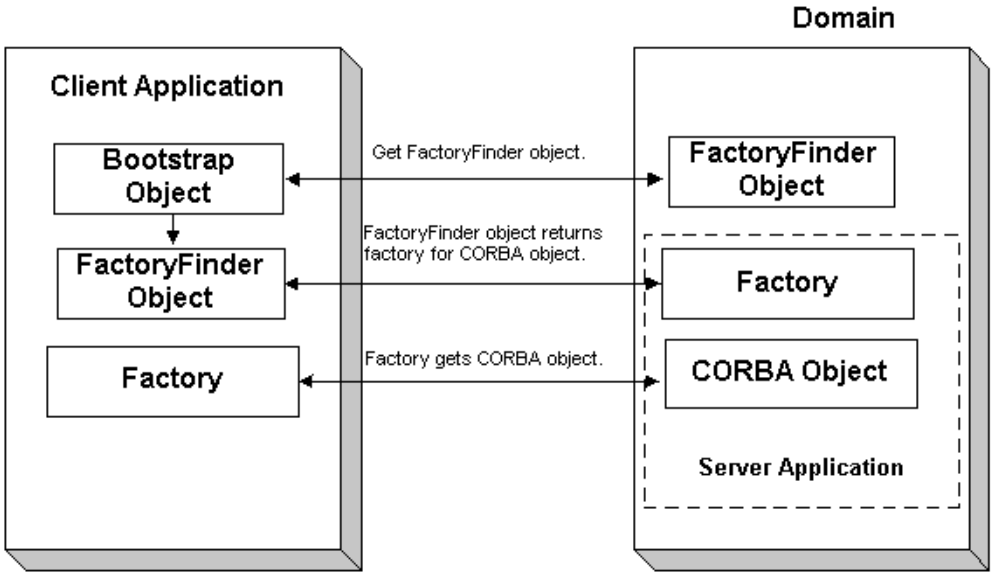
To use a CORBA object, the CORBA client application must be able to locate the factory that creates an object reference for the CORBA object. The BEA Tuxedo software offers the FactoryFinder object for this purpose. The factories available to CORBA client applications are those that are registered with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

The CORBA 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. CORBA client applications query the FactoryFinder object for object references to the desired factory.
3. CORBA client applications then call the factory to obtain an object reference to the CORBA object.

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

Figure 1-5 How Client Applications Use the FactoryFinder Object



Naming Conventions and BEA Tuxedo Extensions to the FactoryFinder Object

The factories available to CORBA client applications are those that are registered with the FactoryFinder object by the CORBA 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 BEA Tuxedo software is defined in the CORBAservices Life Cycle Service. The BEA Tuxedo 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 CORBA Name 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 CORBA client applications to locate factories by extending the interface with simpler method calls.

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

- Let the CORBA client application locate factories by ID, using a simple string parameter for the ID field. This reduces the work needed by the CORBA 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 CORBA 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 CORBA client applications. This method accepts a simple string for factory ID as input and returns one factory to the CORBA client application. The CORBA 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 CORBA 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 CORBA Server Applications*.

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

InterfaceRepository Object

The InterfaceRepository object returns information about the Interface Repository in a BEA Tuxedo 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 CORBA client application is seeking information about a specific CORBA interface or browsing the Interface Repository to find an interface. In either case, the CORBA 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 BEA Tuxedo domain, the CORBA 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.

AFor information about using the InterfaceRepository object in CORBA client applications that use DII, see Using the Dynamic Invocation Interface For a description of the InterfaceRepository object, see the *CORBA Programming Reference*.

SecurityCurrent Object

CORBA C++ client applications use security to authenticate themselves to the BEA Tuxedo 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 BEA Tuxedo domain. The BEA Tuxedo software uses authentication as defined in the CORBAservices Security Service and provides extensions for ease of use.

CORBA client applications use the SecurityCurrent object to log on to the BEA Tuxedo domain and pass security credentials to the domain. The SecurityCurrent object is an BEA Tuxedo implementation of the CORBAservices Security Service. The CORBA security model in the BEA Tuxedo product is based on authentication.

You use the SecurityCurrent object to specify the appropriate level of security for the domain. The following levels of authentication are provided:

- TOBJ_NOAUTH

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

- TOBJ_SYSAUTH

  The CORBA client application must authenticate itself to the BEA Tuxedo domain and must specify a username, client application name, and application password.

- TOBJ_APPAUTH

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

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

In the BEA Tuxedo software, only the PrincipalAuthenticator and Credentials properties on the SecurityCurrent object are supported.

For information about using the SecurityCurrent object in client applications, see *Using Security in CORBA Applications*. For a description of the `SecurityLevel1::Current` and `SecurityLevel2::Current` interfaces, refer to the *CORBA Programming Reference*.

TransactionCurrent Object

The TransactionCurrent object is an BEA Tuxedo implementation of the CORBAservices Object Transaction Service. The TransactionCurrent object maintains a transactional context for the current session between the CORBA client application and the CORBA server application. Using the TransactionCurrent object, the CORBA 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 a CORBA 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 BEA Tuxedo 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 CORBA client applications, see *Using CORBA Transactions*. For a description of the TransactionCurrent object, see the *CORBA Programming Reference*.

NotificationService and Tobj_SimpleEventsService Objects

The `NotificationService` and `Tobj_SimpleEventsService` objects provide access to a CORBA event service. The event service in the CORBA environment of the BEA Tuxedo product offers similar capabilities to those of the EventBroker in the ATMI environment. However, the CORBA event service offers a programming model and interface that is natural for CORBA programmers.

The event service receives event posting messages, filters them, and distributes them to subscribers. A poster is a CORBA application that detects when an event of interest has occurred and reports (posts) it to the event service. A subscriber is a CORBA application that requests some notification action to be taken when an event of interest is posted.

The CORBA event service provides two sets of interfaces:

- The `NotificationService` object provides a minimal subset of the CORBA-based Notification Service interfaces (referred to as the `CosNotification` Service interface).
- The `Tobj_SimpleEventsService` object provides BEA-proprietary interfaces designed to be easy to use.

Both sets of interfaces pass standard, structured events as defined by the CORBA Notification Service specification. The two sets of interfaces are compatible with each other; that is, events posted using the NotificationService interfaces can be subscribed to by the `Tobj_SimpleEventsService` interfaces and vice versa.

For information about using the NotificationServer and Tobj_SimpleEventsService objects, see *Using the CORBA Notification Service*.

NameService Object

The NameService object provides access to a CORBA Name Service which allows CORBA server applications to advertise object references using logical names. CORBA client applications can then locate an object by asking the CORBA Name Service to look up the name.

The CORBA Name Service provides:

- An implementation of the Object Management Group (OMG) Interoperable Name Service (INS) specification.
- Application programming interfaces (APIs) for mapping object references into an hierarchical naming structure (referred to as a namespace).
- Commands for displaying bindings and for binding and unbinding naming context objects and application objects into the namespace.

For information about using the NameService object in a CORBA client application, see *Using the CORBA Name Service*.

CHAPTER 2

Creating CORBA Client Applications

This topic includes the following sections:

Notes: The BEA Tuxedo CORBA Java client and BEA Tuxedo CORBA Java client ORB were deprecated in Tuxedo 8.1 and are no longer supported. All BEA Tuxedo CORBA Java client and BEA Tuxedo CORBA Java client ORB text references, associated code samples, should only be used to help implement/run third party Java ORB libraries, and for programmer reference only.

Technical support for third party CORBA Java ORBs should be provided by their respective vendors. BEA Tuxedo does not provide any technical support or documentation for third party CORBA Java ORBs.

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.
5	Build the CORBA C++ client application.

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

The BEA Tuxedo 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 a BEA Tuxedo 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 CORBA 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 CORBA client application. You can use both types of invocation in a CORBA client application.

For an overview of static and dynamic invocation, see Static and Dynamic Invocation.

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 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 CORBA 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 CORBA 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 *BEA Tuxedo Command Reference*.

When creating CORBA 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 CORBA 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 a CORBA server application, a CORBA 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 BEA Tuxedo ORB.
2. Establish communication with the BEA Tuxedo 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 the CORBA University Sample Applications*. The Basic sample application is located in the following directory on the BEA Tuxedo software kit:

`drive:``\tuxdir\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 CORBA client application is running on a machine that also has CORBA server applications running and the CORBA client application wants to access server applications in another BEA Tuxedo 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`.

Establishing Communication with the BEA Tuxedo Domain

The CORBA 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 BEA Tuxedo 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 address>/<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 2000, 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 Applet

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

where *this* is the name of the Java applet

A BEA Tuxedo domain can have multiple IIOP Listener/Handlers. If you are accessing a BEA Tuxedo 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 a BEA Tuxedo domain. The list of IIOP Listener/Handlers can also be specified in `TOBJADDR`.

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

Note: Third-party client ORBs can also use the CORBA Interoperable Naming Service (INS) mechanism to gain access to a BEA Tuxedo domain and its services. CORBA INS allows third-party client ORBs to use their ORB's `resolve_initial_references()` function to access CORBA services provided by the BEA Tuxedo domain and use stubs generated from standard OMG IDL to act on the instances returned from the domain. For more information about using the Interoperable Naming Service, see the *CORBA Programming Reference*.

Resolving Initial References to the FactoryFinder Object

The CORBA client application must obtain initial references to the environmental objects that provide services for the CORBA application. The Bootstrap object's `resolve_initial_references` operation can be called to obtain references to the FactoryFinder, InterfaceRepository, SecurityCurrent, TransactionCurrent, NotificationService, Tobj_SimpleEventsService, and NameService 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 CORBA 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);
```

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 CORBA 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 CORBA 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 CORBA 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 CORBA 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

CORBA client applications call the factory to get an object reference to a CORBA object. The CORBA 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 for the 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 a CORBA client application executable. The command combines the client stubs for interfaces that use static invocation, and the associated header files with the standard BEA Tuxedo libraries to form a client executable. For the syntax of the `buildobjclient` command, see the *BEA Tuxedo Command Reference*.

Server Applications Acting as Client Applications

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

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

Using Java2 Applets

The CORBA environment in the BEA Tuxedo product supports Java2 applets as clients. 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, 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`.

CHAPTER 3

Using the Dynamic Invocation Interface

This topic includes the following sections:

- When to Use DII
- DII Concepts
- Summary of the Development Process for DII
- Step 1: Loading the CORBA Interfaces into the Interface Repository
- Step 2: Obtaining the Object Reference for the CORBA Object
- Step 3: Creating a Request Object
- Step 4: Sending a DII Request and Retrieving the Results
- Step 5: Deleting the Request
- Step 6: Using the Interface Repository with DII

The information in this topic applies to CORBA C++client applications.

For an overview of the invocation types and DII, see Static and Dynamic Invocation.

For a complete description of the CORBA member functions mentioned in this topic, see the *CORBA Programming Reference*.

Notes: The BEA Tuxedo CORBA Java client and BEA Tuxedo CORBA Java client ORB were deprecated in Tuxedo 8.1 and are no longer supported. All BEA Tuxedo CORBA Java client and BEA Tuxedo CORBA Java client ORB text references, associated code samples, should

only be used to help implement/run third party Java ORB libraries, and for programmer reference only.

Technical support for third party CORBA Java ORBs should be provided by their respective vendors. BEA Tuxedo does not provide any technical support or documentation for third party CORBA Java ORBs.

When to Use DII

There are good reasons to use either static or dynamic invocation to send requests from the CORBA client application. You may find you want to use both invocation types in the same CORBA client application. 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 CORBA client application sends a request and waits until a response is retrieved; the CORBA client application cannot do any other work while it is waiting for the response. In deferred synchronous communication, the CORBA client application sends the request and is free to do other work. Periodically, the CORBA client application checks to see if the request has completed; when the request has completed, the CORBA client application makes use of the result of that request.

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

A CORBA client application can use DII to obtain better performance. For example, the CORBA 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 applications, 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 CORBA client application; static and dynamic invocations are identical from a CORBA 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 CORBA 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 *CORBA Programming Reference*.

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 CORBA 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 *CORBA Programming Reference*.

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 CORBA 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 responses are processed, you want the CORBA 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 CORBA 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 CORBA client application to process other requests while waiting for a specific response.

- `CORBA::ORB::get_response`

  This member function blocks until the response 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 response 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 *CORBA 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 BEA 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 object 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 BEA Tuxedo domain that the CORBA 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 *BEA Tuxedo Command 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 Step 4: Writing the CORBA Client Application.

Step 3: Creating a Request Object

When your CORBA 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 CORBA server application. When your CORBA 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 *CORBA 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 CORBA 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 *CORBA Programming Reference*.

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 CORBA 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 CORBA 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 CORBA server application. For a complete description of the `CORBA::Request::send_oneway` member function, see the *CORBA Programming Reference*.

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 CORBA 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 CORBA 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 CORBA 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 CORBA client application can create, populate, and send requests for objects that were not known to the CORBA client application when it was built. To do this, the CORBA client application uses the

Interface Repository to retrieve information needed to create and populate the requests. The CORBA 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 CORBA client application that browses for objects, or when creating a CORBA 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 BEA Tuxedo system.
2. Use the Bootstrap object to obtain the InterfaceRepository object, which contains a reference to the Interface Repository in a particular BEA Tuxedo 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 CORBA 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.

CHAPTER 4

Handling Exceptions

This topic describes how CORBA C++ client applications handle CORBA exceptions.

Notes: The BEA Tuxedo CORBA Java client and BEA Tuxedo CORBA Java client ORB were deprecated in Tuxedo 8.1 and are no longer supported. All BEA Tuxedo CORBA Java client and BEA Tuxedo CORBA Java client ORB text references, associated code samples, should only be used to help implement/run third party Java ORB libraries, and for programmer reference only.

Technical support for third party CORBA Java ORBs should be provided by their respective vendors. BEA Tuxedo does not provide any technical support or documentation for third party CORBA Java ORBs.

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 CORBA 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 CORBA client application handles exceptions.

CORBA System Exceptions

Table 4-1 lists the CORBA system exceptions.

Table 4-1 CORBA System Exceptions

Exception Name	Description
BAD_CONTEXT	An error occurred 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.

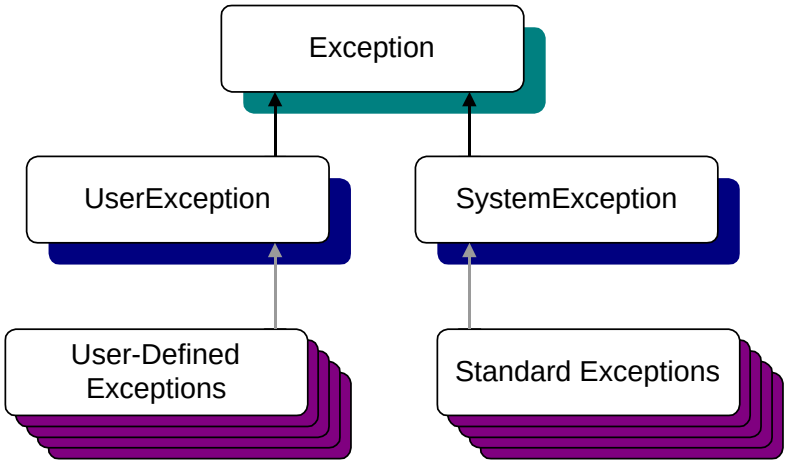
Table 4-1 CORBA System Exceptions (Continued)

Exception Name	Description
NO_RESOURCES	Insufficient resources to process request.
NO_RESPONSE	Response to request not yet available.
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 CORBA client application can narrow from the `Exception` class to a specific `SystemException` or `UserException` class. The C++ Exception inheritance hierarchy is shown in Figure 4-1.

Figure 4-1 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 CORBA 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 BEA Tuxedo CORBA sample applications do not use the `_narrow` operation.

Handling System Exceptions

The CORBA C++ client applications in the BEA Tuxedo 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 CORBA client applications. In C++, `catch` clauses are attempted in the order specified, and the first matching handler is called.

The following example from the CORBA 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;
}
```