OrbixCOMet Desktop Programmer's Guide and Reference

Contents

Part 1 Programmer's Guide

Part II   Programmer's Reference

Preface

OrbixCOMet combines the best of both the Object Management Group (OMG) Common Object Request Broker Architecture (CORBA) and Microsoft COM standards. It provides a high performance bidirectional dynamic bridge which enables two-way integration between COM/Automation and CORBA applications.

OrbixCOMet is designed to allow COM developers—who use tools like Visual C++, Visual Basic, PowerBuilder, Delphi or Active Server Pages on the Windows desktop—to easily access CORBA applications running on Windows, UNIX or OS/390 (formerly MVS) environments. It means COM developers can use the tools familiar to them to build heterogenous systems that use both COM and CORBA components within a COM environment.

Audience

This guide is intended for use by COM developers who wish to familiarise themselves with developing OrbixCOMet applications on the Windows Desktop environment.

Organisation of this Guide

This guide is divided into two main parts.

Part I, Programmer's Guide

Chapter 1, "Introduction to OrbixCOMet"

The COM/CORBA Interworking specification defines a model for transparent two-way interworking between the Object Management Group (OMG) Common Object Request Broker Architecture (CORBA) and Microsoft COM/ Automation environments. OrbixCOMet implements the COM/CORBA

Interworking specification by enabling two-way interworking between CORBA and COM/Automation objects. This chapter explains what interworking means. It also introduces the components involved in OrbixCOMet's implementation of the interworking model, and the concepts and terminology used throughout this guide.

Chapter 2, "Getting Started on Automation"

You can use OrbixCOMet to write Automation client applications for existing CORBA servers implemented, for example, in C++. As an introduction to programming with OrbixCOMet, this chapter illustrates this with a simple example.

Chapter 3, "Getting Started on COM"

You can use OrbixCOMet to write COM client applications for existing CORBA servers implemented, for example, in C++. As a further introduction to programming with OrbixCOMet, this chapter illustrates this with a simple example.

Chapter 4, "Usage Models and Bridge Locations"

You can use OrbixCOMet to develop applications that combine COM/Automation and CORBA in different ways. These combinations are called usage models. You can build client-server applications based on the following two usage models: a COM/Automation client that calls objects in a CORBA server, and a CORBA client that calls objects in a COM/Automation server. This chapter explains how OrbixCOMet supports these usage models.

Chapter 5, "Mapping CORBA Objects to Automation"

CORBA types are defined in OMG IDL. Automation types are defined in Microsoft IDL (MIDL). To allow interworking between CORBA and Automation, OMG IDL types must be translated to MIDL. This chapter outlines how translation of CORBA objects to Automation is achieved.

Chapter 6, "Mapping Automation Objects to CORBA"

Automation types are defined in Microsoft IDL (MIDL). CORBA types are defined in OMG IDL. To allow interworking between Automation and CORBA, MIDL types must be translated to OMG IDL. This chapter outlines how translation of Automation objects to CORBA is achieved.

Chapter 7, "Mapping CORBA Objects to COM"

CORBA types are defined in OMG IDL. COM types are defined in Microsoft IDL (MIDL). To allow interworking between CORBA and COM, OMG IDL types must be translated to MIDL. This chapter outlines how translation of CORBA objects to COM is achieved.

Chapter 8, "Mapping COM Objects to CORBA"

COM types are defined in Microsoft IDL (MIDL). CORBA types are defined in OMG IDL. To allow interworking between COM and CORBA, MIDL types must be translated to OMG IDL. This chapter outlines how translation of COM objects to CORBA is achieved.

Chapter 9, "Development Support Tools"

OrbixCOMet is a high-performance bridge that stores OMG IDL and MIDL type information at the bridging location in an ORB-neutral binary format. The OrbixCOMet type store holds a cache of this information that is used by the dynamic bridge during runtime of your OrbixCOMet applications. This chapter describes the GUI and command line tools that allow you to maintain the type store cache and to create OMG IDL, MIDL, type libraries, smart proxy DLLs and server stub code. It also describes the GUI and command line tools that you can use to replace an existing DCOM server with a CORBA server.

Chapter 10, "Implementing CORBA Clients"

This chapter provides further details about programming OrbixCOMet clients.

Chapter 11, "Exposing DCOM Servers to CORBA Clients"

This chapter explains how to expose an existing DCOM server to CORBA clients. This functionality is particularly important in allowing a CORBA client to talk to applications such as Excel, Word, Access, and so on.

Chapter 12, "Implementing CORBA Servers"

You can use OrbixCOMet to implement COM/Automation servers that appear as CORBA servers. These servers can accept requests from standard COM/Automation clients and from CORBA clients. This chapter provides details about programming OrbixCOMet servers.

Chapter 13, "Error Handling"

Error handling is an important aspect of programming an OrbixCOMet application. Remote method calls are much more complex to transmit than local method calls, so there are many more possibilities for error. This chapter explains how CORBA exceptions can be handled in a client and how a server can raise a user exception.

Chapter 14, "Client Callbacks"

Usually, CORBA clients invoke operations on objects in CORBA servers. However, CORBA clients can implement some of the functionality associated with servers, and all servers can act as clients. A callback invocation is a programming technique that takes advantage of this. This chapter describes client callbacks.

Chapter 15, "Managing the Type Store"

"Development Support Tools" on page 121 describes the tools you can use to populate and remove information from the OrbixCOMet type store in order to create IDL files, type libraries, smart proxy DLLs and server stub code. This chapter describes the general workings of the type store and explains how you can prime it in order to optimise performance at application runtime.

Chapter 16, "OrbixCOMet Configuration"

This chapter describes the keys that are of interest to OrbixCOMet configuration, and their associated default values. It includes details of configuration entries that are either specific to OrbixCOMet or common to multiple IONA products.

Chapter 17, "Deploying your OrbixCOMet Application"

This chapter describes the various models you can adopt when deploying an application you have built using OrbixCOMet. It also describes the steps you must follow to deploy an OrbixCOMet application.

Part II, Programmer's Reference

Chapter 18, "OrbixCOMet API"

This chapter describes the application programming interface (API) to OrbixCOMet. The API is defined in MIDL. This chapter is divided into two main sections. The first section describes the interface entries for Automation. The second section describes the interface entries for COM.

Chapter 19, "Introduction to OMG IDL"

This chapter introduces the CORBA Interface Definition Language (OMG IDL) which is used to describe the interfaces of objects in Orbix.

Chapter 20, "System Exceptions"

This chapter describes system exceptions that are defined by CORBA or specific to Orbix.

Document Conventions

This guide uses the following typographical conventions:

`Constant width`	Constant width (courier font) in normal text represents portions of code and literal names of items such as classes, functions, variables, and data structures. For example, text might refer to the `CORBA::Object` class.
	Constant width paragraphs represent code examples or information a system displays on the screen. For example:

```
#include <stdio.h>
```

Italic	Italic words in normal text represent *emphasis* and *new terms*.
	Italic words or characters in code and commands represent variable values you must supply, such as arguments to commands or pathnames for your particular system. For example:

```
% cd /users/your_name
```

Note: some command examples may use angle brackets to represent variable values you must supply. This is an older convention that is replaced with *italic* words or characters.

This guide may use the following keying conventions:

No prompt	When a command's format is the same for multiple platforms, no prompt is used.
%	A percent sign represents the UNIX command shell prompt for a command that does not require root privileges.
#	A number sign represents the UNIX command shell prompt for a command that requires root privileges.
>	The notation > represents the DOS, Windows NT, or Windows 95 command prompt.

... ⋮	Horizontal or vertical ellipses in format and syntax descriptions indicate that material has been eliminated to simplify a discussion.
[ ]	Brackets enclose optional items in format and syntax descriptions.
{ }	Braces enclose a list from which you must choose an item in format and syntax descriptions.
|	A vertical bar separates items in a list of choices enclosed in { } (braces) in format and syntax descriptions.

Part I

Programmer's Guide
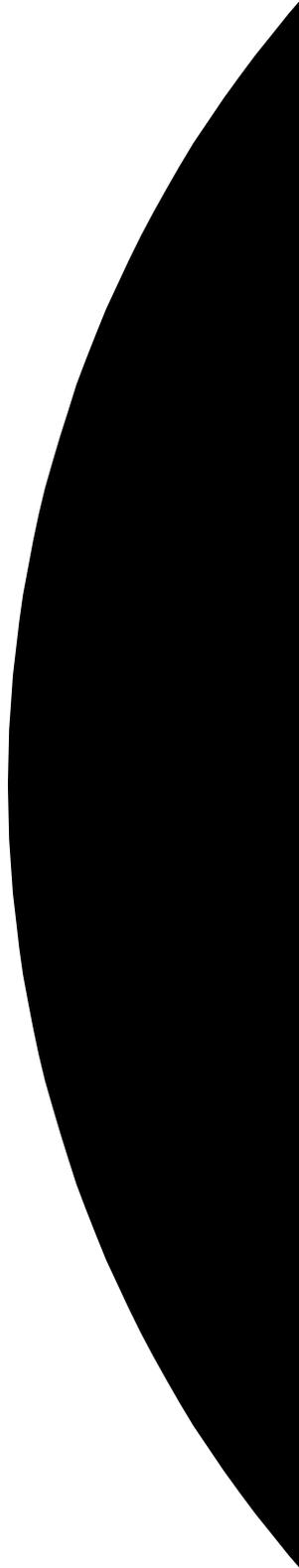
1

Introduction to OrbixCOMet

The COM/CORBA Interworking specification defines a model for transparent two-way interworking between the Object Management Group (OMG) Common Object Request Broker Architecture (CORBA) and Microsoft COM/Automation environments. OrbixCOMet implements the COM/CORBA Interworking specification by enabling two-way interworking between CORBA and COM/Automation objects. This chapter explains what interworking means. It also introduces the components involved in OrbixCOMet's implementation of the interworking model, and the concepts and terminology used throughout this guide.

Subsequent chapters will explain how to use OrbixCOMet's implementation of the model to build distributed applications that combine the CORBA and COM/Automation models.

Note: OrbixCOMet is not a CORBA C++ server-side implementation product. Any C++ examples provided in this book are provided for reference purposes and assume you already have a CORBA server implementation product. The examples provided are for use with the Orbix for Windows product.

Two-way Interworking

Two-way interworking means that CORBA and COM/Automation applications integrate seamlessly. For example:

- A COM/Automation client can call objects in a CORBA server. Because both COM and CORBA support distribution, the COM/Automation client and the CORBA server can be on different machines.

- A CORBA client can call objects in a COM/Automation server. Again, the CORBA client and the COM/Automation server can be on different machines.

You can implement CORBA clients and CORBA servers on any operating system and in any language supported by a CORBA implementation. The range of operating systems supported by Orbix includes UNIX, Windows, and OS/390 (formerly MVS). The range of languages supported by Orbix includes C++, Java, and (using OrbixCOMet) all COM and Automation-based languages.

By providing two-way interworking, OrbixCOMet supports application integration across the boundaries of the network, different operating systems and different programming languages. In particular, it allows you to create new applications to interact with existing applications that were written specifically for the Windows desktop.

OrbixCOMet supports both the Internet Inter-ORB Protocol (IIOP) and Microsoft DCOM. This means any IIOP-compliant Object Request Broker (ORB) can interact with an OrbixCOMet application.

Transparent Interworking

Transparency in the interworking mechanism means transparency between the COM/Automation and CORBA object models. For example:

- A client working in the CORBA model can view and treat a COM/Automation object as if it were a CORBA object. This is because the COM/Automation object has an OMG IDL interface that the CORBA client can understand.

- A client working in the COM/Automation model can view and treat a CORBA object as if it were a COM/Automation object. This is because the CORBA object has a MIDL interface that the COM/Automation client can understand.

Transparency allows clients to work with their familiar object model. They do not have to know that the objects they are using belong to another object system.

The Interworking Model

The COM/CORBA Interworking specification defines the interworking model that specifies how the integration between the COM/Automation and CORBA object models is achieved.
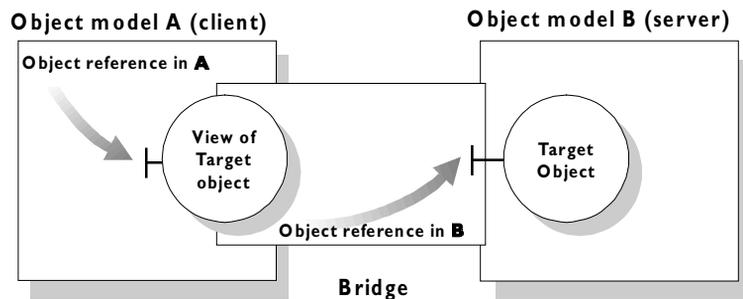
Figure 1.1: *The Interworking Model*

A client in one object system wishes to send a request to an object in the other system. The interworking specification provides a *bridge* that acts as an intermediary between the two object systems. The bridge provides the mappings that are required between the object systems. It provides these mappings transparently so that the client can make requests in its familiar object model.

To implement the bridge, the model provides an object called a *view* in the client's system. The view object exposes the interface of the *target* object in the model understood by the client.

The client makes requests on the view object. The bridge maps these requests into requests in the server's object model and forwards these requests to the target objects across the system boundary. The workings of the bridge are transparent to the client.

How OrbixCOMet Implements the Interworking Model

For a CORBA programmer, OrbixCOMet provides the expected development paradigm for ORB applications. The CORBA programmer starts with an OMG IDL specification. Using OrbixCOMet, the CORBA programmer has the choice to code CORBA clients and servers using any COM-based or Automation-based tool including C++, Visual Basic or PowerBuilder.

OrbixCOMet, therefore, presents a programming model that is familiar to the programmer. Figure 1.2 on page 7 shows the components involved in OrbixCOMet's implementation of the interworking model for allowing COM/Automation clients to make calls on objects in a CORBA server. (Similarly, the interworking model allows for CORBA clients to make calls on objects in a COM/Automation server.)

Bridge

The role of the OrbixCOMet bridge is to map requests in one object system into requests in the other object system. Two-way interworking requires the bridge to provide the mappings and perform translation between CORBA and COM/Automation types.

A bridge contains a COM/Automation object and an Orbix object so that it can expose an appropriate COM/Automation or CORBA interface to its clients. (Refer to Figure 1.3 on page 8.) In OrbixCOMet, the bridge is implemented as a set of DLLs that are capable of dynamically mapping between any COM/Automation and CORBA types.
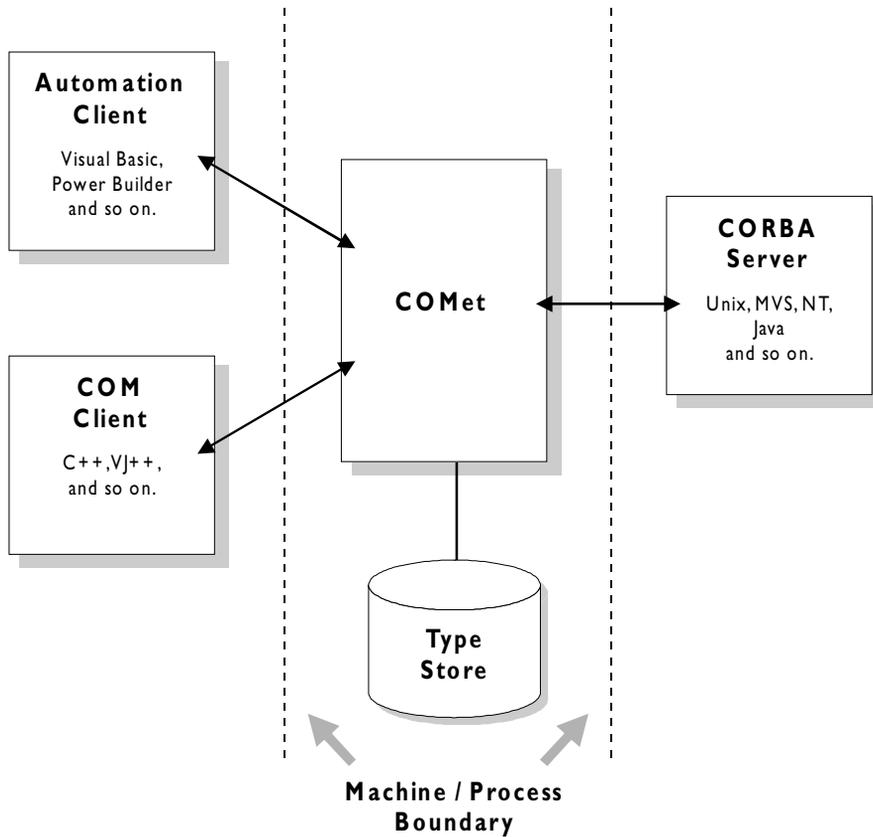
Figure 1.2: *OrbixCOMet's Implementation of the Interworking Model*
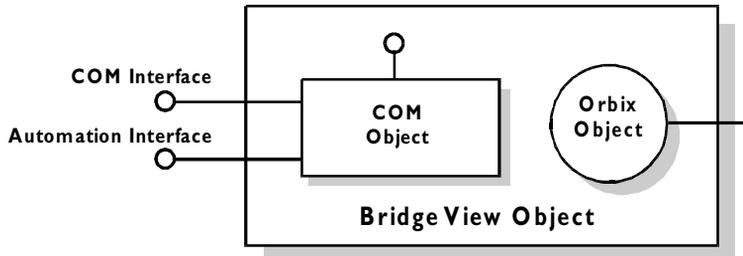
Figure 1.3: *An OrbixCOMet Bridge View Object*

Automation Client

This is a regular Automation client written in a language such as Visual Basic, PowerBuilder, Excel, MFC or any other Automation-compatible language.

COM Client

This is a pure COM client written in C++ or any language that supports COM clients.

COM Library

This is part of the operating system that provides the COM and Automation infrastructure.

CORBA Server

This is a normal CORBA server written in any language supported by an ORB. It can be an Orbix program written in languages that include C++, Java, and Automation-compatible languages such as Visual Basic and PowerBuilder.

Automation Server

This is a regular Automation server written in a language such as Visual Basic, PowerBuilder, Excel, MFC or any other Automation-compatible language.

COM Server

This is a pure COM server written in C++ or any language that supports COM servers.

CORBA Client

This is a CORBA client written in any language supported by an ORB. It can be an Orbix program written in languages that include C++, Java, and Automation-compatible languages such as Visual Basic and PowerBuilder.

As shown in Figure 1.2 on page 7, the Interworking model allows a COM/ Automation client to make calls on objects in a CORBA server. Similarly, a CORBA client can make calls on objects in a COM/Automation server. The bridge is not involved in CORBA to CORBA requests.

2

Getting Started on Automation

You can use OrbixCOMet to write Automation client applications for existing CORBA servers implemented, for example, in C++. As an introduction to programming with OrbixCOMet, this chapter illustrates this with a simple example.

Versions of the Automation client application described in this chapter can be found at the following locations in your OrbixCOMet installation:

Visual Basic	`demo\VB6`
PowerBuilder	`demo\PB6`
Internet Explorer	`demo\IE`

The server application is implemented in C++ and its code is located in the directory `demo\corbasrv\phonebook` of your OrbixCOMet installation. You do not need to understand how the server is implemented in order to follow the example in this chapter.

This chapter assumes that you are familiar with the CORBA Interface Definition Language (OMG IDL). Refer to "Introduction to OMG IDL" on page 313 for more details.

Phone Book Example

Figure 2.1 illustrates the components of a telephone book application. The CORBA server contains an object that supports the interface PhoneBook. Your task is to implement the Automation client that will make requests on the PhoneBook object.

CORBA Server

Automation Client → **COMet** →

PhoneBook Object

numberOfEntries ( )
addNumber ( )
lookupNumber ( )

Orbix Object
(Implemented in C++)
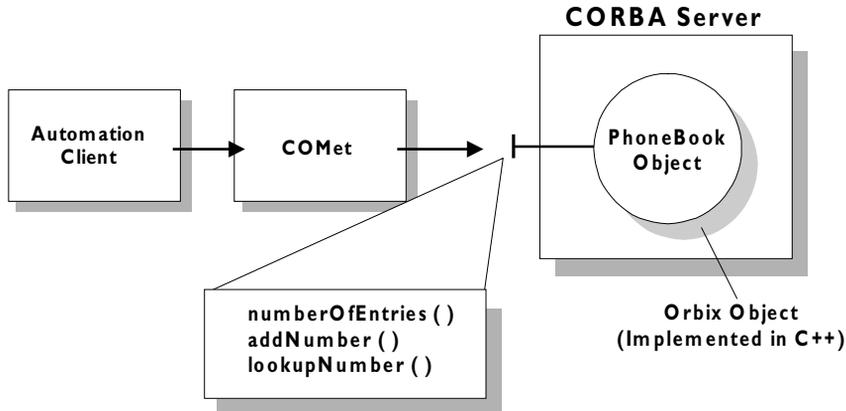
Figure 2.1: *Telephone Book Example*

Creating a Type Library

When using an Automation client, you have the option in some controllers (for example, Visual Basic) of using straight IDispatch interfaces or dual interfaces. Refer to the sections "Late Binding"and "Early Binding" on page 148 for more details.

If you want to use dual interfaces you must create a type library. The Type Store Manager tool (Typeman.exe) allows you to do this. (Refer to "Development Support Tools" on page 121 for more details about Typeman.exe.)

The **OrbixCOMet tools** screen shown in Figure 2.2 on page 13 is opened when you click the **COMet tools** option on your OrbixCOMet start menu.
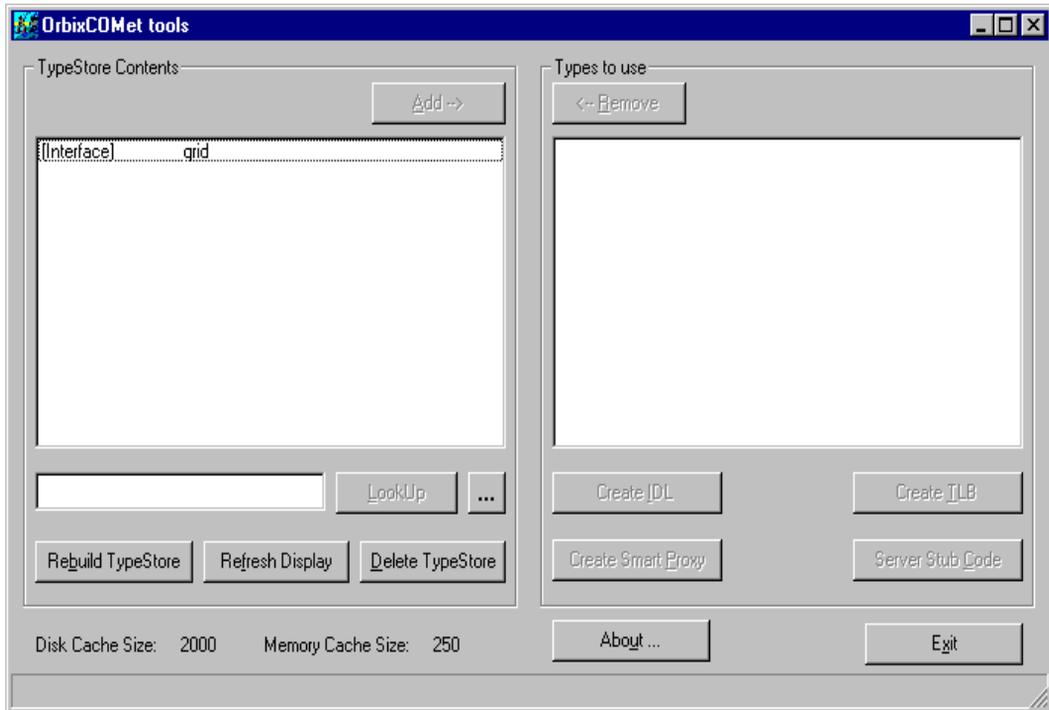
Figure 2.2: *OrbixCOMet Tools Screen*

The **TypeStore Contents** panel lists all the interfaces in the type store. To create a type library:

1. From the **TypeStore Contents** panel, select an interface you want to include in the type library. (In this example, you would add the Phone Book interface.)

2. Select the **Add** button. This adds the interface to the **Types to use** panel.

3. Select the **CreateTLB** button after you have selected all the types you want to use. This opens the **Typelibrary Generator** screen shown in Figure 2.3 on page 14.
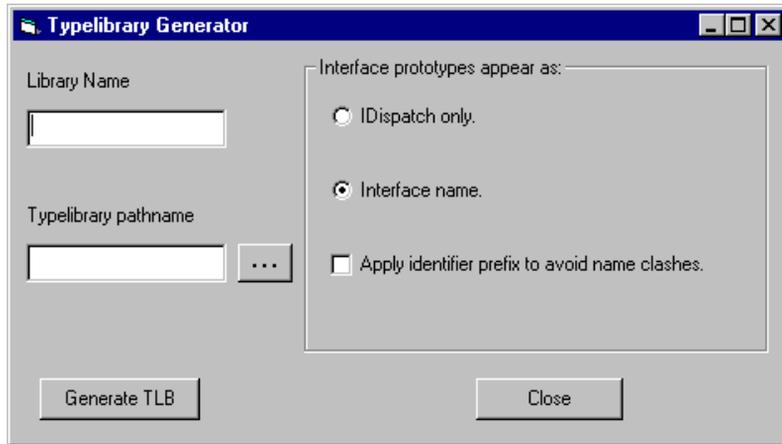
Figure 2.3: *Creating a Type Library*

4. Enter the library name and the type library pathname in the corresponding fields.

5. Select the appropriate radio button to determine whether interface prototypes should appear as `IDispatch` only or use the specific interface name.

6. Select the **Generate TLB** button. This creates the type library.

Refer to "Development Support Tools" on page 121 to find out more about these screens and about creating a type library.

Implementing the Client

The principal task of the client is to obtain a reference to an Automation view object in the bridge that can forward requests to the `PhoneBook` object in the CORBA server. The section "How OrbixCOMet Implements the Interworking Model" on page 6 explained that a client makes method calls on a view object. The bridge forwards these requests to the target object in the server.

In this example, the `PhoneBook` view object exports the Automation interface `DIPhoneBook` generated from the OMG IDL `PhoneBook` interface. You can find details of how CORBA types are translated to Automation in "Mapping CORBA Objects to Automation" on page 41.

Obtaining a Reference to a CORBA Object

This section includes Visual Basic and PowerBuilder examples of how the client obtains a reference to a CORBA object.

Visual Basic

```
Dim ObjFactory As Object
Dim phoneBookObj As Object
...
Set ObjFactory = CreateObject("CORBA.Factory")
...
Set phoneBookObj = ObjFactory.GetObject(
    PhoneBook:PhoneBookSrv:" & host.Text)
```

PowerBuilder

```
OleObject ObjFactory
OleObject phoneBookObj
...
ObjFactory = CREATE OleObject
ObjFactory.ConnectToNewObject("CORBA.Factory")
...
phoneBookObj = CREATE OleObject
phoneBookObj = ObjFactory.GetObject(
    PhoneBook::PhoneBookSrv:" & host.Text)
```

The client first instantiates a CORBA object factory in the bridge. The CORBA object factory is a factory for view objects. It has the ProgID `CORBA.Factory`.

The client then calls `GetObject()` on the CORBA object factory. It passes the name of `PhoneBook` object in the CORBA server in the parameter for `GetObject()`. This parameter has the form:

```
Interface:Marker:Server:Host
```

In this example, GetObject() does not specify a marker (Orbix object name), so the call to GetObject() will look for any object in the PhoneBookSrv server on the host specified in host.Text that supports the PhoneBook interface. Refer to "Implementing CORBA Clients" on page 135 for full details of the string parameter for GetObject().
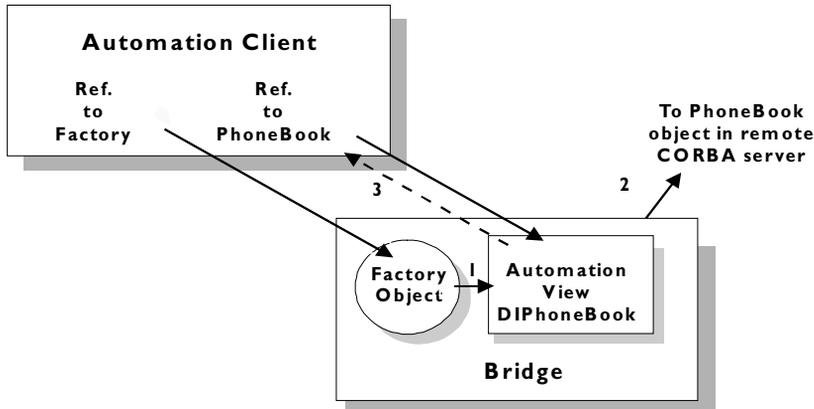
Figure 2.4: *Binding to the Phone Book Object*

The purpose of the call to GetObject() is to achieve the connection between the client's phoneBookObj object reference and the target PhoneBook object in the server. To achieve this, GetObject() does the following:

1. It creates an Automation view object in the phonebookBridge that implements the dual interface DIPhoneBook.

2. It binds the Automation view object to the CORBA implementation object named in GetObject()'s string parameter.

3. It returns a reference to the view object.

After the call to GetObject(), the client can use the phoneBookObj object reference to invoke operations on the target PhoneBook object in the server. For example:

```
phoneBookObj.addNumber(...)
```

The Client Code

This section provides Visual Basic and PowerBuilder versions of the client application. It shows how the code extracts provided earlier in this chapter fit into a full client application. The client presents the interface shown in Figure 2.5.
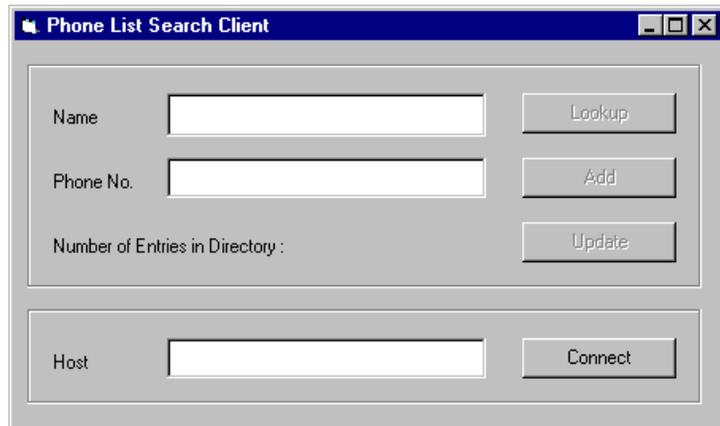
Figure 2.5: *Using the Phone List Search Client Application*

Visual Basic Implementation

General Declarations

Declare a reference to the factory object and to the Automation view object.

```
Dim ObjFactory As Object
Dim phoneBookObj As Object
```

Creating the Form

Create the CORBA factory object when the Visual Basic Form is created.

```
Private Sub Form_Load()
   Set ObjFactory = CreateObject("CORBA.Factory")
End Sub
```

Connecting to the CORBA Server

The implementation of the Connect button connects to the CORBA object specified in `GetObject()`'s parameter.

```
Private Sub ConnectBtn_Click()
   Set phoneBookObj = _
      ObjFactory.GetObject( _
         "PhoneBook:PhoneBookSrv:" & host.Text)
   ...
End Sub
```

Invoking Operations on the PhoneBook Object

The subroutines to implement the Add, LookUp, and Update buttons call OMG IDL operations on the `PhoneBook` object in the CORBA server.

```
Private Sub AddBtn_Click()
   If phoneBookObj.addNumber( _
      PersonalName.Text, Number.Text) Then
         MsgBox "Added " & PersonalName.Text & " successfully"
   Else ...
   End If

   ' Update the display of the current number of
   ' entries in the phonebook
   EntryCount.Caption = phoneBookObj.numberOfEntries
End Sub

Private Sub LookupBtn_Click()
   Dim num
   num = phoneBookObj.lookupNumber(PersonalName.Text)
   ...
End Sub

Private Sub UpdateBtn_Click()
   ' Update the display for the number of entries
   ' in the remote phonebook
   EntryCount.Caption = phoneBookObj.numberOfEntries
End Sub
```

Unloading the Form

The `Form_Unload()` subroutine releases the CORBA factory object and the Automation view object.

```
Private Sub Form_Unload(Cancel As Integer)
   Set ObjFactory = Nothing
   Set phoneBookObj = Nothing
End Sub
```

PowerBuilder Implementation

General Declarations

Declare global variables for the factory object and the Automation view object.

```
OleObject ObjFactory
OleObject phoneBookObj
```

Loading the Window

Create the CORBA factory object within the open event for the Phone List Search Client window.

```
ObjFactory = CREATE OleObject
ObjFactory.ConnectToNewObject("CORBA.Factory")
```

Connecting to the CORBA Server

The clicked event for the Connect button connects to the CORBA server.

```
phoneBookObj = ObjFactory.GetObject(
   "PhoneBook:PhoneBookSrv:" + sle_host.Text)
...
```

Invoking Operations on the PhoneBook Object

The clicked event for the Add, LookUp, and Update buttons call operations on the `PhoneBook` object in the CORBA server.

```
// Add Button
If sle_phone.Text <> "" and sle_name.Text <> "" then
   If phoneBookObj.addNumber(sle_name.Text, sle_phone.Text) Then
     MessageBox ("Success!", "Added " + sle_name.Text
        + " successfully.")
      EntryCount.Text = String(phoneBookObj.numberOfEntries)
```

```
        ...
      End If
   End if


// Lookup Button
if sle_name.Text <> "" then
   ...
   Result = phoneBookObj.lookupNumber(sle_name)
   ...
end if

// Update Button
EntryCount.Text = String(phoneBookObj.numberOfEntries)
```

Unloading the Window

Release the CORBA factory object and the Automation view object when unloading the window.

```
ObjFactory.DisconnectObject()
DESTROY ObjFactory
DESTROY phoneBookObj
```

Building the Client

You can now build your client executable as normal for the language you are using.

Running the Client

To run the client, perform the following steps:

1. Ensure that the Orbix daemon is running on the CORBA server's host. If you have Orbix for Windows installed, you can run the Orbix daemon from the Orbix Desktop Programs group on the Windows Start menu.

2. Register the server with the Implementation Repository on the server's host. (Usually, it will not be necessary to register a server if the server has been written and registered by someone else.)

   You can use the putit utility from the command prompt as follows:

```
putit PhoneBookSrv your_path\phonebook.exe
```

where *your_path* is the full pathname of the directory containing the server's executable file.

Refer to the Orbix documentation set for more information about the `putit` command.

Note: Alternatively, you can use the Orbix Server Manager to register the server. Run the Orbix Server Manager from the GUI Tools Programs group on the Windows Start menu. Register the server with the name `PhoneBookSrv`. The on-line GUI Tools Help explains how use the Orbix Server Manager to register a server.

---

3. Run the client.

   On the Phone List Search Client screen, type the server's host name in the Host textbox and click Connect. You can now add and look up phone book entries.

   If your client is inactive for some time, the `PhoneBookSrv` server will time out and exit. It will be reactivated automatically if the client issues another request.

3

Getting Started on COM

You can use OrbixCOMet to write COM client applications for existing CORBA servers implemented, for example, in C++. As an introduction to programming with OrbixCOMet, this chapter illustrates this with a simple example.

A version of the COM client application described in this chapter is located in the directory `demo\com\phonebook` of your OrbixCOMet installation. This directory contains Visual C++ COM client code.

The server application is implemented in C++ and its code is located in the directory `demo\corbasrv\phonebook` of your OrbixCOMet installation. You do not need to understand how the server is implemented in order to follow the example in this chapter.

This chapter assumes that you are familiar with the CORBA Interface Definition Language (OMG IDL). Refer to "Introduction to OMG IDL" on page 313 for more details.

Phone Book Example

Figure 3.1 illustrates the components of a telephone book application. The CORBA server contains an object that supports the interface `PhoneBook`. Your task is to implement the COM client that will make requests on the `PhoneBook` object.
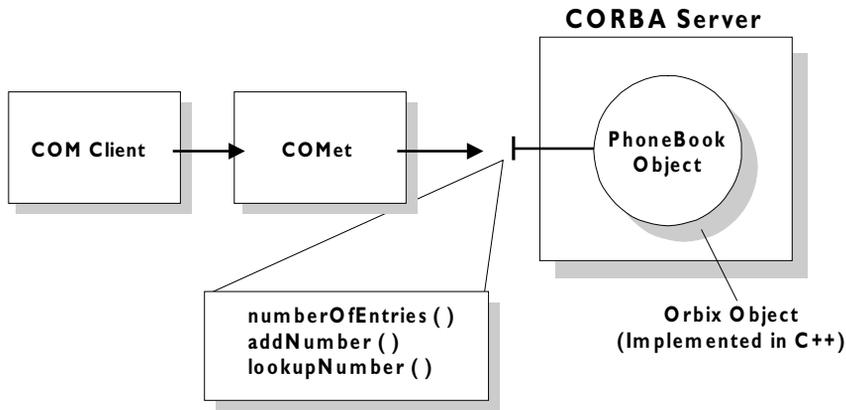
Figure 3.1: *Telephone Book Example*

Obtaining a MIDL Interface

The normal procedure for writing a client in COM is to first obtain a MIDL definition for the interface. OrbixCOMet allows you to extract these MIDL definitions from the OrbixCOMet type store by using the Type Store Manager tool (`Typeman.exe`) described in "Development Support Tools" on page 121. For this example, you need to get a MIDL definition for the phone book example as shown in Figure 3.1.

The **OrbixCOMet tools** screen shown in Figure 3.2 on page 25 is opened when you select the **COMet tools** option on your OrbixCOMet start menu.
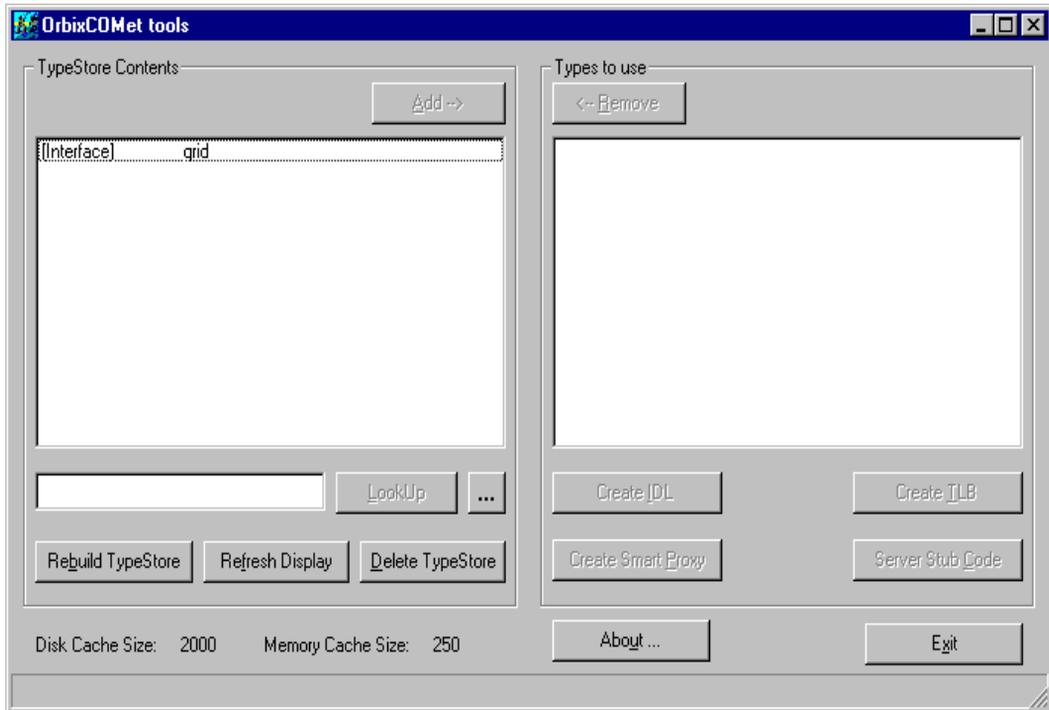
Figure 3.2: *OrbixCOMet Tools Screen*

The **TypeStore Contents** panel lists all the interfaces in the type store. To create an IDL file:

1. From the **TypeStore Contents** panel, select an interface you want to include in the IDL file. (In this example, you would add the Phone Book interface.)

2. Select the **Add** button. This adds the interface to the **Types to use** panel.

3. Select the **CreateIDL** button after you have selected all the types you want to use. This opens the **OrbixCOMet ts2idl client** screen shown in Figure 3.3 on page 26.
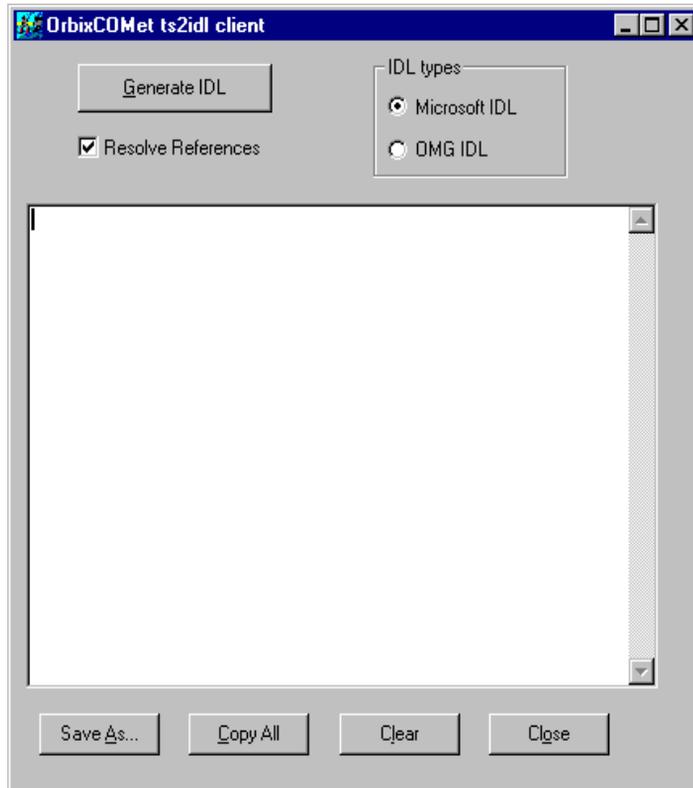
Figure 3.3: *Creating an IDL File*

4. Select the appropriate radio button to indicate the type of IDL file you want to create.

5. Select the **Generate IDL** button. This creates the IDL file.

Note: Refer to "Development Support Tools" on page 121 to find out more about these screens and about generating IDL files.

Building a Proxy/Stub DLL

If the OrbixCOMet bridge is not being loaded in-process to your COM client application, you must create a standard DCOM proxy DLL for the interfaces you are using. This is necessary to allow DCOM to correctly make a connection to the remote OrbixCOMet bridge from the client. OrbixCOMet includes a command line tool called `ts2idl.exe` that can create the sources for the proxy/stub DLL. For this example, you would issue the following command:

```
ts2idl -f PhoneBook.idl -s -p PhoneBook
```

When you are generating a MIDL file from the command line, the `-p` switch allows you to create a Visual C++ makefile that you can use to compile your proxy/stub DLL. For this example, this makefile is called `Phonebook.MK` and is located in the `demo\getstarted\COM` directory.

Note: Refer to "Development Support Tools" on page 121 to find out more about generating smart proxy DLLs and server stub code.

Implementing the Client

The principal task of the client is to obtain a reference to a COM view object in the bridge that can forward requests to the `PhoneBook` object in the CORBA server. The section "How OrbixCOMet Implements the Interworking Model" on page 6 explained that a client makes method calls on a view object. The bridge forwards these requests to the target object in the server.

In this example, the `PhoneBook` view object exposes the COM interface `IPhoneBook` generated from the OMG IDL `PhoneBook` interface. You can find details of how CORBA types are translated to COM in "Mapping CORBA Objects to COM" on page 81.

Obtaining a Reference to a CORBA Object

The following code shows how the client obtains a reference to a CORBA object:

```
//Connecting to the CORBA Factory
hr = CoCreateInstanceEx (IID_ICORBAFactory,
   NULL, ctx, NULL, 1, &mqi);
pCORBAFact = (ICORBAFactory*)mqi.pItf;

//Connecting to the CORBA Server
memset(szMarkerServerHost,'\0',128);
sprintf(szMarkerServerHost,
   "PhoneBook:PhoneBookSrv:%s", hostname);

hr = pCORBAFact->GetObject(szMarkerServerHost,&pUnk);
hr = pUnk->QueryInterface(IID_IPhoneBook, (PPVOID)&pIPhoneBook);
```

The client first instantiates a CORBA object factory in the bridge. The CORBA object factory is a factory for view objects. It has the IID `IID_ICORBAFactory`.

The client then calls `GetObject()` on the CORBA object factory. It passes the name of `PhoneBook` object in the CORBA server in the parameter for `GetObject()`. This parameter has the form:

```
Interface:Marker:Server:Host
```

In this example, `GetObject()` does not specify a marker (Orbix object name), so the call to `GetObject()` will look for any object in the `PhoneBookSrv` server on the host specified in `host.Text` that supports the `PhoneBook` interface. Refer to "Implementing CORBA Clients" on page 135 for full details of the string parameter for `GetObject()`.
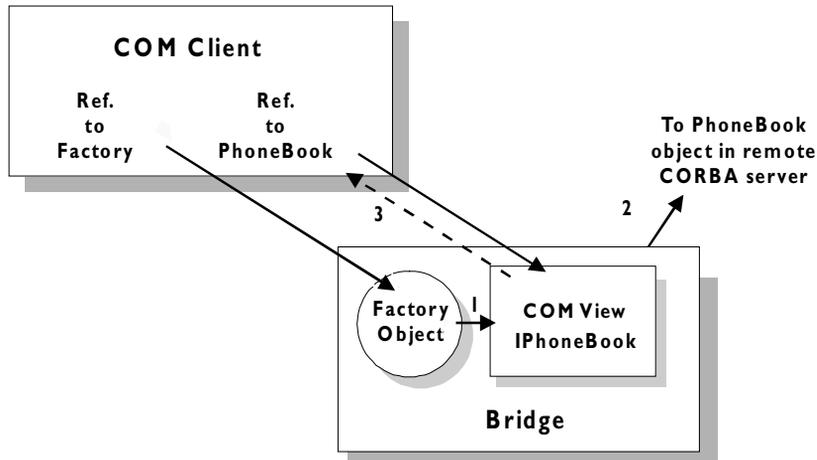
Figure 3.4: *Binding to the Phone Book Object*

The purpose of the call to `GetObject()` is to achieve the connection between the client's `phoneBookObj` object reference and the target `PhoneBook` object in the server. To achieve this, `GetObject()` does the following:

1. It creates a COM view object in the `phonebookBridge` that implements the interface `IPhoneBook`.
2. It binds the COM view object to the CORBA implementation object named in `GetObject()`'s string parameter.
3. It returns a reference to the view object.

After the call to `GetObject()`, the client can use the `phoneBookObj` object reference to invoke operations on the target `PhoneBook` object in the server. For example:

```
phoneBookObj.addNumber(...)
```

Using CoCreateInstance()

The `CORBA.Factory` object allows you to obtain a reference to a CORBA object in a manner that is compliant with the OMG specification. However, OrbixCOMet also allows a COM client to connect directly to a CORBA server using the standard `CoCreateInstance()` COM API call. Refer to "Implementing CORBA Clients" on page 135 for more details.

The Client Code

This section provides a Visual C++ 5.0 version of the client application. It shows how the code extracts provided earlier in this chapter fit into a full client application. The client produces the following output:

```
%%% App beginning --
%%% Using out of process server
About to add IONA Freephone USA
Successfully added the number
There are 12 entries
The number for IONA Freephone USA is 6724948
%%% App end
```

Includes

```
// Header file created from the MIDL file
// generated by the TypeStore Manager Tool
//
#include "phoneBook.h"
```

General Declarations

```
IUnknown*pUnk = NULL;
   IPhoneBook*pIPhoneBook = NULL;
   ICORBAFactory*pCORBAFact = NULL;
   char  szMarkerServerHost[128];
```

Connecting to the CORBA Factory

```
hr = CoCreateInstanceEx (IID_ICORBAFactory,
   NULL, ctx, NULL, 1, &mqi);
pCORBAFact = (ICORBAFactory*)mqi.pItf;
```

Connecting to the CORBA Server

```
memset(szMarkerServerHost,'\0',128);
sprintf(szMarkerServerHost,
    "PhoneBook:PhoneBookSrv:%s", hostname);


hr = pCORBAFact->GetObject(szMarkerServerHost,&pUnk);
hr = pUnk->QueryInterface(IID_IPhoneBook, (PPVOID)&pIPhoneBook);
```

Invoking Operations on the PhoneBook Object

```
boolean lAdded=0;
cout << "About to add IONA Freephone USA" << endl;
hr = pIF->addNumber("IONA Freephone USA",6724948, &lAdded);

if (lAdded)
   cout << "Successfully added the number" << endl;
else
   cout << "Failed to add the number" << endl;

// see how many entries there are in the phonebook
long nNumEntries=0;
hr = pIF->_get_numberOfEntries(&nNumEntries);
cout << "There are " << nNumEntries << " entries" << endl;

// then lookup a couple of numbers number
long phoneNumber=0;
pIF->lookupNumber("IONA Freephone USA", &phoneNumber);
cout << "The number for IONA Freephone USA is " << phoneNumber <<
endl;
```

Building the Client

You can now build your client executable as normal by running the makefile.

Running the Client

Follow these steps to run the client:

1. Ensure that the Orbix daemon is running on the CORBA server's host. You can run the Orbix daemon from the Orbix for Windows Programs group from the Windows Start menu.

2. Register the CORBA server with the Implementation Repository on the server's host. (Usually, it will not be necessary to register a server, if the server has been written and registered by someone else.)

   You can use the `putit` utility from the command prompt as follows:

   ```
   putit PhoneBookSrv your_path\phonebook.exe
   ```

   where *your_path* is the full pathname of the directory containing the server's executable file.

   Refer to the Orbix documentation set for more information about the `putit` command.

Note: Alternatively, you can use the Orbix Server Manager to register the server. Run the Orbix Server Manager from the GUI Tools Programs group on the Windows Start menu. Register the server with the name `PhoneBookSrv`. The on-line GUI Tools Help explains how use the Orbix Server Manager to register a server.

---

3. Run the client. It should produce output like the following:

   ```
   %%% App beginning --
   %%% Using in-process server
   [392: New IIOP Connection (axiom:1570) ]
   [392: New IIOP Connection (192.122.221.51:1570) ]
   [392: New IIOP Connection (axiom:1607) ]
   [392: New IIOP Connection (192.122.221.51:1607) ]
   [392: New IIOP Connection (axiom:1611) ]
   [392: New IIOP Connection (192.122.221.51:1611) ]
   About to add IONA Freephone USA
   Successfully added the number
   There are 11 entries
   The number for IONA Freephone USA is 6724948
   %%% Test end
   ```

4

Usage Models and Bridge Locations

You can use OrbixCOMet to develop applications that combine COM/Automation and CORBA in different ways. These combinations are called usage models. You can build client-server applications based on the following two usage models: a COM/Automation client that calls objects in a CORBA server, and a CORBA client that calls objects in a COM/Automation server. This chapter explains how OrbixCOMet supports these usage models.

Automation Client to CORBA Server
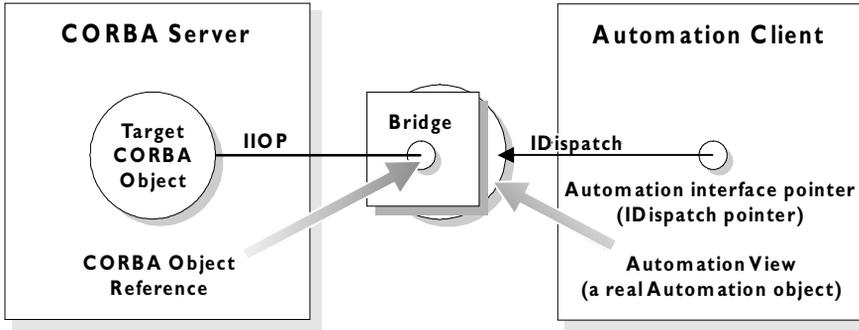
Figure 4.1: *Automation Client to CORBA Server*

The Client

Using this model, an Automation client can use IDispatch to contact a CORBA server. The client makes method calls on an Automation view object in the bridge via an IDispatch pointer. The bridge makes a corresponding operation call on the target object in the CORBA server via a CORBA object reference.

The dynamic marshalling engine of OrbixCOMet allows for automatic mapping of IDispatch pointers to CORBA interfaces and object references at runtime.

The client need not know that the target object is a CORBA object. An Automation client can be written in any Automation-based programming language.

The Server

The CORBA server presents an OMG IDL interface to its objects.

The server application can be developed (or already exist) possibly on platforms other than Windows NT and Windows 95. It can be written in any language supported by a CORBA implementation such as C++, Java, or any Automation-based language.

The Bridge

The bridge can be located on the Automation client, on the CORBA server or on an intermediary machine. It acts as an Automation server because it accepts requests from the Automation client. The bridge also acts as a CORBA client because it translates requests from the Automation client into requests on the CORBA server.

COM Client to CORBA Server
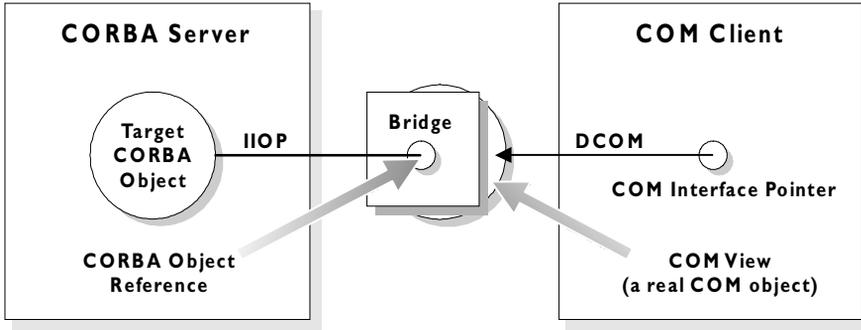
Figure 4.2: *COM Client to CORBA Server*

The Client

Using this model, a COM client can use the DCOM protocol to contact a CORBA server. The client makes method calls on a COM view object in the bridge via a COM interface pointer. The bridge makes a corresponding operation call on the target object in the CORBA server via a CORBA object reference.

The dynamic marshalling engine of OrbixCOMet allows for automatic mapping of COM interface pointers to CORBA interfaces and object references at runtime.

The client need not know that the target object is a CORBA object. A COM client can be written in C++ or any language that supports COM clients.

The Server

The CORBA server presents an OMG IDL interface to its objects.

The server application can be developed (or already exist) possibly on platforms other than Windows NT and Windows 95. It can be written in any language supported by a CORBA implementation such as C++, Java, or any Automation-based language.

The Bridge

The bridge can be located on the COM client, on the CORBA server or on an intermediary machine. It acts as a COM server because it accepts requests from the COM client. The bridge also acts as a CORBA client because it translates requests from the COM client into requests on the CORBA server.

CORBA Client to COM/Automation Server
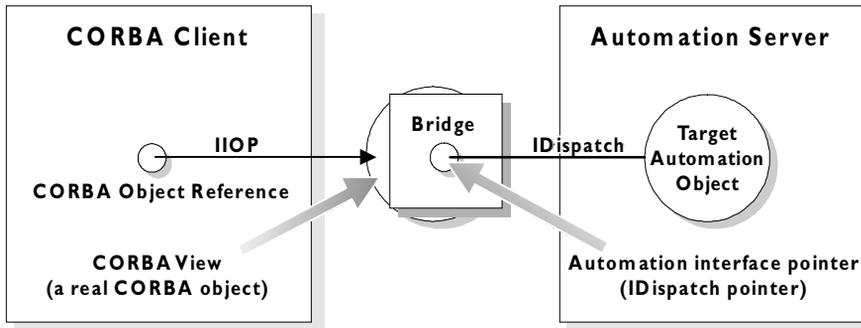
Figure 4.3: *CORBA Client to Automation Server*
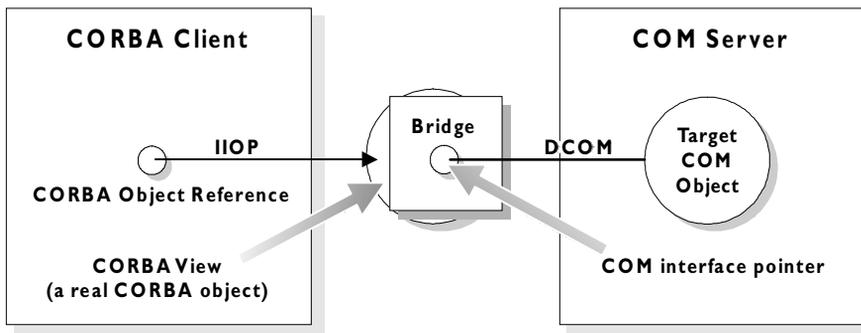
Figure 4.4: *CORBA Client to COM Server*

The Client

Using this model, a CORBA client can use the CORBA IIOP protocol to contact a COM/Automation server. The client makes method calls on a CORBA view object in the bridge via a CORBA object reference. The bridge makes a corresponding operation call on the target object in the COM/Automation server via an Automation (IDispatch) or COM interface pointer.

The dynamic marshalling engine of OrbixCOMet allows for automatic mapping of CORBA interfaces and object references to Automation (IDispatch) and COM interface pointers.

The client need not know that the target object is a COM/Automation object. A CORBA client can be developed on any platform including UNIX, Windows NT and Windows 95. It can be written in any language supported by a CORBA implementation such as C++, Java, or any Automation-based language.

The Server

The COM/Automation server presents a MIDL interface to its objects. An Automation server can be written in any Automation-based language. A COM server can be written in C++ or any language that supports COM servers.

The Bridge

The bridge can be located on the CORBA client (Windows NT or Windows 95 only), on the COM/Automation server or on an intermediary machine. It acts as a CORBA server because it accepts requests from CORBA clients. The bridge also acts as a COM/Automation client because it translates CORBA operation calls into COM/Automation method calls on the COM/Automation server.

5

Mapping CORBA Objects to Automation

CORBA types are defined in OMG IDL. Automation types are defined in Microsoft IDL (MIDL). To allow interworking between CORBA and Automation, OMG IDL types must be translated to MIDL. This chapter outlines how translation of CORBA objects to Automation is achieved.

For the purposes of illustration, this chapter describes a textual mapping between OMG IDL and MIDL. OrbixCOMet itself does not require this textual mapping to take place, because it includes a dynamic marshalling engine. The textual mappings shown in this chapter are actually performed by OrbixCOMet at application runtime.

Translation of Basic Types

OMG IDL basic types translate to compatible types in Automation. Table 5.1 shows the mapping for each type.

OMG IDL	Description	MIDL	Description

`boolean`	Unsigned char, 8-bit
	`0 = FALSE`
	`1 = TRUE`
		`VARIANT_BOOL`	16-bit integer
			`0 = FALSE`
			`-1 = TRUE`

`char`	8-bit quantity	`UI1`[a]	8-bit unsigned integer
`double`	IEEE 64-bit float	`double`	IEEE 64-bit float
`float`	IEEE 32-bit float	`float`	IEEE 32-bit float
`long`	32-bit integer	`long`	32-bit integer
`octet`	8-bit quantity	`UI1`	8-bit unsigned integer
`short`	16-bit integer	`short`	16-bit integer
`unsigned long`	32-bit integer	`long`	32-bit integer
`unsigned short`	16-bit integer	`long`	32-bit integer

Table 5.1: *Translation of OMG IDL Basic Types to Automation*

a.   `UI1` is supported in Windows32.

There is not an exact correspondence between the types supported by OMG IDL and the types supported by Automation.

Automation does not support unsigned types (that is, `unsigned short` or `unsigned long`). Therefore, an OMG IDL `unsigned short` is translated to an Automation `signed long`. An OMG IDL `unsigned long` is translated to its equivalent Automation `signed long`.

These differences mean that the following translations will result in a run-time error:

- Converting Automation `long` to OMG IDL `unsigned long` when the value of the Automation `long` parameter is a negative number.

- Demoting OMG IDL `unsigned long` to Automation `long` when the value of the OMG IDL `unsigned long` parameter is greater than the maximum value of an Automation `long`.

- Demoting Automation `long` to OMG IDL `unsigned short` when the value of the Automation `long` parameter is either negative or greater than the maximum value of an OMG IDL `unsigned short`.

Translation of Strings

An OMG IDL string translates to an Automation `BSTR`. For example:

```
// OMG IDL
// This definition might appear within a struct
// definition.
string address;
```

translates to:

```
// MIDL
BSTR address;
```

A run-time error will occur when mapping a fixed-length OMG IDL `string` if the `BSTR` exceeds the maximum length of the OMG IDL `string`.

Translation of Interfaces

An OMG IDL interface translates to an Automation view interface. For example, the following OMG IDL interface `Bank`:

```
// OMG IDL
interface Bank
{
   // Attributes and operations here;
   . . .
};
```

translates to the Automation view interface `DIBank`:

```
// MIDL
// Definitions that are not of interest here.

[oleautomation, dual, uuid(....)]
interface DIBank : IDispatch
{
   // Properties and methods here.
   ...
}
```
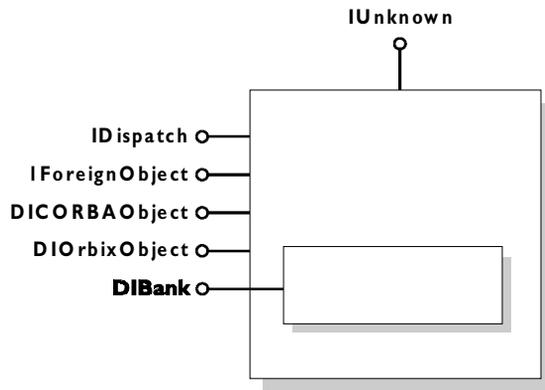

Figure 5.1: *Automation View of Bank Interface*

As shown in Figure 5.1, the Automation view in the bridge supports the interface `DIBank`. Any Automation controller can use the `DIBank` interface to invoke operations on the Automation view. The view forwards the request to the target `Bank` object in the CORBA server.

The interface `DIBank` is an Automation dual interface. A dual interface is a COM vtable-based interface that derives from `IDispatch`. This means that its methods can be either late-bound using `IDispatch::Invoke` or early-bound through the vtable portion of the interface.

The Automation view supports the following interfaces:

- `IUnknown` and `IDispatch` required by all Automation objects.

- `DIForeignObject` required by all views.

- `DICORBAObject` required by all CORBA objects.

- `DIOrbixObject` supported by all Orbix objects.

Translation of Attributes

An OMG IDL attribute translates to an Automation property. For example:

```
// OMG IDL
interface Account
{
   attribute float balance;
   readonly attribute string owner;
   void makeLodgement(in float amount, out float balance);
   void makeWithdrawal(in float amount, out float balance);
};
```

translates to:

```
// MIDL
[oleautomtion, dual, uuid(....)]
interface DIAccount : IDispatch
{
   HRESULT makeLodgement ([in] float amount,
                          [out] float * balance,
                          [optional, out] VARIANT * excep_OBJ);
   HRESULT makeWithdrawal ([in] float amount,
                          [out] float * balance,
                          [optional, out] VARIANT * excep_OBJ);
   [propget] HRESULT balance([retval,out] float * val);
   [propput] HRESULT balance ([in] float balance);
   [propget] HRESULT owner([retval,out] BSTR * val);
}
```

A normal attribute translates to a property that has a method to set the value and a method to get the value.

A `readonly` attribute translates to a property that has a method to get the value. The `get` method returns the attribute value in the parameter tagged `[retval,out]`.

Visual Basic

```
Set accountObj = ... ' Get a reference to an Account object.

Dim myBalance as Single

' Set the balance of accountObj:
accountObj.balance = 150.22

' Get the balance of accountObj:
myBalance = accountObj.balance
```

PowerBuilder

```
... // Get a reference to an Account object.

integer myBalance

myBalance = accountObj.balance
accountObj.balance myBalance
```

Translation of Operations

An OMG IDL operation translates to an Automation method. For example:

```
// OMG IDL
interface Account {
   void makeDeposit(in float amount, out float balance);
   float calculateInterest();
   ...
};
```

translates to:

```
// MIDL
[oleautomation, dual,uuid(...),helpstring("Account")]
interface DIAccount : IDispatch {
   [id(100)] HRESULT makeDeposit (
        [in] float it_amount,
        [in,out] float *it_balance,
        [optional,in,out] VARIANT *IT_Ex );
   [id(101)] HRESULT calculateInterest (
        [optional,in,out] VARIANT *IT_Ex,
        [retval,out] float *IT_retval );
}
```

Parameters

An OMG IDL `in` parameter translates to an Automation `[in]` parameter.

An OMG IDL `out` parameter translates to an Automation `[out]` parameter.

An OMG IDL `inout` parameter translates to an Automation `[in,out]` parameter.

Return Types

An OMG IDL `void` return type does not need any translation.

An OMG IDL return type that is not `void` translates to an Automation `[retval,out]` parameter. A CORBA operation's return value is therefore mapped to the last argument in the corresponding operation of the Automation view interface.

Each Automation method also has an `out` parameter of type `VARIANT`. This parameter appears before the return type and is used to return exception information. Refer to the section "Translation of Exceptions" on page 60 for more information.

If the CORBA operation has no return value, the optional parameter is the last parameter in the corresponding Automation operation. If the CORBA operation does have a return value, the optional parameter appears directly before the return value in the corresponding Automation operation. This is because the return value must always be the last parameter.

Visual Basic

```
Dim interest, amount As Single
...
' Get a reference to an Account object:
accountObj.makeDeposit amount, balance
interest = accountObj.calculateInterest
```

Translation of Inheritance

Single Inheritance

A hierarchy of singly-inherited OMG IDL interfaces is mapped to an identical hierarchy of Automation view interfaces. For example, the following interface account and its derived interface checkingAccount:

```
// OMG IDL
{
   interface account
   {
      attribute float balance;
      readonly attribute string owner;
      void makeLodgement(in float amount, out float balance);
      void makeWithdrawal(in float amount, out float theBalance);
   };

   interface checkingAccount:account
   {
   readonly attribute float overdraftLimit;
   boolean orderChequeBook();
   };
};
```

translate to the following Automation view interfaces:

```
// MIDL
[oleautomation, dual, uuid(...)]
interface account:IDispatch
{
   HRESULT makeLodgement  ([in] float amount,
                           [out] float * balance),
                           [optional, out] VARIANT * excep_OBJ);
```

```
   HRESULT makeWithdrawal ([in] float amount,
                          [out] float * balance),
                          [optional, out] VARIANT * excep_OBJ);
   [propget] HRESULT balance([retval,out] float * val);
   [propput] HRESULT balance([in] float balance);
   [propget] HRESULT owner([retval,out] BSTR * val);
};

[oleautomation, dual, uuid(...)]
interface checkingAccount:account
{
   HRESULT orderChequeBook([optional, out] VARIANT * excep_OBJ,
                          [retval, out] short * val);
   [propget] HRESULT overdraftLimit ([retval, out] short * val);
};
```

Multiple Inheritance

Automation does not support multiple inheritance. Therefore, a direct mapping of a CORBA inheritance hierarchy using multiple inheritance is not possible. This mapping splits such a hierarchy, at the points of multiple inheritance, into multiple singly-inherited strands.

The mechanism for determining which interfaces appear on which strands is based on a left branch traversal of the inheritance tree. Figure 5.2 on page 50 is an example of a CORBA interface hierarchy.

In Figure 5.2 the hierarchy can be read as follows:

- Account and Simple derive from Bank.

- CheckingDetails derives from Account and Simple.

- Test derives from CheckingDetails and Miscellaneous.

```
// OMG IDL
interface Bank {
void opBank();
attribute long val;
};
interface Account : Bank {
void opAccount();
};
interface Simple : Bank {
void opSimple();
};
interface CheckingDetails : Account, Simple {
void opCheckingDetails();
};
interface Miscellaneous {
void opMiscellaneous();
};
interface Test : CheckingDetails, Miscellaneous {
void opTest();
};
```

Bank

Account   **Simple**

CheckingDetails

Miscellaneous

Test

Figure 5.2: *Example of a CORBA Interface Hierarchy*

In this example, the CORBA hierarchy is mapping to two Automation single inheritance hierarchies: `Bank-Account-CheckingDetails` and `Bank-Simple`. The leftmost strand will be the main strand. In this example, this is `Bank-Account-CheckingDetails`. To accomodate access to all of the object's methods, the operations of the secondary strands are aggregated into the interface of the main strand at points of multiple inheritance. In this example, the operations of `Simple` are therefore added to `CheckingDetails`. This means `CheckingDetails` has all the methods of the hierarchy, and an Automation controller holding a reference to `CheckingDetails` will be able to access all the methods of the hierarchy without having to call QueryInterface.

The following OMG IDL code represents a hierarchy that is based on the example shown in Figure 5.2:

```
// OMG IDL
{
   interface Bank
   {
      void Op1a();
      void Op1b();
   };
   interface Account : Bank
   {
      void Op2a();
      void Op2b();
   };
   interface Simple : Bank
   {
      void Op3a();
      void Op3b();
   };
   interface CheckingDetails : Simple, Account
   {
      void Op4a();
      void Op4b();
   };
};
```

This translates to the following two Automation view hierarchies:

```
// MIDL
// strand 1:Bank-Account-CheckingDetails
[oleautomation, dual, uuid(...)]
interface Bank:IDispatch
{
   HRESULT Op1a([optional, out] VARIANT * excep_OBJ);
   HRESULT Op1b([optional, out] VARIANT * excep_OBJ);
}
[oleautomation, dual, uuid(...)]
interface Account:Bank
{
   HRESULT Op2a([optional, out] VARIANT * excep_OBJ);
   HRESULT Op2b([optional, out] VARIANT * excep_OBJ);
}
```

```
[oleautomation, dual, uuid(...)]
interface CheckingDetails:Account
{
   // Aggregated operations of Simple
   HRESULT Op3a([optional, out] VARIANT * excep_OBJ);
   HRESULT Op3b([optional, out] VARIANT * excep_OBJ);
   // Normal operations of CheckingDetails
   HRESULT Op4a([optional, out] VARIANT * excep_OBJ);
   HRESULT Op4b([optional, out] VARIANT * excep_OBJ);
}
// strand 2:Bank-Simple
[oleautomation, dual, uuid(...)]
interface Simple:Bank
{
   HRESULT Op3a([optional, out] VARIANT * excep_OBJ);
   HRESULT Op3b([optional, out] VARIANT * excep_OBJ);
}
```

Translation of Complex Types

Translation is straightforward where there is a direct Automation counterpart
for a CORBA type. However, OMG IDL includes a number of types that do not
have counterparts in MIDL. This section describes how translation is achieved
for the following complex types:

- Constructed types

- Structs

- Unions

- Sequences

- Arrays

- Exceptions

- Anys

Translation of Constructed Types

OMG IDL constructed types such as `struct`, `union` and `exception` translate to pseudo-Automation interfaces. The Automation/CORBA Interworking standard chose this translation because Automation does not allow Automation constructed types as valid parameter types. Pseudo-objects, which implement pseudo-Automation interfaces, do not expose the `IForeignObject` interface. Instead, the matching Automation interface for a constructed type exposes the interface `DIForeignComplexType`.

Creating Constructed OMG IDL Types

To create a complex OMG IDL type, you can use the function `CreateType()` that is defined on `DICORBAFactoryEx`. The `CreateType()` function creates an Automation object that is an instance of an OMG IDL constructed type.

`CreateType()` has the following prototype:

`CreateType([in] IDispatch* scope, [in] BSTR typename)`

The `scope` parameter refers to the scope in which the type should be interpreted. To indicate global scope, pass `Nothing` to this parameter.

The `typename` parameter is the name of the complex type you wish to create.

You can create an object that represents an OMG IDL constructed type in a client in order to pass it as an `in` or `inout` parameter to an OMG IDL operation. You can create an object that represents an OMG IDL constructed type in a server in order to return it as an `out` or `inout` parameter, or return value, from an OMG IDL operation.

Examples of the use of `CreateType()` to create structs, unions and exceptions are shown in "Translation of Structs" on page 53, "Translation of Unions" on page 55 and "Translation of Exceptions" on page 60.

Translation of Structs

An OMG IDL `struct` translates to an Automation interface of the same name that supports the `DICORBAStruct` interface. `DICORBAStruct`, in turn, derives from the `DIForeignComplexType` interface. `DICORBAStruct` does not define any methods. It is used to identify that the interface is translated from a `struct`. For example:

```
// OMG IDL
struct AccountDetails
{
   long number;
   float balance;
};
```

is translated as if it were defined as:

```
// OMG IDL
interface AccountDetails
{
   attribute long number;
   attribute float balance;
};
```

Figure 5.3 shows the Automation view of the translation.
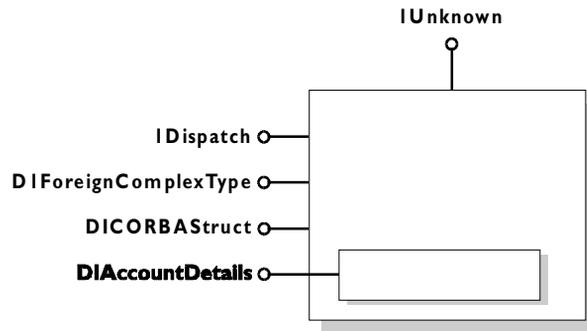
Figure 5.3: *Automation View of the OMG IDL Struct* AccountDetails

Visual Basic

```
Dim ObjFactory As CORBA_Orbix.DICORBAFactoryEx
Dim details As BankBridge.DIAccountDetails
...
Set details = ObjFactory.CreateType(Nothing, "AccountDetails")

details.balance = 1297.66
details.number = 109784
```

Translation of Unions

An OMG IDL union translates to an Automation interface that exposes the `DICORBAUnion` interface. `DICORBAUnion`, in turn, derives from the `DIForeignComplexType` interface. `DICORBAUnion` does not define any methods. It is used to identify that the interface is translated from a union.

In order to describe CORBA `union` types that support multiple case labels per union branch, the `DICORBAUnion2` interface is defined. This provides two additional accessors as follows:

```
// MIDL
[oleautomation, dual, uuid(...)]
interface DICORBAUnion2:DICORBAUnion
{
   HRESULT SetValue([in] long disc, [in] VARIANT val);
   [propget, id(-4)]
   HRESULT CurrentValue([out, retval] VARIANT * val);
};
```

The `SetValue` method can be used to set the discriminant and value simultaneously. The `CurrentValue` method will use the current discriminant value to initialise the `VARIANT` with the union element. All mapped unions should support the `DICORBAUnion2` interface.

The following OMG IDL `union` type:

```
// OMG IDL
interface A { ... };

union U switch(long) {
   case 1: long l;
   case 2: float f;
   default: A obj;
};
```

translates to the following Automation pseudo-union:

```
// MIDL
interface DIU : DICORBAUnion2{
   [propget] HRESULT get_UNION_d([retval,out] long * val);
   [propget] HRESULT get_1([retval,out] long * l);
   [propget] HRESULT put_1([in] long l);
   [propget] HRESULT get_1([retval,out] float * f);
   [propget] HRESULT put_1([in] float f);
```

```
    [propget] HRESULT get_A([retval,out] DIA ** val);
    [propget] HRESULT put_A([in] DIA * val);
};
```

In this case, the mapped Automation dual interface derives from
`DICORBAUnion2`. The property `UNION_d` returns the value of the discriminant.
The discriminant indicates the type of value that the union holds. In this example,
the value of `UNION_d` is 2 if the union `U` contains a `float`.

For each member of the union, a property is generated in the matching MIDL
interface to read the value of the member and to set the value of the member.
The property to set the value of a union member also sets the value of the
discriminant.

It is an error to attempt to read the value of a member using a method that does
not match the type of the discriminant.

The mapping for the OMG IDL default label will be ignored if the cases are
exhaustive over the permissible cases (for example, if the switch type is `boolean`
and a case `TRUE` and case `FALSE` are both defined).

Figure 5.4 shows the Automation view of the translation of the OMG IDL union
`U`.
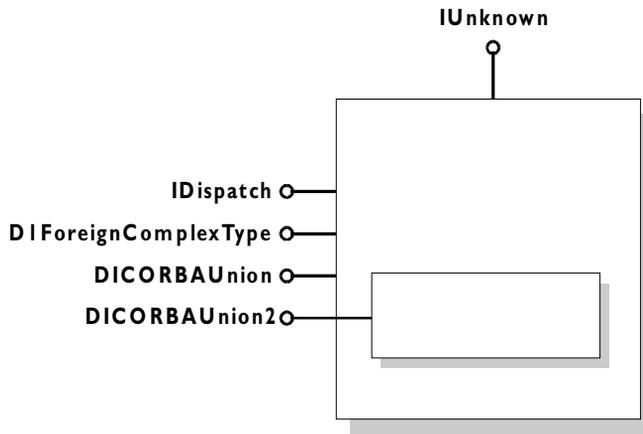
Figure 5.4: *Automation View of the OMG IDL Union* `u`

Visual Basic

```
Dim ObjFactory As CORBA_Orbix.DICORBAFactoryEx
Dim myUnion As DIU
.
.
.
Set myUnion =
   ObjFactory.CreateType(Nothing, "U")

myUnion.s = "This is a string"

Select Case(myUnion.UNION_d())
   Case 1: MsgBox ("Union (long):" & Str$(myUnion.l)
   Case 1: MsgBox ("Union (float):" & Str$(myUnion.f)
   Case 1: MsgBox ("Union (string):" & Str$(myUnion.s)
   Case Else : MsgBox ("Union contains object reference")
End Select
```

Translation of Sequences

An OMG IDL sequence can be mapped to an Automation SafeArray or an OLE collection. The method call `SetSafeArrayMapping` on `DIOrbixORBObject` determines the type of mapping in effect. If `SetSafeArrayMapping` is set to `true`, sequences are mapped to SafeArrays; otherwise, sequences are mapped to OLE collections. You should make this setting only once in your application. Switching `SetSafeArrayMapping` between `true` and `false` could result in undefined behaviour.

Mapping to SafeArrays

If the method call `SetSafeArrayMapping` on `DIOrbixORBObject` is set to `true`, an OMG IDL bounded or unbounded sequence is mapped to a `VARIANT` containing an Automation SafeArray. An OMG IDL bounded sequence is mapped to a fixed-size SafeArray. If you pass a SafeArray that contains a different number of elements than that required by the bounded sequence, it is automatically resized to the correct size. An OMG IDL unbounded sequence is mapped to an empty SafeArray that can grow or shrink to any size.

Mapping to OLE Collections

If the method call `SetSafeArrayMapping` on `DIOrbixORBObject` is set to `false`, an OMG IDL bounded or unbounded sequence is mapped to a `VARIANT` containing an OLE collection object that exposes the `DCollection` interface. Each collection object exposes the following `DCollection` Automation properties and methods:

Method	Type	Description
Count	Read/Write Property	Get/Set the number of elements in the collection.
Item	Read/Write Parameterised Property	Get/Set access to individual elements in the collection.

As an alternative to the `Item` property, each sequence object also exposes the following methods for use in controllers that do not support parameterised properties:

Method	Type	Description
getItem	Method	Get/Set the number of elements in the collection.
setItem	Method	Get/Set access to individual elements in the collection.

Refer to "OrbixCOMet API" on page 241 for a full description of the MIDL definitions for the `DCollection` interface.

The following OMG IDL definition:

```
OMG IDL
module ModBank {
   interface Transaction {...};

   // A bounded sequence
   typedef sequence<Transaction, 30> TransactionList;

   interface Account {
      readonly attribute TransactionList statement;
      readonly attribute float balance;
      ...
   };
```

```
   // An unbounded sequence
   typedef sequence<Account> AccountList;

   interface Bank {
      readonly attribute AccountList personalAccounts;
      AccountList sortAccounts(in AccountList toSort)
      ...
   };
};
```

translates to:

```
// MIDL
typedef [public] VARIANT ModBank_TransactionList

[oleautomation, dual, uuid(...)]
interface DIModBank_Transaction: IDispatch {}

typedef [public] VARIANT ModBank_AccountList;
[oleautomation, dual, uuid(...)]
interface DIModBank_Account: IDispatch {
   [propget] HRESULT statement ([retval, out] IDispatch**
                                 IT_retval);
   [propget] HRESULT balance ([retval, out] float* IT_retval);
};

[oleautomtion, dual, uuid(...)]
interface DIModBank_Bank: IDispatch {
   [propget] HRESULT personalAccounts ([retval,out] IDispatch**
                                        IT_retval);
   HRESULT sortAccounts ([in] IDispatch* toSort,
                         [optional, out] VARIANT* IT_Ex,
                         [retval, out] IDispatch** IT_retval);
};
```

Visual Basic

```
' Visual Basic
Dim myBank As IT_Library_Bank.DIModBank_Bank
Dim myAccounts As Variant
Dim tmpAccount As IT_Library_Bank.DIModBank_Account
Dim myBalance As Single
```

```
' Obtain a reference to a Bank object
Set myBank = ...
Set myAccounts = ORBFactory.CreateType (Nothing,
"ModBank/AccountsList")

For Each acc in myAccounts
   acc.balance = 0.00
Next acc

' Access a member of myAccounts
myBalance = myAccounts(4).balance

' Obtain a reference to a member of myAccounts
Set tmpAccount = myAccounts(7)
myBalance = tmpAccount.balance
```

Translation of Arrays

The mapping for an OMG IDL array is similar to that for an OMG IDL sequence. OMG IDL arrays can map to either Automation SafeArrays or OLE collections.

Mapping to SafeArrays

Multidimensional OMG IDL arrays map to VARIANTs containing multidimensional SAFEARRAYs. The order of dimensions in the OMG IDL array from left to right corresponds to ascending order of dimensions in the SAFEARRAY. An error will occur if the number of dimensions in an input SAFEARRAY does not match the CORBA type.

Mapping to OLE Collections

Only single dimension arrays can be supported when mapping to OLE collections.

Translation of Exceptions

The CORBA model uses exceptions to report error information. Exceptions are classified into two categories as follows:

1. System exceptions can be raised by any operation. A standard set of system exceptions is defined by CORBA, and Orbix provides a number of additional system exceptions. These system exceptions are listed in "System Exceptions" on page 329.

2. User exceptions are defined in OMG IDL, and an OMG IDL operation can optionally specify that it might raise a specific set of user exceptions. An OMG IDL operation can also raise a system exception but this is not defined at the OMG IDL level.

User Exceptions

An OMG IDL user-defined exception translates to an Automation interface that has a corresponding property for each member of the exception. The Automation interface derives from the DICORBAUserException interface. For example:

```
// OMG IDL
exception Reject
{
      string reason;
};
```

translates to:

```
// MIDL
[oleautomation, dual, uuid(...)]
interface DIreject : DICORBAUserException
{
   [propget] HRESULT reason([retval,out] BSTR reason);
}
```

Figure 5.5 on page 62 provides an Automation view of the translation of the exception Bank::Reject:

Exceptions are fully explained in "System Exceptions" on page 329.
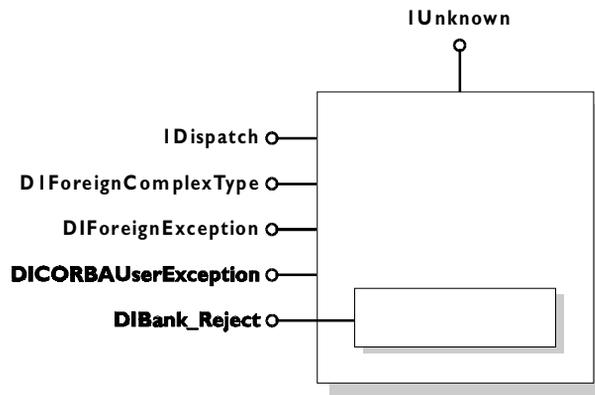
Figure 5.5: *Automation View of Bank_Reject*

System Exceptions

A CORBA system exception translates to the Automation interface
`DICORBASystemException` that derives from `DIForeignException`:

```
// MIDL
[oleautomation, dual, uuid(...)]
interface DICORBASystemException : DIForeignException
{
   [propget] HRESULT EX_minorCode([retval,out] long * val);
   [propget] HRESULT EX_completionStatus([retval,out] long * val);
};
```

The attribute `EX_minorCode` defines the type of system exception raised, while
`EX_completionStatus` has one of the following numeric values:

```
COMPLETION_YES = 0
COMPLETION_NO = 1
COMPLETION_MAYBE = 2
```

These values are specified as an `enum` in the type library information:

```
typedef enum {COMPLETION_YES, COMPLETION_NO,
              COMPLETION_MAYBE}
CORBA_CompletionStatus;
```

This interface is explained in "OrbixCOMet API" on page 241.

Translation of the Any Type

The OMG IDL `any` data type translates to an OLE `VARIANT` type. If the `any` contains a simple data type, this maps to a `VARIANT` containing a corresponding simple type as shown in Table 5.1 on page 42. If the `any` contains a complex type, the `VARIANT` will contain an `IDispatch` view of the CORBA type. If the `any` contains a CORBA `sequence` or `array` type, the `VARIANT` will contain either an Automation SafeArray or an OLE Collection, depending on the setting of the `SetSafeArrayMapping` method call on `DIOrbixORBObject`.

Context Clause

The CORBA standard OMG IDL to Automation mapping does not specify a translation for OMG IDL contexts.

Translation of Object References

When an OMG IDL operation returns an object reference or passes an object reference as an operation parameter, this is translated as a reference to an `IDispatch` interface in MIDL. For example:

```
// OMG IDL
interface Simple
{
   attribute short shortTest;
};
interface ObjRefTest
{
   attribute Simple simpleTest;
   Simple simpleOp(in Simple inTest, out Simple outTest,
                   inout Simple inoutTest);
   };
};
```

translates to:

```
// MIDL
[oleautomation, dual, uuid(...)]
interface DISimple : IDispatch
{
   [propget] HRESULT shortTest([retval,out] short * val);
   [propput] HRESULT shortTest([in] short shortTest);
};
[oleautomation, dual, uuid(...)]
interface DIObjRefTest : IDispatch
{
   HRESULT simpleOp([in] DISimple *inTest,
                    [out] DISimple **outTest,
                    [in,out] DISimple **inoutTest,
                    [optional,out] VARIANT * excep_OBJ,
                    [retval,out] DISimple ** val);
   [propget] HRESULT simpleTest([retval,out] DISimple ** val);
   [propput] HRESULT simpleTest ([in] DISimple * simpleTest);
};
```

Object Reference Parameters and IForeignObject

An Automation view interface must expose the IForeignObject interface in addition to the interface that is isomorphic to the mapped CORBA interface. IForeignObject provides a mechanism to extract a valid CORBA object reference from a view object.

Consider an Automation view object B that is passed as an in parameter to an operation M in view A. Operation M must somehow convert view B to a valid CORBA object reference. The sequence of events involving IForeignObject::GetForeignReference is as follows:

1. The client calls Automation-View-A::M, passing an IDispatch-derived pointer to Automation-View-B.

2. Automation-View-A::M calls IDispatch::QueryInterface for IForeignObject.

3. Automation-View-A::M calls IForeignObject::GetForeignReference to get the reference to the CORBA object of type B.

4. Automation-View-A::M calls CORBA-Stub-A::M with the reference, narrowed to interface type B, as the object reference in parameter.

Visual Basic

```
Dim bankObj As BankBridge.DIBank
Dim accountObj As BankBridge.DIAccount

' Get a reference to a Bank object
Set bankObj = ...

' Get a reference to an Account object as a return value
Set accountObj = bankObj.newAccount "John"

' Use the returned object reference
accountObj.makeDeposit 231.98

' finished, delete the account
bankobj.deleteAccount accountObj
```

Translation of Modules

An OMG IDL definition contained within the scope of an OMG IDL module is translated to its corresponding Automation definition by prefixing the name of the Automation type definition with the name of the module. For example:

```
// OMG IDL
module Finance {
   interface Bank {
   ...
   };
};
```

translates to:

```
[oleautomation, dual, uuid(...), helpstring("Finance_Bank")]
interface DIFinance_Bank : IDispatch {
   ...
}
```

Visual Basic

```
Dim bankObj As DIFinance_Bank
```

Translation of Constants

No Automation code is generated for an OMG IDL constant definition because Automation does not have the concept of a constant. However, code can be generated for an Automation controller, if appropriate.

If an OMG IDL constant is contained within an interface or module, its translated name is prefixed by the name of the interface or module in the Automation controller language. (Refer to "Translation of Scoped Names" on page 68 for more details.)

An OMG IDL constant can be represented as a Visual Basic or PowerBuilder constant definition.

Visual Basic

```
// OMG IDL
const long Max = 1000;
```

can be represented as:

```
' Visual Basic
' In .BAS file
Global Const Max = 1000
```

PowerBuilder

```
// OMG IDL
const long Max = 1000;
```

can be represented as:

```
CONSTANT long Max=100
```

Translation of Enumerated Types

A CORBA enum translates to an Automation enum. For example, the OMG IDL definition:

```
// OMG IDL
{
enum colour { white, blue, red };
   interface foo
   {
      void op1(in colour col);
   };
};
```

translates to:

```
// MIDL
typedef [public,v1_enum] { white, blue, red } colour;
[oleautomation, dual, uuid(...)]
interface foo:IDispatch
{
   HRESULT op1([in] colour col, [optional, out]
   VARIANT * excep_OBJ);
}
```

Because Automation maps enum parameters to the platform's integer type, a run-time error will occur in the following situations:

- If the number of elements in the CORBA enum exceeds the maximum value of an integer.

- If an actual parameter applied to the mapped parameter in the Automation view interface exceeds the maximum value of the enum.

If an OMG IDL enum is contained within an interface or module, its translated name is prefixed by the name of the interface or module in the Automation controller language. (Refer to "Translation of Scoped Names" on page 68 for more details.)

If the enum is declared at global OMG IDL scope, the name of the enum should also be included in the constant name.

Translation of Scoped Names

An OMG IDL scoped name translates to an Automation identifier where the scope operator :: is replaced with _ (that is, an underscore). For example, the following OMG IDL:

```
// OMG IDL
module Finance {
   interface Bank {
      struct PersonnelRecord {
      ...
      };
      void addRecord(in PersonnelRecord r);
   ...
   };
};
```

yields the scoped name Finance::Bank::PersonnelRecord. This would then translate to Finance_Bank_PersonnelRecord.

Translation of Typedefs

The translation of OMG IDL typedef definitions to Automation depends on the OMG IDL type for which the typedef is defined.

No translation is provided for typedef definitions for the basic OMG IDL types that are listed in Table 5.1 on page 42. Therefore, a Visual Basic programmer cannot make use of these typedef definitions for basic types. The following OMG IDL example:

```
// OMG IDL
module MyModule{
   module Module2{
      module Module3{
         interface foo{};
      };
   };
};
typedef MyModule::Module2::Module3::foo bar;
```

can be used as follows:

```
' in Visual Basic
Dim a as Object
Set a = theOrb.GetObject("MyModule/Module2/Module3/foo")
' Release the object
Set a = Nothing
' Create the object using a typedef alias
Set a = theOrb.GetObject("bar")
```

A `typedef` definition is most often used for array and sequence definitions.

6

Mapping Automation Objects to CORBA

Automation types are defined in Microsoft IDL (MIDL). CORBA types are defined in OMG IDL. To allow interworking between Automation and CORBA, MIDL types must be translated to OMG IDL. This chapter outlines how translation of Automation objects to CORBA is achieved.

For the purposes of illustration, this chapter describes a textual mapping between MIDL and OMG IDL. OrbixCOMet itself does not require this textual mapping to take place, because it includes a dynamic marshalling engine. The textual mappings shown in this chapter are actually performed by OrbixCOMet at application runtime.

Translation of Basic Types

Automation basic types translate to compatible types in OMG IDL. Table 6.1 shows the mapping for each type.

MIDL	Description	OMG IDL	Description
`VARIANT_BOOL`	16-bit integer 0 = FALSE -1 = TRUE	`boolean`	Unsigned char, 8-bit 0 = FALSE 1 = TRUE
`UI1`	8-bit unsigned integer	`octet`	8-bit quantity
`short`	16-bit integer	`short`	16-bit integer
`double`	IEEE 64-bit float	`double`	IEEE 64-bit float
`float`	IEEE 32-bit float	`float`	IEEE 32-bit float
`long`	32-bit integer	`long`	32-bit integer
`BSTR`	Length-prefixed string	`string`	Null terminated 8-bit character array
`CURRENCY`	8-byte fixed-point number	`COM::Currency`	OMG IDL struct `currency`
`DATE`	64-bit floating point	`double`	IEEE 64-bit float
`SCODE`	Built-in error type	`long`	32-bit integer

Table 6.1: *Translation of Automation Basic Types to OMG IDL*

There is not an exact correspondence between the types supported by Automation and the types supported by OMG IDL. These differences mean that certain translations can result in a run-time error. For example:

- Translating OMG IDL `COM::Currency` to Automation `CURRENCY`, if the supplied `COM::Currency` value does not translate to a meaningful Automation `CURRENCY` value.

- Translating OMG IDL `double` to Automation `DATE`, if the OMG IDL double value is negative or converts to an impossible date.

Translation of Strings

An Automation BSTR translates to an OMG IDL string. For example:

```
// MIDL
BSTR address;
```

translates to:

```
// OMG IDL
// This definition might appear within a struct
// definition.
string address;
```

Translation of Interfaces

An Automation interface translates in a straightforward fashion to an OMG IDL interface. For example, the Automation interface account:

```
// MIDL
[odl, dual, uuid(...)]
interface account : IDispatch
{
   [propget] HRESULT balance([retval,out] float * ret);
};
```

translates to the following OMG IDL interface:

```
// OMG IDL
interface account
{
   readonly attribute float balance;
};
```

If the Automation interface does not have a parameter with the [retval,out] attributes, its return type is mapped to long. This allows COM SCODE values to be passed through to the CORBA client.
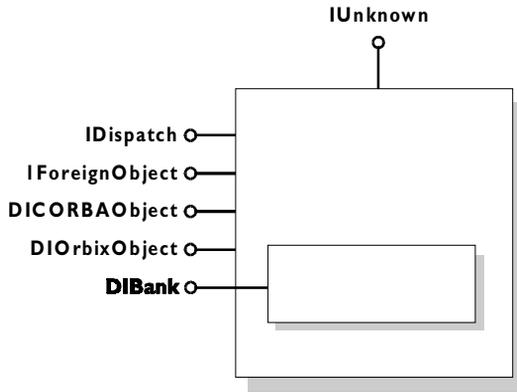
Figure 6.1: *Automation View of Bank Interface*

The Automation view in the bridge supports the interface DIBank. Any Automation controller can use the DIBank interface to invoke operations on the Automation view. The view forwards the request to the target Bank object in the CORBA server.

The Automation view also supports the following interfaces:

- IUnknown and IDispatch required by all Automation objects.

- DIForeignObject required by all views.

- DICORBAObject required by all CORBA objects.

- DIOrbixObject supported by all Orbix objects.

Translation of Properties

An Automation property translates to an OMG IDL attribute. For example:

```
// MIDL
[odl, dual, uuid(...)]
interface DIaccount : IDispatch {
   [propput] HRESULT balance ([in] float balance);
   [propget] HRESULT balance ([retval,out] float * ret);
   [propget] HRESULT owner ([retval,out] BSTR * ret);
```

```
        HRESULT makeLodgement ([in] float amount,
                               [out] float * balance,
                               [optional, out] VARIANT * excep_OBJ);
        HRESULT makeWithdrawal ([in] float amount,
                               [out] float * balance,
                               [optional, out] VARIANT * excep_OBJ);
}
```

translates to:

```
// OMG IDL
interface account
   attribute float balance;
   readonly attribute string owner;
   long makeLodgement(in float amount, out float balance);
   long makeWithdrawal(in float amount, out float balance);
};
```

An Automation property that has a method to get the value, and a method to set the value, translates to a normal OMG IDL attribute.

An Automation property that has a method to get the value translates to an OMG IDL `readonly` attribute.

Translation of Methods

An Automation method translates to an OMG IDL operation.

Parameters

An Automation `[in]` parameter translates to an OMG IDL `in` parameter.

An Automation `[out]` parameter translates to an OMG IDL `out` parameter.

An Automation `[in,out]` parameter translates to an OMG IDL `inout` parameter.

Translation of Inheritance

A hierarchy of Automation interfaces is mapped to an identical hierarchy of OMG IDL view interfaces. For example, the following interface `account` and its derived interface `checkingAccount`:

```
// MIDL
[odl, dual, uuid(...)]
interface account:IDispatch
{
   [propput] HRESULT balance([in] float balance);
   [propget] HRESULT balance([retval,out] float * ret);
   [propget] HRESULT owner([retval,out] BSTR * ret);
   HRESULT makeLodgement([in] float amount,
                         [out] float * balance);
   HRESULT makeWithdrawal([in] float amount,
                          {out} float * balance);
};
interface checkingAccount: account
{
   [propget] HRESULT overdraftLimit ([retval,out] short * ret);
   HRESULT orderChequeBook([retval,out] short * ret);
};
```

translates to the following OMG IDL view interface `account`:

```
// OMG IDL
interface account
{
   attribute float balance;
   readonly attribute string owner;
   long makeLodgement(in float amount, out float balance);
   long makeWithdrawal(in float amount, out float theBalance);
};
interface checkingAccount: account
{
   readonly attribute short overdraftLimit;
   short orderChequeBook();
};
```

Translation of SafeArrays

Automation SafeArrays translate to OMG IDL unbounded sequences.

When SafeArrays are `in` parameters, the view method uses the SafeArray API to dynamically repackage the SafeArray as an OMG IDL sequence.

When arrays are `out` parameters, the view method uses the SafeArray API to dynamically repackage the OMG IDL sequence as a SafeArray.

Multidimensional SafeArrays

Multidimensional SafeArrays are mapped to an identical linear format and then to a sequence in the normal way. This is because the bounding information for each dimension is only available at runtime.

The linearisation of the multidimensional SafeArray is performed as follows:

1. The number of elements in the linear sequence is the product of the dimensions.

2. The position of each element is deterministic. For example, for a SafeArray with dimensions d0, d1 and d2, the location of an element [p0][p1][p2] is defined as follows:

   `pos[p0][p1][p2] = p0*d1*d2 + p1*d2 + p2`

   For example, a SafeArray with dimensions 5, 8, 9 maps to a linear sequence with a run-time bound of 5*8*9=360. This yields valid offsets 0-359. In this example, the real offset to the element at location [4][5][1] is 4*8*9 + 5*9 + 1 = 334.

Translation of Exceptions

Automation exceptions translate to OMG IDL exceptions.

Automation system error codes map to OMG IDL standard exceptions. Automation user-defined error codes map to OMG IDL user exceptions.

An Automation method with a `HRESULT` return value and an argument marked as a `[retval]` maps to an OMG IDL method whose return value is mapped from the `[retval]` argument.

An Automation method with a `HRESULT` return value and no argument marked as a `[retval]` maps to a CORBA interface with a long return value.

Translation of Variants

An Automation VARIANT translates to an OMG IDL any type. If the VARIANT contains a DATE or CURRENCY element, these are mapped as shown in "Translation of Basic Types" on page 72.

Translation of Object References

The following MIDL defines a simple interface and another interface that references simple as an in parameter, an out parameter, an inout parameter and a return value:

```
// MIDL
[odl, dual, uuid(...)]
interface Simple: IDispatch
{
   [propget] HRESULT shortTest([retval, out] short * val);
   [propput] HRESULT shortTest([in] short val);
}

[odl, dual, uuid(...)]
interface ObjRefTest: IDispatch
{
   [propget] HRESULT simpleTest([retval, out] Simple ** val);
   [propput] HRESULT simpleTest([in] Simple *pSimpleTest);
   HRESULT simpleOp([in] Simple *inTest,
                    [out] Simple **outTest,
                    [in,out] Simple **inoutTest,
                    [retval, out] Simple **val);
}
```

This translates to the following OMG IDL:

```
// OMG IDL
interface Simple
{
   attribute short shortTest;
};

interface ObjRefTest
{
   attribute Simple simpleTest;
   Simple simpleOp(in Simple inTest,
                   out Simple outTest,
                   inout Simple inoutTest);
};
```

Translation of Enumerated Types

An Automation enum translates to an OMG IDL enum. For example:

```
// MIDL
typedef enum colour {red=2, green=0, blue=1};
[odl, dual, uuid(...)]
interface foo: IDispatch{
   HRESULT op1([in] colour col);
}
```

translates to:

```
// OMG IDL
enum colour { green, blue, red };
interface foo{
   long op1(in colour col);
};
```

An Automation enumeration is mapped to OMG IDL such that the enumerators are ordered in ascending order of their value. Because OMG IDL does not support explicitly tagged enumerators, the CORBA view of an Automation object must maintain the mapping of the values of the enumeration.

Translation of Typedefs

Automation typedefs map directly to OMG IDL typedefs.

The only exception to this is an Automation `enum` that is required to be a typedef. In this case, the following MIDL:

```
// MIDL
typedef enum {red, green, blue} colour;
[odl, dual, uuid(...)]
interface foo: IDispatch{
   HRESULT op1([in] colour col,
               [optional,out] VARIANT * excep_OBJ);
}
```

translates to:

```
// OMG IDL
enum colour {red, green, blue};
interface foo
{
   void op1(in colour col);
};
```

7

Mapping CORBA Objects to COM

CORBA types are defined in OMG IDL. COM types are defined in Microsoft IDL (MIDL). To allow interworking between CORBA and COM, OMG IDL types must be translated to MIDL. This chapter outlines how translation of CORBA objects to COM is achieved.

For the purposes of illustration, this chapter describes a textual mapping between OMG IDL and MIDL. OrbixCOMet itself does not require this textual mapping to take place, because it includes a dynamic marshalling engine. The textual mappings shown in this chapter are actually performed by OrbixCOMet at runtime.

Translation of Basic Types

OMG IDL basic types translate to compatible types in COM. Table 7.1 shows the mapping for each type.

OMG IDL	Description	MIDL	Description
boolean	Unsigned char, 8-bit 0 = FALSE 1 = TRUE	boolean	16-bit integer 0 = FALSE 1 = TRUE
char	8-bit quantity	char	8-bit quantity
double	IEEE 64-bit float	double	IEEE 64-bit float
float	IEEE 32-bit float	float	IEEE 32-bit float
long	32-bit integer	long	32-bit integer
octet	8-bit quantity	unsigned char	8-bit quantity
short	16-bit integer	short	16-bit integer
unsigned long	32-bit integer	unsigned long	32-bit integer
unsigned short	16-bit integer	unsigned short	16-bit integer
unsigned char	8-bit quantity	unsigned char	8-bit quantity

Table 7.1: *Translation of OMG IDL Basic Types to COM*

Translation of Strings

An OMG IDL string maps to a MIDL LPSTR, which is a null-terminated 8-bit character string.

Unbounded Strings

The following definition for an OMG IDL unbounded string:

```
// OMG IDL
typedef string UNBOUNDED_STRING;
```

translates to:

```
// MIDL
typedef [string, unique] char * UNBOUNDED_STRING;
```

Bounded Strings

The following definition for an OMG IDL bounded string:

```
// OMG IDL
const long N = ...;
typdef string<N>BOUNDED_STRING;
```

translates to:

```
// MIDL
const long N = ...;
typdef [string, unique] char (*BOUNDED_STRING) [N];
```

Translation of Interfaces

An OMG IDL `RepositoryId` translates to a MIDL `IID` (Interface ID) that is similar to the `DCE UUID` (or identical in the case of Windows32).

The mapping is achieved by using a derivative of the RSA Data Security Inc. MD5 Message-Digest algorithm. The `RepositoryId` is fed into the algorithm to produce a 128-bit hash identifier.

When the `RepositoryId` is a `DCE UUID`, the `DCE UUID` is used as the `IID` for a COM view of a CORBA interface.

When the `RepositoryId` is not a `DCE UUID`, the `IID` generated from the `RepositoryId` is used for a COM view of a CORBA interface.

Translation of Attributes

An OMG IDL attribute translates to a MIDL attribute. For example, in the case of the following OMG IDL:

```
// OMG IDL
struct CustomerData
{
   CustomerId Id;
   string Name;
   string SurName;
};

#pragma ID "BANK::Account" "IDL:BANK/Account:3.1"
interface Account
{
   readonly attribute float Balance;
   float Deposit(in float amount) raises(InvalidAmount);
   float Withdrawal(in float amount) raises(InsufFunds,
                                            InvalidAmount);
   float Close();
};

#pragma ID "BANK::Customer" "IDL:BANK/Customer:1.2"
interface Customer
{
   attribute CustomerData Profile:
};
```

the read-write attribute `Profile` maps to the following MIDL:

```
// MIDL
[object,uuid(...),pointer_default(unique)]
interface IBANK_Customer: IUnknown
{
   HRESULT _get_Profile([out] BANK CustomerData * val);
   HRESULT _put_Profile([in] BANK CustomerData * val);
};
```

and the read-only attribute `Balance` maps to the following MIDL:

```
// MIDL
[object,uuid(..)]
interface IBANK Account: IUnknown
{
   HRESULT _get_Balance([out] float * val);
};
```

A normal attribute translates to a property that has a method to set the value and a method to get the value.

A `readonly` attribute translates to a property that has a method to get the value.

The `get` method returns the attribute value in the parameter tagged [out].

Translation of Operations

An OMG IDL operation translates to a MIDL method. For example:

```
// OMG IDL
#pragma ID "BANK::Teller" "IDL:BANK/Teller:1.2"
interface Teller
{
   Account OpenAccount(in float StartingBalance,
                       in AccountTypes AccountType);
   void Transfer(in Account Account1,
                 in Account Account2
                 in float Amount) raises (InSufFunds);
};
```

translates to:

```
// MIDL
[object,uuid(...),pointer_default(unique)]
interface IBANK_Teller: IUnknown
{
   HRESULT OpenAccount([in] float StartingBalance,
                       [in] IBANK_AccountTypes AccountType,
                       [out] IBANK_Account ** ppiNewAccount);
   HRESULT Transfer([in] IBANK_Account * Account1,
                    [in] IBANK_Account * Account2,
                    [in] float Amount,
                    [out] BANK_TellerExceptions ** ppException);
};
```

Parameters

An OMG IDL `in` parameter translates to a MIDL `[in]` parameter.

An OMG IDL `out` parameter translates to a MIDL `[out]` parameter.

An OMG IDL `inout` parameter translates to a MIDL `[in,out]` parameter.

Return Types

An OMG IDL `return` type translates to a MIDL `[out]` parameter as the last parameter in the signature.

Translation of Inheritance

CORBA and COM have different models for inheritance. CORBA interfaces can be multiply inherited but COM does not support multiple interface inheritance. The following rules therefore apply when mapping a hierarchy of interfaces from CORBA to COM:

- Each OMG IDL interface that does not have a parent is mapped to a MIDL interface deriving from `IUnknown`.

- Each OMG IDL interface that inherits from a single parent interface is mapped to a MIDL interface that derives from the mapping for the parent interface.

- Each OMG IDL interface that inherits from multiple parent interfaces is mapped to a MIDL interface deriving from `IUnknown`. This MIDL interface then aggregates both base interfaces.

- For each CORBA interface, the mapping for operations precedes the mapping for attributes.

Figure 7.1 on page 87 shows an example of a CORBA interface hierarchy.

```
// OMG IDL
interface Bank {
void opBank();
attribute long val;
};
interface Account : Bank {
void opAccount();
};
interface Simple : Bank {
void opSimple();
};
interface CheckingDetails : Account, Simple {
void opCheckingDetails();
};
interface Miscellaneous {
void opMiscellaneous();
};
interface Test : CheckingDetails, Miscellaneous {
void opTest();
};
```
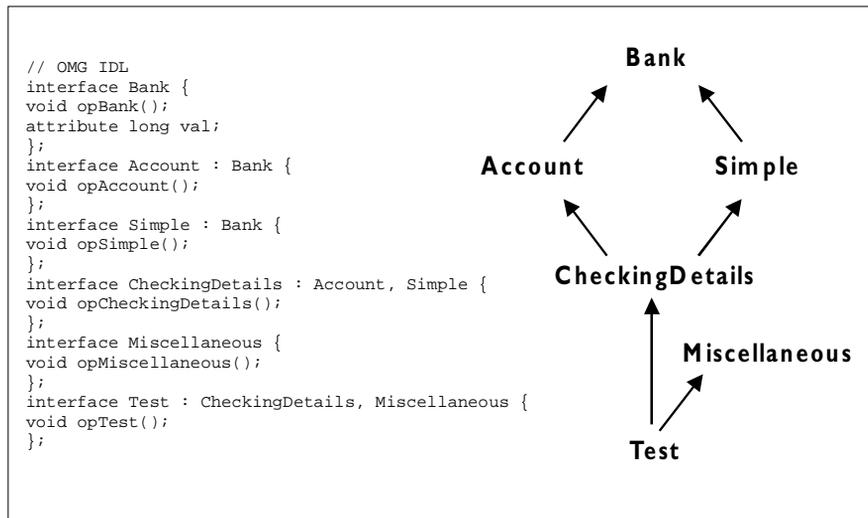
Figure 7.1: *Example of a CORBA Interface Hierarchy*

The following sample of OMG IDL code represents the hierarchy of interfaces shown in Figure 7.1:

```
// OMG IDL
interface Bank
{
   void opBank();
   attribute long val;
};
interface Account:Bank
{
   void opAccount();
};
interface Simple:Bank
{
   void opSimple();
};
```

```
interface CheckingDetails:Account,Simple
{
   void opCheckingDetails();
};
interface Test
{
   void opTest();
};
interface Miscellaneous:CheckingDetails,Test
{
   void opMiscellaneous();
};
```

This translates to the following MIDL:

```
// MIDL
[object,uuid(...)]
interface IBank: IUnknown
{
   HRESULT opBank();
   HRESULT get val([out] long * val);
   HRESULT set val([in] long val);
};
[{object,uuid(...)]
interface IAccount: IBank
{
   HRESULT opAccount();
};
[object,uuid(...)]
interface ISimple: IBank
{
   HRESULT opSimple();
};
[object,uuid(...)]
interface ICheckingDetails: IUnknown
{
   HRESULT opCheckingDetails();
};
[object,uuid(...)]
interface ITest: IUnknown
{
   HRESULT opTest();
};
```

```
[object,uuid(...)]
interface IMiscellaneous: IUnknown
{
   HRESULT opMiscellaneous();
};
```

Note: When the interface defined in OMG IDL is mapped to its corresponding statements in MIDL, the name of the interface is preceded by the letter `I`. If the interface is scoped by OMG IDL modules `::`, this is replaced by an underscore `_` (for example, `foo::bar` maps to `Ifoo_bar`).

Translation of Complex Types

Translation of Constructed Types

OMG IDL constructed types such as `struct`, `union`, `sequence` and `exception` translate to corresponding `struct` types in MIDL as described later in this chapter.

Creating Constructed OMG IDL Types

To create a complex OMG IDL type, you should simply instantiate an instance of its MIDL `struct` type.

You must create an object representing an OMG IDL constructed type in a client in order to pass it as an `in` or `inout` parameter to an OMG IDL operation. You can create an object representing an OMG IDL constructed type in a server in order to return it as an `out` or `inout` parameter, or return value, from an OMG IDL operation.

Translation of Structs

An OMG IDL `struct` translates to a MIDL `struct`. For example:

```
// OMG IDL
typedef ... T0;
typedef ... T1;
typedef ... T2;
...
typedef ... Tn;
struct STRUCTURE
{
   T0 m0;
   T1 m1;
   T2 m2;
...
   Tn mN;
};
```

translates to:

```
// MIDL
typedef ... T0;
typedef ... T1;
typedef ... T2;
...
typedef ... Tn;
typedef struct
   {
      T0 m0;
      T1 m1;
      T2 m2;
      ...
      Tn mN;
   }
   STRUCTURE;
```

Self-referential data types are expanded in the same manner. For example:

```
// OMG IDL
struct A
{
   sequence<A> v1;
};
```

translates to:

```
// MIDL
typedef struct A
{
   struct
   {
      unsigned long cbMaxSize;
      unsigned long cbLengthUsed;
      [size_is(cbMaxSize), length_is(cbLengthUsed), unique]
      struct A * pValue;
   } v1;
} A;
```

Translation of Unions

A discriminated union type in OMG IDL translates to an encapsulated union in MIDL. For example:

```
// OMG IDL
enum UNION_DISCRIMINATOR
{
   dChar=0;
   dShort,
   dLong,
   dFloat,
   dDouble};
union UNION_OF_CHAR_AND_ARITHMETIC
   switch (UNION_DISCRIMINATOR)
   {
      case dChar: char c;
      case dShort: short s;
      case dLong: long l;
      case dFloat: float f:
      case dDouble: double d;
      default: octet v[8];
   };
```

translates to:

```
// MIDL
typedef enum [v1_enum,public]
{
   dchar=o,
   dshort,
   dLong,
   dFloat,
   dDouble,
} UNION_DISCRIMINATOR;

typedef union switch (UNION_DISCRIMINATOR DCE_d)
   {
   case dChar: char c;
   case dShort: short s;
   case dLong: long l;
   case dFloat: float f;
   case dDouble: double d;
   default: byte v[8];
   } UNION_OF_CHAR_AND_ARITH
```

Translation of Sequences

Unbounded Sequences

The following OMG IDL unbounded sequence of type `T`:

```
// OMG IDL
typedef ... T;
typedef sequence<T> UNBOUNDED_SEQUENCE;
```

translates to:

```
// MIDL
typedef ... U;
typedef struct
{
   unsigned long cbMaxSize;
   unsigned long cbLengthUsed;
   [size_is(cbMaxSize), length_is(cbLengthUsed), unique] U *
                                                    pValue;
} UNBOUNDED_SEQUENCE;
```

In the preceding example, the encoding for the unbounded OMG IDL sequence of type `T` is that of a MIDL `struct` containing a unique pointer to a conformant array of type `U`, where `U` is the MIDL mapping of `T`. The enclosing `struct` in the MIDL mapping is necessary to provide a scope in which extent and data bounds can be defined.

Bounded Sequences

The following OMG IDL bounded sequence of type `T` (which can grow to be `N` size):

```
// OMG IDL
const long N = ...;
typedef ... T;
typedef sequence<T,N> BOUNDED_SEQUENCE_OF_N;
```

translates to:

```
// MIDL
const long N = ...;
typedef ... U;
typedef struct
{
   unsigned long reserved;
   unsigned long cbLengthUsed;
   [length_is(cbLengthUsed)] U Value N;
} BOUNDED_SEQUENCE_OF_N;
```

The maximum size of the bounded sequence is declared in the declaration of the array. A `[size_is()]` attribute is therefore not needed.

Translation of Arrays

The following OMG IDL array of some type `T`:

```
// OMG IDL
const long N = ...;
typedef ... T;
typedef T ARRAY_OF_T[N];
```

translates as follows to a MIDL array of type `U`:

```
// MIDL
const long N = ...;
typedef ... U;
typedef U ARRAY_OF_U[N];
```

In this example, the MIDL array of type `U` is the result of mapping the OMG IDL `T` into MIDL.

Note: If the ellipsis were `octet` in the OMG IDL sample, the ellipsis would have to be `byte` in MIDL. That is why the types of the array elements have different names in the two texts.

Translation of Exceptions

The CORBA model uses exceptions to report error information. Exceptions are classified into two categories:

1. System exceptions can be raised by any operation. A standard set of system exceptions is defined by CORBA, and Orbix provides a number of additional system exceptions. These system exceptions are listed in "System Exceptions" on page 329.

2. User exceptions are defined in OMG IDL. An OMG IDL operation can optionally specify that it might raise a specific set of user exceptions. An OMG IDL operation might also raise a system exception but this is not defined at the OMG IDL level.

User Exceptions

An OMG IDL user-defined exception translates to a MIDL interface and an exception structure that appears as the last parameter of any operation mapped from OMG IDL to MIDL.

For example, if an operation in `MyModule::MyInterface` raises a user exception, an exception structure named `MyModule_MyInterfaceExceptions` will be created. This is then mapped as an output parameter to MIDL.

The following OMG IDL code shows the definition of the format used to represent user exceptions:

```
// OMG IDL
module BANK
{
...
   exception InsufficientFunds {float balance};
   exception InvalidAmount {float amount};

   interface Account
   {
      exception NotAuthorised{};
      float Deposit(in float Amount) raises(InvalidAmount);
      float Withdraw(in float Amount) raises(InvalidAmount,
                                             NotAuthorised);
   };
};
```

This translates to:

```
// MIDL
struct BANK_InsufficientFunds
{
   float balance;
};
struct BANK_InvalidAmount
{
   float amount;
};
struct BANK_Account_NotAuthorised
{
};

interface IBANK_AccountUserExceptions: IUnknown
{
   HRESULT get_InsufficientFunds([out] BANK_InsufficientFunds *
                                 exceptionBody);
   HRESULT get_InvalidAmount([out] BANK_InvalidAmount *
                                       exceptionBody);
   HRESULT get_NotAuthorised([out] BANK_Account_NotAuthorised *
                                       exceptionBody);
};
```

```
typedef struct
{
   ExceptionType type;
   LPSTR repositoryId;
   IBANK_AccountUserExceptions * piUserException;
} BANK_AccountExceptions
```

System Exceptions

A CORBA system exception translates to a COM interface defined as follows:

```
// MIDL
SetErrorInfo(OL,NULL); //Initialise the thread-local error object
try
{
   // Call the CORBA operation
}
catch(...)
{
   ...
   CreateErrorInfo(&pICreateErrorInfo);
   pICreateErrorInfo->SetSource(...);
   pICreateErrorInfo->SetDescription(...);
   pICreateErrorInfo->SetGUID(...);
   pICreateErrorInfo->QueryInterface(IID_IErrorInfo,&pIErrorInfo);
   pICreateErrorInfo->SetErrorInfo(OL,pIErrorInfo);
   pIErrorInfo->Release();
   pICreateErrorInfo->Release();
   ...
}
```

A client to a COM view would access the error object as follows:

```
// After obtaining a pointer to an interface on the COM View, the
// client does the following one time
pIMyMappedInterface->QueryInterface(IID_ISupportErrorInfo,
                                    &pISupportErrorInfo);
hr = pISupportErrorInfo->InterfaceSupportsErrorInfo
                                    (IID_MyMappedInterface);
BOOL bSupportsErrorInfo = (hr == NOERROR ? TRUE : FALSE);
...
// Call to the COM operation...
HRESULT hrOperation = pIMyMappedInterface->...
```

```
if (bSupportsErrorInfo)
{
   HRESULT hr = GetErrorInfo(O,&pIErrorInfo);
   // S_FALSE means that error data is not available
   // NO ERROR means it is available
   if (hr == NO_ERROR)
   {
      pIErrorInfo->GetSource(...);
      // Has repository id and minor code
      // hrOperation has the completion status encoded into it
      pIErrorInfo->GetDescription(...);
   }
}
```

Translation of the Any Type

There is no direct translation of the OMG IDL any type to COM. Therefore, it
is mapped to the following interface definition:

```
// MIDL
typedef [v1_enum, public]
enum CORBAAnyDataTagEnum{
   anySimpleValTag=0,
      anyAnyValTag,
      anySeqValTag,
      anyStructValTag,
      anyUnionValTag
} CORBAAnyDataTag;

typedef union CORBAAnyDataUnion
   switch(CORBAAnyDataTag whichOne){
      case anyAnyValTag:ICORBA_Any *anyVal;
      case anySeqValTag:
      case anyStructValTag:
         struct {
            [string, unique] char * repositoryId;
            unsigned long cbMaxSize;
            unsigned long cbLength-Used;
            [size_is(cbMaxSize), length_is(cbLengthUsed),unique]
               union CORBAAnyDatUnion *pVal;
         multiVal;
      case anyUnionValTag;
```

```
          struct{
             [string, unique] char * repositoryId;
             long disc;
             union CORBAAnyDataUnion *value;
          unionVal;
      case anyObjectValTag:
          struct{
             [string, unique] char * repositoryId;
             VARIANT val;
          objectVal;
      case anySimpleValTag: //All other types
          VARIANT simpleVal;
      } CORBAAnyData;
...uuid[...]
interface ICORBA_Any: IUnknown
{
   HRESULT _get_value([out] VARIANT * val);
   HRESULT _put_value([in] VARIANT val);
   HRESULT _get_CORBAAnyData([out] CORBAAnyData * val);
   HRESULT _put_CORBAAnyData([in] CORBAAnyData val);
   HRESULT _get_typeCode([out] ICORBA_TypeCode ** tc);
}
```

Context Clause

The CORBA standard OMG IDL to COM mapping does not specify a translation for OMG IDL contexts.

Translation of Object References

When an OMG IDL operation returns an object reference or it passes an object reference as an operation parameter, this is translated to a reference to an IUnknown based interface in MIDL. For example:

```
// OMG IDL
interface Account {
   ...
};
```

```
interface Bank {
    Account newAccount(in string name);
    deleteAccount(in Account a);
};
```

translates to:

```
// MIDL
[object, uuid(...)]
interface IBank : IUnknown {
    HRESULT newAccount ([in] LPSTR it_name,
                        [out] IAccount ** value);
    HRESULT deleteAccount ([in] IAccount * account);
};
```

C++ COM

```
// Get a pointer to the Bank interface (pIF) using the GetObject
// method of ICORBAFactory
HRESULT hr = NOERROR;
LPSTR szName = "John Smith";
float balance = 0, deposit = 10.0;
IAccount *pAcc = 0;
hr = pIF->newAccount(szName, &pAcc, NULL);
hr = pAcc->makeLodgement(deposit);
hr = pAcc->_get_balance(&balance);
cout << "balance is" << balance << endl;
hr = pIF->deleteAccount(pAcc);
pAcc->Release();
```

Translation of Modules

An OMG IDL definition contained within the scope of an OMG IDL module is translated to its corresponding MIDL definition by prefixing the name of the MIDL type definition with the name of the module. For example:

```
// OMG IDL
module Finance {
    interface Bank {
    ...
    };
};
```

translates to:

```
[object, uuid(...), helpstring("Finance_Bank")]
interface IFinance_Bank : IUnknown {
   ...
}
```

Translation of Constants

OMG IDL const types translate to MIDL const types. For example:

```
// OMG IDL
const short S = ...;
const long L = ...;
const unsigned short US = ...;
const unsigned long UL = ...;
const float F = ...;
const double D = ...;
const char C = ...;
const boolean B = ...;
const string STR = "...";
```

translates to:

```
// MIDL
const short S = ...;
const long L = ...;
const unsigned short US = ...;
const unsigned long UL = ...;
const float F = ...;
const double D = ...;
const char C = ...;
const boolean B = ...;
const LPSTR STR = "...";
```

Translation of Enumerated Types

A CORBA enum translates directly to a COM enum. For example:

```
// OMG IDL
interface MyIntf
{
   enum A_or_B_or_C {A,B,C};
};
```

translates to:

```
// MIDL
[uuid(...), ...]
interface IMyIntf
{
   typedef [v1_enum, public]
   enum MyIntf_A_or_B_or_C {MyIntf_A = 0, MyIntf_B, MyIntf_C}
      MyIntf_A_or_B_or_C;
};
```

CORBA has enums that are not explicitly tagged with values. On the other hand, MIDL supports enums that are explicitly tagged with values. Therefore, any language mapping that permits two enums to be compared, or which defines successor or predecessor functions on enums, must conform to the ordering of the enums as specified in OMG IDL.

The MIDL keyword v1_enum is required in order for an enum to be transmitted as 32-bit values. Microsoft recommends that this keyword is used on 32-bit platforms, because it increases the efficiency of marshalling and unmarshalling data when such an enum is embedded in a structure or union.

CORBA supports enums with up to 2^{32} identifiers but MIDL only supports 2^{16} identifiers. Truncation might therefore result.

Translation of Scoped Names

An OMG IDL scoped name must be fully qualified in MIDL to prevent accidental name collisions. For example:

```
// OMG IDL
module Bank {
   interface ATM {
      enum type {CHECKS,CASH};
      struct DepositRecord {
         string account;
         float amount;
         type kind;
      };
      void deposit(in DepositRecord val);
   };
};
```

translates to:

```
MIDL
[uuid(...), object]
interface IBANK_ATM: IUnknown {
   typedef [v1 enum] enum BANK_ATM_type
      {BANK_ATM_CHECKS, BANK_ATM_CASH} BANK_ATM_type;
   typedef struct
   {
      LPSTR account;
      float amount;
      BANK_ATM_type kind;
   }
   BANK_ATM_DepositRecord;
   HRESULT deposit(in BANK_ATM_DepositRecord * val);
};
```

Translation of Typedefs

A CORBA `typedef` translates directly to a MIDL `typedef`.

A `typedef` definition is most often used for array and sequence definitions. For example:

```
// OMG IDL
interface Account {...};

typedef sequence<Account, 100> AccountList;
```

translates to:

```
[object, UUID(...)]
interface IAccount : IUnknown {...};
Typedef struct {
...
} AccountList;
```

8

Mapping COM Objects to CORBA

COM types are defined in Microsoft IDL (MIDL). CORBA types are defined in OMG IDL. To allow interworking between COM and CORBA, MIDL types must be translated to OMG IDL. This chapter outlines how translation of COM objects to CORBA is achieved.

For the purposes of illustration, this chapter describes a textual mapping between MIDL and OMG IDL. OrbixCOMet itself does not require this textual mapping to take place, because it includes a dynamic marshalling engine. The textual mappings shown in this chapter are actually performed by OrbixCOMet at application runtime.

Translation of Basic Types

COM basic types translate to corresponding types in CORBA. Table 8.1 shows the mapping for each type.

MIDL	Description	OMG IDL	Description

boolean	Unsigned char, 8-bit
	`0 = FALSE`
	`1 = TRUE`
		boolean	Unsigned char, 8-bit
			`0 = FALSE`
			`1 = TRUE`

byte		octet	8-bit quantity
char	8-bit quantity	char	8-bit quantity
double	IEEE 64-bit float	double	IEEE 64-bit float
float	IEEE 32-bit float	float	IEEE 32-bit float
long	32-bit integer	long	32-bit integer
short	16-bit integer	short	16-bit integer
unsigned long	32-bit integer	unsigned long	32-bit integer
unsigned short	16-bit integer	unsigned short	16-bit integer

Table 8.1: *Translation of MIDL Basic Types to OMG IDL*

Translation of Strings

MIDL to OMG IDL `string` mappings are shown in Table 8.2.

MIDL	OMG IDL	Description
`LPSTR [string,unique]char*`	`string`	Null-terminated 8-bit character string.
`BSTR`	`string`	Null-terminated 16-bit character sting.
`LPWSTR [string,unique]wchar t*`	`string`	Null-terminated unicode string.

Table 8.2: *MIDL to OMG IDL String Mappings*

An error will occur if a COM server returns a `BSTR` that contains embedded nulls to a CORBA client.

Unbounded Strings

The following MIDL statement for an unbounded string:

```
// MIDL
typedef [string,unique] char * UNBOUNDED_STRING;
```

translates to:

```
// OMG IDL
typedef string UNBOUNDED_STRING;
```

Bounded Strings

The following MIDL statement for a bounded string:

```
// MIDL
const long N = ...;
typedef [string,unique] char (* BOUNDED_STRING) [N];
```

translates to:

```
// OMG IDL
const long N = ...;
typedef string<N> BOUNDED_STRING;
```

Unicode Unbounded Strings

The following MIDL statement for a unicode unbounded string:

```
// MIDL
typedef [string,unique] LPWSTR UNBOUNDED_UNICODE_STRING;
```

translates to:

```
// OMG IDL
typedef string UNBOUNDED_UNICODE_STRING;
```

Unicode Bounded String

The following MIDL statement for a unicode bounded string:

```
// MIDL
const long N = ...;
typedef [string,unique] wchar t*(BOUNDED_UNICODE_STRING) [N];
```

translates to:

```
// OMG IDL
const long N = ...;
typedef string<N> BOUNDED_UNICODE_STRING;
```

Translation of Interfaces

Translation of Properties

In general, properties are expressed in a COM interface as two separate operations: one to `get` the value and the other to `set` the value. For this reason, these operations are mapped to operations in the CORBA interface.

Translation of Operations

A MIDL operation translates to an OMG IDL operation. For example:

```
// MIDL
interface IFoo: IUnknown
{
   HRESULT stringify([in] VARIANT value, [out,retval] LPSTR *
                                              pszValue);
   HRESULT permute([inout] short * value);
   HRESULT tryPermute([inout] short * value, [out] long *
                                              newValue);
};
```

translates to:

```
// OMG IDL
typedef long HRESULT;
interface IFoo
{
   string stringify(in any value) raises (COM_ERROR, COM_ERROREX);
   void permute(inout short value);
   void tryPermute(inout short value, out long newValue);
};
```

Parameters

A MIDL [in] parameter translates to an OMG IDL in parameter.

A MIDL [out] parameter translates to an OMG IDL out parameter.

A MIDL [inout] parameter translates to an OMG IDL in,out parameter.

A MIDL [retval,out] parameter translates to an OMG IDL return value.

Return Types

All COM interfaces must have a HRESULT return type that is used in COM for exception reporting. Because CORBA has a richer exception hierarchy, the HRESULT types are not included in the mapping. Instead, they are mapped to equivalent CORBA system exceptions. If an operation in the COM interface is marked with an [out,reval] parameter, this parameter will appear as the return value in the CORBA operation.

Translation of Inheritance

CORBA and COM have different models for inheritance. CORBA interfaces can be multiply inherited but COM does not support multiple interface inheritance.

`CORBA::Composite` is a general purpose interface used to provide a standard mechanism for accessing multiple interfaces from a client, eventhough those interfaces are not related by inheritance. It is defined as follows in CORBA:

```
// PIDL
{
   interface Composite
   {
      Object query_interface(in RepositoryId whichOne);
   };
   interface Composable: Composite
   {
      Composite primary_interface();
   };
};
```

The root of a COM interface inheritance tree, when mapped to CORBA, is multiply inherited from `CORBA::Composable` and `CosLifeCycle::LifeCycleObject`. Any COM method parameters that require `IUnknown` interfaces as arguments are mapped in OMG IDL to object references of type `CORBA::Object`.

The following MIDL definiton:

```
// MIDL
interface IFoo: IUnknown
   {
   HRESULT inquire([in] IUnknown *obj);
   };
```

translates to:

```
// OMG IDL
interface IFoo: CORBA::Composable,CosLifeCycle::LifeCycleObject
   {
      void inquire(in Object obj);
   };
```

Translation of Complex Types

Translation of Constructed Types

COM constructed types such as `struct`, `union`, and `array` map to the corresponding CORBA constructed types.

Translation of Structs

A MIDL `struct` translates to a corresponding `struct` in OMG IDL. For example:

```
// MIDL
struct foo {
   long l;
   LPTSTR s;
};
```

translates to:

```
// OMG IDL
struct foo {
   long l;
   string s;
};
```

Translation of Unions

Encapsulated Unions

The following example of a MIDL encapsulated `union`:

```
// MIDL
typedef enum
   {
   dchar,
   dShort,
   dLong,
   dFloat,
   dDouble} UNION_DISCRIMINATOR;
```

```
typedef union switch (UNION_DISCRIMINATOR _d)
   {
   case dChar: char c;
   case dShort: short s;
   case dLong: long l;
   case dFloat: float f;
   case dDouble: double d;
   default: byte v[8];
   } UNION_OF_CHAR_AND_ARITHMETIC;
```

translates to:

```
// OMG IDL
enum UNION_DISCRIMINATOR
   {
   dChar,
   dShort,
   dLong,
   dFloat,
   dDouble
   };
union UNION_OF_CHAR_AND_ARITHMETIC
   switch(UNION_DISCRIMINATOR)
   {
   case dChar: char c;
   case dShort: short s;
   case dLong: long l;
   case dFloat: float f;
   case dDouble: double d;
   default: octet v[8];
   };
```

Non-Encapsulated Unions

MIDL non-encapsulated unions (and MIDL encapsulated unions with non-constant discriminators) translate to an any type in OMG IDL. For example:

```
// MIDL
typedef [switch_type(short)] union
tagUNION_OF_CHAR_AND_ARITHMETIC
   {
   [case(0)] char c;
   [case(1)] short s;
   [case(2)] long l;
```

```
    [case(3)] float f;
    [case(4)] double d;
    [default] byte v[8];
    } UNION_OF_CHAR_AND_ARITHMETIC;
```

translates to:

```
// OMG IDL
typedef any UNION_OF_CHAR_AND_ARITHMETIC;
```

Note: The type of the OMG IDL `any` is determined at runtime during conversion of the MIDL `union`.

Translation of Pointers

A MIDL reference pointer translates to a CORBA sequence containing one element.

A MIDL unique pointer (with no aliases or cycles) translates to a CORBA sequence containing zero or one elements.

A MIDL full pointer (with no aliases or cycles) translates to a CORBA sequence containing zero or one elements.

A run-time error will occur in the following situations:

- If a COM client passes a full pointer containing aliases or cycles to a CORBA server.

- If a COM server attempts to return a full pointer containing aliases or cycles to a CORBA client.

Translation of Arrays

Fixed Arrays

A MIDL fixed-length array translates to an OMG IDL fixed-length array. For example:

```
// MIDL
const long N = ...;
typedef ... U;
typedef U ARRAY_OF_N[N];
typedef float DTYPE[0..10];
```

translates to:

```
// OMG IDL
const long N = ...;
typedef ... T;
typedef T ARRAY_OF_N[N];
typedef float DTYPE[11];
```

Non-Fixed Arrays

A MIDL non-fixed-length array translates to an OMG IDL sequence. For example:

```
// MIDL
typedef short BTYPE[]; // Equivalent to [*];
typedef char CTYPE[*];
```

translates to:

```
// OMG IDL
typedef sequence<short> BTYPE;
typedef sequence<char> CTYPE;
```

Translation of Exceptions

COM exceptions translate to CORBA exceptions. COM system error codes translate to CORBA standard exceptions. COM user-defined error codes translate to CORBA user exceptions.

System Exceptions

COM system exception codes are defined with the FACILITY_NULL and FACILITY_RPC facility codes that translate to CORBA standard exceptions.

Table 8.3 list the mappings from COM FACILITY_NULL exceptions to CORBA standard exceptions.

COM	CORBA
EOUTOFMEMORY	NO_MEMORY
E_INVALIDARG	BAD_PARAM
E_NOTIMPL	NO_IMPLEMENT
E_FAIL	UNKNOWN
E_ACCESSDENIED	NO_PERMISSION
E_UNEXPECTED	UNKNOWN
E_ABORT	UNKNOWN
E_POINTER	BAD_PARAM
E_HANDLE	BAD_PARAM

Table 8.3: *Mapping from COM FACILITY_NULL to CORBA Standard Exceptions*

Table 8.4 list the mappings from COM FACILITY_RPC exceptions to CORBA standard exceptions. (All FACILITY_RPC exceptions not listed in this table are mapped to the new CORBA standard exception COM.)

COM	CORBA
RPC_E_CALL_CANCELED	TRANSIENT
RPC_E_CANTPOST_INSENDCALL	COMM_FAILURE
RPC_E_CANTCALLOUT_INEXTERNALCALL	COMM_FAILURE
RPC_E_CONNECTION_TERMINATED	NV_OBJREF

Table 8.4: *Mapping from COM FACILITY_RPC to CORBA Standard Exceptions*

COM	CORBA
RPC_E_SERVER_DIED	INV_OBJREF
RPC_E_SERVER_DIED_DNE	INV_OBJREF
RPC_E_INVALID_DATAPACKET	COMM_FAILURE
RPC_E_CANTTRANSMIT_CALL	TRANSIENT
RPC_E_CLIENT_CANTMARSHAL_DATA	MARSHAL
RPC_E_CLIENT_CANUNMARSHAL_DATA	MARSHAL
RPC_E_SERVER_CANTMARSHAL_DATA	MARSHAL
RPC_E_SERVER_CANTUNMARSHAL_DATA	MARSHAL
RPC_E_INVALID_DATA	COMM_FAILURE
RPC_E_INVALID_PARAMETER	BAD_PARAM
RPC_E_CANTCALLOUT_AGAIN	COMM_FAILURE
RPC_E_SYS_CALL_FAILED	NO_RESOURCES
RPC_E_OUT_OF_RESOURCES	NO_RESOURCES
RPC_E_NOT_REGISTERED	NO_IMPLEMENT
RPC_E_DISCONNECTED	INV_OBJREF
RPC_E_RETRY	TRANSIENT
RPC_E_SERVERCALL_REJECTED	TRANSIENT
RPC_E_NOT_REGISTERED	NO_IMPLEMENT

Table 8.4: *Mapping from COM FACILITY_RPC to CORBA Standard Exceptions*

User Exceptions

COM user-defined error codes require the use of the `raises` clause in OMG
IDL. The following OMG IDL statement represents such a user exception:

```
// OMG IDL
exception COM_ERROR {long hresult;};
```

Translation of Variants

A COM `VARIANT` translates to a CORBA `any` type.

The allowable `VARIANT` data types are currently limited to the data types supported by Automation. Refer to the documentation for your COM client language for details of the types supported in a `VARIANT`.

An error will occur at runtime if a CORBA client returns an inconvertible `any` type to a COM server.

Translation of Constants

A MIDL `const` type translates to a corresponding OMG IDL `const` type. For example:

```
// MIDL
const short S = ...;
const long L = ...;
const unsigned short US = ...;
const unsigned long UL = ...;
const float F = ...;
const double D = ...;
const char C = ...;
const boolean B = ...;
const string STR = "...";
```

translates to:

```
// OMG IDL
const short S = ...;
const long L = ...;
const unsigned short US = ...;
const unsigned long LS = ...;
const float F = ...;
const double D = ...;
const char C = ...;
const boolean B = ...;
const string STR = "...";
```

Translation of Enumerated Types

A MIDL `enum` translates to an OMG IDL `enum`. For example, the MIDL definition:

```
// MIDL
typedef [v1_enum] enum tagA_or_B_or_C {A=2, B=3, C=1}
    A_or_B_or_C;
```

translates to:

```
// OMG IDL
enum A_or_B_or_C {C,A,B};
```

OMG IDL does not support enums defined with explicit tagged values. The CORBA view of a COM object, therefore, is responsible for maintaining the correct tagged value of the mapped enums as they cross the view.

Translation of Scoped Names

MIDL considers all definitions to be at global scope. Therefore, to avoid collisions across interfaces when translating from MIDL to OMG IDL, nested data types are treated as if they have been prefixed with the name of the scoping level. For example:

```
interface IA: IUnknown
{
    typedef enum {ONE, TWO, THREE} Count;
    HRESULT f([in] Count val);
}
```

is mapped as if it were defined as follows:

```
typdef enum {A_ONE, A_TWO, A_THREE} A_Count;
interface IA: IUnknown
{
    HRESULT f([in] A Count val);
}
```

Translation of Typedefs

A MIDL `typedef` is translated to an OMG IDL `typedef`. For example:

```
// MIDL
interface IAccount : IUnknown {...};
Typedef struct {
...
} AccountList;
```

translates to:

```
// OMG IDL
interface Account {...};

typedef sequence<Account, 100> AccountList;
```

9

Development Support Tools

OrbixCOMet is a high-performance bridge that stores OMG IDL and MIDL type information at the bridging location in an ORB-neutral binary format. The OrbixCOMet type store holds a cache of this information that is used by the dynamic bridge during runtime of your OrbixCOMet applications. This chapter describes the GUI and command line tools that allow you to maintain the type store cache and to create OMG IDL, MIDL, type libraries, smart proxy DLLs and server stub code. It also describes the GUI and command line tools that you can use to replace an existing DCOM server with a CORBA server.

Note: The GUI and command line tools for maintaining the type store cache provide the same functionality. You can choose to use just one or the other, or you can use both if you wish. However, if you are using both the GUI tool and the command line utilities simultaneously, changes you make to the type store cache with the command line tools will not appear automatically on the GUI interface. In this case, you would have to refresh the GUI interface. This is further explained later in this chapter.

Type Store GUI Tools

The OrbixCOMet Tools Screen

From your OrbixCOMet start menu, select **COMet tools** to open the **OrbixCOMet tools** screen shown in Figure 9.1.
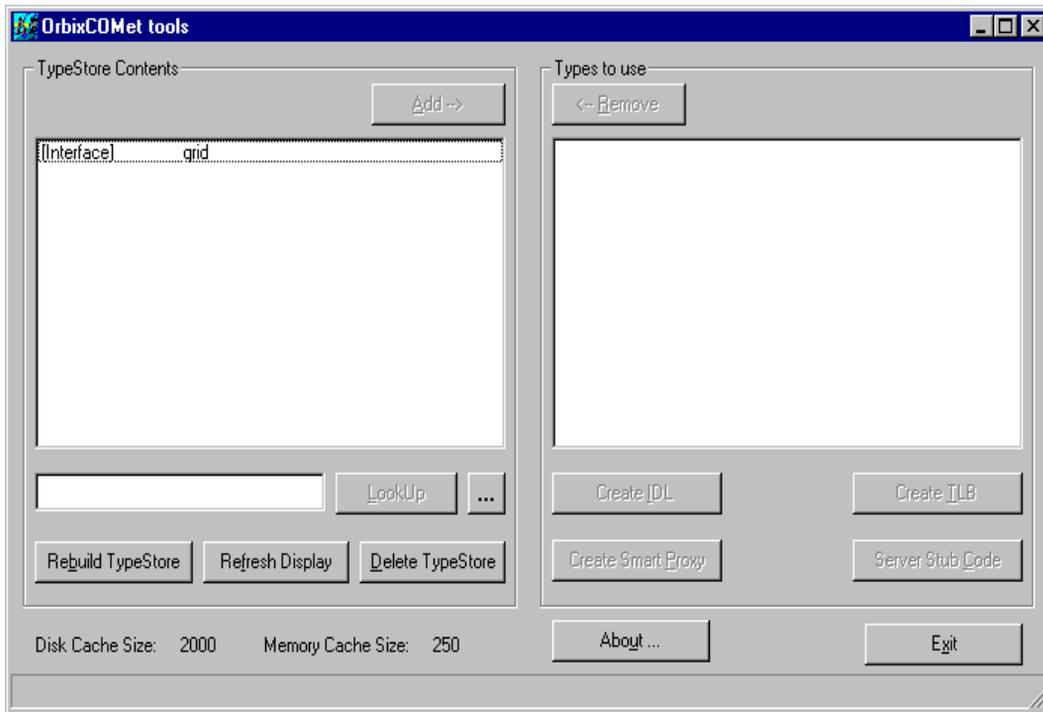
Figure 9.1: *OrbixCOMet Tools Screen*

On the **OrbixCOMet tools** screen, the **TypeStore Contents** panel lists all the type information that is currently held in the type store cache. All type information is held in the cache in one ORB-neutral binary format called metadata, regardless of whether it has originated from OMG IDL or MIDL files. It can consist of module names, interface names or data types.

From this screen, you can perform the following tasks:

- Add new information to the type store.
- Delete the type store contents.
- Rebuild the type store.
- Create an OMG IDL or MIDL file.
- Create a type library.
- Create a smart proxy DLL.
- Create server stub code.

Adding New Information to the Type Store

To add new information to the type store:

1. Enable the **LookUp** button in either of the following ways:
   - In the field beside the **LookUp** button, enter the name of the interface you want to find.
   - Select the browse button that is marked by an ellipsis (that is, …). This provides you with a list of type library names. Select a type library name to return it to the field.

2. Select the **LookUp** button. This searches the Interface Repository and the type store cache for the specified name. If the relevant name is found in the Interface Repository and it is not already in the cache, it is then automatically added to the cache.

Refreshing the Display

If you are using the command line tools and GUI tool simultaneously, any changes you make to the type store cache with the command line tools do not appear automatically in the **TypeStore Contents** panel on the **OrbixCOMet tools** screen. In this case, you can select the **Refresh Display** button to reflect any changes that you made via the command line.

Deleting the Type Store Contents

To delete the entire contents of the cache, select the **Delete TypeStore** button.

Rebuilding the Type Store

To automatically refresh or rebuild the type store from a record of existing entries, select the **Rebuild TypeStore** button.

Creating an IDL File

The normal procedure for writing a CORBA or COM client is to first obtain an OMG IDL or Microsoft IDL (MIDL) definition for the interface. You should ensure that each IDL file contains a module in order to minimise manual lookups.

To create an IDL file from the **OrbixCOMet tools** screen in Figure 9.1 on page 122:

1. From the **TypeStore Contents** panel, select an item of type information you want to include in the IDL file.

2. Select the **Add** button. This adds the item to the **Types to use** panel.

   Repeat steps 1 and 2 until you have added all the items of type information that you want to include in the IDL file.

3. Select the **Create IDL** button. This opens the **OrbixCOMet ts2idl client** screen shown in Figure 9.2 on page 125.

Note: Creating MIDL here allows you to create a standard DCOM proxy/stub DLL that can be installed with your DCOM client application. This means you do not have to install any CORBA components on the client machine. The distribution model is exactly the same as it would be for a standard DCOM application. This means it includes a COM client and a proxy/stub DLL.
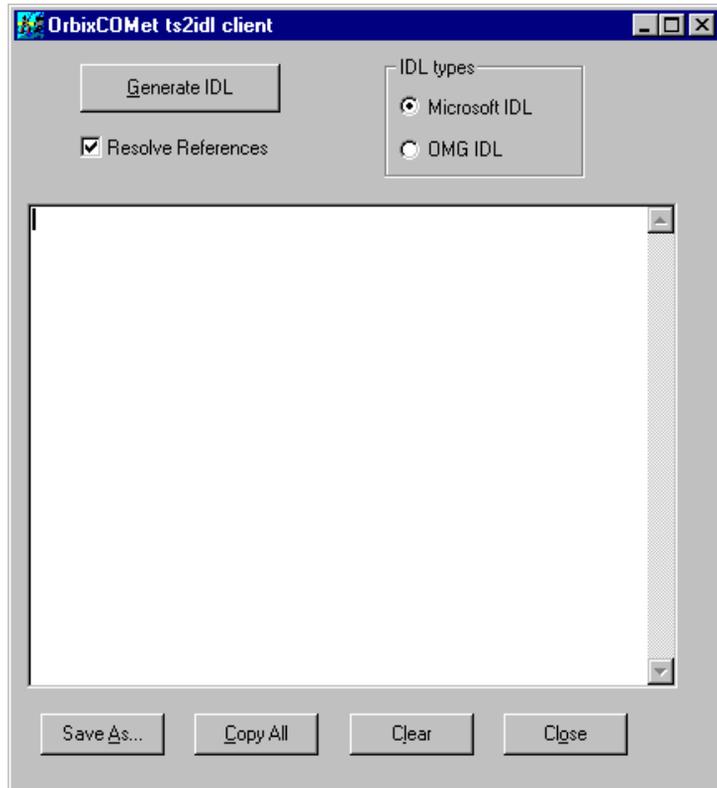
Figure 9.2: *Creating an IDL File*

4. If you want to:

  ♦ Create a Microsoft IDL file, select the **Microsoft IDL** radio button.

  ♦ Create an OMG IDL file, select the **OMG IDL** radio button.

  ♦ Ensure IDL is created for all dependent types not defined within the scope of (for example) your interface, select the **Resolve References** check box.

  ♦ Copy the contents of the IDL file to your development environment, select the **Copy All** button.

- ◆ Refresh the screen, select the **Clear** button.
- ◆ Assign an IDL filename, select the **Save As** button.
5. Select the **Generate IDL** button. This creates the IDL file.

Creating a Type Library

When using an Automation client, you have the option in some controllers (for example, Visual Basic) of using straight `IDispatch` interfaces or dual interfaces. If you are only using straight `IDispatch` interfaces, there is no need to create a type library. This is because OrbixCOMet will automatically put the related type information into the type store when it performs a lookup using `GetObject`, as in the following example:

```
' Visual Basic requesting an Automation object
' reference to OMG IDL interface mod::CorbaSrv
srvobj = factory.GetObject ("mod/CorbaSrv")
```

However, if you want to use dual interfaces you must create a type library.

To create a type library from the **OrbixCOMet tools** screen in Figure 9.1 on page 122:

1. From the **TypeStore Contents** panel, select an item of type information you want to include in the type library.
2. Select the **Add** button. This adds the item to the **Types to use** panel.

   Repeat steps 1 and 2 until you have added all the items of type information that you want to include in the type library.
3. Select the **Create TLB** button. This opens the **Typelibrary Generator** screen shown in Figure 9.3 on page 127.
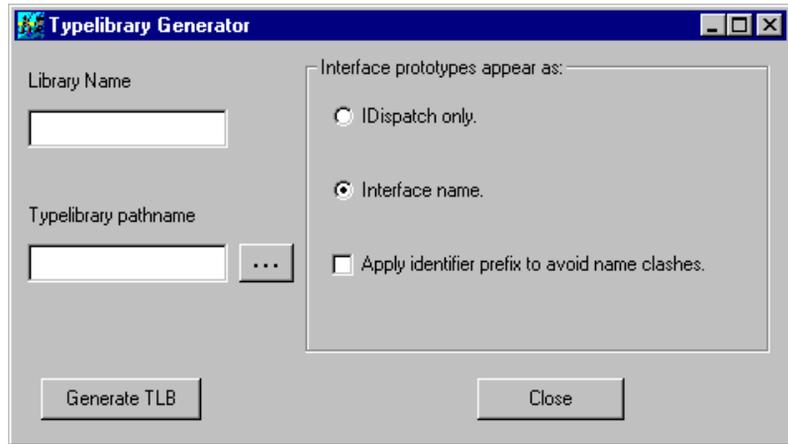
Figure 9.3: *Creating a Type Library*

4. In the **Library Name** field, enter the internal library name. This can be the same as the type library pathname if you wish, but make sure the library does not have the same name as any of the types that it contains.

5. In the **Typelibrary pathname** field, enter the full pathname for the type library.

6. If you want interface prototypes to:
   - Appear as `IDispatch`, select the **IDispatch only** radio button.
   - Use the specific interface name, select the **Interface name** radio button.

7. To apply an identifier prefix to avoid name clashes, select the corresponding check box. This helps to avoid potential name clashes between OMG IDL and MIDL keywords.

8. Select the **Generate TLB** button. This creates the type library.

Generating a Smart Proxy

Proxy objects are an Orbix-specific feature that are implemented in the stub code for the client process. A normal proxy marshals the `in` and `inout` parameters from the client request, transmits the request package to the implementation object in the server, receives the reply package back from the server, and unmarshals the `out` and `inout` parameters and return value for use by the client. In other words, it fools the client into thinking that the distributed object is local to the client process. A smart proxy goes further in that it can also act as a cache of low-level state information and attribute values from the distribution object in the server.

If the OrbixCOMet bridge is not being loaded in-process to your COM client application, you must create a standard DCOM proxy DLL for the interfaces you are using. This is necessary to allow DCOM to correctly make a connection to the remote OrbixCOMet bridge from the client.

To create a smart proxy from the **OrbixCOMet tools** screen in Figure 9.1 on page 122:

1. From the **TypeStore Contents** panel, select an item of type information you want to include in the source for the smart proxy.

2. Select the **Add** button. This adds the item to the **Types to use** panel.

   Repeat steps 1 and 2 until you have added all the items of type information that you want to include in the source for the smart proxy.

3. Select the **Create Smart Proxy** button. This opens the **Smart Proxy Generator** screen shown in Figure 9.4 on page 129.

4. Specify the name you want to assign to the handler DLL file in either of the following ways:

   ♦ In the **Handler Name** field, enter the name of the handler DLL file.

   ♦ Select the browse button that is marked by an ellipsis (that is, …). This provides you with a list of filenames. Select a name to return it to the **Handler Name** field.

5. Select the target directory where you want the output files to be created.

6. Select any additional source files that you want to include.

7. Select the **Generate** button. This creates the smart proxy in the target directory you have selected.
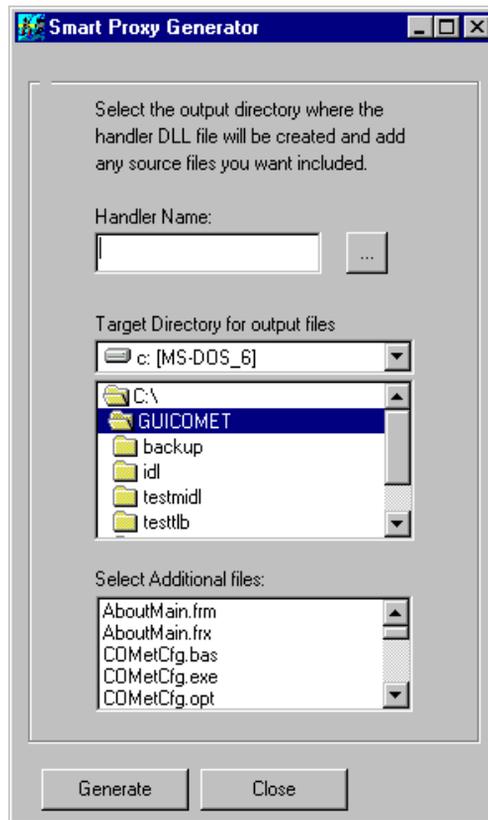
Figure 9.4: *Generating a Smart Proxy*

Generating Server Stub Code and Support for Callbacks

When you want your application to be a server application or to have callback functionality, you must provide an implementation for the server/callback objects. The `cometcfg` tool allows you to generate starting point skeleton code for these object implementations. Currently it generates skeleton code for Visual Basic 5.0 and PowerBuilder.

To create server stub code from the **OrbixCOMet tools** screen in Figure 9.1 on page 122:

1. From the **TypeStore Contents** panel, select an item of type information you want to include in the server stub code.

2. Select the **Add** button. This adds the item to the **Types to use** panel.

   Repeat these steps until you have added all the items of type information that you want to include in the server stub code.

3. Select the **Server Stub Code** button. This opens the **Server Stub Code Generator** screen shown in Figure 9.5.
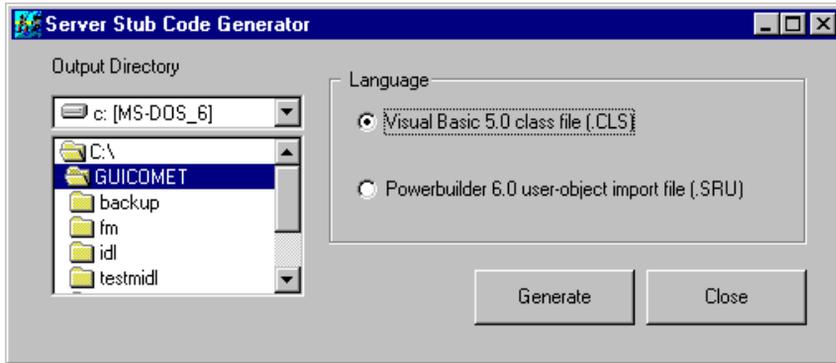
Figure 9.5: *Generating Server Stub Code*

4. To indicate the development language you are using, select the corresponding radio button in the **Language** panel.

5. Select the target directory where you want the output files to be created.

6. Select the **Generate** button. This generates the skeleton code in the target directory you have selected.

Type Store Command Line Tools

This section describes the command line utilities that you can use as an alternative to the GUI tool for maintaining the type store cache and creating OMG IDL files, MIDL files, type libraries and smart proxies.

These utilities can be found in `%ORBIXCOMET%`\bin, where `%ORBIXCOMET%` represents the name of the installation directory you have chosen.

Adding New Information to the Type Store

The following command would add the `grid` interface to the type store cache:

```
typeman -e grid
```

Refer to "Managing the Type Store" on page 211 for more details about adding information to the type store cache. (This is also called *priming* the cache.)

Deleting the Type Store Contents

Either of the following commands would delete the entire contents of the type store:

```
typeman -w
```

or

```
del c:\temp\typeman.*
```

The second command in this case is assuming the typeman data files are held in `c:\temp`. The typeman data files include the following:

`typeman._dc`	This is the disk cache data file.
`typeman.idc`	This is the disk cache index.
`typeman.edc`	This is the disk cache empty record index.
`typeman.map`	This is the UUID name mapper.

Creating an IDL File

The following command would create a `grid.idl` file for the interface `grid`:

```
ts2idl -f grid.idl grid
```

The following is an example of the usage string for `ts2idl`:

```
Usage:
ts2idl [options] <type name> [[<type name>] ...]
Options:
     -b : Pass object references as type Object in OMG IDL
     -c : Don't connect to the IFR (e.g. if cache is fully primed)
     -r : Resolve referenced types
     -m : Generate MIDL <default>
     -p : Generate makefile for proxy/stub DLL
     -f : <filename>
     -v : Print this message
     Tip : use -p to generate a makefile for the marshalling DLL!!
```

For more complicated interfaces that use user-defined types, you can use the `-r` switch to completely resolve those types and produce MIDL for them also.

You can use the `-b` switch when generating OMG IDL based on type library information stored in the type store. Its purpose is to attempt to keep the number of generated lines of OMG IDL to a minimum. It specifies that interface pointers passed as parameters to operations described in the type library are mapped as type `CORBA::Object` in the generated OMG IDL rather than as their dynamic type. Use this switch in conjunction with the `-r` switch. For an example of its use, see the Excel CORBA client in the `demos\corbaclient\excel` directory.

Creating a Type Library

The following command would create a `grid.tlb` file in library `IT_grid` for the interface `grid`:

```
       ts2tlb -f grid.tlb -l IT_grid grid
```

The following is an example of the usage string for `ts2tlb`:

```
Usage:
ts2tlb [options] <type name> [[<type name>] ...]
        -f : file name (defaults to <type name #1>.tlb)
        -l : library name (defaults to IT_Library_<type name #1>
        -p : prefix parameter names with "it_"
        -i : Pass a pointer to interface Foo as IDispatch*
             rather than DIFoo* - necessary for some controllers
        -v : Print this message
        Tip : use tlibreg.exe to register your type library!!
```

Generating a Smart Proxy

The following command would generate a smart proxy for the `grid` interface in the `test` module:

```
ts2sp test::grid
```

Replacing an Existing DCOM Server

At some stage, it might become necessary to replace an existing DCOM server with a CORBA server, without the opportunity to modify existing DCOM clients. However, such clients will not be aware of the `(D)ICORBAFactory` interface that has so far been the usual way for clients to obtain initial references to CORBA objects. The solution is to allow such clients to continue to use their normal `CoCreateInstanceEx()` or `CreateObject()` calls. This means you must retrofit the bridge to serve these clients' activation requests. In other words, you must alias the bridge to the legacy DCOM server. This ensures that when the client is subsequently run, the bridge is activated in response to the client's `CoCreateInstanceEx()`/`CreateObject()` calls.

In OrbixCOMet, this is achieved through a GUI tool called `srvAlias` that allows the user to place some entries in the registry to allow such server 'aliasing' to occur. The `srvAlias` tool is shown in Figure 9.6 on page 134. Type `srvalias` and then press **Enter** in the `%ORBIXCOMET%`\bin directory of your OrbixCOMet installation to open this screen. You must enter the `CLSID` for the server to be replaced and then supply, in the appropriate text box, the string that would be passed to `(D)ICORBAFactory::GetObject()` if the CORBA factory were being used. This string is then stored in the registry under a `COMetInfo` subkey under the server's `CLSID` entries. In addition, `ITUnknown.dll` is registered as the server executable. Nothing else is required.

The `SrvAlias` tool allows users to save the new registry entries in binary format, so that an accompanying command line tool (`aliassrv.exe`) can be used at application deployment time to restore the entries on the destination machine. For example, given a file called `replace.reg` that contains the modified registry entries, the following command would alias the specified `CLSID` to OrbixCOMet:

```
aliassrv -r replace.reg -c {F7B6A75E-90BF-11D1-8E10-0060970557AC}
```

The next time a DCOM client of the server is run, OrbixCOMet will be used instead. Refer to the `demo\corbasrv\replace` directory of your OrbixCOMet installation for an example of `srvalias.exe` and `aliassrv.exe` in action.

Figure 9.6: *Aliasing the Bridge*

10

Implementing CORBA Clients

"Getting Started on Automation" on page 11 and "Getting Started on COM" on page 23 explained how to write a simple CORBA client program in an Automation-compatible language and COM C++ respectively. This chapter provides further details about programming OrbixCOMet clients.

The topics covered in this chapter include:

- How programs communicate with the ORB to obtain services or to modify the ORB's default behaviour.

- The interfaces that CORBA and COM/Automation view objects support.

- How a client can narrow an object reference when the object referred to is a derived type of the client's reference type.

- How a CORBA client can obtain a reference to an object in a CORBA server. This chapter describes a number of ways, including the use of the Naming Service.

This chapter also shows how to implement Visual Basic, PowerBuilder and C++ COM client examples for the Bank server that is developed in "Implementing CORBA Servers" on page 173.

Interface to the ORB

An OrbixCOMet program can obtain a reference to the ORB in order to communicate with it and to modify its settings. This functionality is provided by the following interfaces:

DIORBObject / IORBObject

These interfaces contain a set of methods defined by the COM/CORBA Interworking standard.

(D)`IORBObject` includes methods to convert an Interoperable Object Reference (IOR) to a string known as a stringified IOR, and to convert a stringified IOR back into an IOR.

DIOrbixORBObject / IOrbixORBObject

These interfaces contain a set of methods that are specific to OrbixCOMet for controlling the ORB.

(D)`IOrbixORBObject` includes methods to configure Orbix dynamically, to optimise calls when the client and server are located in the same process, to help with interface matching, and to control the diagnostic level. It also includes a set of methods that allow a client to control connections to a server.

A full description of (D)`IORBObject` and (D)`IOrbixORBObject` is provided in "OrbixCOMet API" on page 241.

Obtaining a Reference to the ORB

The ORB has the ProgID `CORBA.ORB.2`. The following code shows how you can obtain and use a reference to the ORB.

Visual Basic

```
Dim theORB as CORBA_Orbix.DIOrbixORBObject
Set theORB = CreateObject("CORBA.ORB.2")
```

You can now make calls such as:

```
' Do not output any diagnostic messages:
theORB.SetDiagnostics 0  ' No diagnostics
```

PowerBuilder

```
OleObject theOrb
theOrb = CREATE OleObject
theOrb.ConnectToNewObject("CORBA.ORB.2")
```

You can now make calls such as:

```
// Do not check that target object exists when binding:
theORB.PingDuringBind(0)
```

C++ COM

```
// Connect to the ORB and get the regisration interface
IOrbixORBObject * poOrb=0;
hr = CoCreateInstance(IID_IOrbixORBObject, NULL, ctx,
                    IID_IOrbixORBObject, (void**)&poOrb);
CheckHRESULT("Connecting to ORB", hr, FALSE);

hr = poOrb->PingDuringBind(false);
CheckHRESULT("getting Server API", hr, FALSE);
```

Finding Object References

Normally, a client's first task is to locate an object reference in a server. The following are some of the ways in which a client can obtain an object reference:

- The (D)ICORBAFactory interface.
- The Naming Service.
- IDL operations.

The following subsections discuss each of these in turn.

The (D)ICORBAFactory Interface

The COM/CORBA Interworking specification defines the interfaces `DICORBAFactory` and `ICORBAFactory` that provide the methods `GetObject()` and `CreateObject()` to allow a client to obtain references to CORBA objects.

GetObject()

```
// MIDL
interface DICORBAFactory : IDispatch {
   ...
   HRESULT GetObject([in] BSTR objectName,
      [optional,in,out]  VARIANT* IT_Ex,
      [retval,out] IDispatch** IT_retval);
}
```

As explained in "Getting Started on Automation" on page 11 and "Getting Started on COM" on page 23, `GetObject()` performs the following functions:

1. It creates a COM/Automation view in the bridge. This means it creates an object that presents a COM/Automation view of the target CORBA object to the client.
2. It binds the view to the CORBA implementation object in the server.
3. It returns a reference to the view to the caller.

Parameter to GetObject()

The parameter to `GetObject()` is a string that identifies the target object by specifying its Orbix object name or its IOR.

Specifying the Orbix Object Name

The parameter has two forms:

1   `"Interface[[[:Marker]:ServerName]:HostMachine]"`
2   `"Interface:<TAG>:<Tag_data>"`

The components of the string are interpreted as follows:

   `Interface`   This is the IDL interface that the target object should support.

Marker This is the name of the target Orbix object. Every Orbix object has a name that is either chosen by Orbix or set (usually) at the time the object is created. See `SetObjectImpl()` and `DIOrbixObject::Marker()` for details.

The `Marker` component is optional. If specified, `GetObject()` will fail if an object having that marker cannot be found. If not specified, OrbixCOMet chooses any object that supports the desired interface and matches the remaining components.

ServerName This is the name of the Orbix server in which the object is implemented. This is the name of the server that is registered with the Implementation Repository.

This component is optional. If not specified, the server name defaults to the name specified in `Interface`.

HostMachine This is the Internet host name or Internet address of the host on which the server is located. If the string is in the format `xxx.xxx.xxx.xxx`, where `x` is a decimal digit, it is interpreted as an Internet address.

This component is optional. If not specified, OrbixCOMet will use its default locator to search the network for a host supporting the desired server. However, this requires that locator configuration information is available. If no configuration information is available, the search will be confined to the local host. Refer to the Orbix documentation set for an explanation of how to configure the default locator.

<TAG>   Two types of tags are allowed. Each type has a different form of <Tag_data>. Valid tag types are:

- `IOR`—the tag data is the hexadecimal string for the stringified IOR. For example:
  `fact.GetObject("employee:IOR:123456789......")`

- `NAME_SERVICE`—the tag data is the `NAME_SERVICE` compound name separated by ".". For example:
  `fact.GetObject("employee:NAME_SERVICE:`
  `IONA.employees.PD.Ronan")`

Parameter String Examples

The following are some examples of how to use the string parameter for the Orbix object name:

`"Bank"`

> Obtain a reference to any object on any host in the network. The object should implement the OMG IDL interface `Bank` and be located in a server named `Bank`.

`"Bank:alpha"`

> Obtain a reference to any object on host alpha (in the current domain). The object should implement the OMG IDL interface `Bank` and be located in a server named `Bank`.

`"Bank:HighStreet:bankSrv:alpha"`

> Obtain a reference to an object named `HighStreet` in the server `bankSrv` on host `alpha` (in the current domain). The object must implement the OMG IDL interface `Bank`.

`"Bank:HighStreet:Credit:"`

> Locate an object named `HighStreet` in the server `Credit` somewhere in the network. The object must implement the OMG IDL interface `Bank`.

"Bank:alpha.mc.com"

> Obtain a reference to any object on host `alpha` in the domain `mc.com`. The object should implement the OMG IDL interface `Bank` and be located in a server named `Bank`.

"Bank:123.456.789.012"

> Obtain a reference to any object at the `Bank` server at the host whose Internet address is `123.456.789.012`. The object must implement the OMG IDL interface `Bank`.

CreateObject()

```
// MIDL
interface DICORBAFactory : IDispatch {
   HRESULT CreateObject([in] BSTR factoryName,
      [optional,in,out]  VARIANT* IT_Ex,
      [retval,out] IDispatch** IT_retval);
...
}
```

In OrbixCOMet, `DICORBAFactory::CreateObject()` behaves in the same way as `DICORBAFactory::GetObject()`. Therefore, it can be used exactly as described for `GetObject()`.

The Naming Service

A CORBA server can assign a name to an object and register the name and the object with the Naming Service. (The Naming Service is one of the CORBA services defined by the OMG.) A client that knows the object name can resolve it in the Naming Service to obtain a reference to the object. You need an implementation of the Naming Service, such as OrbixNames, to use this method. Refer to the *OrbixNames Programmer's and Administrator's Guide* for details of the Naming Service terminology used here and for full details of how to use OrbixNames.

In this case, a simple example of using the Naming Service from OrbixCOMet is provided.

An object registered with the Naming Service has a name that is defined in OMG IDL as follows:

```
// OMG IDL
module CosNaming {
...
   typedef string Istring;
   struct NameComponent {
      Istring id;
      Istring kind;
   };
   typedef sequence<NameComponent> Name;
...
}
```

To locate an object using the Naming Service, your client must create a `CosNaming::Name` that names the desired object. The client must then resolve the name with the Naming Service.

Creating a CosNaming::Name

In the following code extracts, assume that the client wants to bind to a `Bank` object that is registered with the name `Commercial.Trust`. The Naming Service's bridge in the example is called `NamingBridge`.

Note: The following code creates an IDL sequence of `NameComponent`s to construct a `CosNaming::Name`. Refer to "Mapping CORBA Objects to Automation" on page 41 and "Mapping CORBA Objects to COM" on page 81 for more details of how to create an OMG IDL sequence in an Automation or COM application.

Visual Basic   Create an empty sequence of `CosNaming::NameComponent`s as follows:

```
Dim ObjFactory As DICORBA_Orbix.DICORBAFactory
Set ObjFactory = CreateObject("CORBA.Factory")

Dim bankName As DICosNaming_Name
Set bankName = ObjFactory.CreateType(Nothing,"CosNaming/Name")
```

Create a NameComponent as follows:

```
Dim bankNameComp As DICosNaming_NameComponent
Set bankNameComp = ObjFactory.CreateType(Nothing,
                                     "CosNaming/NameComponent")
```

Populate the NameComponent and insert it in the Name sequence as follows:

```
bankNameComp.id = "Commercial"
bankNameComp.kind = ""
bankName.setItem 0, bankNameComp

bankNameComp.id = "Trust"
bankNameComp.kind = ""
bankName.setItem 1, bankNameComp
```

PowerBuilder Create an empty sequence as follows:

```
bankName = CREATE OleObject
bankName = ObjFactory.CreateType(Nothing,"CosNaming/Name")
```

(Refer to the section "Translation of Constructed Types" on page 53 for a description of using CreateType().)

Create a NameComponent as follows:

```
bankNameComp = CREATE OleObject
bankNameComp = ObjFactory.CreateType(Nothing,
                                     "CosNaming NameComponent")
```

Populate the NameComponents and insert them in the Name sequence as follows:

```
bankName.Count = 2

bankNameComp.id = "Commercial"
bankNameComp.kind = ""
bankName.setitem(0, bankNameComp)

bankNameComp2.id = "Trust"
bankNameComp2.kind = ""
bankName.setitem(1, bankNameComp)
```

COM C++ Create an empty sequence of CosNaming::NameComponents as follows:

```
CosNaming_Name bankName;
CosNaming_NameComponent BankNameComp;

bankName.cbMaxSize = 2;
```

```
bankName.cbLengthUsed = 2;
bankName.pValue = new CosNaming_NameComponent
                    [bankName.cbLengthUsed];
BankNameComp.id="Commercial";
BankNameComp.kind="";
bankName.pValue[0]=BankNameComp;

BankNameComp.id="Trust";
BankNameComp.kind="";

bankName.pValue[1]=BankNameComp;
```

Resolving the Name

The client obtains a reference to the target object by resolving the Name in the Naming Service. The following code extracts illustrate how to do this:

Visual Basic
```
Dim myNS as DICosNaming_NamingContext
Dim NSObj as Object

Dim theORB as CORBA_Orbix.DIOrbixORBObject
Set theORB = CreateObject("CORBA.ORB.2")

Set myNS = ObjFactory.GetObject(
   "CosNaming NamingContext:NS:host")

Set NSObj = myNS.resolve(bankName)

Set theBank = NSObj
```

The first step is to obtain a reference to a `NamingContext`, usually the Naming Service's root context. The client then calls `resolve()` on the `NamingContext` to obtain a reference to the object. The object reference that is returned by the call to `resolve()` must be narrowed to obtain a reference to the desired interface. (Refer to "Narrowing Object References" on page 148 for more details.)

PowerBuilder
```
OleObject ObjFactory
ObjFactory = CREATE OleObject

OleObject theORB
theORB = CREATE OleObject
```

```
myNS = CREATE OleObject
myNS = ObjFactory.GetObject("CosNaming/NamingContext:NS:host")

NSObj = myNS.resolve(bankName)

theORB.ConnectToNewObject("CORBA.ORB.2")

theBank = theORB.Narrow(NSObj,"Bank")
```

COM C++   The desired interface is obtained using `QueryInterface` after you have called `Resolve()`. **For example:**

```
ICosNaming_NamingContext myNS;
IUnknown *NSObj;
Ibank *pIBasic = NULL;

hr = pCORBAFact->GetObject("CosNaming/Namingcontext:NS:host",
                          &myNS);
if(!CheckErrInfo(hr, pCORBAFact, IID_ICORBAFactory))
{
   pCORBAFact->Release();
   return;
}
pCORBAFact->Release();

NSObj=myNS->Resolve(bankName);
hr = NSObj->QueryInterface(IID_Ibank, (PPVOID)&pIBasic);
if(!CheckErrInfo(hr, NSObj, IID_Ibank))
{
   NSObj->Release();
   return;
}
NSObj->Release();
   try
   {
      pIBasic->newAccount(...)
   }
   catch(...)
   {...
```

IDL Operations

A typical client program first obtains a reference to a server object by binding to the object via `DICORBAFactory::GetObject()` or `DICORBAFactory::CreateObject()`, or by using the Naming Service. This object is known as a root object and a client might need to obtain references to more than one root object. Thereafter, the client usually obtains other object references through its interaction with the root object(s).

A client can obtain an object reference from an IDL operation's return value, from an `inout` or `out` parameter, or from an attribute value. When a client receives an object reference in one of these ways, an Automation or COM view is created in the bridge and reference to the Automation or COM view is returned to the client.

The following code extract from a client of the `Bank` server illustrates this method. A fuller version of the code is provided in "A Visual Basic Client Program" on page 150.

Visual Basic

```
Dim ObjFactory As CORBA_Orbix.DICORBAFactory
Dim bankObj As BankBridge.DIBank
Dim bankAccount As BankBridge.DIAccount

Set ObjFactory = CreateObject("CORBA.Factory")
...
Set bankObj = ObjFactory.GetObject("Bank:" & _
   marker.Text & ":" & server_name.Text & _
   ":" & host_name.Text)
...
' Get an object reference as a return value:
Set bankAccount = bankObj.getAccount(txtOwner.Text)
...
' Use the object reference:
TxtBalance.Text = bankAccount.balance
txtOwner.Text = bankAccount.owner
```

Interworking Interfaces on Objects

Orbix objects support the interface defined in their IDL file. In addition, all Orbix objects support the following interfaces:

(D)ICORBAObject   Support for these interfaces is mandated by the COM/ CORBA Interworking standard. These interfaces include important functions to convert object references to string format and to convert object reference strings to object references.

(D)IOrbixObject   OrbixCOMet provides a number of additional methods that are supported by all Orbix objects. These include functions to bind to an object in an Orbix server, to find the object's marker name, to close the underlying communications connection to the server and to determine whether the communications channel between the client and server is open.

A COM/Automation view object supports the additional interfaces DIForeignObject and IForeignObject. The purpose of these interfaces is to provide a way for the view to find the foreign object reference in a proxy. (The term *foreign* refers to the CORBA system in this case.)

Refer to "OrbixCOMet API" on page 241 for details of all interfaces supported in OrbixCOMet.

Implementing CORBA Clients in Automation

Late Binding

Late binding is where you use the `IDispatch` interface on an Automation object. It means that all invocations through the object will require the parameters to be marshalled through `IDispatch` and then to CORBA.

Early Binding

If you make a call on an early bound object, you avoid the `IDispatch` marshalling overhead. This improves performance if the bridge is loaded in-process in your client application.

The code samples shown in "Getting Started on Automation" on page 11 used late binding (via the `IDispatch` interface) and declared all references as `Object`. In this chapter, because Visual Basic allows early binding by calling methods directly through the vtable, the types are specified in the declarations.

For example, to obtain a reference to a view of type `DIAccount`, declare a reference `accountObj` as follows:

```
' Visual Basic
Dim accountObj As BankBridge.DIAccount
```

Narrowing Object References

A client that holds a reference to a view can assign the reference to a derived interface if the implementation object referred to is an instance of the derived interface.

CORBA refers to such an assignment as *narrowing* the object reference. For example, suppose the client holds a reference to an `Account` view, but knows that the implementation object is actually a `CheckingAccount`. The following code extracts demonstrate how the client can obtain a `CheckingAccount` interface pointer:

Visual Basic
```
Dim accountObj as BankBridge.DIAccount
Dim theORB as Object CORBA_Orbix.DIOrbixORBObject
```

```
            Set theORB = CreateObject("CORBA.ORB.2")

            Dim myCheckingAccount as _
                    DICheckingAccount
            ...

            Set accountObj = ... ' Obtain a reference somehow

            Set myCheckingAccount = accountObj
            if myCheckingAccount = Nothing
               then 'Narrow failed
            endif
```

PowerBuilder
```
            OleObject theORB
            theORB = CREATE OLEObject
            theORB.ConnectToNewObject("CORBA.ORB.2")

            OleObject accountObj
            accountObj = ... //get

            OleObject myCheckingAccount
            myCheckingAccount = CREATE OleObject

            myCheckingAccount = theORB
            Narrow("CheckingAccount",accountObj)
            if IsEmpty(myCheckingAccount)
               then // Narrow failed
            endif
```

DIOrbixObject::Narrow()

Refer to the entry for `DIOrbixObject::Narrow()` in "OrbixCOMet API" on page 241 for an alternative way of narrowing an object reference.

A Visual Basic Client Program

This section shows Visual Basic code extracts for a client of the `bank` server that is developed in "Implementing CORBA Servers" on page 173. The code in this section is based on the Bank form shown in Figure 10.1 on page 151.

Recall the interface that the `bank` server presents to its clients:

```
// OMG IDL
interface Account {
   attribute float balance;
   attribute string owner;
   void makeDeposit(in float amount);
   void makeWithdrawal(in float amount);
};
interface CheckingAccount {
   attribute float overDraftLimit;
};
interface Bank {
   exception Reject { string reason; };
   Account newAccount(in string owner) raises (Reject);
   Account newCheckingAccount(in string owner) raises (Reject);
   Account getAccount(in string owner);
   void deleteAccount(in string owner);
};
```

Figure 10.1: *Bank Form Presenting the User's View of the Bank Service*

General Declarations

Note: If your Automation client requires type libraries to be registered, you must add a reference to the type library for early binding. In Visual Basic, use `Project>References` to add references.

```
Dim ObjFactory As CORBA_Orbix.DICORBAFactory
Dim bankObj As BankBridge.DIBank
Dim bankAccount As BankBridge.DIAccount
```

Creating the Form

The `Form_Load()` subroutine, which is called when the Bank form is loaded, creates a CORBA factory object in the bridge that will be used to create Automation views.

```
Private Sub Form_Load()
   ...
   Set ObjFactory = CreateObject("CORBA.Factory")
End Sub
```

Connecting to the CORBA Server

When a user clicks the Connect button in Figure 10.1 on page 151, the client connects to the `bankSrv` server on the host named in the Host textbox and binds to the `Bank` object whose marker is specified in the Marker textbox. (The client binds to any `Bank` object in the `bankSrv` server if no marker is specified.)

The client uses the `DICORBAFactory::GetObject()` method to bind to the Bank object.

It is important to handle errors that can be raised by the call to `GetObject()`. A call to `GetObject()`, or any other remote call, could fail for a number of reasons because of the complexity of making a call across a network. CORBA exceptions raised in the server are mapped into Automation exceptions by the bridge. In Visual Basic, these exceptions can be trapped using the `On Error` statement and handled using the standard `Visual Basic Err` object. "Error Handling" on page 189 explains CORBA exceptions and alternative ways of handling them in a client.

```
Private Sub cmdConnect_Click()
On Error GoTo errorTrap
   Set bankObj = ObjFactory.GetObject("Bank:" & _
      marker.Text & ":" & server_name.Text & _
      ":" & host_name.Text)
   ...
errorTrap:
   MsgBox (Err.Description & " occurred in " & Err.Source)
End Sub
```

Invoking Operations on Remote CORBA Objects

The following subroutines respond to user requests to create bank accounts and deposit and withdraw from accounts.

The subroutine cmdNew_Click() creates an Account or a CheckingAccount, depending on whether the CheckingAccount check box is clicked.

The IDL definitions specify that the operations Bank::newAccount() and Bank::newCheckingAccount() can raise the user exception Bank::Reject if the bank fails to create an account. In the following code sample, this exception is trapped using the On Error statement. "Error Handling" on page 189 shows a better way to handle this exception that provides more information to the user.

```
' New Account button
Private Sub cmdNew_Click()
On Error GoTo errorTrap
   If checking_acc.Value = 0 Then ' Create ordinary Account
      Set bankAccount = bankObj.newAccount(txtOwner.Text)
   ...
   Else
      If txtLimit.Text = "" Then
         MsgBox ("Enter Overdraft Limit!")
      Exit Sub
      Else ' Create CheckingAccount
         Set bankAccount = bankObj.newCheckingAccount( _
         ...
      End If
   End If
   checking_acc.Value = 0
   End If
Exit Sub
errorTrap:
    MsgBox (Err.Description & " occurred in " & Err.Source)
End Sub
```

```
' Get Details button
Private Sub cmdGet_Click()
On Error GoTo errorTrap
    ...
    Set bankAccount = bankObj.getAccount(txtOwner.Text)
    ...
Exit Sub
errorTrap:
    MsgBox (Err.Description & " occurred in " & Err.Source)
End Sub

' Deposit button
Private Sub cmdDeposit_Click()
On Error GoTo errorTrap
    ...
    bankAccount.makeDeposit CSng(txtAmount.Text)
    ...
Exit Sub
errorTrap:
    MsgBox (Err.Description & " occurred in " & Err.Source)
End Sub

' Withdraw button
Private Sub cmdWithdraw_Click()
On Error GoTo errorTrap
     ...
    bankAccount.makeWithdrawal CSng(txtAmount.Text)
    TxtBalance.Text = bankAccount.balance
    ...
errorTrap:
    MsgBox (Err.Description & " occurred in " & Err.Source)
End Sub
' Delete Account button
Private Sub cmdDelete_Click()
On Error GoTo errorTrap
     ...
    bankObj.deleteAccount (txtOwner.Text)
    Set bankAccount = Nothing
    ...
Exit Sub
errorTrap:
    MsgBox (Err.Description & " occurred in " & Err.Source)
End Sub
```

Disconnecting from the CORBA Server

Release the views in the bridge when the user disconnects from the bankSrv server.

```
Private Sub cmdDisconnect_Click()
    ...
    Set bankObj = Nothing
    Set bankAccount = Nothing
End Sub
```

Exiting the Application

Release the CORBA factory object when the user exits the application.

```
Private Sub Form_Unload(Cancel As Integer)
    Set ObjFactory = Nothing
End Sub
```

A PowerBuilder Client Program

This section shows PowerBuilder code extracts for a client of the bank server that is developed in "Implementing CORBA Servers" on page 173. The code in this section is based on the Bank form shown in Figure 10.1 on page 151.

The simplified PowerBuilder example shown here implements a client for a server that does not support the CheckingAccount interface.

Recall the interface that the bank server presents to its clients:

```
// OMG IDL
interface Account {
   attribute float balance;
   attribute string owner;

   void makeDeposit(in float amount);
   void makeWithdrawal(in float amount);
};

interface CheckingAccount {
   attribute float overDraftLimit;
};
```

```
interface Bank {
   exception Reject { string reason; };

   Account newAccount(in string owner) raises (Reject);

   Account newCheckingAccount(in string owner)
      raises (Reject);
   Account getAccount(in string owner);
   void deleteAccount(in string owner);
};
```

General Declarations

Declare global variables for the factory object and the CORBA object.

```
OleObject ObjFactory
OleObject bankObj
```

Loading the Window

Create the CORBA factory object within the open event for the Bank window.

```
ObjFactory = CREATE OleObject
ObjFactory.ConnectToNewObject("CORBA.Factory")
```

Connecting to the CORBA Server

When a user clicks the Connect button in the Bank window in Figure 10.1 on page 151, the client connects to the `bankSrv` server on the host specified in the Host textbox and binds to the `Bank` object whose marker is specified in the Marker textbox. (The client binds to any `Bank` object in the `bankSrv` server if no marker is specified.)

The client uses the `DICORBAFactory::GetObject()` method to bind to the Bank object.

```
bankObj = CREATE OleObject
bankObj = ObjFactory.GetObject(
         "Bank:bankSrv:" + host_name.Text)
```

Invoking IDL Operations

The following code implements the New Account, Delete Account, and Deposit and Withdraw buttons, responding to user requests to create bank accounts and to deposit and withdraw from accounts. Corresponding OMG IDL operations are invoked on `Bank` and `Account` objects in the CORBA server.

```
// New Account button
If owner.Text = "" Then
   MessageBox ("Error","Enter account owner's name")
Else
   accountObj = bankObj.newAccount( owner.Text )
   balance.Text = String( accountObj.balance )
   ...
End If
// Delete Account button
If owner.Text = "" Then
   MessageBox ("Error","Enter account name first")
Else
   bankObj.deleteAccount( accountObj )
   DESTROY accountObj
   ...
End If
// Deposit button
If enter_amt.Text = "" Then
   MessageBox("Error", "Enter amount" )
Else
   accountObj.makeDeposit( integer(enter_amt.Text) )
   balance.Text = String(accountObj.balance)
End If
   ...
   // Update balance after transaction.
   balance.Text = String( accountObj.balance )
End If
// Withdraw button
If enter_amt.Text = "" Then
   MessageBox( "Error", "Enter amount" )
Else
   accountObj.makeWithdrawal( Integer(enter_amt.Text) )
   // Update balance after transaction.
   balance.Text = String( accountObj.balance )
End If
```

157

Disconnecting from the Server

Release the Automation views when the user disconnects from the `bankSrv` server.

```
bankObj.DisconnectObject()
DESTROY accountObj
DESTROY bankObj
```

Unloading the Window

Release the CORBA factory object when the user exits the application.

```
ObjFactory.DisconnectObject()
DESTROY ObjFactory
```

Implementing CORBA Clients in COM

COM Apartments and Threading

COM and Automation view objects exposed by the bridge are marked with the `Both` attribute in the registry. This means these objects can be hosted in either an apartment-threaded or free-threaded client application. Refer to the Microsoft DCOM documentation for a fuller discussion of COM apartments and threading models.

Narrowing Object References

In CORBA, the process of converting a base object to a more derived instance is called *narrowing* an object reference. CORBA provides an API for doing this to ensure that type unsafe casts are not needed.

When using the COM mapping, CORBA objects do not explicitly need to be narrowed to a derived interface. If the object is actually an instance of the derived type, making a call to `QueryInterface` with the IID of the derived interface will be sufficient. If `QueryInterface` fails, this object cannot be validly converted to an instance of the derived type.

A C++ COM Client Program

The remainder of this section shows code extracts for a C++ COM client of the
`bank` server that is developed in "Implementing CORBA Servers" on page 173.

Recall the interface that the bank server presents to its clients:

```
// OMG IDL
interface Account {
   attribute float balance;
   attribute string owner;

   void makeDeposit(in float amount);
   void makeWithdrawal(in float amount);
};

interface CheckingAccount {
   attribute float overDraftLimit;
};

interface Bank {
   exception Reject { string reason; };

   Account newAccount(in string owner) raises (Reject);
   Account newCheckingAccount(in string owner)
      raises (Reject);
   Account getAccount(in string owner);
   void deleteAccount(in string owner);
};
```

Includes

```
// Include
#include <iostream.h>
#include <stdio.h>
#include <oaidl.h>
#include "bank.h"
```

General Declarations

```
// General Declaration
HRESULT hr=NOERROR;
IUnknown *pUnk=NULL;
```

```
ICORBAFactory *pCORBAFact=NULL;

// our custom interface
Ibank *pIBasic=NULL;
MULTI_QI mqi;
```

Connecting to the CORBA Factory

```
// In Process
memset (&mqi, 0x00, sizeof (MULTI_QI));
mqi.pIID = &IID_ICORBAFactory;
hr=CoCreateInstanceEx (IID_ICORBAFactory, NULL,
                       CLSCTX_INPROC_SERVER, NULL, 1, &mqi);
CheckHRESULT("CoCreateInstanceEx()", hr, FALSE);
pCORBAFact = (ICORBAFactory*)mqi.pItf;


// Out Process
memset (&mqi, 0x00, sizeof (MULTI_QI));
mqi.pIID = &IID_ICORBAFactory;
hr = CoCreateInstanceEx (IID_ICORBAFactory, NULL,
                       CLSCTX_LOCAL_SERVER | CLSCTX_REMOTE_SERVER,
                       NULL, 1, &mqi);
CheckHRESULT("CoCreateInstanceEx()", hr, FALSE);
pCORBAFact = (ICORBAFactory*)mqi.pItf;
```

Connecting to the CORBA Server

```
hr = pCORBAFact->GetObject("bank:bank:" & hostname,&pUnk);
if(!CheckErrInfo(hr, pCORBAFact, IID_ICORBAFactory))
{
   pCORBAFact->Release();
   return;
}
pCORBAFact->Release();
```

```
hr = pUnk->QueryInterface(IID_Ibank, (PPVOID)&pIBasic);
if(!CheckErrInfo(hr, pUnk, IID_Ibank))
{
   pUnk->Release();
   return;
}
pUnk->Release();
```

Invoking Operations on Remote CORBA Objects

```
Iaccount *pAcc = 0;
IcurrentAccount *pCurrAcc = 0;
IOrbixObject *pOrbixObj = 0;
float balance = 0, overdraft = 0, deposit = 1000000;
hr = pIBasic->newAccount("Ronan", &pAcc, NULL);
CheckErrInfo(hr, pIBasic, IID_Ibank);

if (SUCCEDED(pAcc->QueryInterface(IID_IOrbixObject,
                (PPVOID)&pOrbixObj)))
{
   LPSTR marker = 0, host = 0;
   hr = pOrbixObj->_get_Marker($marker);
   CheckErrInfo(hr, pOrbixObj, IID_IOrbixObject);
   cout << "Our marker is " << marker << endl;
   CoTaskMemFree(marker);
   hr = pOrbixObj->_get_Host(&host);
   CheckErrInfo(hr, pOrbixObj, IID_IOrbixObject);
   cout << "Our host is " << host << endl;
   CoTaskMemFree(host);
   pOrbixObj->Release();
}
else
   cout << "FAIL: QI for IID_IOrbixObject failed" << endl;

cout << "Calling makeLodgement()" << endl;
hr = pAcc->makeLodgement(deposit);
CheckErrInfo(hr, pAcc, IID_Iaccount);
cout << "Calling _get_balance()" << endl;
hr = pAcc->_get_balance(&balance);
CheckErrInfo(hr, pAcc, IID_Iaccount);
cout << "balance was " << balance << endl;
if(balance != deposit)
   cout << "FAIL: balance is not correct" << endl;
```

```
// now use QueryInterface() to see if we have really been given a
// CurrentAccount (this is like doing a _narrow in CORBA)
if (SUCCEEDED(pAcc->QueryInterface(IID_IcurrentAccount,
               (PPVOID)&pCurrAcc)))
{
   cout << "We have a current Account" << endl;
   hr = pCurrAcc->_get_overdraftLimit(&overdraft);
   CheckErrInfo(hr, pCurrAcc, IID_IcurrentAccount);
   cout << "Our overdraft limit is " << overdraft << endl;

// call a couple of methods from our base interface, i.e. account
cout << "Calling makeLodgement()" << endl;
hr = pCurrAcc->makeLodgement(deposit);
CheckErrInfo(hr, pCurrAcc, IID_IcurrentAccount);
cout << "Calling _get_balance()" << endl;
hr = pCurrAcc->_get_balance(&balance);
CheckErrInfo(hr, pCurrAcc, IID_IcurrentAccount);
cout << "balance was " << balance << endl;
if(balance != 2*deposit)
   cout << "FAIL: current account's balance is not correct!" <<
         endl;
pCurrAcc->Release();

// finally, just to prove that all the above happened to the same
// object, call account::balance
cout << "Calling _get_balance()" << endl;
hr = pAcc->_get_balance(&balance);
CheckErrInfo(hr, pAcc, IID_Iaccount);
cout << "balance was " << balance << endl;
if(balance != 2*deposit)
   cout << "FAIL: balance is not correct" << endl;
}
```

Disconnecting from the CORBA Server

```
hr = pIBasic->deleteAccount(pAcc);
CheckErrInfo(hr, pIBasic, IID_Ibank);
pAcc->Release();
pIBasic->Release();
```

Exiting the Application

```
CoUninitialize();
```

11

Exposing DCOM Servers to CORBA Clients

This chapter explains how to expose an existing DCOM server to CORBA clients. This functionality is particularly important in allowing a CORBA client to talk to applications such as Excel, Word, Access, and so on.

It used to be the case that developers wishing to expose DCOM objects to CORBA clients had to use the (D)IOrbixServerAPI interface to register their DCOM objects with the bridge. However, this is no longer required. You can now expose DCOM objects to CORBA clients without needing to write any such wrapper code. In addition, the existing DCOM server remains unchanged. The following is a summary of the main steps you need to follow to expose DCOM servers to CORBA clients:

- Build and register the DCOM server and any proxy/stub DLLs.

- Prime the OrbixCOMet type store with the correct type library.

- Register custsur.exe in the Implementation Repository under a given server name.

- Generate OMG IDL.

- Bind to the server and call operations.

An Existing DCOM Server

IONA ships some pure DCOM applications with OrbixCOMet in the comet_*x.x*\dcomapp directory (where *x.x* represents the release number). These are primarily intended to serve as diagnostic tools that allow trouble-shooting of DCOM installations without the added variable of a COM/CORBA bridge. A DCOM (local) server called fortune is provided in the dcomapp\testExe\server directory. This server is written using ATL and exposes objects supporting the following MIDL interface:

```
[
   object,
   uuid(F7B6A75D-90BF-11D1-8E10-0060970557AC),
   dual,
   helpstring("IIT_DcomTest Interface"),
   pointer_default(unique)
]
interface IIT_DcomTest : IDispatch
{
   [propget, id(1), helpstring("property fortune")]
   HRESULT fortune([out, retval] BSTR *pVal);
};
```

This chapter uses the example of the fortune server. When the COM C++ client in the dcomapp\testexe\client directory is run, the output is as follows:

```
[c:\iona\comet_x.x\dcomapp\testexe\client]client advice

Your fortune is :

    This fortune intentionally left blank :-) :-)
```

To build the above server, do the following:

```
//build the server exe
[c:\iona\comet_x.x] cd dcomapp\testexe\server
[c:\iona\comet_x.x\dcomapp\testexe\server] nmake -f
                                 IT_DcomApp.mak
//build the P/S dll
[c:\iona\comet_x.x\dcomapp\testexe\server] nmake -f
                                 IT_DcomAppps.mk
```

At this point, you might wish to check the server's operation using the DCOM client as described above.

Exposing the DCOM Server to CORBA

When talking to a CORBA server from COM/Automation, the Interface Repository must be populated with the required OMG IDL definitions so that the OrbixCOMet type store can obtain them the first time an application is run. (For more details about the OrbixCOMet type store, refer to "Development Support Tools" on page 121 and "Managing the Type Store" on page 211.) You can also populate the type store in advance by using the following command:

```
typeman -e <typename>
```

Because you want to contact a DCOM server, all the marshalling code will be based on the type library (in this case, `IT_DcomApp.tlb`). You must prime the type store with this type library as follows:

```
typeman -e c:\iona\comet_x.x\dcomapp\testexe\
               server\IT_DcomApp.tlb
```

Note: The full path to the type library must be supplied.

The next step is to decide on a CORBA server name, and to create an entry in the Implementation Repository under that name. In this case, the server name is `fortune`, which is an arbitrary choice. OrbixCOMet supplies a generic Orbix server (`custsur.exe`) that can masquerade as any server, receiving CORBA requests and making the corresponding call on the correct DCOM server. This is the server executable that you use when creating the entry in the Implementation Repository. The `custsur.exe` server has a dual personality because it can also act as a DCOM surrogate executable. This makes it a generic DCOM server as well as a generic Orbix server.

Type the following to register your server:

```
[c:\iona\comet_x.x\bin] putit fortune c:\iona\cometx.x\
                                      bin\custsur.exe
```

This server currently has an infinite timeout value. This means it stays around forever after it has been launched by the Orbix daemon.[1]

---

1. In future release of OrbixCOMet, there might be some switches that will allow this server to be timed-out after a specific period of inactivity.

To expose the server to CORBA, you just need to do the following:

- Register the type library.
- Register `custsur.exe` in the Implementation Repository under a server name.

Using the Server from CORBA

If you want to contact the server and invoke requests, you need some OMG IDL definitions. The `ts2idl.exe` can produce these by applying the COM/Automation->CORBA mapping rules to the type information stored in the type store (that is, `IT_DcomApp.tlb`).

The following command:

```
>ts2idl -i -r -f fortune.idl IT_DCOMAPPLib::IT_DcomTest
```

produces an OMG IDL file (`fortune.idl`) that will have, in this case, two interfaces called `IT_DCOMAPPLib::IIT_DcomTest` and `IT_DCOMAPPLib::IT_DcomTest` (coclass pseudo interface).

Both of these interfaces are scoped within a module (`IT_DCOMAPPLib`) that is the internal type library name. (You can check this using `oleview` if you wish.)

The OMG IDL shows the following:

```
// within module IT_DCOMAPPLib

interface IIT_DcomTest : CosLifeCycle::LifeCycleObject,
                         CORBA_COM::Composable
{
   readonly attribute string fortune;
};

// manufactured interface for coclass

interface IT_DcomTest : CosLifeCycle::LifeCycleObject,
                        CORBA_COM::Composable
{
   readonly attribute IT_DCOMAPPLib::IIT_DcomTest it_default;

};
```

There are several points to note here:

- The original `propget` (`fortune`) of type `BSTR` has been mapped to a readonly attribute of type `string`. This is as expected.

- All mapped interfaces inherit from `CosLifeCycle::LifeCycleObject`, which is one of the interfaces specified in the CORBA lifecycle service. This is because of DCOM and CORBA's differing approaches to reference counting.

  DCOM uses distributed reference counting. This means that when all outstanding references to an object are released (even for references that were held by remote clients), the server object's reference count will fall to zero and the object will be destroyed. When all objects in a DCOM server have been destroyed, the server typically shuts down.

  CORBA, however, takes a different view. Client calls to `_duplicate()` and `release()` should in no way affect the reference count of an object in the server. This can present problems in a COM/CORBA bridge where it launches DCOM servers in response to requests from CORBA clients, because the bridge would not know when to release DCOM interface pointers. The solution to this problem lies in the lifecycle interfaces, especially the method `CosLifeCycle::LifeCycleObject::remove()`. When a CORBA client is finished with a particular object reference, it should call `remove()` to release the DCOM interface pointer in the bridge, and thus allow the DCOM server to shut down if necessary.

- The presence of so-called "coclass pseudo interface". Coclasses are a feature of Microsoft IDL. They provide a listing of the interfaces that an object supports. The object itself is identified by its `CLSID`, which is provided in its UUID attribute, and each interface is marked with either "default" or "source" attributes. In the interface shown previously, `IIT_DcomTest` is the default interface for the coclass `IT_DcomTest` (the object that serves up fortune strings), and is represented by a readonly attribute on the pseudo coclass object. Any other interfaces supported by the coclass object (in this example there are no others) would also be represented by readonly attributes. You should think of these coclasses as your initial point of contact; for example, these are what you `_bind()` to from an Orbix client.

- All interfaces inherit from `Composable`, as mandated by the COM/ CORBA specification. This allows CORBA programmers to navigate between the various interfaces supported by the COM object in the absence of an inheritance relationship between those interfaces.

Writing a Client to Talk to the DCOM Server

You can write a client to talk to the DCOM server in the same way that you write any other CORBA client. You should firstly obtain an initial object reference. The following example uses `_bind()` to do this but you can also use `custsur.exe` to generate IORs for non-Orbix clients, (Refer to "Connection and Usage from Other ORBs" on page 170 for more details). After you have obtained an IOR you can then invoke operations. For example:

```
// C++

using namespace IT_DCOMAPPLib;

IT_DcomTest_var  dcomTestVar;
IIT_DcomTest_var defaultVar;

// _bind to the coclass pseudo object in server "fortune" on host
// "advice.iona.com"
dcomTestVar =  IT_DcomTest::_bind(":fortune", "advice.iona.com");

// now get the default interface of the coclass - IIT_DcomTest
// in our case

defaultVar = dcomTestVar->it_default();
if(!CORBA::is_nil(defaultVar))
{
   cout << "got default interface...calling fortune()" << endl;
   // call fortune()
   cout << "fortune is " << defaultVar->fortune() << endl;
   // lifecycle support - signal that we are finished with
   // this objref
   defaultVar->remove();
}
// lifecycle support - after this call, the DCOM server will
// have shut down...
dcomTestVar->remove();
```

If you were to examine the task list during the running of this client you would see that `IT_DcomApp.exe` appeared briefly and disappeared after the second call to `remove()` shown above. This means it was correctly shut down due to the lifecycle support.

CORBA Client Example Using Composable Support

The following is an example of a CORBA client of the `fortune` DCOM server that uses `composable` support (rather than the pseudo coclass object support described in "Writing a Client to Talk to the DCOM Server" on page 168):

```
#include "fortune.hh"
#include <iostream.h>
#include <stdlib.h>

int main (int argc, char **argv) {

   if (argc < 2) {
      cout << "usage: " << argv[0] << " <hostname>" << endl;
      exit (-1);
   }

   try {
      using namespace IT_DCOMAPPLib;

      CORBA::Object_var pObj;
      IT_DcomTest_var dcomTestVar;
      IIT_DcomTest_var defaultVar;
      dcomTestVar = IT_DcomTest::_bind(":fortune", argv[1]);

   cout << "_bind succeeded; calling query_interface()..." <<
   endl;
   pObj = dcomTestVar->query_interface
           ("IT_DCOMAPPLib::IIT_DcomTest");
   if(!CORBA::is_nil(pObj))
   {
      defaultVar = IIT_DcomTest::_narrow(pObj);
      if(CORBA::is_nil(defaultVar))
         cerr << "got nil obj ref after q_i()" << endl;
      else
      {
         cout << "fortune is " << defaultVar->fortune() << endl;
```

```
        defaultVar->remove();
      }
    }

      // lifecycle support
      dcomTestVar->remove();
  } catch (CORBA::SystemException &sysEx) {
    cerr << "Unexpected system exception" << endl;
    cerr << &sysEx;
    exit(1);
  } catch(...) {
    // an error occurred while trying to bind to the IT_DcomTest
    // object.
    cerr << "Bind to object failed" << endl;
    cerr << "Unexpected exception " << endl;
    exit(1);
  }
  return 0;
}
```

Connection and Usage from Other ORBs

You can use `custsur.exe` to generate IORs for non-Orbix clients. The following switches apply:

- `-g`   This generates an IOR.
- `-m`   This specifies the marker name.
- `-i`   This specifies the interface name.
- `-s`   This specifies the server name.
- `-f`   This specifies the filename.

For example, the following command would generate an IOR for the `IT_DcomTest` interface in the `fortune` server and write it to the `fortune.ior` file:

```
    custsur -g -i IT_DcomApplib::IT_DcomTest
        -s fortune -f c:\temp\fortune.ior
```

The following is an example of a VisiBroker client:

```
ifstream in(argv[1], ios::nocreate);
// read in the IOR, then do a string_to_object
if(!in.is_open())
{
    cerr << "Unable to open file " << argv[1] << endl;
    return 1;
}
in >> ior;
in.close();

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

objVar = orb->string_to_object(ior);
if(CORBA::is_nil(objVar))
{
    cerr << "string_to_object() returned a nil objref" << endl;
    return 1;
}

dcomTestVar= IT_DCOMAPPLib::IT_DcomTest::_narrow(objVar);
if(CORBA::is_nil(dcomTestVar))
{
    cerr << "_narrow() returned a nil objref" << endl;
    return 1;
}

cout << "About to get the default interface " << endl;

defaultVar= dcomTestVar->it_default();

if(!CORBA::is_nil(defaultVar))
{
    cout << "got default interface...calling fortune()" << endl;
    cout << "fortune is " << defaultVar->fortune() << endl;
    // lifecycle support
    defaultVar->remove();
}

// lifecycle support
dcomTestVar->remove();
```

12

Implementing CORBA Servers

You can use OrbixCOMet to implement COM/Automation servers that appear as CORBA servers. These servers can accept requests from standard COM/Automation clients and from CORBA clients. This chapter provides details about programming OrbixCOMet servers.

Note: If you wish to expose an existing DCOM server to CORBA clients without writing any wrapper code, refer to "Exposing DCOM Servers to CORBA Clients" on page 163.

A CORBA server presents an OMG IDL interface to its clients. To implement a CORBA server, your first step is to define the interfaces to your objects in OMG IDL.

You can use the Type Store Manager tool (`typeman`) to generate skeleton code to help you implement the interfaces to your objects. (Refer to "Generating Skeleton Code" on page 203 for more details.) You now have a standard Automation server.

To allow your COM/Automation server to accept CORBA requests, you can instantiate one or more COM/Automation objects and register them with OrbixCOMet, and then activate the server to receive CORBA requests. You can also register the server in the Orbix Implementation Repository. Your server is now a CORBA server.

The example provided in this chapter illustrates how to do this.

Defining the Interfaces

The example in this chapter represents a bank and its accounts. For example:

```
// OMG IDL
interface Account {
   attribute float balance;
   attribute string owner;

   void makeDeposit(in float amount);
   void makeWithdrawal(in float amount);
};

interface CheckingAccount : Account {
   attribute float overDraftLimit;
};

interface Bank {
   exception Reject { string reason; };

   Account newAccount(in string owner) raises (Reject);
   Account newCheckingAccount(in string owner)
      raises (Reject);
   Account getAccount(in string owner);
   void deleteAccount(in string owner);
};
```

A Visual Basic version of this example is available in the \demo\VB6\bank
directory of your OrbixCOMet installation. A COM C++ version can be found
in the \demo\com\idl_demo directory of your OrbixCOMet installation.

Generating the Skeleton Code

Generating skeleton code automates the task of translating your OMG IDL
interface definitions into equivalent definitions in your implementation language.
It also ensures that all parameters are available in order and that they are passing
the correct types. For more details about generating skeleton code, refer to
"Development Support Tools" on page 121.

Implementing CORBA Servers in Automation

Implementing the Interfaces

To implement the OMG IDL interfaces, you implement a class in your chosen implementation language, exactly as you would for a normal Automation server.

The interfaces defined in your OMG IDL file define the interface that (remote) CORBA clients use to interact with your server objects. You must provide implementations of these interfaces, and each of their operations and attributes, in your chosen implementation language.

You might also need to implement supporting classes, functions or subroutines to complete your application. In the following Visual Basic example, the collections `colAccounts` and `colCheckingAccounts` are needed to maintain a collection of `Account` and `CheckingAccount` objects owned by the `Bank`. The code for `colAccounts` and `colCheckingAccounts` is not shown here. It can be found in the `accounts.cls` and `checkingaccounts.cls` files in the `demo\VB6\banksvr\` directory of your OrbixCOMet installation.

In the code samples in the following subsections, the additions to the generated code are shown in bold text.

Implementing the Account Interface

```
' Visual Basic (account.cls)
Private accBalance As Single
Private accOwner As String

Public Property Get balance() As String
   balance = accBalance
End Property

Public Property Let balance( _
   ByVal var_balance As String)
   accBalance = var_balance
End Property

Public Property Get owner() As String
   owner = accOwner
End Property
```

```
Public Property Let owner( _
   ByVal var_owner As String)
   accOwner = var_owner
End Property

Public Sub makeDeposit( _
   ByVal var_amount As Single, _
   Optional ByRef IT_Ex As Variant)
   accBalance = accBalance + var_amount
End Sub

Public Sub makeWithdrawal( _
   ByVal var_amount As Single, _
   Optional ByRef IT_Ex As Variant)
   If ((accBalance - var_amount) >= 0) Then _
      accBalance = accBalance - var_amount
End Sub
```

Implementing the CheckingAccount Interface

The interface CheckingAccount inherits from Account. To implement the
CheckingAccount interface, you must reimplement the properties and methods
inherited from Account. You must also implement the overdraftLimit
property that the CheckingAccount interface adds.

```
' Visual Basic (checkingaccount.cls)
Private parentAcc As New Account
Private accLimit As Single

Public Property Let overdraftLimit(
      ByVal var_overdraftLimit As Single)
   accLimit = var_overdraftLimit
End Property

Public Property Get overdraftLimit() As Single
   overdraftLimit = accLimit
End Property

Public Property Get balance()
   balance = parentAcc.balance
End Property
```

```
Public Property Let owner(ByVal owner As String)
    parentAcc.owner = owner
End Property

Public Property Get owner() As String
    owner = parentAcc.owner
End Property

Public Sub makeDeposit(ByVal amount As Single, _
      Optional IT_Ex As Variant)
    parentAcc.makeDeposit amount
End Sub

Public Sub makeWithdrawal(ByVal amount As Single, _
      Optional IT_Ex As Variant)
   If ((parentAcc.balance(amount-overdraftLimit))>=0) Then _
        parentAcc.balance = parentAcc.balance - amount
End Sub
```

Implementing the Bank Interface

The operations newAccount() and newCheckingAccount() on interface Bank
raise an exception if the bank fails to create an account. The code to raise an
exception is not included here. "Error Handling" on page 189 deals with this
topic in detail.

```
' Visual Basic (Bank.cls)
...
Public Function newAccount( _
      ByVal var_owner As String, _
      Optional ByRef IT_Ex As Variant) As Object
   ...
   ' Raise Reject exception here, if Bank cannot
   ' create account. .
   ...
   Set newAccount = Accounts.Add(var_owner)
   frmBankSrv.details.AddItem "Created new _
     account for Customer : " & newAccount.owner
End Function
```

```
Public Sub deleteAccount( _
     ByVal var_owner As String, _
     Optional ByRef IT_Ex As Variant)
   Accounts.Remove var_owner
End Sub

Public Function getAccount( _
     ByVal var_owner As String, _
     Optional ByRef IT_Ex As Variant) As Object
   Set getAccount = Accounts.Item(var_owner)
End Sub
```

Registering with OrbixCOMet

When you have implemented your OMG IDL interfaces, you have developed an
Automation server. To make your Automation server appear as a CORBA
server, you must instantiate your implementation Automation object and
register it with OrbixCOMet. (If it makes sense for your application, you might
want to create more than one implementation object.)

Visual Basic

```
Dim orb As Object
Dim bankobj As New Bank
Dim serverAPI As Object

Private Sub Form_Load()
On Error GoTo errorTrap
   Set orb = CreateObject("CORBA.ORB.2")

   Set serverAPI = orb.GetServerAPI
   Set orb = Nothing
   Call serverAPI.SetObjectImpl("bank", "", bankObj)
   Call serverAPI.Activate("bank")
   Exit Sub
errorTrap:
   MsgBox (Err.Description & " in " & Err.Source)
   Err.Clear
End Sub
```

```
Private Sub Form_Unload(Cancel As Integer)
   Call serverAPI.Deactivate("bank")
   Set serverAPI = Nothing
End Sub
```

PowerBuilder

This section demonstrates how to use PowerBuilder to transform an Automation server to a CORBA server.

Note: In PowerBuilder, your implementation (user) objects must be exposed with valid ProgIDs using the PowerBuilder tool PBGENREG.EXE.

```
// Get a reference to the ITServerAPI object
OleObject orb
orb = CREATE OleObject
orb.ConnectToNewObject("CORBA.ORB.2")
ServerAPI=orb.GetServerAPI()

// Instantiate a Bank object.
// You first need to use PBGENREG.EXE to expose the Bank
// object with the ProgID 'bankSrv.bankImplObject'
OleObject bankObj
bankObj = CREATE OleObject
bankObj.ConnectToNewObject("bankSrv.bankImplObject")

// Register bankObj with the Bridge.
serverAPI.setObjectImpl("Bank", "", bankObj)

// Activate the server so that bankObj
// can receive incoming calls from CORBA clients.
serverAPI.Activate("Bank")

// Deactivate the server when finished.
serverAPI.Deactivate("Bank")
```

The code instantiates a Bank object and registers it with the bridge by calling SetObjectImpl() on the bridge's ITServerAPI interface.

`SetObjectImpl()` specifies the IDL interface that the registered object supports in its first parameter and specifies the object's marker in its second parameter. No marker is specified in this example. Therefore, Orbix will choose the marker for the `Bank` object.

The next step is to activate the server so that any objects registered with the bridge will receive incoming requests from CORBA clients. In this case, the call to `Activate()` gives the server the name `bankSrv`. This is also the name with which the server will be registered in the Implementation Repository. (Refer to "Registering the CORBA Server in the Implementation Repository" on page 188 for more details.)

When your application no longer needs to receive CORBA client requests, you can deactivate the server by calling `Deactivate()`.

Implementing CORBA Servers in COM

Implementing the Interfaces

To implement the OMG IDL interfaces, you implement a class in your chosen implementation language, exactly as you would for a normal COM server.

The interfaces defined in your OMG IDL file define the interface that (remote) CORBA clients use to interact with your server objects. You must provide implementations of these interfaces, and each of their operations and attributes, in your chosen implementation language.

You might also need to implement supporting classes, functions or subroutines to complete your application.

Implementing the Account Interface

```
ICOMAccountImpl :: ICOMAccountImpl(char *szOwner)
: i_nRef(0)
{
   m_fBalance=0;
   if(strlen(szOwner)<20)
      m_szOwner=szOwner;
   else
   {
      *(szOwner+20)=\0;
      m_szOwner=szOwner;
   }
}


ICOMAccountImpl :: ~ICOMAccountImpl() {
}

ICOMAccountImpl* ICOMAccountImpl :: Create(char *szOwner) {
   ICOMAccountImpl * pObj;
   if ((pObj = new ICOMAccountImpl(szOwner)) == NULL)
      return NULL;
   pObj->AddRef();
   return pObj;
}

STDMETHODIMP ICOMAccountImpl :: QueryInterface (REFIID riid, void
                                                  ** ppv) {
   *ppv=NULL;
   if (IsEqualIID(riid, IID_IUnknown) || IsEqualIID(riid,
                  IID_Iaccount)) {

      *ppv=this;
      AddRef();
      return NOERROR;
   }
   return E_NOINTERFACE;
}

STDMETHODIMP_(ULONG) ICOMAccountImpl :: AddRef() {
   return InterlockedIncrement(&i_nRef);
}
```

```
STDMETHODIMP_(ULONG) ICOMAccountImpl :: Release() {
   if (0!=InterlockedDecrement(&i_nRef))
      return i_nRef;
   delete this;
   return 0;
}

// custom methods
STDMETHODIMP ICOMAccountImpl :: makeLodgement (float f) {
   m_fBalance+=f;
   return NOERROR;
}

STDMETHODIMP ICOMAccountImpl :: makeWithdrawal (float f) {
   if(m_fBalance>=f)
      m_fBalance-=f;
   return NOERROR;
}

STDMETHODIMP ICOMAccountImpl :: _get_balance (float *val) {
   *val=m_fBalance;
   return NOERROR;
}
```

Implementing the CheckingAccount Interface

The interface CheckingAccount inherits from Account. To implement the CheckingAccount interface, you must reimplement the properties and methods inherited from Account. You must also implement the overdraftLimit property that the CheckingAccount interface adds.

```
ICOMCheckingAccountImpl :: ICOMCheckingAccountImpl(char *szOwner,
                              float fOverLimit)
: i_nRef(0)
{
   m_fBalance=0;
   if(strlen(szOwner)<20)
      m_szOwner=szOwner;
   else
   {
      *(szOwner+20)=\0;
      m_szOwner=szOwner;
   }
   m_fOverLimit=fOverLimit;
}

ICOMCheckingAccountImpl :: ~ICOMCheckingAccountImpl() {
}

ICOMCheckingAccountImpl* ICOMCheckingAccountImpl :: Create() {
   ICOMAccountImpl * pObj;
   if ((pObj = new ICOMCheckingAccountImpl) == NULL)
      return NULL;
   pObj->AddRef();
   return pObj;
}

STDMETHODIMP ICOMCheckingAccountImpl :: QueryInterface (REFIID
                                    riid, void ** ppv) {
   *ppv=NULL;
   if (IsEqualIID(riid, IID_IUnknown) ||
       IsEqualIID(riid, IID_Iaccount) ||
       IsEqualIID(riid, IID_IcurrentAccount)) {

      *ppv=this;
      AddRef();
      return NOERROR;
   }
   return E_NOINTERFACE;
}

STDMETHODIMP_(ULONG) ICOMCheckingAccountImpl :: AddRef() {
   return InterlockedIncrement(&i_nRef);
}
```

```
STDMETHODIMP_(ULONG_ ICOMCheckingAccountImpl :: Release() {
   if (0!=InterlockedDecrement(&i_nRef))
      return i_nRef;
   delete this;
   return 0;
}

// custom methods
STDMETHODIMP ICOMCheckingAccountImpl :: makeLodgement (float f) {
   m_fBalance+=f;
   return NOERROR;
}

STDMETHODIMP ICOMCheckingAccountImpl :: makeWithdrawal (float f) {
   if((m_fBalance-f)>m_fOverLimit)
      m_fBalance-=f;
   else
      return -1;
      return NOERROR;
}

STDMETHODIMP ICOMCheckingAccountImpl :: _get_balance (float *val)
{
   *val=m_fBalance;
   return NOERROR;
}

STDMETHODIMP ICOMCheckingAccountImpl :: _get_overdraftLimit (float
                                        *val) {
   *val=m_fOverLimit;
   return NOERROR;
}
```

Implementing the Bank Interface

The operations `newAccount()` and `newCheckingAccount()` on interface Bank
raise an exception if the bank fails to create an account. The code to raise an
exception is not included here. "Error Handling" on page 189 deals with this
topic in detail.

```
ICOMBankImpl :: ICOMBankImpl()
: i_nRef(0)
{
}

ICOMBankImpl :: ~ICOMBankImpl() {
}

ICOMBankImpl* ICOMBankImpl :: Create() {
   ICOMBankImpl * pObj;
   if ((pObj = new ICOMBankImpl) == NULL)
      return NULL;
   pObj->AddRef();
   return pObj;
}

STDMETHODIMP ICOMBankImpl :: QueryInterface (REFIID riid, void **
                                                ppv) {
   *ppv=NULL;
   if (IsEqualIID(riid, IID_IUnknown) ||
      IsEqualIID(riid, IID_Ibank)) {
     *ppv=this;
     AddRef();
     return NOERROR;
   }
   return E_NOINTERFACE;
}

STDMETHODIMP_(ULONG) ICOMBankImpl :: AddRef() {
   return InterlockedIncrement(&i_nRef);
}

STDMETHODIMP_(ULONG) ICOMBankImpl :: Release() {
   if (0!=InterlockedDecrement(&i_nRef))
      return i_nRef;
   delete this;
   return 0;
}
```

```
// custom methods
STDMETHODIMP ICOMBankImpl :: deleteAccount
   (ICOMCheckingAccountImpl *a)
{
   a->Release();
   return NOERROR;
}


STDMETHODIMP ICOMBankImpl :: newAccount (LPSTR name,
   ICOMAccountImpl **val, ICOMBankUserExceptionsImpl
   **ppException)
{
   *val=new ICOMAccountImpl(name);
   return NOERROR;
}

STDMETHODIMP ICOMBankImpl :: newCheckingAccount (LPSTR name,
   float limit, ICOMCheckingAccountImpl **val,
   ICOMBankUserExceptionsImpl **ppException)
{
   *val=new ICOMCheckingAccountImpl(name,limit);
   return NOERROR;
}
```

Registering with OrbixCOMet

When you have implemented your OMG IDL interfaces, you have developed a
COM server. To make your COM server appear as a CORBA server, you must
instantiate an implementation COM object and register it with OrbixCOMet. (If
it makes sense for your application, you might want to create more than one
implementation object.)

The following code shows how to complete your implementation:

```
ICOMBankImpl* m_Bank=new ICOMBankImpl();

IOrbixORBObject * poOrb=0;
hr = CoCreateInstance(IID_IOrbixORBObject, NULL, ctx,
                 IID_IOrbixORBObject, (void**)&poOrb);
CheckHRESULT("Connecting to ORB", hr, FALSE);

IOrbixServerAPI * poAPI=0;
```

```
hr = poOrb->GetServerAPI(&poAPI);
CheckHRESULT("getting Server API", hr, FALSE);
poOrb->Release();

// register our COM object as the CORBA interface 'ClientObject'
poAPI->setObjectImpl("Bank", "", m_Bank);

poAPI->Activate("banksvr");

poAPI->Release();
```

The code instantiates a `Bank` object and registers it with the bridge by calling `SetObjectImpl()` on the bridge's `ITServerAPI` interface.

`SetObjectImpl()` specifies the IDL interface that the registered object supports in its first parameter and specifies the object's marker in its second parameter. No marker is specified in this example. Therefore, OrbixCOMet will choose the marker for the `Bank` object.

The next step is to activate the server so that any objects registered with the bridge will receive incoming requests from CORBA clients. In this case, the call to `Activate()` gives the server the name `bankSrv`. This is also the name with which the server will be registered in the Implementation Repository. (Refer to "Registering the CORBA Server in the Implementation Repository" on page 188 for more details.)

When your application no longer needs to receive CORBA client requests, you can deactivate the server by calling `Deactivate()`.

Running the Server

You can now build your server executable as normal for the language you are using. Your server project name will be used as the first part of the ProgID for your server's Automation objects.

Registering the **CORBA** Server in the Implementation Repository

Your server executable must be registered in the Orbix Implementation Repository so that the Orbix daemon knows how to activate it when a CORBA client makes a request on one of its objects.

You must register your server with the name that was specified in the call to `Activate()`. In this example, the server must be registered with the name `bankSrv`.

You can register your server as follows using `putit`:

```
putit bankSrv executable_file
```

where `executable_file` is the full path to the server program.

13

Error Handling

Error handling is an important aspect of programming an OrbixCOMet application. Remote method calls are much more complex to transmit than local method calls, so there are many more possibilities for error. This chapter explains how CORBA exceptions can be handled in a client and how a server can raise a user exception.

CORBA defines a standard set of system exceptions that can be raised by the ORB during the transmission of remote operation calls and reported to a client or server. These exceptions range between reporting network problems and failure to marshal operation parameters.

CORBA also allows users to define application-specific exceptions that allow an application to define the set of exception conditions associated with it. These exceptions are defined in the `raises` clause of an OMG IDL operation as explained in the Orbix documentation set.

Applications do not (and should not) explicitly raise system exceptions. However, they should handle system exceptions and user exceptions appropriately.

CORBA Exceptions

A client application should handle user exceptions, defined in an OMG IDL `raises` clause, that can be raised by a call to an OMG IDL operation.

A client should also handle system exceptions that can be raised by OrbixCOMet itself, either during a remote invocation or by calls to OrbixCOMet. OrbixCOMet might raise a system exception if, for example, it encounters a problem with the network.

Example of User Exception

Recall the `Bank` interface defined in "Implementing CORBA Servers" on page 173.

```
// OMG IDL
interface Bank {

   exception Reject {
      string reason;
   };

   Account newAccount(in string owner) raises (Reject);
   ...
};
```

Note: An operation can raise more than one exception. For example:
```
      Account newAccount(in string owner) raises (Reject,
            BankClosed);
```

If the bank fails to create an account (for example, because the owner already has an account at the bank), the operation `newAccount()` raises the `Reject` user exception. The `Reject` exception contains one member, of type `string`, that is used to specify the reason why the request for a new account was rejected.

The `newAccount()` operation can, of course, raise a system exception if some problem is encountered by OrbixCOMet during the operation invocation. However, system exceptions are not listed in a `raises` clause, and user code should never raise a system exception.

The Automation view of these OMG IDL definitions is as follows:

```
// MIDL
interface DIBank : IDispatch {
   HRESULT newAccount(
      [in] BSTR owner,
      [optional,out] VARIANT* IT_Ex,
      [retval,out] IDispatch** IT_retval);
   ...
}
...
interface DIBank_Reject : DICORBAUserException {
   [propput] HRESULT reason( [in] BSTR reason );
   [propget] HRESULT reason(
      [retval,out] BSTR* IT_retval);
}
```

The COM view of these OMG IDL definitions is as follows:

```
// OMG IDL
interface Ibank: IUnknown
{
   typedef struct tagbank_reject
   {
      LPSTR reason;
   } bank_reject;
   HRESULT deleteAccount([in] Iaccount *a);
   HRESULT newAccount([in, string] LPSTR name,
                      [out] Iaccount **val,
                      [in,out,unique] bankExceptions
                            **ppException);
   HRESULT newCurrentAccount([in, string] LPSTR name,
                         [in] float limit,
                         {out] IcurrentAccount **val,
                         [in,out,unique] bankExceptions
                               **ppException);
};
```

Refer to "Mapping CORBA Objects to Automation" on page 41 for details of how OMG IDL interfaces and exceptions translate to Automation. Refer to "Mapping CORBA Objects to COM" on page 81 for details of how OMG IDL interfaces and exceptions translate to COM.

Exception Properties

System exceptions and user exceptions have a number of properties that allow you to find information about an exception that has occurred. Both system exceptions and user exceptions expose the interface `(D)IForeignException` that is defined as follows:

```
interface DIForeignException : DIForeignComplexType {
   [propget] HRESULT EX_majorCode(
      [retval,out] long* IT_retval);

   [propget] HRESULT EX_Id(
      [retval,out] BSTR* IT_retval);
};
```

The method `EX_majorCode()` indicates the category of exception raised. It can be one of the following defined in the file `ITStdInterfaces.tlb`:

```
      EXCEPTION_NO
      EXCEPTION_USER
      EXCEPTION_SYSTEM
```

The method `EX_Id()` indicates the type of exception raised. For example, `CORBA::COMM_FAILURE` is an example of a system exception. `Bank::Reject` is an example of a user exception (based on the `Bank` interface in "Example of User Exception" on page 190).

System Exception Properties

System exceptions have the following additional properties defined in `(D)ICORBASystemException`:

```
interface DICORBASystemException : DIForeignException {
   [propget] HRESULT EX_minorCode(
      [retval,out] long* IT_retval);
   [propget] HRESULT EX_completionStatus(
      [retval,out] long* IT_retval);
};
```

`EX_completionStatus()` indicates the status of the operation at the time the system exception was raised. The status can be as follows:

`COMPLETION_YES`	This means the operation had completed before the exception was raised.
`COMPLETION_NO`	This means the operation was never initiated.
`COMPLETION_MAYBE`	This means the operation was initiated but it cannot be determined whether or not it had completed.

`EX_minorCode()` returns a code describing the type of system exception that has occurred. Each of these codes is assigned a descriptive identifier that is declared as a constant in the file `_orbix.bas`. A minor code can be looked up in the error messages file, `ERRMSGS`, to find a textual description of the code.

Exception Handling in Automation

CORBA exceptions are mapped into Automation exceptions by the bridge. This allows exceptions raised by calls to CORBA objects to be handled in whatever way your development tool handles Automation exceptions.

User exceptions can define members as part of their OMG IDL definition. However, using Automation's native exception handling, these members cannot be accessed by a caller.

Exception Handling in Visual Basic

In Visual Basic, exceptions can be trapped using the `On Error GoTo` clause and handled using the standard `Err` object. For example:

```
' Visual Basic
Dim accountObj As BankBridge.DIAccount
Dim bankObj As BankBridge.DIBank
On Error Goto errorTrap

' Obtain a reference to a Bank object:
Set bankObj = ...
Set accountObj = bankObj.newAccount(owner)
...
```

```
Exit Sub
errorTrap:
   MsgBox(Err.Description & _
        " occurred in " & Err.Source)
End Sub
```

The Err Object

The details of any exception that occurs are available as properties of the standard `Err` object. (Refer to your Visual Basic documentation for full details of the `Err` object.) For example:

- `Err.Description` provides details of the exception, including the name of the exception; for example, `CORBA::COMM_FAILURE` or `Bank::Reject`.

  For a user exception, an example of the string in `Err.Description` is:

  `CORBA User Exception :[Bank::Reject]`

  For a system exception, an example is:

  ```
  CORBA System Exception :[CORBA::COMM_FAILURE]
  minor code [10087][NO]
  ```

- `Err.Source` indicates the operation that raised the exception; for example, `Bank.newAccount`.

Inline Exception Handling

The second approach to handling exceptions in an Automation client is to use the exception parameter directly.

Recall that an OMG IDL operation translates to an Automation method that has an additional optional parameter. For example:

```
interface Account {
   ...
   void makeDeposit(in float amount,
      out float balance);
};
```

translates to:

```
// MIDL
interface DIAccount : IDispatch {
   ...
   HRESULT makeDeposit(
      [in] float amount,
      [out] float* balance,
      [optional, in, out] VARIANT* IT_Ex);
}
```

A client can pass this parameter in a method call, and check if it contains an exception after the call. The error-handling code must be written inline. The ability to handle user exceptions inline is useful because user exceptions can be thrown to indicate logical errors rather than unrecoverable errors.

However, it allows the caller to get additional information about a user exception that has occurred. A user exception can define one or more members that translate to MIDL methods that can be used by the caller to extract this additional information. (Refer to "Mapping CORBA Objects to Automation" on page 41 and "Mapping Automation Objects to CORBA" on page 71 for a description of the mapping between OMG IDL and MIDL user exceptions.)

Standard Automation expection handling is disabled when the exception parameter is passed in a method. This allows the value of the exception to be examined inline.

The code extracts in the following subsections show two ways of checking for and handling an exception using this exception parameter. Assume that `newAccount()` can raise the user exception `Reject` defined as follows:

```
// OMG IDL
interface Bank {
   exception Reject {
      string reason;
   };
...
};
```

Using Type Information

You can use type information to check the type of exception that occurred.

```
' Visual Basic

Dim excep as Variant
Set excep = Nothing
...
Set bankObj = ...
Set accountObj = bankObj.newAccount owner, excep

If TypeOf excep Is DIBank_Reject
   MsgBox "User Exception: " + excep.reason
ElseIf TypeOf excep Is CORBA_Orbix.DICORBASystemException And _
   Not excep.EX_majorCode = 0 Then
      MsgBox ("error creating account : " & _
         excep.EX_Id)
End If
```

Using the Standard Interfaces

You can call the methods defined on the standard interfaces supported by
`DICORBAUserException` and `DICORBASystemException` directly. (Refer to
"Exception Properties" on page 192.)

Visual Basic

```
Dim excep As Variant
Set excep = Nothing
...
Set bankObj = ...
Set accountObj = bankObj.newAccount owner, excep

If excep.EX_majorCode() = IT_SystemException
Then ' this is a system exception
   MsgBox(excep.EX_Id)
Else If excep.EX_majorCode() = IT_UserException
Then ' this is a user exception
   If excep.EX_Id = "[Bank::Reject]" Then
        MsgBox(excep.EX_Id & excep.reason)
```

```
    Else
        MsgBox(excep.EX_repositoryID)
    End If
End If
```

A program can check for a specific minor code as follows:

```
If excep.EX_minorCode = NARROW_FAILED Then
    ...
```

Exception Handling in COM

Catching COM Exceptions

The bridge will translate the exception into a standard COM exception. There
are two parts to the exception. The first part, HRESULT, gives the class of
exception. The second part is a human-readable form of the exception exposed
through the IErrorInfo interface. For example:

```
HRESULT hRes;
IErrorInfo *pIErrInfo = 0;
ISupportErrorInfo *pISupportErrInfo = 0;

if(SUCCEEDED(hr))
    return TRUE;

if(SUCCEEDED(pUnk->QueryInterface(IID_ISupportErrorInfo,
                            (PPVOID)&pISupportErrInfo)))
{
    if(SUCCEEDED(pISupportErrInfo->InterfaceSupportsErrorInfo
                                (riid)))
    {
        hRes = GetErrorInfo(0, &pIErrInfo);
        if(hRes == S_OK)
        {
            pIErrInfo->GetSource(&src);
            pIErrInfo->GetDescription(&desc);
            mbsrc = WSTR2CHAR(src);
            mbdesc = WSTR2CHAR(desc);
            SysFreeString(src);
            SysFreeString(desc);
```

```
                    mbmsg = new char [strlen(mbsrc) + strlen(mbdesc)+strlen
                                    (" : ")+1];
                    sprintf(mbmsg, "%s : %s", mbsrc, mbdesc);
                    pIErrInfo->Release();
                    CheckHRESULT(mbmsg, hr);
                    delete [] mbsrc;
                    delete [] mbdesc;
                    delete [] mbmsg;
                }
                else
                    cout << "No error object found" << endl;


            }
            pISupportErrInfo->Release{};
        }
        CheckHRESULT("Error : ", hr);
```

Using Direct-to-COM Support in Visual C++ 5.0

In this case, CORBA exceptions are mapped to the standard _com_error
exception. For example:

```
try
{
    short h, w;
    DIbankPtr bank;
    DIaccountPtr acc;
    DICORBAFactoryPtr fact;

    fact.CreateInstance("CORBA.Factory");
    bank = fact->GetObject("bank:bank:", NULL);
    acc = bank->newAccount("Ronan", NULL);
    cout << "Created new account 'Ronan'" << endl;
    acc->makeLodgement(100, NULL);
    cout << "Deposited $100" << endl;
    cout << "New balance is " << acc->Getbalance() << endl;
    bank->deleteAccount(acc, NULL);
    cout << "Deleted account" << endl;
}
```

```
catch (_com_error &e)
{
   print_error(e);
}
catch (...)
{
   cerr << "Caught unknown exception " << endl;
}
```

Raising an Exception in a Server

When an OMG IDL operation definition specifies a `raises` clause, the server's implementation of that operation should raise the exception(s) specified in an appropriate way.

In the `Bank` example, the implementation of the OMG IDL operation `newAccount()` raises the `Reject` exception when it fails to create an account.

To raise the exception, create an exception object using the `(D)ICORBAFactoryEx::CreateType()` method. (Refer to "Translation of Constructed Types" on page 53 and page 89 for more details.)

If the OMG IDL exception defines members, you must assign appropriate data to these members to provide details about the exception to the caller. You then assign the exception into the `IT_ex` parameter whose purpose is to transmit system and user exceptions back to the caller. It is good practice to exit the function immediately after you raise an exception.

Automation Exceptions

```
' Visual Basic
Dim ObjFactory As CORBA_Orbix.DICORBAFactory

Public Function newAccount( _
     ByVal var_owner As String, _
     Optional ByRef IT_Ex As Variant) As Object
   ...
   If ...' owner has account at the bank
     If Not IsMissing(IT_Ex) Then
        Dim excep As BankBridge.DIBank_Reject
```

```
            Set excep = ObjFactory.CreateType(Nothing,_
                "Bank/Reject")
            excep.reason = "Account already exists!"
            Set IT_Ex = excep
            Exit Function
          End If
       Else ... ' create new account
       ...
    End Function
```

COM Exceptions

```
try
{
   short h, w;
   DIbankPtr bank;
   DIaccountPtr acc;
   DICORBAFactoryPtr fact;

   fact.CreateInstance("CORBA.Factory");
   bank = fact->GetObject("bank:bank", NULL);
   acc = bank->newAccount("Ronan", NULL);
   cout << "Created new account 'Ronan'" << endl;
   acc->makeLodgement(100, NULL);
   cout << "Deposited $100" << endl;
   cout << "New balance is " << acc->Getbalance() << endl;
   bank->deleteAccount(acc, NULL);
   cout << "Deleted account" << endl;
}
catch (_com_error &e)
{
   print_error(e);
}
catch (...)
{
   cerr << "Caught unknown exception " << endl;
}
```

14

Client Callbacks

Usually, CORBA clients invoke operations on objects in CORBA servers. However, CORBA clients can implement some of the functionality associated with servers, and all servers can act as clients. A callback invocation is a programming technique that takes advantage of this. This chapter describes client callbacks.

A callback is an operation invocation made from a server to an object that is implemented in a client. A callback allows a server to send information to clients without forcing clients to explicitly request the information.

Callbacks are typically used to allow a server to notify a client to update itself. For example, in the `bank` application, clients might maintain a local cache to hold the balance of accounts for which they hold references[1]. Each client that uses the server's account object maintains a local copy of its balance. If the client accesses the balance attribute, the local value is returned if the cache is valid. If the cache is invalid, the remote balance is accessed and returned to the client, and the local cache is updated.

When a client makes a deposit or withdrawal from an account, it invalidates the cached balance in the remaining clients that hold a reference to that account. These clients must be informed that their cached value is invalid. To do this, the real account object in the server must notify (that is, call back) its clients whenever its balance changes.

---

1. A bridge holds an Orbix proxy as well as a COM/Automation view for each implementation object to which it has a reference. The client could maintain a cache by implementing a smart proxy. Refer to the Orbix documentation set for details about writing smart proxies.

Implementing Callbacks

To implement callbacks, you must do the following:

- Define the OMG IDL interfaces for the server's objects and the client's objects.

- Write a client.

- Write a server and register it in the Implementation Repository.

The OMG IDL Interfaces

The client implements an interface that the server will use to call back clients. A suitable interface might be defined as:

```
// OMG IDL
interface NotifyCallback{
    oneway void notifyClient();
}
```

The `notifyClient()` operation is declared to be `oneway` because it is important that the server is not blocked when it calls back its clients.

The server implements an interface that allows it to maintain a list of clients that should be notified of changes in its objects' data. A suitable interface might be defined as:

```
// OMG IDL
interface RegisterCallback{
    void registerClient(in NotifyCallback client);
    void unregisterClient(in NotifyCallback client);
}
```

The operation `registerClient()` registers a client with the server. The parameter to `registerClient()` is of type `NotifyCallback` so that the client can pass a reference to itself to the server. The server can maintain this reference in a list of clients who should be notified of events of interest.

The operation `unregisterClient()` tells the server that the client is no longer interested in receiving callbacks. The server can remove the client from its list of interested clients.

Generating Skeleton Code

As in the case of creating a server, you should use the Type Store Manager tool to create your skeleton code for your callback objects. This will ensure that your interfaces have the correct parameters in the correct order and so on. Refer to "Development Support Tools" on page 121 for more details.

Writing a Client

To write a client, you must implement the `NotifyCallback` interface for the client objects. You can use the skeleton code generated by the Type Store Manager tool as a template, as if the client were a CORBA server.

Visual Basic

```
Dim clientObj as New NotifyCallback

Dim ObjFactory As Object
Set ObjFactory = CreateObject("CORBA.Factory")
...
Dim serverObj as clientBridge.DIRegisterCallback
Set serverObj = ObjFactory.GetObject(
                  "RegisterCallback:callbackSrv:h")
serverObj.registerClient clientObj
... 'Your code goes here
Public Sub notifyClient(Optional ByRef IT_Ex As Variant)

End Sub
...
```

PowerBuilder

```
OleObject NotifyCallback
NotifyCallback = CREATE OleObject
NotifyCallback.ConnectToNewObject("serverImplObject")
```

```
OleObject ObjFactory
ObjFactory = CREATE OleObject

serverObj = CREATE OleObject
serverObj = ObjFactory.GetObject
            ("RegisterCallback:callbackServer:h")
serverObj.resolve(NotifyCallback)
```

C++ COM

```
ICallBack *pIF = NULL;
...

hr = CoCreateInstanceEx (IID_ICORBAFactory, NULL, ctx, NULL, 1,
                         &mqi);
CheckHRESULT("CoCreateInstanceEx()", hr, FALSE);

pCORBAFact = (ICORBAFactory*)mqi.pItf;

// connect to the target CORBA server
memset(szMarkerServerHost, '\0',128);
sprintf(szMarkerServerHost, "CallBack:callback:%s", hostname);
hr = pCORBAFact->GetObject(szMarkerServerHost,&pUnk);
if(!CheckErrInfo(hr, pCORBAFact, IID_ICORBAFactory))
{
   pCORBAFact->Release();
   return;
}
pCORBAFact->Release();

hr = pUnk->QueryInterface(IID_ICallBack, (PPVOID)&pIF);
if(!CheckErrInfo(hr, pUnk, IID_ICallBack))
{
   pUnk->Release();
   return;
}
pUnk->Release();

// Create our implementation for the callback object
ICOMCallBackImpl * poImpl = ICOMCallBackImpl::Create();
```

```
// make the call to the server passing in our object
pIF->Register(poImpl);
// wait around until we explicitly quit for the none console
// application
StartCOMServerLOOP(10000);
poImpl->Release();
```

The client creates an implementation object, `clientObj`, of type `NotifyCallback`.

It binds to an object of type `RegisterCallback` in the server. At this point, the client holds an implementation object for type `NotifyCallback` and a reference to an Automation View (`serverObj`) for an object of type `RegisterCallback`.

To allow the server to invoke operations on the `NotifyCallback` object, the client must pass a reference to its implementation object to the server. Thus, the client calls the operation `registerClient()` on the `serverObj` View and passes it a reference to its implementation object `clientObj`.

Because it implements an interface, the client is acting as a server. However, the client does not have to register its implementation object with the bridge and it is not registered in the Implementation Repository. Therefore, the server will not be able to bind to the client's implementation object.

Implementing the Server

The server application can be coded as a normal OrbixCOMet server as described in "Implementing CORBA Servers" on page 173.

In particular, you must provide an implementation class for the `RegisterCallback` interface using the skeleton code generated by the Type Store Manager tool as a template.

The implementation of the `registerClient()` operation receives an object reference from the client. When this object reference enters the server address space, a COM/Automation view for the client's `NotifyCallback` object is created in the server's bridge. The server uses this view to call back to the client. The implementation of `RegisterClient()` should store the reference to the view for this purpose.

Visual Basic

```
Public Sub registerClient(ByVal var_client As Object,
   Optional ByRef IT_Ex As Variant)
   // Store reference to var_client
   ...
End Sub

Public Sub unregisterClient(ByVal var_client As Object,
   Optional ByRef IT_Ex As Variant)
   // Remove reference to var_client
   ...
End Sub
```

PowerBuilder

```
// Create two functions passing a user object
registerClient (...
unregisterClient (...
```

COM C++

```
void CallBack_i::Register (IClientObject * obj)
{
   cout << "in Server, about to call back to client" << endl;

   // Register reference
   ...
}

void CallBack_i::UnRegister (IClientObject * obj)
{
   cout << "in Server, about to call back to client" << endl;

   // Remove the reference
   ...
}
```

Invoking the Operation

After the view is created in the server address space, the server can invoke the operation `notifyClient()` on the view. For example, the server might initiate this call in response to an incoming event (such as a request to make a deposit or withdrawal to a bank account).

Visual Basic

```
Dim callbackObj as serverBridge.DINotifyCallback

' Get reference to the client from server's stored data:
Set callbackObj = ...

' Call back to client
callbackObj.notifyClient
```

PowerBuilder

```
// get the reference to the client from the server's stored data
OleObject CallbackObj
...
CallbackObj.ConnectToNewObject(...)
...
CallbackObj.notify()
```

COM C++

```
try
{
   obj->op1("This is the server calling");
}
catch (CORBA(SystemException) & oEx)
{
   cout << oEx;
}
catch (...)
{
   cout << "Unknown exception" << endl;
}
cout << "in Server, back from client" << endl;
```

The callback can be sent directly to the client. The callback does not need to be routed through an Orbix daemon, so the client does not have to be registered in the Orbix Implementation Repository. Therefore, the server will not be able to bind to the client's implementation object.

Registering the Callback Object Server

Finally, the server instantiates an object of type `RegisterCallback`, registers it with the bridge and activates itself as a CORBA server.

Visual Basic

```
Dim serverObj As New RegisterCallback
Dim serverAPI as Object

...
Set serverAPI = CreateObject("serverBridge.ITServerAPI")
serverAPI.SetObjectImpl("RegisterCallback", "", serverObj)
serverAPI.Activate("CallbackServer")
```

The server should be registered in the Implementation Repository with the name specified in the `Activate()` call.

PowerBuilder

```
// Get a reference to the ITServerAPI object
OleObject serverAPI
serverAPI = CREATE OleObject
serverAPI.ConnectToNewObject("serverBridge.ITServerAPI")

// Instantiate a Bank object.
// You first need to use PBGENREG.EXE to expose the
// object with the ProgID 'CallbackSrv.CallbackImplObject'
OleObject Obj
Obj = CREATE OleObject
Obj.ConnectToNewObject("CallbackSrv.CallbackImplObject")

// Register Obj with the Bridge.
serverAPI.setObjectImpl("RegisterCallback", "", Obj)
```

```
// Activate the server so that bankObj
// can receive incoming calls from CORBA clients.
serverAPI.Activate("CallbackServer")

//Deactivate the server when finished.
serverAPI.Deactivate("CallbackServer")
```

COM C++

```
CallBack_i* m_pObj=new CallBack_i();

IOrbixORBObject * poOrb=0;
hr = CoCreateInstance(IID_IOrbixORBObject, NULL, ctx,
                      IID_IOrbixORBObject, (void**)&poOrb);
CheckHRESULT("Connecting to ORB", hr, FALSE);

IOrbixServerAPI * poAPI=0;

hr = poOrb->GetServerAPI(&poAPI);
CheckHRESULT("getting Server API", hr, FALSE);
poOrb->Release();

// register our COM object as the CORBA interface 'ClientObject'
poAPI->setObjectImpl("RegisterCallback", "", m_pObj);

poAPI->Activate("CallbackServer");

poAPI->Release();

delete m_pObj;
```

15

Managing the Type Store

"Development Support Tools" on page 121 has already described the tools you can use to populate and remove information from the OrbixCOMet type store in order to create IDL files, type libraries, smart proxy DLLs and server stub code. This chapter describes the general workings of the type store and explains how you can prime it in order to optimise performance at application runtime.

OMG IDL files are text files that contain CORBA Interface Definition Language (OMG IDL). MIDL files are text files that contain Microsoft Interface Definition Language (MIDL). The OMG IDL compiler is used to compile OMG IDL into skeleton source code for client and server applications. The MIDL compiler (`midl.exe`) is used to compile MIDL into skeleton source code for client and server proxies, stubs, and so on. In CORBA, you can use the `putidl` command to store OMG IDL in the Interface Repository in binary format. In COM, when you run `midl.exe`, it automatically creates a type library that can store MIDL in binary format. (Note that `midl.exe` is now replacing `mktypelib.exe`.)

The OrbixCOMet type store holds a cache of OMG IDL and MIDL type information in an ORB-neutral binary format called metadata. This type information is then used by the dynamic bridge at application runtime.

> **Note:** Type libraries are not required if you are only using straight `IDispatch` interfaces. In this case, OrbixCOMet automatically puts type information into the type store when it does a lookup using `GetObject`, as in the following example:
> ```
> mod::CorbaSrv
> 'Visual Basic
> srvobj = factory.GetObject ("mod/CorbaSrv")
> ```

The Caching Mechanism

It is recognised that a possible performance bottleneck might result at application runtime when the Interface Repository is being contacted to obtain OMG IDL type information, and the type library is being contacted to obtain MIDL type information. This is because every query might involve multiple remote invocations. To avoid this, OrbixCOMet uses a memory and disk cache of type information. This means the Interface Repository only has to be contacted once per OMG IDL definition, and the type library only has to be contacted once per MIDL definition. When a new OMG IDL type is encountered, OrbixCOMet queries the Interface Repository for its type information. When a new MIDL type is encountered, OrbixCOMet queries the type library for its type information.

The OrbixCOMet Type Store Manager tool (`TypeMan`) converts all OMG IDL and MIDL type information into metadata, and caches it in memory. At runtime, when OrbixCOMet is marshalling information, and method invocations are being made, the type store cache holds the required type information in memory. The type information is handled on a first in-first out (FIFO) basis in the memory cache. This means the most recently accessed information becomes the most recent in the queue. On exiting the application process (or when the memory cache size limit has been reached), new entries in the memory cache are written out to persistent storage and are reloaded on the next run of an OrbixCOMet application.

The memory cache and disk cache are quite separate. Initially, on starting up, the memory cache is primed with the most recently accessed elements of the disk cache. (The number of elements in the memory cache depends on the configuration settings, as described in "OrbixCOMet Configuration" on

page 219.) When lookups are performed, if the required type information is not already in the memory cache, the `TypeMan` tool pulls it out of the disk cache. If the required type information is not already in the memory or disk cache, the `TypeMan` tool pulls it out of the Interface Repository or type library (depending on whether it is an OMG IDL or MIDL type definition). The related type information then becomes the most recent item in the queue in the type store memory cache.

Type Store Configuration Issues

Refer to "OrbixCOMet Configuration" on page 219 for a description of the keys that are of interest to OrbixCOMet configuration, and their associated default values.

Inserting Information into the Type Store

"Development Support Tools" on page 121 has already explained how to insert type information into the OrbixCOMet type store. To recap, you can use the GUI tool (**OrbixCOMet tools** screen) or the command line utilities.

Remember that if you use the command line tools and GUI tool simultaneously, any changes you make with the command line tools will not appear automatically on the **OrbixCOMet tools** screen. In this case, you would have to click the **Refresh Display** button on the **OrbixCOMet tools** screen to update the list of type store contents displayed.

Removing the Contents of the Type Store

"Development Support Tools" on page 121 has already explained how to delete the contents of the type store. To recap, you can use any of the following three ways:

- Click the **Delete TypeStore** button on the **OrbixCOMet tools** screen. (Refer to "Development Support Tools" on page 121 for more details.)

213

- Type `typeman -w` at the command line. (This is the "wipe" command in typeman.exe.)

- Type `del c:\temp\typeman.*` at the command line. (This is assuming the data files are in the default location `c:\temp`.) If the data files are not in `c:\temp`, you can find out where they are stored by checking the setting for `COMet.TypeMan.TYPEMAN_CACHE_FILE` in the `\iona\config\orbixcometx.x.cfg` file (where *x.x* represents the version number). (Refer to "OrbixCOMet Configuration" on page 219 for more details.)

Priming the Bridge Cache

While it is possible to allow the type store cache to obtain its information on an as-needed basis (that is, at runtime on the first run of an application), it is possible that users might want to prime the cache before executing their program. If this is done, there will be a notable performance difference on the first run of an application.

Note: This is only an issue for the first run of an application that is using previously unseen OMG IDL or MIDL types. After OrbixCOMet has obtained the type information from the Interface Repository or type library, either through cache priming or during the first run of an application, all subsequent queries will be satisfied by the cache.

To allow users to prime the cache, OrbixCOMet is shipped with a command line utility, `TypeMan.exe`, found in the `%ORBIXCOMET%\bin` directory (where `%ORBIXCOMET%` represents the name of the installation directory you have chosen). The help message for this command is as follows:

```
[c:\iona\comet\bin]typeman /?

TypeMan [filename | -e name] [-r] [-i] [-c[n][u]] [-b] [-h]
        [-l[+]] [-v[s[i] method]] [-z]

        filename: name of input text file.
        -e:   lookup Entry (name or uuid)
```

```
        -r:   Resolve all references (use to generate static
              bridge compatible names for CORBA sequences)
        -i:   always connect to IFR.
        -c[n][u]: list disk cache Contents, n: Natural order,
                  u: display uuid.
        -v[s[i] method]: log v-table for interface/struct
              [s:search for method]
          [i]: Ignore case. use -v with -e option.
        -b:   Log mem cache hash-table bucket sizes.
        -h:   Log cache hits/misses.
        -z:   Log mem cache siZe after each addition.
        -l[+]: log TS basic contents ['+' shows new's/delete's]
        -w:   Delete (wipe) cache contents.
        -u:   (Unattended) continue without pause at program
              exit.
        -?2:  Priming input file format info.
```

The cache file for interface type information is determined by the `Comet.TypeMan.TYPEMAN_CACHE_FILE` entry in the `\iona\config\orbixcomet`*x.x*`.cfg` file (where *x.x* represents the version number). A typical value would be `C:\temp\typeman._dc`. (For more details about OrbixCOMet configuration entries, refer to "OrbixCOMet Configuration" on page 219.)

If you wish, you can use one single text file to contain all Interface Repository and type library entries, and this can then be used to rebuild the cache. In the text file, you can comment out a line by putting `//` at the start of it. If you insert a double blank line, it will be treated as the end of the file.

Note: The Type Store Manager tool (`typeman`) does not currently support selective removal of type store entries. However, use of this file format for priming the cache is a useful aid in managing the cache. Simply delete the typeman cache files `typeman.edc`, `typeman.idc`, `typeman.map`, `typeman._dc` (usually in `\temp`) using the `del c:\temp typeman.*` command or the `typeman -w` command. If you wish, you can then re-prime the cache using a modified list of types.

Prime from the Interface Repository

In this case, the command expects either an individual OMG IDL typename or a text file that lists all the fully scoped OMG IDL interface names with which the cache is to be primed.

For example, assume you want to prime the cache with the type information for interfaces `grid`, `bank`, `account` and `foobar::myInterface`. You would edit a text file (called, for example, `prime.txt`) where each interface name is placed on a separate line as follows:

```
grid
bank
account
foobar::myInterface
```

Then you would run the following command:

```
[c:\] typeman  prime.txt
```

Assuming that the IDL for the interfaces listed in `prime.txt` is installed in the Interface Repository, the cache would be primed with this type information.

Alternatively, you can supply an individual typename on the command line. For example, to prime the cache for interface `grid` you would type the following:

```
[c:\] typeman -e grid
```

Assuming that the Interface Repository has been updated with the OMG IDL for interface `grid`, the OrbixCOMet disk cache would be primed with metadata for the `grid` interface. Thereafter, no interaction with the Interface Repository occurs for applications using that interface.

Prime from Type Libraries

In this case, the command expects either an individual type library name (including its full path) or a text file that contains a list of type library names (including their full paths) with which the cache is to be primed.

For example, assume you want to prime the cache with the type information for type libraries `grid.tlb`, `bank.tlb`, `account.tlb` and `foobar.tlb`, which are all held, for example, in c:\temp. You would edit a text file (called, for example, `prime.txt`) where each type library path is placed on a separate line as follows:

```
c:\temp\grid.tlb
```

```
c:\temp\bank.tlb
c:\temp\account.tlb
c:\temp\foobar.tlb
```

Then you would run the following command:

```
[c:\] typeman  prime.txt
```

The cache would then be primed with this type information.

Alternatively, you can supply an individual type library pathname on the command line. For example, to prime the cache for type library grid.tlb held in c:\temp, type the following:

```
[c:\] typeman -e c:\temp\grid.tlb
```

The OrbixCOMet disk cache would then be primed with metadata for that type library.

Dumping Contents of the Cache

The typeman tool is also a useful diagnostic utility in that it allows for dumping contents of the cache. For example, the following command would print the methods of interface grid in alphabetical order and also in vtable order (this order is determined by the *COM/CORBA Interworking* document):

```
[c:\] typeman -e grid -v

MD5/Name or IFR look up: grid


        Name sorted                    V-table  DispId  Offset


                get                        get      1       0
        height  get                        set      2       1
                set                     height      3       2
         width  get                      width      4       3
```

Note: The second column in this example denotes attribute get operations. The absence of height set and width set implies that these are read-only attributes.

16

OrbixCOMet Configuration

This chapter describes the keys that are of interest to OrbixCOMet configuration, and their associated default values. It includes details of configuration entries that are either specific to OrbixCOMet or common to multiple IONA products.

OrbixCOMet Keys

This section describes the configuration variables that are specific to OrbixCOMet. They are held in the `\iona\config\orbixcomet`*x.x*`.cfg` file (where *x.x* represents the version number). They are declared within various scopes within the scope `COMet{...}`. As shown in this section, you can also use the prefix `COMet.`*Scope name*`.` to refer to individual entries in this file.

Key   `COMet.Licensing.IT_KEY="134217728-1476395008-134217728-134217728-`
                              `2147483710"`

Description   This denotes either an evaluation or full licence key for using the product. When you install OrbixCOMet you are asked for a valid OrbixCOMet licence code that should be included in your product package. If you supply an invalid code, the installation defaults to a 21-day evaluation licence. You can contact `shipping@iona.com` if you need to obtain a new licence code. When you receive a full licence code you should enter it in this key.

Key   COMet.Config.COMET_HANDLER_LOCATION="COMet\Handlers"

Description   This key is used to specify handler DLLs for smart proxies, filters, transformers, I/O callbacks and so on (for example, calls to configure Orbix dynamically). It specifies a key stored in HKEY_CLASSES_ROOT that stores where these DLLs are located. The default value is "COMet*x.x*\Handler DLLs", where *x.x* represents the version number. It is placed in HKEY_CLASSES_ROOT so that users without administrative privileges can read and modify the values. The values would look like the following:

```
[HKEY_CLASSES_ROOT\COMet\x.x\Handler DLLs]
grid="c:\foo\bar.dll"
bank="c:\foo\bar2.dll"
```

Key   COMet.Config.COMET_DAEMON_HOST=""

Description   This specifies the host where the Orbix daemon is running. The locator on this host will be used in calls to GetObject(). This is used by GetObject() when the Marker:Server:Host format is being used (but not when stringified IORs are being used, because this information is contained in the IOR). This allows IIOP-on-the-wire to be used by OrbixCOMet, without a local daemon. The locator on the host specified by COMET_DAEMON_HOST will be used by OrbixCOMet to launch the server.

Key   COMet.Config.COMET_DEFAULT_PROTOCOL="IIOP"

Description   This is the default protocol used by OrbixCOMet to connect to a CORBA server and the IFR. Valid settings are:

    "IIOP"   Internet Inter-ORB Protocol

    "POOP"   Plain Old Orbix Protocol

Key   COMet.Config.COMET_ROOT="c:\iona\comet\bin"

Description   This is the full pathname of the OrbixCOMet installation directory. This is used by the Uninstall package to indicate where OrbixCOMet is located.

Key	`Comet.Config.IT_DAEMON_PROTOCOL="IIOP"`
Description	This is used by OrbixCOMet to connect to the daemon. The daemon then launches the server and returns a `COMET_DEFAULT_PROTOCOL` type connection between OrbixCOMet and the server. In general, this setting should be identical to the `COMET_DEFAULT_PROTOCOL` setting. Valid settings are:

`"IIOP"`	Internet Inter-ORB Protocol
`"POOP"`	Plain Old Orbix Protocol

Key	`COMet.Config.COMET_SHUTDOWN_POLICY="implicit"`
Description	The valid settings for this key are as follows:

`"implicit"`	This is the default setting. It means OrbixCOMet will shut down the first time `DllCanUnloadNow` is about to return 'yes'.
`"explicit"`	This means you must make a call to `ORB::ShutDown()` to force OrbixCOMet to shut down.
`"Disabled"`	This means OrbixComet will not shut down the ORB when it thinks it is about to unload. That is, the DLL is not unloaded when `DllCanUnloadNow` is called by the COM runtime. Visual Basic and Internet Explorer do this to cache the DLLs.
	A problem will arise, however, if the DLL is re-used because Orbix has already been shut down.
`"atExit"`	This means that the OrbixCOMet bridge will only shut down at process exit time. This is the recommended setting when running inside the Visual Basic development environment.

Key	`COMet.Config.COMET_UPDATE_LEVEL="3-0b4-00"`
Description	This includes information about the version of OrbixCOMet, the patch level of Orbix against which OrbixCOMet has been built, and the patch level of OrbixCOMet. You should quote this value whenever posting to `support@iona.com` or to the comet-users newsgroup.

Key   `COMet.Mapping.UseSAFEARRAYMapping="yes"`

Description The Automation mapping for OMG IDL sequences and arrays is to Automation compatible `SAFEARRAY`s as described in the *COM/CORBA Interworking* document. Existing code from the Orbix Desktop product used the alternative mapping to collections. This mapping to collections has been deprecated in the current specification, but it is supported in OrbixCOMet for existing users. To specify that the collections mapping should be used, you should set this value to "no".

Key   `COMet.Mapping.KEYWORDS="grid, DialogBox, bar, Foobar, height"`

Description This allows you to enter a list of words that you want to be prefixed in order to avoid clashes when using `ts2idl` to generate IDL.

Key   `COMet.TypeMan.TYPEMAN_CACHE_FILE="C:\TEMP\typeman._dc"`

Description OrbixCOMet uses a memory and disk cache for efficient access of type information. This entry specifies the name and location of the file used.

Key   `COMet.TypeMan.TYPEMAN_DISK_CACHE_SIZE="2000"`

`COMet.TypeMan.TYPEMAN_MEM_CACHE_SIZE="250"`

Description These two keys specify the maximum number of entries allowed in the disk cache/memory cache. When these values are exceeded, entries can be flushed from the cache. The nature of the applications using the bridge will affect the values these keys should have. However, as a general rule, the disk cache size should be about eight to ten times greater than the the memory cache. Furthermore, to avoid unnecessary swapping into and out from disk, you should ensure the memory cache size is no smaller than 100. An "entry" in this case corresponds to a user-defined type. For example, a union defined in OMG IDL would result in one entry in the cache. An interface containing the definition of a structure would result in two entries. A good rule of thumb is that 1000 cache entries (given a representative cross section of user-defined types) would correspond to approximately 2 MB of disk space. Therefore, the default disk cache size of 2000 allows for a maximum disk cache file size of approximately 4 MB. When the cache is primed with type libraries for DCOM servers, the size could be considerably larger. It depends on the size of the type libraries, and this can vary considerably. Typically, a primed type library will be over three times the size of the original type library because the information is stored in a format that optimises speed.

Key	COMet.TypeMan.TYPEMAN_IFR_HOST=""
Description	To allow for ease of deployment and for an easy upgrade path (for example, when new interfaces are exposed by a server implementor), a common requirement is to use a central Interface Repository (IFR). This raises the need to get OrbixCOMet to use an IFR on a machine other than that on which OrbixCOMet is installed. If it is preferable that an IFR on another machine should be used, simply create an entry in the orbix.hst file for use by the locator and specify the host that should be contacted. For example, to use the IFR on the machine advice.iona.com, the orbix.hst file would look like: IFR:advice.iona.com:
	However, use of the Orbix locator requires an orbixd on the local machine. This might not always be the case, and OrbixCOMet allows for this by providing the TYPEMAN_IFR_HOST configuration file entry that can be used to specify the host on which the IFR should be contacted. The value for this key should specify the host in question.
Key	COMet.TypeMan.TYPEMAN_IFR_IOR_FILENAME=""
Description	This key only needs to be set if you are using the stand-alone COMetIFR that ships with OrbixCOMet. This is the full pathname to the file containing the stringified version of the COMetIFR Interoperable Object Reference (IOR).
Key	Comet.TypeMan.TYPEMAN_IFR_NS_NAME=""
Description	This is the name of the IFR in the Naming Service. This is needed if you are using the Naming Service to resolve the IFR. You should register an IOR for the IFR in the Naming Service under a compound name. This key should contain that compound name.
Key	Comet.TypeMan.TYPEMAN_READONLY="no"
Description	This key determines whether readonly mode is to be used for the type store.
Key	Comet.Services.NameService=""
Description	This is the full pathname to the file containing the IOR for the Naming Service. This is needed if you are using the Naming Service to resolve the IFR.

223

Common Keys

This section describes the configuration variables that are common to multiple IONA products, including OrbixCOMet. They are held in the `\iona\config\common.cfg` file and are declared within the scope `Common{...}`. As shown in this section, you can also use the prefix `Common.` to refer to individual entries in this file.

Key   `Common.IT_DAEMON_PORT="1570"`

Description   This is the TCP port number that OrbixCOMet will use to contact an Orbix daemon.

Key   `Common.IT_DAEMON_SERVER_BASE="1570"`

Description   This is the starting port number for servers launched by the Orbix daemon. This key must be set if you are using the stand-alone COMetIFR. This location is used by the COMetIFR to keep repository records.

Key   `Common.IT_IMP_REP_PATH=cfg_dir + "Repositories\ImpRep"`

Description   This is the full pathname of the Implementation Repository directory.

Key   `Common.IT_INT_REP_PATH=cfg_dir + "Repositories\IFR"`

Description   This is the full pathname of the Interface Repository directory.

Key   `Common.IT_LOCATOR_PATH=cfg_dir`

Description   This is the full pathname of the directory holding the locator files.

Key   `Common.IT_LOCAL_DOMAIN=""`

Description   This is the name of the local Internet domain. This should be the same for both the client and server sides. An empty value is a valid value.

Key   `Common.IT_JAVA_INTERPRETER="c:\iona\bin\jre.exe"`

Description   This is the full pathname to the JRE binary executable that installs with Orbix.

Key   `Common.IT_DEFAULT_CLASSPATH=cfg_dir + "C:\IONA\bin\bongo.zip;`
`C:\IONA\bin\marimba.zip;C:\IONA\bin\NSclasses.zip;`
`C:\IONA\bin\utils.zip;C:\IONA\bin\rt.jar;C:\IONA\bin\orbixweb.jar`
`;C:\IONA\Tools\NamingServiceGUI\NSGUI.jar"`

Description   This the default classpath to be used when Java servers are automatically launched by the daemon.

> **Note:** After installation, the `common.cfg` file provides default settings for the main environment variables required. You can change these default settings by manually editing the configuration file, or by using the Configuration Explorer, or by setting a variable in the user environment. If an environment variable is set, it takes precedence over the value set in the configuration file. Environment variables are not scoped with a `Common.` prefix.

Orbix Keys

This section describes configuration variables that are common to both Orbix and OrbixCOMet. They are held in the `\iona\config\orbix3.cfg` file and are declared within the scope `Orbix{...}`. By default, the configuration variables in this file are scoped with the `Orbix.` prefix.

Key   `Orbix.IT_ERRORS=cfg_dir + "ErrorMsgs"`

Description   This is the pathname for the error messages file.

Key   `Orbix.IT_CONNECT_ATTEMPTS="10"`

Description   This is the maximum number of retries that Orbix makes to connect to a server.

17

Deploying your OrbixCOMet Application

This chapter describes the various models you can adopt when deploying an application you have built using OrbixCOMet. It also describes the steps you must follow to deploy an OrbixCOMet application.

Deployment Models

OrbixCOMet supports communication using either the DCOM protocol or the CORBA IIOP protocol. Therefore, when it comes to deploying your applications, you have a great degree of flexibility in terms of how you might choose to install OrbixCOMet. For example, you can install it on each of your client machines or on one central machine that is separate from your clients. Alternatively, you could install it on your server machine.

Internet Deployment

When it comes to deploying your OrbixCOMet application on the Internet you have two options to choose from:

- Download the entire bridge onto the client machine.
- Download only the DLLs and leave the bridge on the Internet server machine.

Using OrbixCOMet with Internet Explorer

Before reading this section, refer to the section "DCOM On-the-Wire with OrbixCOMet" on page 232.

The DLL `CCIExWrapper.DLL` contains a control for wrapping `CoCreateInstanceEx()` that can be referenced in HTML files (using the `OBJECT` tag) supplying attribute values that specify the object name, location, type, and so on. The `CODEBASE` attribute identifies the code base for the object by supplying a URL. (The machine name might need to be modified in the HTML file before the demonstration will work.) The `CLASS ID` attribute identifies the object implementation. Its syntax is `CLSID:class-identifier` for registered ActiveX controls. For example:

```
<OBJECT ID="bridge" <
 CLASSID="CLSID:3DA5B85F-F2FC-11D0-8D97-0060970557AC"
 # change this to reflect the location of CCIExWrapper.dll on your
   machine
 CODEBASE="\\INSTALLATION_MACHINE_NAME\IONA\COMet\bin\CIExWrapper.dll"
>
</OBJECT>
```

When the HTML file is first downloaded, the control is also retrieved and registers itself on your machine (subject to you agreeing, of course). This allows use of OrbixCOMet from client machines with zero configuration effort required on the client's part. The only requirement is that the developer configures OrbixCOMet on the server side with respect to type information, access permissions, and so on, and places a HTML file on a server. This HTML file can contain VBScript or JavaScript for calling methods on the remote CORBA objects. (DCOM will be used on the wire.) For example, the following VBScript is used for connecting to the `grid` object on machine "advice.iona.com" and obtaining the height and the width of the grid:

```
<SCRIPT LANGUAGE="VBScript">
<!--

Dim Grid
Dim fact

Sub btnConnect_Onclick
    lblStatus.Value = "Connecting..."

  # DCOM on the wire...
    # the parameter should be the name of the
    #  machine where the bridge is located
Set fact = bridge.IT_CreateRemoteFactory("advice.iona.com")

# IIOP on the wire
Set fact = CreateObject("CORBA.Factory")

    Set Grid = fact.GetObject("grid")
    lblStatus.Value = "Obtaining dimensions..."
    sleWidth.Value  = Grid.width
    sleHeight.Value = Grid.height
    lblStatus.Value = "Connected..."
End Sub

-->
</SCRIPT>
```

The full example can be found in:

```
%ORBIXCOMET%\demos\ie\grid\griddemo.htm
```

(where %ORBIXCOMET% represents the installation directory you have chosen).

In order to use it you must set your Internet Explorer security settings to "Medium" as follows:

1. Click **Options** on the View menu.
2. Select the Security tab and click Safety level.
3. Set the security to medium.
4. Click OK to close the options dialog box.

A security setting of medium means you will be prompted whenever executable content is being downloaded. That is all you need to do. You do not need to have Orbix installed. You can now open the file:

`\\`*INSTALLATION_MACHINE_NAME*`\OrbixCOMet\demos\iexplorer\griddemo.h`

You will need to edit the file to specify the name of the machine that you want to be contacted when the demonstration is downloaded.

The two changes are on the following lines in `griddemo.htm`:

```
CODEBASE="\\MYMACHINENAME\IONA\COMet\bin\CIExWrapper.dll"
```

and

```
Set fact = bridge.IT_CreateInstanceEx("{A8B553C5-3B72-11CF-BBFC-
444553540000}", "MACHINENAME")
```

or

```
Set fact = bridge.IT_CreateRemoteFactory("MACHINENAME")
```

`IT_CreateInstanceEx` in the preceding example takes a stringified `CLSID` as the first parameter, which in this case is the `CLSID` for the CORBA Factory, whereas `IT_CreateRemoteFactory` has the `CLSID` for `CORBA.Factory` hard-coded in its implementations.

When these changes have been made, this file can be accessed from any Windows NT 4.0 or Windows 95 machine with Internet Explorer. Neither Orbix nor OrbixCOMet are required on the client side for this demonstration to work.

The first time the page is accessed, a dialog box opens to tell you that unsigned executable content is being downloaded. That is acceptable in this case. You should be presented with a simple GUI, somewhat similar to the Visual Basic or PowerBuilder GUI described earlier. To use the demonstration you can click Connect, fill in x and y values for the grid, and then click Set/Get and so on. You can click Disconnect when you are finished.

Bridge on Each Client Machine

In this model, the OrbixCOMet runtime is installed on every machine. Clients communicate with servers using the CORBA communications protocol, IIOP, on the wire. Figure 17.1 provides an overview of this model.

The OrbixCOMet runtime is what is required to deploy an OrbixCOMet application. It requires considerably less disk space than an installation of OrbixCOMet on a development machine.
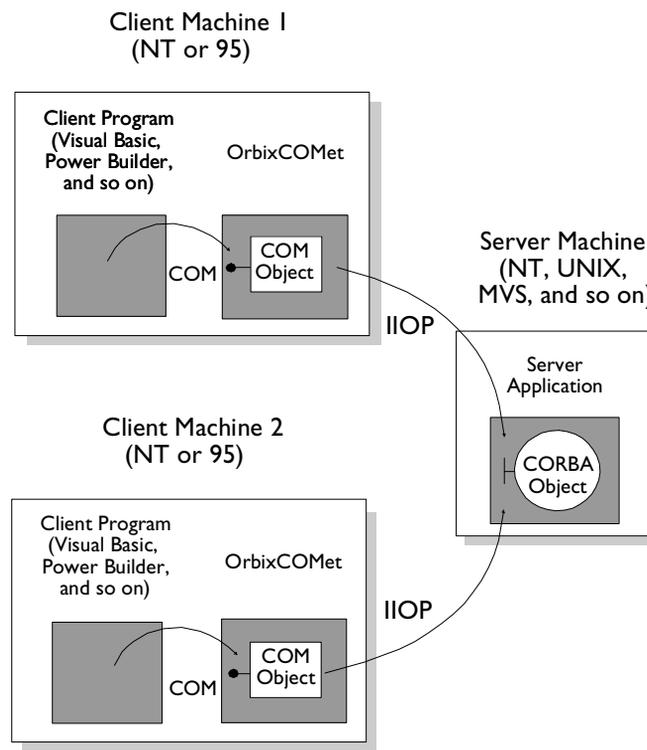

Figure 17.1: *Bridge on Each Client Machine*

DCOM On-the-Wire with OrbixCOMet

This section describes how OrbixCOMet can be used to write applications that launch the bridge out-of-process, either on the local machine or on a remote machine.

A DLL called `CCIExWrapper.dll` has been provided with your OrbixCOMet installation. This DLL essentially exposes the functionality of `CoCreateInstanceEx()` to PowerBuilder/Visual Basic and Delphi programmers. Use of this functionality allows programmers to specify the machine on which the OrbixCOMet bridge should be launched, thus allowing use of DCOM on-the-wire. Programmers can of course elect to use IIOP on-the-wire instead. Both configurations are equally easy to use from the client programmers' point of view. The decision about which one is to be used can be made at runtime. It is simply a matter of whether the bridge is launched as an in-process server, a local server or a remote server[1]. For example, consider the following Visual Basic code, where you use a check button (`inprocess`) to let the user decide whether to launch the bridge in-process (and hence talk IIOP on-the-wire) or out-of-process (and hence talk DCOM on-the-wire):

```
Private Sub ConnectBtn_Click()
On Error GoTo errortrap
    If inprocess.Value <> Checked Then
      Dim wrapper As Object
set wrapper = CreateObject("IT_CCIExWrap.IT_CCIExWrap.1")
       set objFactory =
wrapper.IT_CreateRemoteFactory(HostName.Text)
     set wrapper = Nothing
    Else
        Set objFactory = CreateObject("CORBA.Factory")
    End If
    inprocess.Enabled = False
```

---

1. Using DCOM on-the-wire to another machine requires that DCOM security issues are addressed. Security can be dealt with by using `DCOMCNFG.EXE`, programmatically via security API functions, or by a combination of the two approaches. To help in this regard, refer to the *OrbixCOMet Desktop Getting Started* book which provides details on some DCOM-only applications shipped with OrbixCOMet that can be used to experiment with configuring DCOM. However, a full treatment of COM security is outside the scope of this book. Refer to the COM security FAQ at `http://support.microsoft.com/support/kb/articles/q158/5/08.asp` for more details.

```
    Set srvObj = objFactory.GetObject("grid:grid:" &
HostName.Text)
StartBtn.Enabled = True
    ConnectBtn.Enabled = False
    Exit Sub
errortrap:
    MsgBox (Err.Description & ", in " & Err.Source)
End Sub
```

In the preceding example, the same hostname is supplied to the `GetObject` call and the `IT_CreateRemoteFactory` call. This is purely to keep the example simple. Remember that the hostname passed to `GetObject()`, as shown in the preceding example, specifies the host on which the CORBA server you wish to contact is registered. The hostname passed to `IT_CreateRemoteFactory` in the preceding example specifies the host on which you want to create an instance of the `CORBA.Factory` object (that is, the host (local or remote) on which you want to launch the bridge). In practice, the two hosts can be different. When `IT_CreateRemoteFactory()` is used as in the preceding example, the OrbixCOMet DLLs are hosted by a surrogate executable (`custsur.exe` found in `%ORBIXCOMET%\bin\`) on the local or remote host. Furthermore, the code in `CCIExWrapper.DLL` is completely independent of Orbix, and can therefore be used on dedicated DCOM client machines. This is of particular use when using OrbixCOMet with Internet Explorer. When a user accesses a given web page that references the wrapper object, the DLL is downloaded automatically to the client's machine.

Use of OrbixCOMet in this fashion requires zero configuration effort on the client's machine.

Surrogate

As already mentioned, when the bridge is launched out of process, the OrbixCOMet DLLs are hosted by a surrogate process (`%ORBIXCOMET%\bin\custsur.exe`)rather than the default surrogate `DLLHOST.exe`.

This is indicated by the following registry value that is set during installation:

```
HKEY_CLASSES_ROOT\AppID\{A8B553C5-3B72-11CF-BBFC-444553540000}
[DllSurrogate] = c:\iona\comet\bin\custsur.exe
```

233

Bridge Shared by Multiple Clients

In this model, OrbixCOMet is installed on one central machine that is separate from your clients. In this case, you only need to be able to create a remote instance of the CORBA Factory object on your client machines. This is normally done using the DCOM `CoCreateInstanceEx()` method. OrbixCOMet provides a simple wrapper for this function for any languages (such as Visual Basic Script or PowerBuilder) that do not directly support this DCOM call. Figure 17.2 provides an overview of this model.
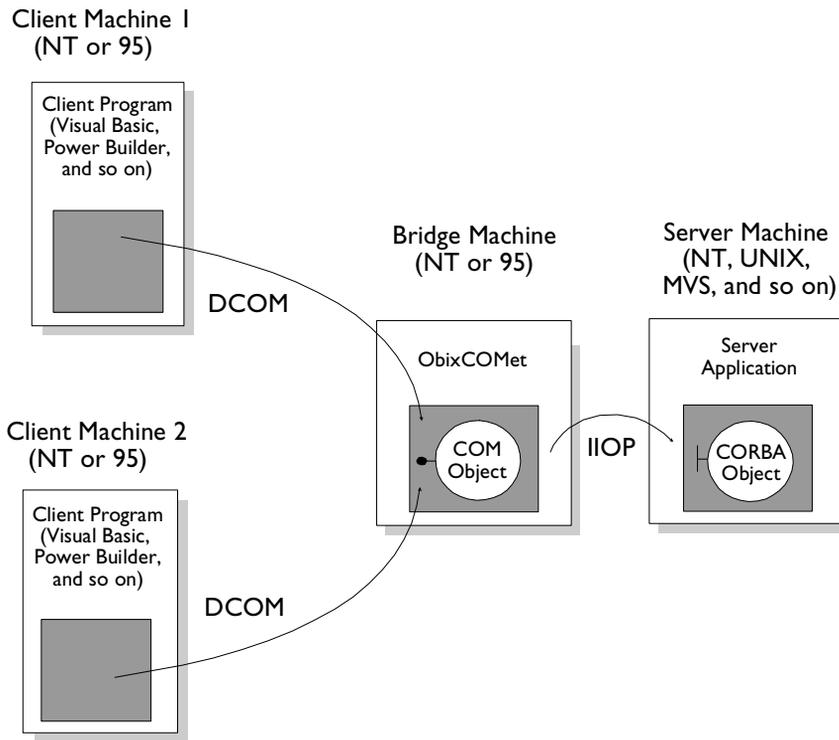

Figure 17.2: *Bridge Shared by Multiple Clients*

Bridge on Server Machine

In this model, OrbixCOMet is only installed on your server machine. In this case, your server should be running on Windows NT. The DCOM protocol is used to communicate across the network.

Note: The OrbixCOMet Server product would provide a better solution for implementing this deployment model because OrbixCOMet Desktop is designed specifically as a client-side product.

Deployment Steps

To install an application developed with OrbixCOMet you must install:

- Your application's runtime.
- The development language's runtime.
- The OrbixCOMet runtime.

You must also set the OrbixCOMet configuration variables required by your OrbixCOMet application at the location where the OrbixCOMet runtime is installed. These are described in "OrbixCOMet Configuration" on page 219.

Installing Your Application Runtime

The components associated with your OrbixCOMet application consist of:

- Your application executables.
- Any other DLLs needed by your application.

Installing the Development Language Runtime

The run-time requirements for your development language normally consist of:

- Run-time libraries (such as Visual Basic or PowerBuilder run-time libraries).

- Support libraries (such as Roguewave tools or extra libraries).

Details of the run-time requirements of your development language can be found in the documentation set for the specific development language.

Installing the OrbixCOMet Runtime

The run-time requirements of OrbixCOMet are the following DLLs:

- `CCIExWrapper.dll*`
- `custsur.exe`
- `DSIMM23C.dll`
- `ITCplx.dll*`
- `ITGeneric.dll*`
- `ITGLM23C.dll`
- `ITLicense.dll*`
- `ITMisc.dll`
- `ITM_M23C.dll`
- `ITOLM23C.dll`
- `ITStdObjs.dll*`
- `ITStdPS.dll*`
- `ITts2tlb.dll*`
- `ITUnknown.dll`
- `it_licps.dll*`
- `MSVCIRT.dll`

Note: The files marked with * must be explicitly registered with COM using `regsvr32 dllname`.

Depending on the version of OrbixCOMet you have installed, these DLLs can also be found in CAB file format at `installdir/redist/cometCAB` in your OrbixCOMet installation. Alternatively, you can find them on the OrbixCOMet web site at `www.iona.OrbixCOMet`.

Minimising Your Client-Side Footprint

In certain scenarios, OrbixCOMet allows you to deploy your client application without requiring any OrbixCOMet footprint on the client machine. This is normally referred to as a zero install configuration. This means you can use a centralised installation of the OrbixCOMet bridge for your clients that provides the deployment option of using DCOM as the wire protocol for communication.

Internet-Based Deployment

This deployment scenario allows you to download your client application over the Internet. Because OrbixCOMet supports the DCOM wire protocol, your Web-based clients can use DCOM to communicate with your installation of OrbixCOMet which will then forward the calls to the appropriate CORBA server.

If your scripting language supports the creation of a remote DCOM object, no OrbixCOMet runtime needs to be downloaded to that machine. At time of writing, the main scripting language is VB-Script which does not have this capability. For this reason, OrbixCOMet includes a simple wrapper DLL called `CCIExWrapper.DLL` that is a small (less than 20K) ActiveX that can be automatically downloaded with your web page and allows connection to a remote instance of the OrbixCOMet bridge. The samples provided in the `demo/IE` directory of your OrbixCOMet installation show how this can be achieved.

Automation-Based Clients

If you are developing client applications that use Automation late binding (that is, the `IDispatch` interface), you have the option to use DCOM-on-the-wire. In this scenario, you do not need any OrbixCOMet installation on your client machine provided the Automation language supports connection to a remote DCOM object (which in this case is the OrbixCOMet bridge). PowerBuilder 6.0 is currently the only main Automation client language that supports this. Visual Basic does not allow direct connection to a remote DCOM object.

As in the case of Internet-based deployment, you can use the `CCIExWrapper.DLL` supplied in your OrbixCOMet installation to limit the OrbixCOMet footprint to less than 20K.

If you are using early binding (that is, dual interfaces), you must include the Automation type library that you created with the `COMetCfg` tool or the `ts2tlb` command line tool. This will allow DCOM to use the standard type library `Marshaller` to manage the client side marshalling of your client.

COM-Based Clients

The normal DCOM deployment rules state that you need to deploy and register a proxy/stub DLL for all the COM interfaces your client uses. OrbixCOMet can automatically generate the MIDL definitions and makefile, which are needed to create this DLL, using the `COMetCfg` tool or the `ts2idl` command line tool.

If your COM client application uses the standard OrbixCOMet interfaces, such as `ICORBAFactory`, you must also include the OrbixCOMet proxy/stub DLL. This is called `ITStdPS.DLL` and is located in the `\bin` directory of your OrbixCOMet installation.

If your COM client uses pure DCOM calls, you must register forwarding entries in your client-side registry to indicate the OrbixCOMet CORBA location information for your CORBA server. The extra registry entries can be created by using the OrbixCOMet `SrvAlias.exe` tool. For deployment purposes, an additional tool `AliasSrv.exe` can be used to restore these settings during installation. See the `demo\COM\coCreate` demonstration for details. (Refer to "Replacing an Existing DCOM Server" on page 133 for more information about these tools.)

Part II

Programmer's Reference
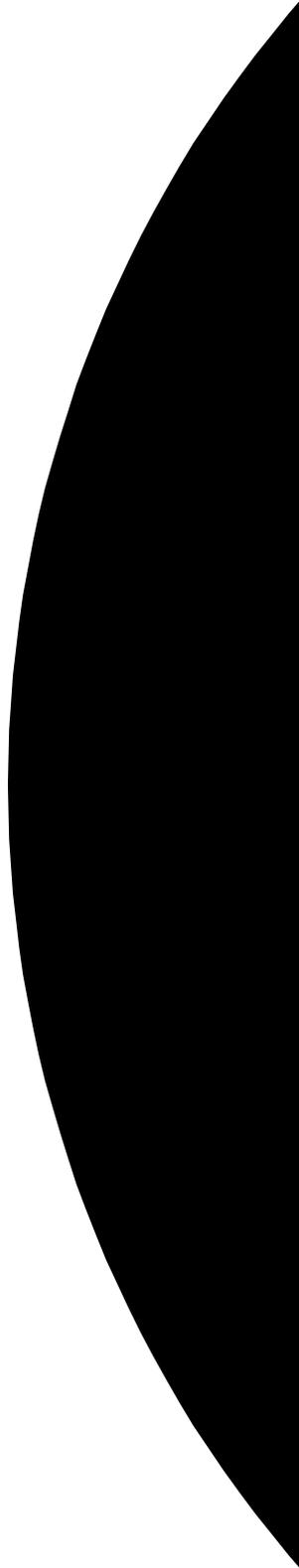
18

OrbixCOMet API

*This chapter describes the application programming interface (API)
to OrbixCOMet. The API is defined in MIDL. This chapter is divided
into two main sections. The first section describes the interface
entries for Automation. The second section describes the interface
entries for COM.*

Automation Interfaces

DIOrbixServerAPI

Note: You no longer need to use `DIOrbixServerAPI` to register your DCOM
objects with the bridge. (Refer to "Exposing DCOM Servers to CORBA
Clients" on page 163 for more details.) Because the use of this interface
is deprecated, it is mainly used for backwards compatibility purposes.

Synopsis
```
[oleautomation,dual,uuid(...)]
interface DIOrbixServerAPI : IDispatch
{
   HRESULT Activate ([in] BSTR cServerName,
                     [optional,in,out] VARIANT *IT_Ex);
   HRESULT Deactivate ([in] BSTR cServerName,
                       [optional,in,out] VARIANT *IT_Ex);
```

```
        HRESULT DispatchEvents ([optional,in,out] VARIANT *IT_Ex);
        HRESULT SetObjectImpl ([in] BSTR cIFace,
                               [in] BSTR cMarker,
                               [in] VARIANT poImpl,
                               [optional,in,out] VARIANT *IT_Ex);
        HRESULT ActivatePersistent ([optional,in,out] VARIANT *IT_Ex);

        HRESULT SetObjectImplPersistent ([in] BSTR cIFace,
                                         [in] BSTR cmarker,
                                         [in] BSTR cServer,
                                         [in] VARIANT poImpl,
                                         [in] BSTR cIORFileName,
                                         [optional,in,out] VARIANT
                                                              *IT_Ex);
};
```

Description   Bridges expose an Automation interface that allows them to act as CORBA servers. This interface can be obtained using the ProgID `ServerAPI`.

The Automation server should instantiate an object of this type and use it to control the Automation server's behaviour as a CORBA server.

Methods

`Activate()`   This activates an Automation server as a CORBA server using the `cServerName` parameter. This name should be the same name that is used to register the application in the Implementation Repository: that is, the name passed to `putit` or the server manager tool.

Once `Activate()` has been called, your server is ready to receive incoming requests from CORBA clients.

It is recommended that you register all your implementation objects using `SetObjectImpl()` before calling `Activate()`.

`Deactivate()`	This deactivates your application as a CORBA server. Once `Deactivate()` has been called, your application can no longer process incoming requests from CORBA clients.
	`cServerName` is the name of the CORBA server. The server must be registered with this name in the Orbix Implementation Repository.
`DispatchEvents()`	This causes any outstanding CORBA events to be dispatched to a client or server application for processing. It might be necessary to call this method in a client application if the client is asynchronously receiving callbacks from a server object. This will depend primarily on your development environment. Single-threaded development environments (for example, Visual Basic or PowerBuilder) require this to correctly dispatch incoming events.
`SetObjectImpl()`	This registers an Automation object with the bridge. The `poimpl` parameter identifies the Automation object and exposes it to the CORBA object space as the interface `CIFace` with the Orbix marker `cMarker`. (Markers are used to uniquely identify different instances of the same interface.) If no marker is passed, Orbix will automatically select a unique marker for the object. The marker names chosen by Orbix consist of a string composed entirely of decimal digits. To ensure that a new marker is different from any chosen by Orbix, do not use marker strings that consist entirely of digits. Marker names cannot contain a colon ":" or a null character.
`ActivatePersistent()`	This allows servers to be started without the need for `orbixd`.
`SetObjectImplPersistent()`	See `ActivatePersistent()`.

243

DCollection

Synopsis
```
[odl,uuid(...)]
interface DCollection : IDispatch {
    [propget] long Count();
    [propget,id(0)] VARIANT Item ([in] long index);
    [propput,id(0)] void Item ([in] long index, [in] VARIANT val);
    [id(-4)] IEnumVARIANT* _NewEnum();
    VARIANT getItem ([in] long index);
    void setItem ([in] long index, [in] VARIANT val);
};

[oleautomation,dual,uuid(...)]
interface DCollection : DIForeignComplexType {
    [propget,id(100)] HRESULT Count([retval,out] long* IT_retval);
    [propget,id(0)] HRESULT Item ([in] long index,
                          [retval,out] VARIANT* IT_retval);
    [propput,id(0)] HRESULT Item ([in] long index,
                          [in] VARIANT val);
    [id(101)] HRESULT getItem ([in] long index,
                          [retval,out] VARIANT* IT_retval);
    [id(102)] HRESULT setItem ([in] long index,
                          [in] VARIANT val);
    [id(-4)] HRESULT _NewEnum([out,retval] IUnknown** IT_retval);
};
```

Description   Automation interfaces that result from the translation of an OMG IDL sequence
support the interface `DCollection`.

Methods

`Count()`	This returns the number of items in a collection (that is, the number of items in the sequence).
`Item()`	This returns the collection member at the specified index (`propget`) or inserts an item into the collection at the specified index (`propput`).
`GetItem()`	This returns the collection member at the specified index.
`SetItem()`	This inserts an item into the collection at the specified index.

UUID   {E977F909-3B75-11CF-BBFC-444553540000}

Notes   Automation/CORBA compliant.

DICORBAAny

<table>
<tr><td>**Synopsis**</td><td>

```
typedef enum {
    tk_null, tk_void, tk_short, tk_long, tk_ushort,
    tk_ulong, tk_float, tk_double, tk_octet, tk_any,
    tk_typeCode, tk_principal, tk_objref, tk_struct,
    tk_union, tk_enum, tk_string, tk_sequence, tk_array,
    tk_alias, tk_except, tk_boolean, tk_char
} CORBATCKind;

[oleautomation,dual,uuid(...)]
interface DICORBAAny : DIForeignComplexType {
    [id(0),propget] HRESULT value([retval,out] VARIANT* IT_retval);
    [id(0),propput] HRESULT value([in] VARIANT val);
    [propget] HRESULT kind([retval,out] CORBATCKind* IT_retval);

//  tk_objref, tk_struct, tk_union, tk_alias, tk_except
    [propget] HRESULT id([retval,out] BSTR* IT_retval);
    [propget] HRESULT name([retval,out] BSTR* IT_retval);

//  tk_struct, tk_union, tk_enum, tk_except
    [propget] HRESULT member_count([retval,out] long* IT_retval);
    HRESULT member_name([in] long index,
                        [retval,out] BSTR* IT_retval);
    HRESULT member_type([in] long index,
                        [retval,out] VARIANT* IT_retval);

//  tk_union
    HRESULT member_label([in] long index,
                        [retval,out] VARIANT* IT_retval);
    [propget] HRESULT discriminator_type(
                        [retval,out] VARIANT* IT_retval);
    [propget] HRESULT default_index([retval,out] long* IT_retval);

//  tk_string, tk_array, tk_sequence
    [propget] HRESULT length([retval,out] long* IT_retval);

//  tk_array, tk_sequence, tk_alias
    [propget] HRESULT content_type(
                        [retval,out] VARIANT* IT_retval);
};
```

</td></tr>
</table>

Description The OMG IDL type `any` translates to the Automation interface `DICORBAAny`. Details about the type of value stored by an `any` can be found using the methods defined on `DICORBAAny`. The particular methods that can be called on a `DICORBAAny` depend on the kind of value contained in the `DICORBAAny`. The kind of value that the `DICORBAAny` contains can be found using the method `kind()`. This method returns an enumerated value of type `CORBATCKind`. For example, a `DICORBAAny` containing a struct has the kind `tk_struct`; a `DICORBAAny` containing an object has the kind `tk_objref`; a `DICORBAAny` containing a `typedef` has the kind `tk_alias`.

A `BadKind` exception is raised if a method is called on `DICORBAAny` that is not appropriate to the kind of value it contains.

Methods

`value()` These `propput` and `propget` methods can be called on every kind of `DICORBAAny`.

The `propget` method returns the actual value stored in `DICORBAAny`.

The `propput` method inserts a value into a `DICORBAAny`.

`kind()` This can be called on every kind of `DICORBAAny`.

It finds the type of OMG IDL definition described by the `any`. It returns an enumerated value of type `CORBATCKind`. For example, an `any` that contains a sequence has the kind `tk_sequence`. Once the kind of value stored by the `any` is known, the methods that can be called on the `any` are determined.

`id()` This can be called on a `DICORBAAny` that has the kind `tk_objref`, `tk_struct`, `tk_union`, `tk_enum`, `tk_alias` or `tk_except`. If called on a `DICORBAAny` of a different kind, it raises a `BadKind` exception.

It returns the Interface Repository ID that globally identifies the type.

This method requires run-time access to the Interface Repository.

name()

This can be called on a DICORBAAny that has the kind tk_objref, tk_struct, tk_union, tk_enum, tk_alias or tk_except. If called on a DICORBAAny of a different kind, it raises a BadKind exception.

It returns the name that identifies the type. The name returned does not contain any scoping information.

member_count()

This can be called on a DICORBAAny that has the kind tk_struct, tk_union, tk_enum or tk_except. If called on a DICORBAAny of a different kind, it raises a BadKind exception.

It returns the number of members that make up the type.

member_name()

This can be called on a DICORBAAny that has the kind tk_struct, tk_union, tk_enum or tk_except. If called on a DICORBAAny of a different kind, it raises a BadKind exception.

The method member_name() returns the name of the member identified by the index parameter. The name returned does not contain any scoping information.

A Bounds exception is raised if the index parameter is greater than or equal to the number of members that make up the type. Note that the index starts at 0.

member_type()

This can be called on a DICORBAAny that has the kind tk_struct, tk_union or tk_except. If called on a DICORBAAny of a different kind, it raises a BadKind exception.

It returns the type of the member identified by the index parameter.

A Bounds exception is raised if the index parameter is greater than or equal to the number of members that make up the type. Note that the index starts at 0.

member_label()	This can be called on a DICORBAAny that has the kind tk_union. If called on a DICORBAAny of a different kind, it raises a BadKind exception.
	The method member_label() returns the case label of the union member identified by index. (The case label is an integer, char, boolean or enum type.)
	A Bounds exception is raised if the index parameter is greater than or equal to the number of members that make up the type. Note that the index starts at 0.
discriminator_type()	This can be called on a DICORBAAny that has the kind tk_union. If called on a DICORBAAny of a different kind, it raises a BadKind exception.
	It returns the type of the union's discriminator.
default_index()	This can be called on a DICORBAAny that has the kind tk_union. If called on a DICORBAAny of a different kind, it raises a BadKind exception.
	The method default_index() returns the index of the default member; it returns –1 if there is no default member.
length()	This can be called on a DICORBAAny that has the kind tk_string, tk_sequence or tk_array.
	For a bounded string or sequence, length() returns the bound; a return value of 0 indicates an unbounded string or sequence.
	For an array, length() returns the length of the array.

content_type() This can be called on a DICORBAAny that has the
kind tk_sequence, tk_array or tk_alias. If
called on an any of a different kind, it raises a
BadKind exception.

For a sequence or array, content_type() returns
the type of element contained in the sequence or
array. For an alias, it returns the type aliased by the
typedef definition.

UUID   {A8B553C4-3B72-11CF-BBFC-444553540000}

Notes   Automation/CORBA compliant.

DICORBAFactory

Synopsis   [oleautomation,dual,uuid(...)]
interface DICORBAFactory : IDispatch
{
    HRESULT CreateObject([in] BSTR factoryName,
                         [optional,in,out] VARIANT* IT_Ex,
                         [retval,out] IDispatch** IT_retval);
    HRESULT GetObject([in] BSTR objectName,
                      [optional,in,out] VARIANT* IT_Ex,
                      [retval,out] IDispatch** IT_retval);
}

Description DICORBAFactory is a factory class that provides a way to obtain a reference to a
CORBA object.

The Automation/CORBA-compliant ProgID for this class is CORBA.Factory.

In OrbixCOMet, the name CORBA.Factory.Orbix is also registered as an alias
for CORBA.Factory. This allows access to the Orbix instance after a subsequent
installation of an ORB other than Orbix.

Methods

CreateObject() This is the same as GetObject().

`GetObject()`   The OMG *COM/CORBA Interworking* document at
`WWW.OMG.ORG` specifies that `GetObject()` should take a
string as one parameter and return a pointer to the
`IDispatch` interface on the created object. However, it
does not specify the format for the string. In OrbixCOMet,
the formats for the parameter to `GetObject()` are as
follows:

- Old format (for backwards compatibility with the
  Orbix/ActiveX Integration product):

  `"broker.interface[[[:marker]:`
  `server]:host]"`

  (Broker is ignored.)

- COMet format:

  `"interface[[[:marker]:server]:`
  `host]"`

- Tagged format:

  `"interface:TAG:Tag data"`

  where `TAG` is one of the following:

  `IOR`—The data is the hexadecimal string for the
  stringified IOR. For example:
  `fact.GetObject("employee:IOR:123456789.")`

  `NAME_SERVICE`—The data is the `NAME_SERVICE`
  compound name separated by ".". For example:
  `fact.GetObject("employee:NAME_SERVICE:`
  `IONA:employees.PD.Ronan")`

<table>
<tr>
<td>

`GetObject()`
(continued)

</td>
<td>

•Simple Format:

`"interface"`

This assumes the server name is the same as the interface. It also assumes the Orbix locator is used to find the host name. If there is no Orbix daemon running in the client machine, the configuration setting `COMET_DAEMON_HOST` should point at a machine where a daemon is running with its locator configured.

Note that if the interface were scoped (for example, `"Module::Interface"`) the interface token above would be `"Module/Interface"`.

</td>
</tr>
</table>

UUID   {204F6241-3AEC-11CF-BBFC-444553540000}

Notes   Automation/CORBA compliant.

See Also   `DIOrbixORBObject::pingDuringBind()`

DICORBAFactoryEx

Synopsis
```
[oleautomation,dual,uuid(...)]
interface DICORBAFactoryEx : DICORBAFactory {
   HRESULT CreateType([in] IDispatch* scopingObj,
                      [in] BSTR typeName,
                      [optional,in,out]  VARIANT* IT_Ex,
                      [retval,out] VARIANT* IT_retval);
   HRESULT CreateTypeById([in] IDispatch* scopingObj,
                          [in] BSTR repID,
                          [optional,in,out]  VARIANT* IT_Ex,
                          [retval,out] VARIANT* IT_retval);
};
```

Description   `DICORBAFactoryEx` is a factory class that allows Automation objects representing the OMG IDL complex types, `struct`, `union` and `exception` to be created.

You can create an object representing an OMG IDL complex type in a client in order to pass it as an `in` or `inout` parameter to an OMG IDL operation. You can create an object representing an OMG IDL complex type in a server in order to return it as an `out` or `inout` parameter or return value from an OMG IDL operation.

The methods of `DICORBAFactoryEx` can be called on an instance of the interface `DICORBAFactory`.

Methods

`CreateType()`	This creates an Automation object that is an instance of an OMG IDL complex type.
	The `scopingObj` parameter indicates the scope in which the type in `typeName` should be interpreted. Global scope is indicated by passing the parameter `Nothing`.
`CreateTypeById()`	This creates an instance of a complex type based on its repository ID. The repository ID can be determined using a call to `DIForeignComplexType::INSTANCE_repositoryID()`.
	This method requires run-time access to the IFR.

UUID {A8B553C5-3B72-11CF-BBFC-444553540000}

Notes Automation/CORBA compliant.

DICORBAObject

Synopsis
```
[oleautomation,dual,uuid(...)]
interface DICORBAObject : IDispatch {
   HRESULT GetInterface([optional,in,out]  VARIANT* IT_Ex,
                   [retval,out] IDispatch** IT_retval);
   HRESULT GetImplementation([optional,in,out]  VARIANT* IT_Ex,
                       [retval,out] BSTR* IT_retval);
   HRESULT IsA([in] BSTR repositoryID,
          [optional,in,out]  VARIANT* IT_Ex,
          [retval,out] VARIANT_BOOL* IT_retval);
   HRESULT IsNil([optional,in,out]  VARIANT* IT_Ex,
            [retval,out] VARIANT_BOOL* IT_retval);
```

```
    HRESULT IsEquivalent([in] IDispatch* obj,
                         [optional,in,out]  VARIANT* IT_Ex,
                         [retval,out] VARIANT_BOOL* IT_retval);
    HRESULT NonExistent([optional,in,out]  VARIANT* IT_Ex,
                         [retval,out] VARIANT_BOOL* IT_retval);
    HRESULT Hash([in] long maximum,
                 [optional,in,out]  VARIANT* IT_Ex,
                 [retval,out] long* IT_retval);
};
```

Description   All CORBA objects expose the interface `DICORBAObject`. It provides a number of Automation/CORBA compliant methods that all CORBA (and hence, Orbix) objects support.

Methods

`GetInterface()`	This returns a reference to an object in the IFR that provides type information about the target object. This method requires run-time access to the IFR.
`GetImplementation()`	This finds the name of the target object's server as registered in the Implementation Repository. For a local object in a server, this will be that server's name if it is known. For an object created in a client program, it will be the process identifier of the client process.
`IsA()`	This returns `true` if the object is either an instance of the type specified by the `repositoryID` parameter or an instance of a derived type of the type in `repositoryID`. Otherwise, it returns `false`.
`IsNil()`	This returns `true` if an object reference is nil. Otherwise, it returns `false`.
`IsEquivalent()`	This returns `true` if the target object reference is known to be equivalent to the object reference in the parameter `obj`.
	A return value of `false` indicates that the object references are distinct; it does not necessarily mean that the references indicate distinct objects.

253

`NonExistent()`	This returns `true` if the object has been destroyed. Otherwise, it returns `false`.
`Hash()`	Every object reference has an internal identifier associated with it—a value that remains constant throughout the lifetime of the object reference.
	`Hash()` returns a hashed value, determined via a hashing function, from the internal identifier. Two different object references can yield the same hashed value. However, if two object references return different hash values, you will know that these object references are for different objects.
	The `Hash()` function allows you to partition the space of object references into sub-spaces of potentially equivalent object references.
	The parameter `maximum` specifies the maximum value that is to be returned from the `Hash()` function. For example, by setting `maximum` to 7, the object reference space is partitioned into a maximum of 8 sub-spaces (since the lower bound of the function is 0).

UUID   {204F6244-3AEC-11CF-BBFC-444553540000}

Notes   Automation/CORBA compliant.

See Also   `DIOrbixObject`

DICORBAStruct

Synopsis   `[oleautomation,dual,uuid(...)]`
`interface DICORBAStruct : DIForeignComplexType {};`

Description An Automation interface that results from the translation of an OMG IDL `struct` definition supports the interface `DICORBAstruct`. Its purpose is to identify that the interface is translated from an OMG IDL `struct`.

UUID   {A8B553C1-3B72-11CF-BBFC-444553540000}

Notes   Automation/CORBA compliant.

DICORBASystemException

Synopsis   `[oleautomation,dual,uuid(...)]`
`interface DICORBASystemException : DIForeignException {`
`    [propget] HRESULT EX_minorCode([retval,out] long* IT_retval);`
`    [propget] HRESULT EX_completionStatus(`
`                        [retval,out] long* IT_retval);`
`};`

Description   An Automation interface that represents a system exception supports the
interface `DICORBASystemException`. (Note that system exceptions are not
defined in OMG IDL.)

Methods

`EX_minorCode()`	This describes the system exception.
`EX_completionStatus()`	This indicates the status of the operation at the time the exception occurred. Possible return values are:

COMPLETION_YES = 0

COMPLETION_NO = 1

COMPLETION_MAYBE = 2

The value COMPLETION_YES indicates that the operation had completed before the exception was raised.

The value COMPLETION_NO indicates that the operation was never initiated.

The value COMPLETION_MAYBE indicates that the operation was initiated, but whether it completed cannot be determined.

UUID   {A8B553C9-3B72-11CF-BBFC-444553540000}

Notes   Automation/CORBA compliant.

DICORBATypeCode

Synopsis
```
[oleautomation,dual,uuid(...)]
interface DICORBATypeCode : DIForeignComplexType {
// tk_objref, tk_struct,
// tk_union, tk_alias,
// tk_except
[propget] HRESULT id ([retval,out] BSTR * val);
[propget] HRESULT name ([retval,out] BSTR * val);

// tk_struct, tk_union,
// tk_enum, tk_except
[propget] HRESULT member_count ([retval,out] long* val);
        HRESULT member_name ([in] long index,
                            [retval,out] BSTR* val);
        HRESULT member_type ([in] long index,
                            [retval,out] DICORBATypeCode** val);

// tk_union
HRESULT member_label ([in] long index,
                     [retval,out] VARIANT* val);
[propget] HRESULT discriminator_type ([retval,out] IDispatch **
                                                    val);
[propget] HRESULT default_index ([retval,out] long* val);

// tk_string, tk_array,
// tk_sequence
[propget] HRESULT length ([retval,out] long* val);

// tk_array, tk_sequence,
// tk_alias
[propget] HRESULT content_type ([retval,out] IDispatch** val);
};
```

Description   An Automation interface that results from the translation of an OMG IDL
typecode definition supports the interface DICORBATypeCode.

Methods

id()

This can be called on a DICORBATypeCode that has the kind tk_objref, tk_struct, tk_union, tk_enum, tk_alias or tk_except. If called on a DICORBATypeCode of a different kind, it raises a BadKind exception.

It returns the IFR repository ID that globally identifies the type.

This method requires run-time access to the IFR.

name()

This can be called on a DICORBATypeCode that has the kind tk_objref, tk_struct, tk_union, tk_enum, tk_alias or tk_except. If called on a DICORBATypeCode of a different kind, it raises a BadKind exception.

It returns the name that identifies the type. The name returned does not contain any scoping information.

member_count()

This can be called on a DICORBATypeCode that has the kind tk_struct, tk_union, tk_enum or tk_except. If called on a DICORBATypeCode of a different kind, it raises a BadKind exception.

It returns the number of members that make up the type.

member_name()

This can be called on a DICORBATypeCode that has the kind tk_struct, tk_union, tk_enum or tk_except. If called on a DICORBATypeCode of a different kind, it raises a BadKind exception.

The method member_name() returns the name of the member identified by the parameter index. The name returned does not contain any scoping information.

A Bounds exception is raised if the parameter index is greater than or equal to the number of members that make up the type. Note that the index starts at 0.

`member_type()`	This can be called on a `DICORBATypeCode` that has the kind `tk_struct`, `tk_union` or `tk_except`. If called on a `DICORBATypeCode` of a different kind, it raises a `BadKind` exception.
	It returns the type of the member identified by the parameter `index`.
	A `Bounds` exception is raised if the parameter `index` is greater than or equal to the number of members that make up the type. Note that the index starts at `0`.
`member_label()`	This can be called on a `DICORBATypeCode` that has the kind `tk_union`. If called on a `DICORBATypeCode` of a different kind, it raises a `BadKind` exception.
	The method `member_label()` returns the case label of the union member identified by `index`. (The case label is an integer, char, boolean or enum type.)
	A `Bounds` exception is raised if the parameter `index` is greater than or equal to the number of members that make up the type. Note that the index starts at `0`.
`discriminator_type`	This can be called on a `DICORBATypeCode` that has the kind `tk_union`. If called on a `DICORBATypeCode` of a different kind, it raises a `BadKind` exception.
	It returns the type of the union's discriminator.
`default_index`	This can be called on a `DICORBATypeCode` that has the kind `tk_union`. If called on a `DICORBATypeCode` of a different kind, it raises a `BadKind` exception.
	The method `default_index()` returns the index of the default member; it returns −1 if there is no default member.
`length`	This can be called on a `DICORBATypeCode` that has the kind `tk_string`, `tk_sequence` or `tk_array`.
	For a bounded string or sequence, `length()` returns the bound; a return value of `0` indicates an unbounded string or sequence.
	For an array, `length()` returns the length of the array.

258

content_type	This can be called on a DICORBATypeCode that has the kind tk_sequence, tk_array or tk_alias. If called on a DICORBATypeCode of a different kind, it raises a BadKind exception.
	For a sequence or array, content_type() returns the type of element contained in the sequence or array. For an alias, it returns the type aliased by the typedef definition.

UUID {A8B553C3-3B72-11CF-BBFC-444553540000}

Notes Automation/CORBA compliant.

DICORBAUnion

Synopsis
```
[oleautomation,dual,uuid(...)]
interface DICORBAUnion : DIForeignComplexType {};
```

Description An Automation interface that results from the translation of an OMG IDL union definition supports the interface DICORBAUnion. Its purpose is to identify that the interface is translated from an OMG IDL union.

UUID {A8B553C2-3B72-11CF-BBFC-444553540000}

Notes Automation/CORBA compliant.

DICORBAUserException

Synopsis
```
[oleautomation,dual,uuid(...)]
interface DICORBAUserException : DIForeignException {};
```

Description An Automation interface that results from the translation of an OMG IDL exception definition supports the interface DICORBAUserException. Its purpose is to identify that the interface is translated from an OMG IDL exception.

UUID {A8B553C8-3B72-11CF-BBFC-444553540000}

Notes Automation/CORBA compliant.

DIForeignComplexType

Synopsis
```
[oleautomation,dual,uuid(...)]
interface DIForeignComplexType : IDispatch {
    [propget] HRESULT INSTANCE_repositoryId(
                                    [retval,out] BSTR* IT_retval);
    HRESULT INSTANCE_clone([in] IDispatch* obj,
                        [optional,in,out]  VARIANT* IT_Ex,
                        [retval,out] IDispatch** IT_retval);
};
```

Description   An Automation interface that results from the translation of OMG IDL complex
types (`struct`, `union`, and `exception`) supports the interface
`DIForeignComplexType`.

Methods

`INSTANCE_repositoryId()`	This returns the repository ID of a complex type.
`INSTANCE_clone()`	This creates a new instance that is an identical copy of the target instance.

Note: Both of these methods are deprecated as of CORBA 2.2. The approved
way to get a repository ID is through `DIObjectInfo::unique_id()` and
to clone using `DIObjectInfo::clone()`.

UUID   {A8B553C0-3B72-11CF-BBFC-444553540000}

Notes   Automation/CORBA compliant.

DIForeignException

Synopsis
```
[oleautomation,dual,uuid(...)]
interface DIForeignException : DIForeignComplexType {
    [propget] HRESULT EX_majorCode([retval,out] long* IT_retval);
    [propget] HRESULT EX_Id([retval,out] BSTR* IT_retval);
};
```

Description   An Automation interface that represents either a user-defined or system
exception supports the interface `DIForeignException`.

Methods

`EX_majorCode()`	This defines the category of exception raised. Possible return values are:

> `IT_NoException`
> `IT_UserException`
> `IT_SystemException`

`EX_Id()`	This returns a unique string that identifies the exception.

UUID   {A8B553C7-3B72-11CF-BBFC-444553540000}

Notes   Automation/CORBA compliant.

DIObject

Synopsis   `[oleautomation,dual,uuid(...)]`
`interface DIObject : IDispatch {};`

Description   This is the object wrapper for the OMG IDL `Object` type.

UUID   {49703179-4414-a552-1ddf-90151ac3b54b}

Notes   Automation/CORBA compliant.

DIObjectInfo

Synopsis
```
[oleautomation,dual,uuid(...)]
interface DIObjectInfo : DICORBAFactoryEx {
   HRESULT type_name ([in] IDispatch* target,
                      [optional,in,out] VARIANT * IT_Ex,
                      [retval,out] BSTR* typeName);
   HRESULT scoped_name ([in] IDispatch* target,
                        [optional,in,out] VARIANT * IT_Ex,
                        [retval,out] BSTR* repositoryID);
   HRESULT unique_id ([in] IDispatch* target,
                      [optional,in,out] VARIANT * IT_Ex,
                      [retval,out] BSTR* uniqueID);
};
```

Description   This provides helper functions for retrieving information about a composite data type (such as a `union`, `structure` or `exception`) that is held as an `IDispatch` pointer.

Methods

`type_name()`	This retrieves the simple type name of the data type.
`scoped_name()`	This retrieves the scoped name of the data type.
`unique_id()`	This retrieves the repository ID of the data type.
`clone()`	This creates an identical copy of the data type.

UUID   {6dd1b940-21a0-11d1-9d47-00a024a73e4f}

Notes   Automation/CORBA compliant.

DIOrbixObject

Synopsis
```
[oleautomation,dual,uuid(...)]
interface DIOrbixObject : DICORBAObject {
   HRESULT Bind([optional,in] VARIANT marker,
                [optional,in] VARIANT host,
                [optional,in,out] VARIANT * IT_Ex,
                [retval,out] short* IT_retval);
   [propget] HRESULT Marker([retval,out] BSTR* marker);
   [propput] HRESULT Marker([in] BSTR marker);
   [propget] HRESULT Host([retval,out] BSTR* marker);
   [propput] HRESULT Host([in] BSTR marker);
   HRESULT CloseChannel();
   HRESULT FileDescriptor([optional,in,out] VARIANT *IT_Ex,
                          [retval,out] short * rval);
   HRESULT HasValidOpenChannel([optional,in,out] VARIANT *IT_Ex,
                               [retval,out] VARIANT_BOOL * val);
}
```

Description   This allows Orbix-specific operations to be performed on the object.

Methods

Bind()

This provides a way to bind to an object in an Orbix server. It can be used as an alternative to `DICORBAObject::GetObject()` with the `marker:server:host` parameter.

The `markerServer` parameter has the format `marker:server`.

See `DICORBAObject::GetObject()` for an explanation of how the values set in `marker`, `server` and `host` affect the search for the object.

The following Visual Basic example shows how to use `Bind()` to obtain a reference to an Orbix object named `m` in server `s` on host `h`: (The Orbix object supports the interface `A`.)

- Create a view for the target Orbix object in the bridge:

```
Dim RealRef as DIA
Set RealRef as CreateObject("A")
```

- Set a reference of type `CORBA_Orbix.DIOrbixObject` pointing to the view:

```
Dim Binder as CORBA_Orbix.DIOrbixObject
Set Binder = RealRef
```

- Call `Bind()` to bind the view to the target object and release the `DIOrbixObject` reference.

```
Binder.Bind "m:s", "h"
Set Binder = Nothing

RealRef.someOperation(...)
```

263

Narrow()

This method is currently missing in version 1.0 but it will feature in a later release of the product.

A client that holds an object reference for an object of one type, and knows that the (remote) implementation object is a derived type, can narrow the object reference to the derived type.

The following Visual Basic code shows how to use this function:

```
' Given an object reference of some base
type
Dim BaseRef as ABridge.DIBase
Set BaseRef = ... ' Obtain object
reference

' Create an object of the derived type
Dim DerivedRef as ABridge.DIDerived
Set DerivedRef = _
    CreateObject("ABridge.Derived")

' Get a reference to the new object's
DIOrbixObject
' interface
Dim OrbixObjectRef as
CORBA_Orbix.DIOrbixObject
Set OrbixObjectRef = DerivedRef

' Call Narrow on this reference
OrbixObjectRef.Narrow BaseRef

' Set the reference to the DIOrbixObject
interface
' to null
Set OrbixObjectRef = Nothing
```

264

`Marker()`	The `propget` method finds the object's marker name.
	The `propput` method sets the object's marker name.
	If, when setting the object's marker, you choose a marker that is already in use for an object of the same interface within the server, OrbixCOMet will (silently) assign a different marker to the object. (The object with the original marker will not be affected.) You might wish to check for this when assigning a new marker.
	The `propput` method should be used with care. Every incoming request to a server is dispatched to the appropriate object within the server on the basis of the marker included in the request. Thus, if an object is made known to a remote client (for example, via `Bind()`, `DICORBAFactory::GetObject()`, as a return value, or as an `out` or `inout` parameter of an operation), and the object's marker is subsequently changed within the server by a call to `Marker()`, a subsequent request from the remote client will fail because the client will have used the original value of the marker. Thus, you should change the marker name of an object before knowledge of the existence of the object is passed from the server to any client.
	A marker should not consist entirely of digits and cannot contain a colon or null character.
`Host()`	This returns the host on which the object's server is located.

`CloseChannel()`	This requests Orbix to close the underlying communications connection to the server. This function is intended as an optimisation so that a connection between a client and server that is rarely used is not kept open for long periods between uses.
	The channel is automatically reopened when an invocation is made on the object. Note that if the client holds proxies for other objects in the same server, the channel is closed for all such proxies; it is automatically reopened when a subsequent invocation is made on one of these proxies
`FileDescriptor()`	This gets the file descriptor of the object.
`HasValidOpenChannel()`	This determines whether the communications channel between the client and server is open.
	(This channel can be closed to avoid having an unnecessary connection left open for long periods between an idle client and server. The channel is automatically reopened when an invocation is made on the object.)

UUID   {036A6A33-0BB3-CF47-1DCB-A2C4E4C6417A}

Notes   Orbix-specific.

See Also   `DICORBAObject`

DIOrbixORBObject

Synopsis
```
[oleautomation,dual,uuid(...)]
interface DIOrbixORBObject : DIORBObject {
   HRESULT ConnectionTimeout([in] long timeout,
                             [optional,in,out]  VARIANT* IT_Ex,
                             [retval,out] long* IT_retval);
   HRESULT MaxConnectRetries([in] long numTries,
                             [optional,in,out]  VARIANT* IT_Ex,
                             [retval,out] long* IT_retval);
```

```
HRESULT PingDuringBind([in] VARIANT_BOOL pingOn,
                   [optional,in,out]  VARIANT* IT_Ex,
                   [retval,out] VARIANT_BOOL* IT_retval);
HRESULT ReSizeObjectTable([in] long size,
                       [optional,in,out]  VARIANT* IT_Ex);
HRESULT NoReconnectOnFailure([in] VARIANT_BOOL OffOn,
                       [optional,in,out]  VARIANT* IT_Ex,
                       [retval,out] VARIANT_BOOL* IT_retval);
HRESULT ReclaimCallbackStore([optional,in,out] VARIANT* IT_Ex);
HRESULT AbortSlowConnects([in] VARIANT_BOOL OnOff,
                       [optional,in,out] VARIANT *IT_Ex,
                       [retval,out] VARIANT_BOOL *IT_retval);
HRESULT ActivateCVHandler([in] BSTR identifier,
                          [optional,in,out] VARIANT *IT_Ex );
HRESULT DeactivateCVHandler([in] BSTR identifier,
                          [optional,in,out] VARIANT *IT_Ex );
HRESULT ActivateOutputHandler([in] BSTR identifier,
                          [optional,in,out] VARIANT *IT_Ex);
HRESULT PlaceCVHandlerAfter([in] BSTR handler,
                       [in] BSTR afterThisHandler,
                       [optional,in,out] VARIANT *IT_Ex );
HRESULT PlaceCVHandlerBefore([in] BSTR handler,
                          [in] BSTR beforeThisHandler,
                          [optional,in,out] VARIANT *IT_Ex );
HRESULT DeactivateOutputHandler ([in] BSTR identifier,
                            [optional,in,out] VARIANT *IT_Ex);
HRESULT BaseInterfacesOf([in] BSTR derived,
                       [optional,in,out] VARIANT *IT_Ex,
                       [retval,out] VARIANT** IT_retval);
HRESULT IsBaseInterfaceOf([in] BSTR derived,
                       [in] BSTR maybeBase,
                       [optional,in,out] VARIANT *IT_Ex,
                       [retval,out] VARIANT_BOOL * IT_retval);
HRESULT CloseChannel([in] long fd,
                   [optional,in,out] VARIANT *IT_Ex);
HRESULT Collocated([in] VARIANT_BOOL OnOff,
                   [optional,in,out] VARIANT *IT_Ex,
                   [retval,out] VARIANT_BOOL * IT_retval);
HRESULT DefaultTxTimeout([in] long timeout,
                   [optional,in,out] VARIANT *IT_Ex,
                   [retval,out] long* IT_retval);
```

```
            HRESULT EagerListeners([in] VARIANT_BOOL OnOff,
                            [optional,in,out] VARIANT *IT_Ex,
                    [retval,out] VARIANT_BOOL * IT_retval);
            HRESULT GetConfigValue([in] BSTR name, [out] BSTR *value,
                            [optional,in,out] VARIANT *IT_Ex,
                            [retval,out] VARIANT_BOOL * IT_retval);
            HRESULT SetConfigValue([in] BSTR name, [in] BSTR value,
                            [optional,in,out] VARIANT *IT_Ex,
                            [retval,out] VARIANT_BOOL * IT_retval);
            HRESULT Output([in] VARIANT value, [in] short level,
                    [optional,in,out] VARIANT *IT_Ex);
            HRESULT ReinitialiseConfig([optional,in,out] VARIANT *IT_Ex);
            HRESULT SetDiagnostics([in] short level,
                            [optional,in,out] VARIANT *IT_Ex,
                            [retval,out] short * IT_retval);
            HRESULT StartUp([optional,in,out] VARIANT *IT_Ex,
                    [retval,out] VARIANT_BOOL * IT_retval);
            HRESULT ShutDown([optional,in,out] VARIANT *IT_Ex,
                    [retval,out] VARIANT_BOOL * IT_retval);
            HRESULT GetServerAPI([optional,in,out] VARIANT *IT_Ex,
                        [retval,out] IDispatch ** IT_retval);
            HRESULT LoadHandler([in] BSTR handlerName,
                        [optional,in,out] VARIANT *IT_Ex);
        };
```

Description `DIOrbixORBObject` provides Orbix-specific methods that allow a programmer to control some aspects of the ORB (Orbix) or request the ORB to perform actions. These methods augment the Automation/CORBA compliant methods defined in the interface `DIORBObject`.

The ORB has the ProgID CORBA.ORB.2.

CORBA.ORB.2 is the Automation/CORBA compliant name. In Orbix COMet, the name CORBA.ORB.Orbix is registered as an alias for CORBA.ORB.2. This allows access to the Orbix instance after a subsequent installation of an ORB other than Orbix.

Methods

ConnectionTimeout()

This sets the time limit, in seconds, for establishing that a connection from a client to a server is fully operational. The default is 30 seconds. This should be adequate in the majority of cases.

The value set by this function comes into effect if, for example, the server crashes after the transport (for example, TCP/IP) connection has been made but before the full Orbix connection has been established.

The value set by ConnectionTimeout() is independently used by AbortSlowConnect() when setting up the transport connection.

If clients of a single-threaded server are continually timed out because the server is busy, it might be that existing connections are being favoured over new connection attempts. The function EagerListeners() addresses this problem.

MaxConnectRetries()

If an operation call cannot be made on the first attempt because the transport (for example, TCP/IP) connection cannot be established, Orbix will retry the attempt every two seconds until either the call can be made or until there have been too many retries. The function MaxConnectRetries() sets the maximum number of retries. The default number of retries is ten.

As an alternative, the IT_CONNECT_ATTEMPTS entry in the Orbix configuration file or as an environment variable can be used to control the maximum number of retries. The value set by MaxConnectRetries() takes precedence over this. The IT_CONNECT_ATTEMPTS value will only be used if it is set to zero.

PingDuringBind()

By default, _bind() raises an exception if the object on which the _bind() is attempted is unknown to Orbix. Doing so requires Orbix to ping the desired object. The ping operation is defined by Orbix and it has no effect on the target object. The pinging will cause the target server process to be activated if necessary, and will confirm that this server recognises the target object. Pinging can be enabled using PingDuringBind() by passing 1 to the parameter pingOn. Pinging can be disabled by passing 0 to pingOn. The previous setting is returned in the parameter IT_retval.

You might wish to disable pinging to improve efficiency by reducing the overall number of remote invocations. In this case, Orbix will check the object's availability only when a method is invoked on the object and not at the time that the bind attempt is made.

Note that if PingDuringBind(false) is called:

- A _bind() to an unavailable object will not immediately raise an exception, but subsequent requests using the object reference returned from _bind() will fail by raising a system exception (CORBA::INV_OBJREF).

- If a host name is specified to _bind(), the _bind() will not itself make any remote calls; it simply sets up a proxy with the required fields.

- If a host name is not specified, Orbix uses it locator to find a suitable server, but _bind() does not interact with that server to determine if the required object exists within it.

ReSizeObjectTable()

All Orbix implementation objects in an address space are registered in its object table—a hash table that maps from object identifiers to the location of objects in virtual memory. It is important that this table is not too small for the number of objects in a process, because long overflow chains lead to inefficiencies. The default size of the object table is defined as the value:

CORBA_OBJECT_TABLE_SIZE_DEFAULT

in the file CORBA.h.

If you anticipate that a program will handle a much larger number of objects than the default size (which is about 1000), you can use this function to resize the table.

NoReconnectOnFailure()

When an Orbix client first contacts a server, a single communication channel is established between the client and the server. This connection is used for all subsequent communications between the client and server. The connection is closed only when either the client or the server exits.

When a server exits while a client is still connected, the next invocation by the client will raise a system exception of type CORBA::COMM_FAILURE. If the client attempts another invocation, Orbix will automatically try to re-establish the connection.

This default behaviour can be changed by passing the value 0 (false) to NoReconnectOnFailure(). Then, all client attempts to contact a server subsequent to closure of the communications channel will raise a CORBA::COMM_FAILURE system exception.

`ReclaimCallbackStore()`	When an Automation object is passed as a callback object to a server, Orbix creates internal structures to facilitate the callback. When this facility is no longer required, you can call `ReclaimCallbackStore()` to free the memory allocated by Orbix.
`AbortSlowConnects()`	This aborts TCP/IP connection attempts that exceed the timeout specified in `DIOrbixORBObject:: ConnectionTimeout()`. The default value for this timeout is 30 seconds.
	A TCP/IP connect can block for a considerable time if a node, known to the local node, is down or unreachable.
	Set `OnOff` to `1` to abort slow connection attempts.
`ActivateCVHandler()`	Thia activates the configuration value handler identified in identifier.
	You must call `ReinitialiseConfig()` before modifications by this function take effect.
	Refer to the Orbix documentation set for information on configuration handlers.
`DeactivateCVHandler()`	This deactivates the configuration value handler identified in `identifier`.
	You must call `ReinitialiseConfig()` before modifications by this function take effect.
	Refer to the Orbix documentation set for information on configuration handlers.
`ActivateOutputHandler()`	This activates the output handler specified in `identifier`.
	Refer to the Orbix documentation set for information on output handlers.

`PlaceCVHandlerAfter()`	This modifies the order in which configuration handlers are called. If not explicitly rearranged, configuration handlers are called in reverse order to that in which they are instantiated in an application.
	You must call `ReinitialiseConfig()` before modifications by this function take effect.
	Refer to the Orbix documentation set for information on configuration handlers.
`PlaceCVHandlerBefore()`	See `PlaceCVHandlerAfter()`.
`DeactivateOutputHandler()`	This deactivates the output handler specified in `identifier`.
	Refer to the Orbix documentation set for information on output handlers.
`BaseInterfacesOf()`	This returns a list of interfaces that are base interfaces of the interface named in `derived`. The interface `derived` is included in the list, because it is considered a base interface of itself.
`IsBaseInterfaceOf()`	This determines whether the interface `maybeBase` is a base interface of the `derived` interface.
	`IsBaseInterfaceOf()` returns `1` if `maybeBase` is a base interface of `derived` (or if `derived` and `maybeBase` are the same). Otherwise, it returns `0`.

`CloseChannel()`	This requests Orbix to close the underlying communications connection to the server. This function is intended as an optimisation so that a connection between a client and server that is rarely used is not kept open for long periods between uses.
	The channel is automatically reopened when an invocation is made on the object. Note that if the client holds proxies for other objects in the same server, the channel is closed for all such proxies; it is automatically reopened when a subsequent invocation is made on one of these proxies.
`Collocated()`	This determines whether collocation is enforced.
	Set `OnOff` to `1` to disallow binding to objects outside the address space of the current process.
	Set `OnOff` to `0`, to allow binding to objects outside the address space of the current process. This is the default.
`DefaultTxTimeout()`	This sets the timeout for all remote calls and returns the previous setting.
	By default, there is no timeout set for remote calls; that is, the default timeout is infinite.
	The value set by this function is ignored when making a connection between a client and a server. It comes into effect only when the connection has been established.

`EagerListeners()`	By default, established connections to a server are given priority over requests for new connections. As a result, busy single-threaded servers (for example, processing long running operations) might not service new connection attempts and consequently clients attempting to make a connection might be continually timed out.
	`EagerListeners()` allows equal fairness to be given to both established connections and to new connection attempts. This avoids discrimination against new connections.
	This feature is not necessary in multi-threaded versions of Orbix.
	Set `OnOff` to `1` to enable eager listening. This means that attempts to establish new connections are given equal priority to processing existing connections.
	Set `OnOff` to `0` to give priority to established connections.
	`EagerListeners()` returns the previous setting.
`GetConfigValue()`	This obtains the value of the configuration entry in `name`.
	Refer to the Orbix documentation set for information on configuration values.

275

`SetConfigValue()`	This sets the value of the configuration entry specified in `name` for this process only. (It does not set the configuration entry in the Orbix configuration file or in the Windows registry.)
	The following configuration entries can be set by `SetConfigValue()`:

```
IT_DAEMON_SERVER_BASE
IT_DAEMON_SERVER_RANGE
IT_DAEMON_PORT
IT_ERRORS
IT_IMP_REP_PATH
IT_LOCATOR_PATH
IT_INT_REP_PATH
IT_LOCAL_DOMAIN
```

	`ReinitialiseConfig()` invalidates the effect of a call to this function.
	Refer to the Orbix documentation set for information on configuration values.
`Output()`	This outputs application's diagnostic and other output via the active output handlers.
	Unless overridden by an implementation of the C++ class `CORBA::ORB::UserOutput`, all output is directed to the standard output stream via the default output handler `ITStdOutHandler`.
	Refer to the Orbix documentation set for information on output handlers.

ReinitialiseConfig()	This effects modifications to the arrangement or activation of configuration value handlers.
	It must be called in order for changes made by ActivateCVHandler(), DecactivateCVHandler(), PlaceCVHandlerBefore() and PlaceCVHandlerAfter() to take effect.
	Refer to the Orbix documentation set for information on configuration handlers.
SetDiagnostics()	This controls the level of diagnostic messages output to the cout stream by the Orbix libraries. The previous setting is returned.
	Level 1—Output no diagnostics.
	Level 2—Output simple diagnostics (default).
	Level 3—Output full diagnostics.
	Diagnostic messages are output for events such as operation requests, connections and disconnections from a client.
	An interleaved history of activity across the distributed system can be obtained from the full diagnostic output, for example from a client to a server, by redirecting the diagnostic messages from both the client and the server to files and then sorting a merged copy of these files.
StartUp()	This allows users to programatically initialise the bridge.
ShutDown()	This allows users to programatically shut down the bridge. This might be necessary if you are experiencing hang-on-exit problems.
LoadHandler()	This forces OrbixCOMet to load the specified handler DLL into memory. Handlers can contain smart proxies, filters, transformers and so on.

`Narrow()`	This narrows the object reference specified in `srcObj` to an object reference for the interface whose ProgID is specified in `cDestProgID`.

UUID   {036A6A33-0BB3-CF47-1DCB-A2C4E4C6417A}

Notes   Orbix-specific.

DIORBObject

Synopsis
```
[oleautomation,dual,uuid(...)]
interface DIORBObject : IDispatch {
   HRESULT ObjectToString([in] IDispatch* obj,
                          [optional,in,out]  VARIANT* IT_Ex,
                          [retval,out] BSTR* IT_retval);
   HRESULT StringToObject([in] BSTR ref,
                          [optional,in,out]  VARIANT* IT_Ex,
                          [retval,out] IDispatch** IT_retval);
   HRESULT GetInitialReferences([optional,in,out]  VARIANT* IT_Ex,
                             [retval,out] VARIANT* IT_retval);
   HRESULT ResolveInitialReference([in] BSTR name,
                             [optional,in,out]  VARIANT* IT_Ex,
                             [retval,out] IDispatch** IT_retval);
   HRESULT GetCORBAObject([in] IDispatch* obj,
                             [optional,in,out]  VARIANT* IT_Ex,
                       [retval,out] IDispatch** IT_retval);
};
```

Description The interface `DIORBObject` provides Automation/CORBA-compliant methods that request the ORB to perform actions.

The ORB has the ProgID `CORBA.ORB.2`.

In OrbixCOMet, the name `CORBA.ORB.Orbix` is registered as an alias for `CORBA.ORB.2`. This allows access to the Orbix instance after a subsequent installation of an ORB other than Orbix.

Methods

ObjectToString()

This converts the target object's reference to a string. An Orbix stringified object reference has the form:

```
:\host:serverName:marker:IFR_host:
IFR_server:interfaceMarker
```

The fields can be described as follows:

- host—This is the host name of the target.

- serverName—This is the name of the target object's server as registered in the Implementation Repository and also as specified to `CORBA::BOA::impl_is_ready()`, `CORBA::BOA::object_is_ready()` or set by `setServerName()`. For a local object in a server, this will be that server's name (if that server's name is known); otherwise, it will be the identifier of the process. Note that the server name will be known if the server is launched by Orbix; or if it is launched manually and the server name is passed to `impl_is_ready()` or the server name has been set by `CORBA::ORB::setServerName()`.

- marker—This is the object's marker name. This can be chosen by the application, or it will be a string of digits chosen by Orbix.

- IFR_host—This is the name of a host running an IFR that stores the target object's OMG IDL definition. Normally, this is blank.

279

`ObjectToString()` (continued)	•`IFR_server`—This is the string "`IFR`".
	•`interface_Marker`—This is the target object's interface. If called on a proxy, this might not be the object's true (most derived) interface: it can be a base interface.
	This method can also produce stringified IOR if IIOP is being used.
`StringToObject()`	This accepts a string produced by `ObjectToString()` and returns the corresponding object reference.
	(See `ObjectToString` for a description of the fields in a stringified object reference.)
`GetInitialReferences()`	The IFR and the CORBA services can only be used by first obtaining an object reference to an object through which the service can be used. The Automation/CORBA standard defines `GetInitialReferences()` as a way to list the services that are available.
	(CORBA services are optional extensions to ORB implementations that are specified by CORBA. They include the Naming Service and Event Service.)
`ResolveInitialReference()`	This returns an object reference through which a service (for example, the IFR or one of the CORBA services) can be used. The `ref` parameter names the desired service. A list of supported services can be obtained using `DIORBObject::GetInitialReferences()`.
`GetCORBAObject()`	This returns an object that allows access to the methods defined on the interfaces `DICORBAObject` and `DIOrbixObject`.

UUID	{204F6246-3AEC-11CF-BBFC-444553540000}
Notes	Automation/CORBA compliant.
See Also	`DIOrbixORBObject`

IForeignObject

Synopsis
```
interface IForeignObject : IUnknown {
    HRESULT GetForeignReference([in] objSystemIDs systemIDs,
        [out] long* systemID,
        [out] BSTR* objRef);
    HRESULT GetRepositoryId([out] BSTR* repositoryId);
};
```

Description
The `IForeignObject` interface must be supported by all view objects.

As well as having an Automation view, a bridge holds an Orbix proxy for each implementation object for which the client holds a reference. The `IForeignObject` interface provides a way for a view to find the foreign object reference in a proxy.

Methods

`GetForeignReference()`	This extracts an object reference from a proxy in string form.
	The parameter `systemIDs` is an array of long values, where a value in the array identifies an object system (for example, CORBA) for which the caller is interested in obtaining object references. The value for the CORBA object system is the long value `1`. The order of IDs in the array, `systemIDs`, indicates the caller's order of preference in the event that the proxy could be a proxy for an object in more than one object system.
	The `out` parameter `systemID` identifies an object system for which the proxy can produce an object reference. If the proxy can produce a reference for more than one object system, the order of preference specified in the `systemIDs` parameter is used to determine the value returned in this parameter.

`GetForeignReference()` (continued)	The `out` parameter `objRef` contains the object reference in string form. In the case of the CORBA object system, this is a stringified interoperable object reference (IOR).
`GetRepositoryId()`	This returns an IFR identifier for the object. This method requires run-time access to the IFR.

UUID {204f6242-3aec-11cf-bbfc-444553540000}

Notes Automation/CORBA compliant.

COM Interfaces

IOrbixServerAPI

Note: You no longer need to use IOrbixServerAPI to register your DCOM
objects with the bridge. (Refer to "Exposing DCOM Servers to CORBA
Clients" on page 163 for more details.) Because the use of this interface
is deprecated, it is mainly used for backwards compatibility purposes.

Synopsis
```
[object, uuid(...)]
interface IOrbixServerAPI : IUnknown
{
   HRESULT Activate ([in] LPSTR cServerName);
   HRESULT Deactivate ([in] LPSTR cServerName);
   HRESULT DispatchEvents ();
   HRESULT SetObjectImpl ([in] LPSTR CIFace,
                          [in] LPSTR cMarker,
                          [in] IUnknown* poImpl);
   HRESULT ActivatePersistent ([optional,in,out] VARIANT *IT_Ex);
   HRESULT SetObjectImplPersistent ([in] LPSTR cIFace,
                                    [in] LPSTR cmarker,
                                    [in] LPSTR cSrv,
                                    [in] IUnknown *poImpl,
                                    [in] LPSTR cIORFileName);
};
```

Description Bridges expose a COM interface that allows them to act as CORBA servers.
This interface can be obtained using the ProgID ServerAPI.

The COM server should instantiate an object of this type and use it to control
the COM server's behaviour as a CORBA server.

Methods

`Activate()`	This activates a COM server as a CORBA server using the `cServerName` parameter. This name should be the same name that is used to register the application in the Implementation Repository: that is, the name passed to `putit` or the server manager tool.
	Once `Activate()` has been called, your server is ready to receive incoming requests from CORBA clients.
	It is recommended that you register all your implementation objects using `SetObjectImpl()` before calling `Activate()`.
`Deactivate()`	This deactivates your application as a CORBA server. Once `Deactivate()` has been called, your application can no longer process incoming requests from CORBA clients.
	`cServerName` is the name of the CORBA server. The server must be registered with this name in the Orbix Implementation Repository.
`DispatchEvents()`	This causes any outstanding CORBA events to be dispatched to a client or server application for processing. It might be necessary to call this method in a client application if the client is asynchronously receiving callbacks from a server object. This will depend primarily on your development environment. Single-threaded development environments require this to correctly dispatch incoming events.

SetObjectImpl()	This registers a COM object with the bridge. The poimpl parameter identifies the COM object and exposes it to the CORBA object space as the interface CIFace with the Orbix marker cMarker. (Markers are used to uniquely identify different instances of the same interface.) If no marker is passed, Orbix will automatically select a unique marker for the object. The marker names chosen by Orbix consist of a string composed entirely of decimal digits. To ensure that a new marker is different from any chosen by Orbix, do not use marker strings that consist entirely of digits. Marker names cannot contain a colon ":" or a null character.
ActivatePersistent()	This allows servers to be started without the need for orbixd.
SetObjectImplPersistent()	See ActivatePersistent().

UUID   {127e2a6c-c1fe-b9f2-1d63-fb97cfc58b84}

Notes   Orbix-specific.

ICORBA_Any

Synopsis
```
typedef [public,v1_enum] enum CORBAAnyDataTagEnum {
    anySimpleValTag=0,
    anyAnyValTag,
    anySeqValTag,
    anyStructValTag,
    anyUnionValTag,
    anyObjectValTag
}CORBAAnyDataTag;

interface ICORBA_ANY;
interface ICORBA_TypeCode;
```

285

```
typedef union CORBAAnyDataUnion switch(CORBAAnyDataTag whichOne) {
  case anyAnyValTag:
     ICORBA_Any *anyVal;
  case anySeqValTag:
     struct tagMultiVal {
        [string,unique] LPSTR repositoryId;
        unsigned long cbMaxSize;
        unsigned long cbLengthUsed;
        [size_is(cbMaxSize),length_is(cbLengthUsed),unique]
           union CORBAAnyDataUnion * pVal;
     ] multiVal;
  case anyUnionValTag:
     struct tagUnionVal {
        [string,unique] LPSTR repositoryId long disc;
        union CORBAAnyDataUnion * pVal;
     } unionVal;
  case anyObjectValTag:
     struct tagObjectVal {
        [string,unique] LPSTR repositoryId VARIANT val;
     } objectVal;
  case anySimpleValTag:
     VARIANT simpleVal;
} CORBAAnyData;

[object,uuid(...),pointer_default(unique)]
interface ICORBA_Any : IUnknown
{
   HRESULT _get_value([out] VARIANT * val);
   HRESULT _put_value([in] VARIANT val);
   HRESULT _get_CORBAAnyData([out] CORBAAnyData * val);
   HRESULT _put_CORBAAnyData([in] CORBAAnyData val);
   HRESULT _get_typeCode([out] ICORBA_TypeCode ** tc);
};
```

Description   Interface for CORBA/Orbix any type.

Methods

_get_value()	This returns the value of a CORBA any.
_put_value()	This sets the value of a CORBA any.
_get_CORBAAnyData()	This returns the data stored in the CORBA any.
_put_CORBAAnyData()	This sets the data stored in the CORBA any.

_get_typeCode()   This returns the type of the `any`.

UUID   {74105f50-3c68-11cf-9588-aa0004004a09}

Notes   COM/CORBA compliant.

ICORBAFactory

Synopsis
```
[object,uuid(...)]
interface ICORBAFactory : IUnknown
{
    HRESULT CreateObject ([in] LPSTR factoryName, [out] IUnknown **
                                                         val);
    HRESULT GetObject ([in] LPSTR objectName, [out] IUnknown **
                                                         val);
};
```

Description   This supports general, simple mechanisms for creating new CORBA object
instances and accessing existing instances of CORBA object references by name.

Methods

GetObject()   The OMG *COM/CORBA Interworking* document at
                           `WWW.OMG.ORG` specifies that `GetObject()` should take
                           a string as one parameter and return a pointer to the
                           `IDispatch` interface on the created object. However,
                           it does not specify the format for the string. In
                           OrbixCOMet, the formats for the parameter to
                           `GetObject()` are as follows:

                           • Old format (for backwards compatibility with the
                             Orbix/ActiveX Integration product):

                             "broker.interface[[[:marker]:
                             server]:host]"

                             (Broker is ignored.)

                           • COMet format:

                             "interface[[[:marker]:server]:
                             host]"

GetObject()
(continued)

•Tagged format:

`"interface:TAG:Tag data"`

where `TAG` is one of the following:

`IOR`—The data is the hexadecimal string for the stringified IOR. For example:
`fact.GetObject("employee:IOR:123456789.")`

`NAME_SERVICE`—The data is the `NAME_SERVICE` compound name separated by ".". For example:
`fact.GetObject("employee:NAME_SERVICE:`
`IONA:employees.PD.Ronan")`

•Simple Format:

`"interface"`

This assumes the server name is the same as the interface. It also assumes the Orbix locator is used to find the host name. If there is no Orbix daemon running in the client machine, the configuration setting `COMET_DAEMON_HOST` should point at a machine where a daemon is running with its locator configured.

Note that if the interface were scoped (for example, `"Module::Interface"`) the interface token above would be `"Module/Interface"`.

CreateObject()

This is the same as `GetObject()`.

UUID   {204F6240-3AEC-11CF-BBFC-444553540000}

Notes   COM/CORBA compliant.

ICORBAObject

Synopsis
```
[object,uuid(...)]
interface ICORBAObject : IUnknown
{
    HRESULT GetInterface ([out] IUnknown ** val);
    HRESULT GetImplementation ([out] LPSTR * val);
    HRESULT IsA ([in] LPSTR repositoryID, [out] boolean* val);
    HRESULT IsNil ([out] boolean* val);
```

```
HRESULT IsEquivalent ([in] IUnknown* obj, [out] boolean* val);
HRESULT NonExistent ([out] boolean* val);
HRESULT Hash ([in] long maximum, [out] long* val);
};
```

Description   This allows COM clients access to operations on the CORBA object references.

Methods

GetInterface()	This returns a reference to an object in the IFR that provides type information about the target object. This method requires run-time access to the IFR.
GetImplementation()	This finds the name of the target object's server as registered in the Implementation Repository. For a local object in a server, this will be that server's name if it is known. For an object created in a client program, it will be the process identifier of the client process.
IsA()	This returns `true` if the object is either an instance of the type specified by the `repositoryID` parameter or an instance of a derived type of the type in the `repositoryID`. Otherwise, it returns `false`.
IsNil()	This returns `true` if an object reference is nil. Otherwise, it returns `false`.
IsEquivalent()	This returns `true` if the target object reference is known to be equivalent to the object reference in the parameter `obj`.
	A return value of `false` indicates that the object references are distinct; it does not necessarily mean that the references indicate distinct objects.
NonExistent()	This returns `true` if the object has been destroyed. Otherwise, it returns `false`.

Hash()

Every object reference has an internal identifier associated with it—a value that remains constant throughout the lifetime of the object reference.

`Hash()` returns a hashed value, determined via a hashing function, from the internal identifier. Two different object references can yield the same hashed value. However, if two object references return different hash values, you will know that these object references are for different objects.

The `Hash()` function allows you to partition the space of object references into sub-spaces of potentially equivalent object references.

The parameter `maximum` specifies the maximum value that is to be returned from the `Hash()` function. For example, by setting `maximum` to 7, the object reference space is partitioned into a maximum of 8 sub-spaces (since the lower bound of the function is 0).

UUID   {204F6243-3AEC-11CF-BBFC-444553540000}

Notes   COM/CORBA compliant.

ICORBA_TypeCode

Synopsis
```
[uuid(...), object, pointer_default(unique)]
interface ICORBA_TypeCode : IUnknown
{
    HRESULT equal ([in] ICORBA_TypeCode * pTc,
                   [out] boolean * pval,
                   [out] CORBA_TypeCodeExceptions ** ppExcept);
    HRESULT kind ([out] CORBA_TCKind * pval,
                  [out] CORBA_TypeCodeExceptions ** ppExcept);
    HRESULT id ([out] LPSTR * pId,
                [out] CORBA_TypeCodeExceptions ** ppExcept);
    HRESULT name ([out] LPSTR * pName,
                  [out] CORBA_TypeCodeExceptions ** ppExcept);
    HRESULT member_count ([out] unsigned long * pCount,
                          [out] CORBA_TypeCodeExceptions ** ppExcept);
```

```
       HRESULT member_name ([in] unsigned long nIndex,
                         [out] LPSTR * pName,
                    [out] CORBA_TypeCodeExceptions ** ppExcept);
       HRESULT member_type ([in] unsigned long nIndex,
                         [out] ICORBA_TypeCode ** pRetval,
                         [out] CORBATypeCodeExceptions ** ppExcept);
       HRESULT member_label ([in] unsigned long nIndex,
                         [out] ICORBA_Any ** pRetval,
                    [out] CORBA_TypeCodeExceptions ** ppExcept);
       HRESULT discriminator_type ([out] ICORBA_TypeCode ** pRetval,
                         [out] CORBA_TypeCodeExceptions ** ppExcept);
       HRESULT default_index ([out] unsigned long * pRetval,
                         [out] CORBA_TypeCodeExceptions ** ppExcept);
       HRESULT length ([out] unsigned long * nLen,
                    [out] CORBA_TypeCodeExceptions ** ppExcept);
       HRESULT content_type ([out] ICORBA_TypeCode ** pRetval,
                         [out] CORBA_TypeCodeExceptions ** ppExcept);
   };
```

Description   This describes arbitrary complex OMG IDL type structures at runtime.

Methods

equal()	This returns `true` if the `TypeCode`s are equal. Otherwise, it returns `false`.
kind()	This can be called on every kind of `ICORBA_TypeCode`.
	It finds the type of OMG IDL definition described by the `typecode`. It returns an enumerated value of type `CORBATCKind`. For example, a `typecode` that contains a sequence has the kind `tk_sequence`. Once the kind of value stored by the `typecode` is known, the methods that can be called on the `typecode` are determined.

291

id()

This can be called on an ICORBA_TypeCode that has the kind tk_objref, tk_struct, tk_union, tk_enum, tk_alias or tk_except. If called on an ICORBA_TypeCode of a different kind, it raises a BadKind exception.

It returns the IFR repository ID that globally identifies the type.

This method requires run-time access to the IFR.

name()

This can be called on an ICORBA_TypeCode that has the kind tk_objref, tk_struct, tk_union, tk_enum, tk_alias or tk_except. If called on an ICORBA_TypeCode of a different kind, it raises a BadKind exception.

It returns the name that identifies the type. The name returned does not contain any scoping information.

member_count()

This can be called on an ICORBA_TypeCode that has the kind tk_struct, tk_union, tk_enum or tk_except. If called on an ICORBA_TypeCode of a different kind, it raises a BadKind exception.

It returns the number of members that make up the type.

member_name()

This can be called on an ICORBA_TypeCode that has the kind tk_struct, tk_union, tk_enum or tk_except. If called on an ICORBA_TypeCode of a different kind, it raises a BadKind exception.

The method member_name() returns the name of the member identified by the parameter index. The name returned does not contain any scoping information.

A Bounds exception is raised if the parameter index is greater than or equal to the number of members that make up the type. Note that the index starts at 0.

`member_type()`	This can be called on an `ICORBA_TypeCode` that has the kind `tk_struct`, `tk_union` or `tk_except`. If called on an `ICORBA_TypeCode` of a different kind, it raises a `BadKind` exception.
	It returns the type of the member identified by the parameter `index`.
	A `Bounds` exception is raised if the parameter `index` is greater than or equal to the number of members that make up the type. Note that the index starts at `0`.
`member_label()`	This can be called on an `ICORBA_TypeCode` that has the kind `tk_union`. If called on an `ICORBA_TypeCode` of a different kind, it raises a `BadKind` exception.
	The method `member_label()` returns the case label of the union member identified by `index`. (The case label is an integer, char, boolean or enum type.)
	A `Bounds` exception is raised if the parameter `index` is greater than or equal to the number of members that make up the type. Note that the index starts at `0`.
`discriminator_type()`	This can be called on an `ICORBA_TypeCode` that has the kind `tk_union`. If called on an `ICORBA_TypeCode` of a different kind, it raises a `BadKind` exception.
	It returns the type of the union's discriminator.
`default_index()`	This can be called on an `ICORBA_TypeCode` that has the kind `tk_union`. If called on an `ICORBA_TypeCode` of a different kind, it raises a `BadKind` exception.
	The method `default_index()` returns the index of the default member; it returns `-1` if there is no default member.

293

`length()`	This can be called on an `ICORBA_TypeCode` that has the kind `tk_string`, `tk_sequence` or `tk_array`.
	For a bounded `string` or `sequence`, `length()` returns the bound; a return value of `0` indicates an unbounded string or sequence.
	For an array, `length()` returns the length of the array.
`content_type()`	This can be called on an `ICORBA_TypeCode` that has the kind `tk_sequence`, `tk_array` or `tk_alias`. If called on an `any` of a different kind, it raises a `BadKind` exception.
	For a sequence or array, `content_type()` returns the type of element contained in the sequence or array. For an alias, it returns the type aliased by the `typedef` definition.

UUID {9556EA21-3889-11cf-9586AA0004004A09}

Notes COM/CORBA compliant.

ICORBA_TypeCodeExceptions

Synopsis
```
typedef struct tagTypeCodeBounds {long 1;} TypeCodeBounds;
typedef struct tagTypeCodeBadKind {long 1;} TypeCodeBadKind;

[object, uuid(...), pointer_default(unique)]
interface ICORBA_TypeCodeExceptions : IUnknown
{
   HRESULT _get_Bounds([out] TypeCodeBounds * pExceptionBody);
   HRESULT _get_BadKind([out] TypeCodeBadKind * pExceptionBody);
};
typedef struct tagCORBA_TypeCodeExceptions
{
   CORBA_ExceptionType type;
   LPSTR repositoryId;
   ICORBA_TypeCodeExceptions *pUserException;
} CORBA_TypeCodeExceptions;
```

Description   This allows for the raising of exceptions that can occur with an
                       `ICORBA_TypeCode` at runtime.

Methods

       `_get_Bounds()`   This returns a `Bounds` exception that results if the
                       parameter index is greater than or equal to the
                       number of members that make up the type.

       `_get_BadKind()`   This returns a `BadKind` exception that results from
                       doing a method call on an `ICORBA_TypeCode` that has
                       the wrong kind for that method.

UUID   {9556ea20-3889-11cf-9586-aa0004004a09}

Notes   COM/CORBA compliant.

IForeignObject

Synopsis
```
typedef [public] struct objSystemIDs {
    unsigned long cbMaxSize;
    unsigned long cbLengthUsed;
    [size_is(cbMaxSize), length_is(cbLengthUsed), unique]
    long * pValue;
} objSystemIDs;

[object, uuid(...), pointer_default(unique)]
interface IForeignObject : IUnknown
{
    HRESULT GetForeignReference ([in] objSystemIDs systemIDs,
                                 [out] long * systemID,
                                 [out] LPSTR * objRef);
    HRESULT GetUniqueId ([out] LPSTR * uniqueId);
};
```
Description   This provides bridges access to object references from foreign object systems
                       that are encapsulated in proxies.

Methods

`GetForeignReference()`	This extracts an object reference from a proxy in string form.
	The parameter `systemIDs` is an array of long values where a value in the array identifies an object system (for example, CORBA) for which the caller is interested in obtaining object references. The value for the CORBA object system is the long value `1`. The order of IDs in the array, `systemIDs`, indicates the caller's order of preference in the event that the proxy could be a proxy for an object in more than one object system.
	The `out` parameter `systemID` identifies an object system for which the proxy can produce an object reference. If the proxy can produce a reference for more than one object system, the order of preference specified in the `systemIDs` parameter is used to determine the value returned in this parameter.
	The `out` parameter `objRef` contains the object reference in string form. In the case of the CORBA object system, this is a stringified interoperable object reference (IOR).
`GetUniqueId()`	This returns a unique identifier for the object.

UUID   {204f6242-3aec-11cf-bbfc-444553540000}

Notes   COM/CORBA compliant.

IMonikerProvider

Synopsis
```
[object, uuid(...)]
interface IMonikerProvider : IUnknown
{
    HRESULT get_moniker([out] IMoniker ** val);
};
```

Description   This allows COM clients to persistently save object references for later use without needing to keep the view in memory.

The moniker returned by `IMonikerProvider` must support at least the `IMoniker` and `IPersistStorage` interfaces. To allow object reference monikers to be created with one COM/CORBA interworking solution and later restored using another, `IPersist::GetClassID` must return the following `CLSID`:

> {a936c802-33fb-11cf-a9d1-00401c606e79}

Methods

`get_moniker()`	This returns a COM moniker that allows the CORBA object to be converted to persistent form for storage in a file and so on. Once stored to persistent form using this moniker, the CORBA object can be reconnected to again, using the standard COM moniker semantics.

UUID   {ecce76fe-39ce-11cf-8e92-080000970dac7}

Notes   COM/CORBA compliant.

IOrbixObject

Synopsis
```
[object, uuid(...)]
interface IOrbixObject : ICORBAObject
{
   HRESULT _get_Marker ([out] LPSTR *marker);
   HRESULT _put_Marker ([in] LPSTR marker);
   HRESULT _get_Host ([out] LPSTR *marker);
   HRESULT _put_Host ([in] LPSTR marker);
   HRESULT CloseChannel();
   HRESULT FileDescriptor ([out] short * rval);
   HRESULT HasValidOpenChannel ([out] boolean * val);
};
```
Description   This allows Orbix-specific operations to be performed on the object.

Methods

`_get_Marker()` `_put_Marker()`	Both `_get_Marker` and `_put_Marker` allow you to access the marker on the object. (Refer to `ICORBAFactory::GetObject()` on page 287 for more details.)
`_get_Host()` `_put_Host()`	Both `_get_Host` and `_put_Host` allow you to access the host part of the object reference (that is, the host on which the object lives).
`CloseChannel()`	This requests Orbix to close the underlying communications connection to the server. This function is intended as an optimisation so that a connection between a client and server that is rarely used is not kept open for long periods between uses.
	The channel is automatically reopened when an invocation is made on the object. Note that if the client holds proxies for other objects in the same server, the channel is closed for all such proxies; it is automatically reopened when a subsequent invocation is made on one of these proxies.
`FileDescriptor()`	This gets the set of file descriptors scanned by Orbix to detect incoming events. Programmers using libraries or systems that depend on the UNIX `select()` system call might need to know which file descriptors are scanned by Orbix.
	Note that this function is defined only if the following preprocessor directive is issued in the C++ file before including `CORBA.h`.
`HasValidOpenChannel()`	This determines whether the communications channel between the client and server is open.
	(This channel can be closed to avoid having an unnecessary connection left open for long periods between an idle client and server. The channel is automatically reopened when an invocation is made on the object.)

UUID   {036A6A34-0BB3-CF47-1DCB-A2C4E4C6417A}

Notes   Orbix-specific.

IOrbixORBObject

Synopsis   ```
[object, uuid(...)]
interface IOrbixORBObject : IORBObject
{
    HRESULT ConnectionTimeout ([in] long timeout,
                               [out] long* IT_retval);
    HRESULT MaxConnectRetries ([in] long numTries,
                               [out] long* IT_retval);
    HRESULT PingDuringBind ([in] BOOLEAN ping On,
                            [out] BOOLEAN* IT_retval);
    HRESULT ReSizeObjectTable ([in] long size);
    HRESULT NoReconnectOnFailure ([in] BOOLEAN OffOn,
                                  [out] BOOLEAN* IT_retval);
    HRESULT AbortSlowConnects ([in] BOOLEAN OnOff,
                               [out] BOOLEAN *IT_retval);
    HRESULT ActivateCVHandler ([in] LPSTR identifier);
    HRESULT DeactivateCVHandler ([in] LPSTR identifier);
    HRESULT ActivateOutputHandler ([in] LPSTR identifier);
    HRESULT PlaceCVHandlerAfter ([in] LPSTR handler,
                                 [in] LPSTR afterThisHandler);
    HRESULT PlaceCVHandlerBefore ([in] LPSTR handler,
                                  [in] LPSTR beforeThisHandler);
    HRESULT DeactivateOutputHandler ([in] LPSTR identifier);
    HRESULT BaseInterfacesOf ([in] LPSTR derived,
                              [out] VARIANT* IT_retval);
    HRESULT IsBaseInterfaceOf ([in] LPSTR derived,
                               [in] LPSTR maybeABase,
                               [out] BOOLEAN * IT_retval);
    HRESULT CloseChannel ([in] long fd);
    HRESULT Collocated ([in] BOOLEAN OnOff,
                        [out] BOOLEAN *IT_retval);
    HRESULT DefaultTxTimeout ([in] long timeout,
                              [out] long* IT_retval);
    HRESULT EagerListeners ([in] BOOLEAN OnOff,
                            [out] BOOLEAN * IT_retval);
    HRESULT GetConfigValue ([in] LPSTR name,
                            [out] LPSTR *value,
                            [out] BOOLEAN * IT_retval);
```

```
                HRESULT SetConfigValue ([in] LPSTR name,
                                        [in] LPSTR value,
                                        [out] BOOLEAN * IT_retval);
            HRESULT Output ([in] LPSTR value,
                            [in] short level);
            HRESULT ReinitialiseConfig ();
            HRESULT SetDiagnostics {[in] short level,
                                    [out] short * IT_retval);
            HRESULT StartUp ([out] BOOLEAN * IT_retval);
            HRESULT ShutDown ([out] BOOLEAN * IT_retval);
            HRESULT GetServerAPI ([retval,out] IDispatc ** IT_retval);
            HRESULT LoadHandler ([in] LPSTR keyName);
        };
```

Description   This is used for Orbix-specific operations.

Methods

ConnectionTimeout()   This sets the time limit, in seconds, for
                                   establishing that a connection from a client to a
                                   server is fully operational. The default is 30
                                   seconds. This should be adequate in the majority
                                   of cases.

                                   The value set by this function comes into effect
                                   if, for example, the server crashes after the
                                   transport (for example, TCP/IP) connection has
                                   been made but before the full Orbix connection
                                   has been established.

                                   The value set by ConnectionTimeout() is
                                   independently used by AbortSlowConnect()
                                   when setting up the transport connection.

                                   If clients of a single-threaded server are
                                   continually timed out because the server is busy,
                                   it might be that existing connections are being
                                   favoured over new connection attempts. The
                                   function EagerListeners() addresses this
                                   problem.

MaxConnectRetries()

If an operation call cannot be made on the first attempt because the transport (for example, TCP/IP) connection cannot be established, Orbix will retry the attempt every two seconds until either the call can be made or until there have been too many retries. The function `MaxConnectRetries()` sets the maximum number of retries. The default number of retries is ten.

As an alternative, the `IT_CONNECT_ATTEMPTS` entry in the Orbix configuration file or as an environment variable can be used to control the maximum number of retries. The value set by `MaxConnectRetries()` takes precedence over this. The `IT_CONNECT_ATTEMPTS` value will only be used if it is set to zero.

PingDuringBind()

By default, `_bind()` raises an exception if the object on which the `_bind()` is attempted is unknown to Orbix. Doing so requires Orbix to ping the desired object. The ping operation is defined by Orbix and it has no effect on the target object. The pinging will cause the target server process to be activated if necessary, and will confirm that this server recognises the target object. Pinging can be enabled using `PingDuringBind()` by passing 1 to the parameter `pingOn`. Pinging can be disabled by passing 0 to `pingOn`. The previous setting is returned in the parameter `IT_retval`.

You might wish to disable pinging to improve efficiency by reducing the overall number of remote invocations. In this case, Orbix will check the object's availability only when a method is invoked on the object and not at the time that the bind attempt is made.

Note that if `PingDuringBind(false)` is called:

- A `_bind()` to an unavailable object will not immediately raise an exception, but subsequent requests using the object reference returned from `_bind()` will fail by raising a system exception (`CORBA::INV_OBJREF`).

- If a host name is specified to `_bind()`, the `_bind()` will not itself make any remote calls; it simply sets up a proxy with the required fields.

- If a host name is not specified, Orbix uses its locator to find a suitable server, but `_bind()` does not interact with that server to determine if the required object exists within it.

ReSizeObjectTable()　　　All Orbix implementation objects in an address space are registered in its object table—a hash table that maps from object identifiers to the location of objects in virtual memory. It is important that this table is not too small for the number of objects in a process, because long overflow chains lead to inefficiencies. The default size of the object table is defined as the value:

CORBA_OBJECT_TABLE_SIZE_DEFAULT

in the file CORBA.h.

If you anticipate that a program will handle a much larger number of objects than the default size (which is about 1000), you can use this function to resize the table.

NoReconnectOnFailure()　　　When an Orbix client first contacts a server, a single communications channel is established between the client-server pair. This connection is then used for all subsequent communications between the client and the server. The connection is closed only when either the client or the server exists.

When a server exists while a client is still connected, the next invocation by the client will raise a system exception of type CORBA::COMM_FAILURE. If the client attempts another invocation, Orbix will automatically try to re-establish the connection.

This default behaviour can be changed by passing the value 0 (false) to the function CORBA::ORB::NoReconnectOnFailure(). Then, all client attempts to contact a server subsequent to closure of the communications channel will raise a CORBA::COMM_FAILURE system exception.

303

`AbortSlowConnects()`	This aborts TCP/IP connection attempts that exceed the timeout specified in `DIOrbixORBObject:: ConnectionTimeout()`. The default value for this timeout is 30 seconds.
	A TCP/IP connect can block for a considerable time if a node, known to the local node, is down or unreachable.
	Set `OnOff` to `1` to abort slow connection attempts.
`ActivateCVHandler()`	This activates the configuration value handler identified in `identifier`.
	You must call `ReinitialiseConfig()` before modifications by this function take effect.
	Refer to the Orbix documentation set for information on configuration handlers.
`DeactivateCVHandler()`	This deactivates the configuration value handler identified in `identifier`.
	You must call `ReinitialiseConfig()` before modifications by this function take effect.
	Refer to the Orbix documentation set for information on configuration handlers.
`ActivateOutputHandler()`	This activates the output handler specified in `identifier`.
	Refer to the Orbix documentation set for information on output handlers.

`PlaceCVHandlerAfter()`	This modifies the order in which configuration handlers are called. If not explicitly rearranged, configuration value handlers are called in reverse order to that in which they are instantiated in an application.
	You must call `ReinitialiseConfig()` before modifications by this function take effect.
	Refer to the Orbix documentation set for information on configuration handlers.
`PlaceCVHandlerBefore()`	See `PlaceCVHandlerAfter`.
`DeactivateOutputHandler()`	This deactivates the output handler specified in `identifier`.
	Refer to the Orbix documentation set for information on output handlers.
`BaseInterfacesOf()`	This returns an OMG IDL sequence of strings in the parameter `interfaces`, listing the base interfaces of `derived`. The interface `derived` is returned in the sequence because it is considered a base interface of itself.
`IsBaseInterfaceOf()`	This determines whether the interface `maybeABase` is a base interface of the interface `derived`.
	`IsBaseInterfaceOf()` returns `1` if `maybeABase` is a base interface of `derived` (or if `derived` and `maybeABase` are the same). Otherwise, it returns `0`.

`CloseChannel()`	This requests Orbix to close the underlying communications connection to the server. This function is intended as an optimisation so that a connection between a client and server that is rarely used is not kept open for long periods between uses.
	The channel is automatically reopened when an invocation is made on the object. Note that if the client holds proxies for other objects in the same server, the channel is closed for all such proxies; it is automatically reopened when a subsequent invocation is made on one of these proxies.
`Collocated()`	This determines whether collocation is enforced.
	Set `OnOff` to `1` to disallow binding to objects outside the address space of the current process.
	Set `OnOff` to `0`, to allow binding to objects outside the address space of the current process. This is the default.
`DefaultTxTimeout()`	This sets the timeout for all remote calls and returns the previous setting.
	By default, there is no timeout set for remote calls; that is, the default timeout is infinite.
	The value set by this function is ignored when making a connection between a client and a server. It comes into effect only when the connection has been established.

`EagerListeners()`	By default, currently established connections to a server are given priority over requests for new connections. As a result, busy single-threaded servers (for example, processing long running operations) might not service new connection attempts and consequently clients attempting to make a connection might be continually timed out.
	`EagerListeners()` allows equal fairness to be given to both established connections and to new connection attempts. This avoids discrimination against new connections.
	This feature is not necessary in multi-threaded versions of Orbix.
	Set `OnOff` to `1` to enable eager listening. This means that attempts to establish new connections are given equal priority to processing existing connections.
	Set `OnOff` to `0` to give priority to established connections.
	`EagerListeners()` returns the previous setting.
`GetConfigValue()`	This obtains the value of the configuration entry in `name`.
	Refer to the Orbix documentation set for information on configuration values.

SetConfigValue()

This sets the value of the configuration entry specified in name for this process only. (It does not set the configuration entry in the Orbix configuration file or in the Windows registry.)

The following configuration entries can be set by SetConfigValue():

```
IT_DAEMON_SERVER_BASE
IT_DAEMON_SERVER_RANGE
IT_DAEMON_PORT
IT_ERRORS
IT_IMP_REP_PATH
IT_LOCATOR_PATH
IT_INT_REP_PATH
IT_LOCAL_DOMAIN
```

ReinitialiseConfig() invalidates the effect of a call to this function.

Refer to the Orbix documentation set for information on configuration values.

Output()

This outputs application's diagnostic and other output via the active output handlers.

Unless overridden by an implementation of the C++ class CORBA::ORB::UserOutput, all output is directed to the standard output stream via the default output handler ITStdOutHandler.

Refer to the Orbix documentation set for information on output handlers.

ReinitialiseConfig()

This effects modifications to the arrangement or activation of configuration value handlers.

It must be called in order for changes made by ActivateCVHandler(), DecactivateCVHandler(), PlaceCVHandlerBefore() and PlaceCVHandlerAfter() to take effect.

Refer to the Orbix documentation set for information on configuration handlers.

	`SetDiagnostics()`	This controls the level of diagnostic messages output to the `cout` stream by the Orbix libraries. The previous setting is returned.
		Level 1—Output no diagnostics.
		Level 2—Output simple diagnostics (default).
		Level 3—Output full diagnostics.
		Diagnostic messages are output for events such as operation requests, connections and disconnections from a client.
		An interleaved history of activity across the distributed system can be obtained from the full diagnostic output, for example from a client to a server, by redirecting the diagnostic messages from both the client and the server to files and then sorting a merged copy of these files.
	`StartUp()`	This initialises the ORB. Invoking this method is optional. If `StartUp` is not invoked, the ORB is automatically initialised when the first object is created. However, it is a CORBA guideline that an ORB should be initialised before being used. IONA recommends you call this method before doing anything else (that is, before you make any calls to `GetObject` or `CreateType` on `ICORBAFactory`).
	`ShutDown()`	This unitialises the ORB. Once this is called, no more invocations can be made using CORBA.
	`GetServerAPI()`	This returns a COM/Automation interface that allows you to turn your application into a CORBA server.
	`LoadHandler()`	This forces OrbixCOMet to load the specified handler DLL into memory. Handlers can contain smart proxies, filters, transformers and so on.
UUID	`{4ea7b110-1a93-f447-1dc7-c8c8b25be06f}`	
Notes	Orbix-specific.	

IORBObject

Synopsis

```
[public] typedef struct tagCORBA_ORBObjectIdList {
    unsigned long cbMaxSize;
    unsigned long cbLengthUsed;
    [size_is(cbMaxSize), length_is(cbLengthUsed), unique]
        LPSTR *pValue;
} CORBA_ORBObjectIdList;

[object, uuid(...)]
interface IORBObject : IUnknown
{
    HRESULT ObjectToString ([in] IUnknown* obj,
                            [out] LPSTR* val);
    HRESULT StringToObject ([in,string] LPSTR cStr,
                            [out] IUnknown ** val);
    HRESULT GetInitialReferences ([out] CORBA_ORBObjectIdList*
                                                        val);
    HRESULT ResolveInitialReference ([in,string] LPSTR name,
                                     [out] IUnknown** IT_retval);
};
```

Description   This provides COM clients access to the operations on the ORB pseudo-object.

Methods

ObjectToString()   This converts the target object's reference to a
                                string. An Orbix stringified object reference has
                                the form:

                                `:\host:serverName:marker:IFR_host:`
                                `IFR_server:interfaceMarker`

                                The fields can be described as follows:

                                - `host`—This is the host name of the
                                  target.

ObjectToString()
(continued)

•serverName—This is the name of the target object's server as registered in the Implementation Repository and also as specified to `CORBA::BOA::impl_is_ready()`, `CORBA::BOA::object_is_ready()` or set by `setServerName()`. For a local object in a server, this will be that server's name (if that server's name is known); otherwise, it will be the identifier of the process. Note that the server name will be known if the server is launched by Orbix; or if it is launched manually and the server name is passed to `impl_is_ready()` or the server name has been set by `CORBA::ORB::setServerName()`.

•marker—This is the object's marker name. This can be chosen by the application, or it will be a string of digits chosen by Orbix.

•IFR_host—This is the name of a host running an IFR that stores the target object's OMG IDL definition. Normally, this is blank.

•IFR_server—This is the string "IFR".

•interface_Marker—This is the target object's interface. If called on a proxy, this might not be the object's true (most derived) interface: it can be a base interface.

This method can also produce stringified IOR if IIOP is being used.

StringToObject()

This converts the stringified object reference `obj_ref_string` to an object reference.

(See `ObjectToString` for a description of the fields in a stringified object reference.)

`GetInitialReferences()`	The IFR and the CORBA services can only be used by first obtaining an object reference to an object through which the service can be used. The Automation/CORBA standard defines `GetInitialReferences()` as a way to list the services that are available.
	(CORBA services are optional extensions to ORB implementations that are specified by CORBA. They include the Naming Service and Event Service.)
`ResolveInitialReference()`	This returns an object reference through which a service (for example, the IFR or one of the CORBA services) can be used. The parameter `ref` names the desired service. A list of supported services can be obtained using `DIORBObject::GetInitialReferences()`.

UUID   {204F6245-3AEC-11CF-BBFC-444553540000}

Notes   COM/CORBA compliant.

19

Introduction to OMG IDL

This chapter describes the CORBA Interface Definition Language (OMG IDL) that is used to describe the interfaces of objects in Orbix.

The OMG IDL language itself is part of the Object Management Group (OMG) Common Object Request Broker Architecture (CORBA) specification. OMG IDL is not a programming language because it cannot be used to implement the interfaces that are defined in it. The use of OMG IDL does not replace the roles of programming languages such as C++, OLE Automation, Visual Basic, Smalltalk, Java, and Ada. An advantage of OMG IDL is that it allows interfaces to be defined independently of the languages used to implement and use these interfaces. It therefore makes it easy to support language interoperability.

OMG IDL does not have many complex features. This makes it an easy language to learn and helps programmers to write clear interfaces.

OMG IDL Interfaces

An OMG IDL interface provides a description of the functionality that is provided by an object. An interface definition provides all of the information needed to develop clients that use the interface. An interface definition typically specifies the attributes and operations belonging to that interface, as well as the parameters of each operation. Defining the interfaces between components is the most important aspect of distributed system design. Interfaces, therefore, are the single most important feature of OMG IDL.

Consider a simple banking application that will manage bank accounts. A user of an account object will wish to make deposits and withdrawals. An account object will also need to hold the balance of the account and perhaps the name of the account's owner. A sample interface is as follows:

```
// OMG IDL
interface Account {
   // Attributes to hold the balance and the name
   // of the account's owner.
   attribute float balance;
   readonly attribute string owner;

   // The operations defined on the interface.
   void makeDeposit(in float amount,
           out float newBalance);
   void makeWithdrawal(in float amount,
           out float newBalance);
};
```

The `Account` interface defines attributes `balance` and `owner`; these are properties of an `Account` object. The attribute `balance` can take values of type `float` which is one of the basic types of OMG IDL and represents a floating point type (such as `102.31`). The attribute `owner` is of type `string` and is defined to be `readonly`.[1]

Two operations, `makeDeposit()` and `makeWithdrawal()`, are provided. Each of these has two parameters of type `float`. Each parameter must specify the direction in which the parameter is passed. The possible parameter passing modes are as follows:

`in`	The parameter is passed from the caller (client) to the called object.
`out`	The parameter is passed from the called object to the caller.
`inout`	The parameter is passed in both directions.

---

1. An attribute declaration typically maps to two functions in the programming language: one to retrieve the value of the attribute and the other to set the value of the attribute. The `readonly` keyword specifies that there is only a function to retrieve the value. A `readonly` attribute need not be a constant: two reads of an attribute where there is an interleaving operation call can return different values.

In this example, `amount` is passed as an `in` parameter to both functions and the new balance is returned as an `out` parameter. The parameter passing mode must be specified for each parameter, and it is used both to improve the "self-documentation" of an interface and to help guide the code to which the OMG IDL is subsequently translated.

Line comments are introduced with the characters `//` as shown in the example. Comments spanning more than one line are delimited by `/*` and `*/`. For example:

```
// OMG IDL
/* This commment
   spans more than
   one line. */
```

Multiple OMG IDL interfaces can be defined in a single source file, but it is common to define each interface in its own file.

Oneway Operations

Normally, the caller of an operation is blocked while the call is being processed by the receiver. However, an OMG IDL operation can be defined to be `oneway` so that the caller is not blocked and can continue in parallel to the server. For example, you could provide a oneway operation on the `Account` interface to send a notice to the account:

```
// OMG IDL
interface Account {
   // Details as before.
   // Send notice to account.
   oneway void notice(in string notice);
};
```

A `oneway` operation must specify a `void` return type. It cannot have `out` or `inout` parameters, or a `raises` clause.

Oneway operations are supported because it is sometimes important to be able to communicate with a remote object without waiting for a reply. A `oneway` operation differs from a normal operation (that is, an operation not designated as `oneway`) that has no `out` or `inout` parameters and a `void` return type. Calls to the normal operation will block until the operation request has been carried out.

Context Clause

The use of context is not specified in the *COM/CORBA Interworking Specification.* Contexts are therefore deprecated.

Modules

An interface can be defined within a module. This allows interfaces and other OMG IDL type definitions to be grouped in logical units. This can be convenient because names defined within a module do not clash with names defined outside the module (that is, a module defines a naming scope). This allows sensible names for interfaces and other definitions to be chosen without clashing with other names.

The following example illustrates the use of a module (where the interfaces related to banks are defined within a module called `Finance`):

```
// OMG IDL
module Finance {
   interface Bank {
      . . .
   };
   interface Account {
      . . .
   };
};
```

The full (or *scoped*) name of `Account` is then `Finance::Account`.

Exceptions

An OMG IDL operation can raise an exception indicating that an error has occurred. To illustrate exceptions, the banking application will now be extended by providing a `Bank` interface as follows:

```
// OMG IDL
interface Bank {
   exception Reject {
      string reason;
```

```
    };
    exception TooMany {}; // Too many accounts.
    Account newAccount(in string name)
                raises (Reject, TooMany);
    void deleteAccount(in Account a);
};
```

The `Bank` interface defines two operations:

`newAccount()`	This creates an account whose owner is the person or company whose name is passed as the parameter. The operation returns a reference to an `Account` object.
`deleteAccount()`	This deletes an account.

The `newAccount()` operation specifies via the `raises` expression that it can raise two exceptions called `Reject` and `TooMany`. The exceptions `Reject` and `TooMany` are defined within the `Bank` interface. The `Reject` exception defines a member of type `string`, which is used to specify the reason that the bank rejected the request to create a new account. The `TooMany` exception does not define any members.

As well as user-defined exceptions, a set of standard exceptions is defined by CORBA. These correspond to standard run-time errors that can occur during the execution of a request. Refer to "System Exceptions" on page 329 for more details.

Exceptions provide a clean way to allow an operation to raise an error to a caller. It allows an operation to specify that it can raise a set of possible error conditions. Because OMG IDL provides a separate syntax for exceptions, this can be translated into exception handling code in programming languages that support them (such as C++ and Ada).

Inheritance

The banking application example also needs to consider that there are many types of bank account (for example, checking (or current) accounts and savings accounts). Both checking accounts and savings accounts share the properties of an account and respond to the same operations but these operations have different behaviour. They can also have additional properties and operations.

The relationships among these interfaces can be described in a type hierarchy as shown in Figure 19.1. The `Account` interface is called a *base interface* of `CheckingAccount` and `SavingsAccount`. The interfaces `CheckingAccount` and `SavingsAccount` are called *derived interfaces* of `Account`.

Figure 19.1: *Inheritance*

You can define interface `CheckingAccount` as follows:

```
// OMG IDL
interface CheckingAccount : Account {
   readonly attribute overdraftLimit;
   boolean orderChequeBook();
};
```

It defines one attribute `overdraftLimit` but it inherits the attributes `balance` and `owner` defined in its base interface `Account`. Similarly, it inherits the operations `makeDeposit()` and `makeWithdrawal()` from `Account`, and defines a new operation `orderChequebook()`. An implementation of interface `CheckingAccount` can provide code that is different to an implementation of interface `Account`.

You can define interface `SavingsAccount` as follows:

```
// OMG IDL
interface SavingsAccount : Account {
   float calculateInterest();
};
```

An interface can be derived from any number of base interfaces. This is known as multiple inheritance. For example, a premium account might have the properties of both a checking account and a savings account. This means it is an interest earning account that can also have a cheque book. Thus the multiple inheritance hierarchy is as shown in Figure 19.2.

Figure 19.2: *Multiple Inheritance*

The SavingsAccount interface is defined as follows:

```
// OMG IDL
interface SavingsAccount : Account {
   float calculateInterest();
};
```

The PremiumAccount interface can then be specified as follows:

```
// OMG IDL
interface PremiumAccount :
     CheckingAccount, SavingsAccount {
   // New attributes and operations defined here.
};
```

If an interface inherits from two interfaces that contain a definition (constant, type, or exception) of the same name, references to this interface in the derived interface will be ambiguous unless the name of the definition is qualified by its interface name (that is, unless a scoped name is provided). It is illegal to inherit from two interfaces with the same operation or attribute name.

OMG IDL inheritance differs considerably from C++ inheritance. The latter has variations such as private, protected, public and virtual that are not reflected in OMG IDL. Public virtual inheritance in C++ is similar to OMG IDL inheritance. An instance of a derived interface must behave as an instance of all of its base interfaces. All of the attributes and operations on base interfaces are available on instances of a derived interface.

The Basic Types of OMG IDL

Table 19.1 lists the basic types supported in OMG IDL.

Type	OMG IDL Identifier	Description
Floating point	`float`	IEEE single-precision floating point numbers.
	`double`	IEEE double-precision numbers.
Integer	`long`	$-2^{31}...2^{31}-1$ (32bit)
	`short`	$-2^{15}...2^{15}-1$ (16bit)
	`unsigned long`	$0...2^{32}-1$ (32bit)
	`unsigned short`	$0...2^{16}-1$ (16bit)
Char	`char`	An 8-bit quantity.
Boolean	`boolean`	`TRUE` or `FALSE`
Octet	`octet`	An 8-bit quantity that is guaranteed not to undergo any conversion during transmission.
Any	`any`	The `any` type allows the specification of values that can express an arbitrary OMG IDL type.

Table 19.1: *OMG IDL Basic Types*

The `any` type allows an interface to specify that a parameter or result type can contain an arbitrary type of value to be determined at runtime. For example:

```
// OMG IDL
interface I {
   void op(in any a);
};
```

A process that receives an `any` must determine what type of value it contains and then extract the value.

Constructed Types

As well as the basic types listed above, OMG IDL provides three constructed types: `struct`, `union` and `enum`.

Structures

A `struct` data type allows related items to be packaged together. For example:

```
// OMG IDL
struct PersonalDetails {
   string name;
   short age;
};

interface Bank {
   exception Reject {
      string reason;
   };
   Account newAccount(in string name,
           in short age) raises (Reject);
   PersonalDetails getPersonalDetails(
           in string name);
   void deleteAccount(in account a);
};
```

The `struct PersonalDetails` has two members: `name` of type `string`, and `age` of type `short`. The operation `getPersonalDetails()` returns one of these structs.

Enumerated Types

An enumerated type allows the members of a set of values to be depicted by identifiers. For example:

```
// OMG IDL
enum colour { red, green, blue, yellow, white };
```

This is more readable than defining `colour` as a `short`. The order in which the identifiers are named in the specification of an enumerated type defines the relative order of the identifiers. This order can be used by a specific programming language mapping that allows two enumerators to be compared.

Unions

The OMG IDL `union` type provides a space-saving type whereby the amount of storage required for a `union` is the amount necessary to store its largest element. A tag field is used to specify which member of a `union` instance is currently assigned a value. For example:

```
// OMG IDL
union token switch (long) {
    case 1 : long l;
    case 2 : float f;
    default: string str;
};
```

The identifier following the `union` keyword defines a new legal type. A `union` type can also be named using a `typedef` declaration.

OMG IDL unions must be discriminated. This means the `union` header must specify a tag field that determines which `union` member is assigned a value. In the preceding example, the tag is called `token` and is of type `long`. Each expression that follows the `case` keyword must be compatible with the tag type. The type specified in parentheses after the `switch` keyword must be an integer, char, boolean or enum type. A default case can appear at most once in a `union` declaration.

Arrays

OMG IDL provides multi-dimensional fixed-size arrays to hold lists of elements of the same type. The size of each dimension should be specified in the definition. Some sample array types are as follows:

```
// OMG IDL
// A 1-dimensional array.
Account bankAccounts[100];

// A 2-dimensional array.
short gridArr[10][20];
```

Types `bankAccounts` and `gridArr` can be used, for example, to define parameters to an operation.

Template Types

OMG IDL provides two template types, `sequence` and `string`, that are described in the following subsections.

Sequences

An OMG IDL `sequence` data type allows lists of items to be passed between objects. A sequence is similar to a one-dimensional array. It has two characteristics: a maximum size that is fixed at compile time and a length that is determined at runtime. A sequence differs from an array in that a sequence is not of fixed size (although a bounded sequence has a fixed maximum size). Hence, a sequence is a more flexible data type, and should be used in preference to an array except when the list of elements to be passed is always of the same size.

A sequence can be bounded or unbounded, depending on whether the maximum size is specified. For example, the following type declaration:

```
// OMG IDL
sequence<long, 10> vectorTen;
```

defines a bounded sequence of size 10. The sequence `vectorTen` can be of any length up to the bound (that is, 10).

323

The following type declaration defines an unbounded sequence:

```
// OMG IDL
sequence<long> vector;
```

A sequence that is used within an interface definition must be named by a typedef declaration before it can be used as the type of an attribute definition or as a parameter to an operation. For example:

```
// OMG IDL
typedef sequence<long, 10> vectorTen;

attribute vectorTen vector;

// The following definition is not allowed:
attribute sequence<long, 10> illegalVector;
```

A sequence that appears within a struct or union definition does not have to be named.

Strings

The `string` type has already been used. It is similar to a sequence of `char`. A `string` can be bounded or unbounded depending on whether its length is specified in the declaration. A length can be specified for a `string` as shown in the following example:

```
// OMG IDL
interface Bank {
   // Other details as before.

   // A bounded string.
   attribute string sortCode<10>;

   // An unbounded string.
   attribute string address;
};
```

Constants

A constant can be defined as follows:

```
// OMG IDL
interface Bank {
    const long MaxAccounts = 100000;
    // Rest of definition here.
};
```

The value of an OMG IDL constant cannot change. Constants can be defined in an interface or module, or at global or file-level scope (outside of any interface or module).

Constants of type `long`, `unsigned long`, `short`, `unsigned short`, `char`, `boolean`, `float`, `double` and `string` can be declared. Constants of type `octet` cannot be declared.

Typedef Declaration

A `typedef` declaration can be used to define a meaningful or a more simple name for a basic or a user-defined type. For example:

```
// OMG IDL
typedef short size;
```

defines `size` as a synonym for `short`. Consequently the parameter declaration:

```
// OMG IDL
in size i
```

is equivalent to:

```
// OMG IDL
in short i
```

The definition:

```
// OMG IDL
typedef Account Accounts[100];
```

allows a subsequent definition (for example, as a member of a structure):

```
// OMG IDL
Accounts bankAccounts;
```

Forward Declaration

An interface must be declared before it is referenced. A forward declaration declares the name of an interface without defining it. This allows the definition of interfaces that mutually reference each other. The syntax is the keyword `interface` followed by the identifier that names the interface. For example:

```
// OMG IDL
interface Bank;
```

The interface definition must appear later in the specification.

Scoped Names

An OMG IDL file forms a naming scope in which an identifier is defined and can be referred to. Every OMG IDL identifier must be unique within a scope but an identifier can be reused in distinct scopes. An interface is considered to represent a distinct scope. Thus, names defined within an interface do not clash with names defined outside of that interface (for example, in another interface or at file level). The following type definitions also represent distinct scopes: module, structure, union, operation and exception. The following type definitions are treated as being scoped: types, constants, enumeration values, exceptions, interfaces, attributes and operations.

A qualified or *scoped name* has the form `<scoped_name>::<identifier>`. Within a scope, a name can be used in its unqualified form.

The Preprocessor

OMG IDL provides preprocessing directives that allow macro substitution, conditional compilation and source file inclusion. The OMG IDL preprocessor is based on the C++ preprocessor. For example, the `#include` directive allows an OMG IDL file to be included in other files.

As for a C++ include file, the following directives should be used in an OMG IDL file that is potentially included in many other OMG IDL files:

```
#ifndef <some_unique_name>
#define <some_unique_name>

Body of the idl file.

#endif
```

Other preprocessing directives available in OMG IDL are `#define`, `#undef`, `#include`, `#if`, `#ifdef`, `#ifndef`, `#elif`, `#else`, `#endif`, `#defined`, `#error`, and `#pragma`.

The Orb.idl Include File

The interface names for the CORBA pseudo types `NamedValue`, `Principal` and `TypeCode` are available in an OMG IDL file only if it includes the following directive:

```
#include <orb.idl>
```

The interface name `Object` (the implicit base interface of all interfaces) is available in all files.

20

System Exceptions

This chapter describes system exceptions that are defined by CORBA or specific to Orbix.

Exceptions Defined by CORBA

Identifier	Exception	Description
10000	UNKNOWN	The unknown exception.
10020	BAD_PARAM	An invalid parameter was passed.
10040	NO_MEMORY	Dynamic memory allocation failure.
10060	IMP_LIMIT	Violated implementation limit.
10080	COMM_FAILURE	Communication failure.
10100	INV_OBJREF	Invalid object reference.
10120	NO_PERMISSION	No permission for attempted operation.
10140	INTERNAL	ORB internal error.
10160	MARSHAL	Error marshalling parameter/result.
10180	INITIALIZE	ORB initialisation failure.
10200	NO_IMPLEMENT	Operation implementation unavailable.
10220	BAD_TYPECODE	Bad TypeCode.
10240	BAD_OPERATION	Invalid operation.

10260	NO_RESOURCES	Insufficient resources for request.
10280	NO_RESPONSE	Response to request not yet available.
10300	PERSIST_STORE	Persistent storage failure.
10320	BAD_INV_ORDER	Routine invocations out of order.
10340	TRANSIENT	Transient failure—reissue request.
10360	FREE_MEM	Cannot free memory.
10380	INV_IDENT	Invalid identifier syntax.
10400	INV_FLAG	Invalid flag was specified.
10420	INTF_REPOS	Error accessing Interface Repository.
10440	BAD_CONTEXT	Error processing context object.
10460	OBJ_ADAPTOR	Failure detected by object adaptor.
10480	DATA_CONVERSION	Data conversion error.

Table 20.1: *CORBA-Defined Exceptions*

Orbix-Specific Exceptions

Identifier	Exception	Description
10500	FILTER_SUPPRESS	Suppress exception raised in per-object pre-filter.
10520	LOCATOR	Locator error.
10540	ASCII_FILE	ASCII file error.
10560	LICENCING	Licencing error.
10580	VXWORKS_EX	VxWorks error.
10600	IIOP	IIOP error.
10620	NO_CONFIG_VALUE	No configuration value set for one of the mandatory configuration entries.

Table 20.2: *Orbix-Specific Exceptions*

Index