OrbixCOMet Desktop Programmer's Guide and Reference

IONA Technologies PLC
September 2000

M 2 4 2 3

Contents

Preface xiii

- **Audience** xiii
- **Contact Information** xiii
- **Organization of this Guide** xiv
- **Document Conventions** xix

Part I Introduction

Chapter 1 Introduction to OrbixCOMet 3

- **Two-way Interworking** 4
- **Transparent Interworking** 4
- **The Interworking Model** 5
- **How OrbixCOMet Implements the Interworking Model** 6
  - Bridge 8
  - Automation Client 9
  - COM Client 9
  - COM Library 9
  - CORBA Server 9
  - CORBA Client 10
  - Automation Server 10
  - COM Server 10

Chapter 2 Usage Models and Bridge Locations 11

- **Automation Client to CORBA Server** 12
- **COM Client to CORBA Server** 14
- **CORBA Client to COM or Automation Server** 16

Part II Programmer's Guide

Chapter 3 Getting Started 21

Server-Side Requirements 21
Registering OMG IDL Type Information 22
Implementing Automation Clients 22
Writing a Client Using PowerBuilder 23
Writing a Client Using Visual Basic 26
Running the Client Application 28
Using DCOM On-the-Wire with OrbixCOMet 29
DCOM Security 30
The Surrogate Executable 31
Using OrbixCOMet with Internet Explorer 31
Automation Dual Interface Support 34
Implementing COM Clients 36
Generating COM IDL Definitions from OMG IDL 36
Writing COM Clients 37
Priming the OrbixCOMet Type Store Cache 40
DCOM Trouble-Shooting 40
Miscellaneous Configuration Tips 43

Chapter 4 Developing a Client in Automation 45

The Telephone Book Example 46
Creating a Type Library 47
Implementing the Client 47
Obtaining a Reference to a CORBA Object 48
The Visual Basic Client Code in Detail 50
The PowerBuilder Client Code in Detail 52
Building the Client 54
Running the Client 54

Chapter 5 Developing a Client in COM 55

The Telephone Book Example 56
Obtaining a COM IDL Interface 57
Building a Proxy/Stub DLL 57

Implementing the Client 58
Obtaining a Reference to a CORBA Object 58
Using CoCreateInstance() 60
The COM C++ Client Code in Detail 60
Building the Client 62
Running the Client 63

Chapter 6 Implementing CORBA Clients 65
Interfaces to the ORB 66
Obtaining Object References 68
The (D)ICORBAFactory Interface 68
The Naming Service 70
IDL Operations 74
Interworking Interfaces on Objects 75
Implementing CORBA Clients in Automation 76
Late Binding 76
Early Binding 76
Narrowing Object References 77
A Visual Basic Client Program 78
Implementing CORBA Clients in COM 82
COM Apartments and Threading 82
Narrowing Object References 82
A COM C++ Client Program 83

Chapter 7 Exposing DCOM Servers to CORBA Clients 89
The Supplied DCOM Server 90
Building the DCOM Server and Proxy Stub DLLs 90
Priming the Type Store 91
Registering the Server 91
Generating OMG IDL 92
Writing a Client to Talk to the DCOM Server 94
CORBA Client Example Using Composable Support 95
Connection and Usage with the Custsur Executable 97

Chapter 8 Implementing CORBA Servers 99
Steps to Implementing a CORBA Server 99
Defining and Registering OMG IDL Interfaces 100
Generating a Type Library or COM IDL 101

Generating Server Skeleton Code **101**
Implementing the Server Interfaces **102**
Implementing the Account Interface 102
Implementing the CurrentAccount Interface 103
Implementing the Bank Interface 104
Registering the Server with OrbixCOMet **105**
Running the Server **108**
Registering the CORBA Server in the Implementation Repository **108**

Chapter 9 Exception Handling **109**
CORBA Exceptions **110**
Example of a User Exception **110**
Exception Properties **112**
Exception Handling in Automation **113**
Exception Handling in Visual Basic 114
Inline Exception Handling 115
Exception Handling in COM **118**
Catching COM Exceptions 118
Using Direct-to-COM Support in Visual C++ 119
Raising an Exception in a Server **120**
Automation Exceptions 120
COM Exceptions 121

Chapter 10 Implementing Client Callbacks **123**
Defining OMG IDL Interfaces **124**
Generating Skeleton Code for Callback Objects **125**
Writing a Client **125**
Visual Basic 125
PowerBuilder 126
COM C++ 126
Writing the Server **128**
Implementing the RegisterCallback Interface 128
Invoking the Operation to Notify the Client 129
Registering the Callback Object Server **131**
Visual Basic 131
PowerBuilder 131
COM C++ 132

Chapter 11 SSL Support 133
Enabling SSL in an OrbixCOMet Application 134
OrbixCOMet SSL Handler DLLs 135
Secure CORBA Clients Accessing Existing DCOM Servers 136
Specifying the Custsur.exe Certificate 137
Specifying the Corresponding Private-Key Password 138
OrbixCOMet Type Store Manager and the Secure IFR 138

Chapter 12 Deploying an OrbixCOMet Application 139
Deployment Models 139
Bridge on Each Client Machine 140
Bridge on Server Machine 142
Bridge on Intermediary Machine 144
Internet Deployment 146
Deployment Steps 146
Installing Your Application Runtime 147
Installing the Development Language Runtime 147
Installing the Orbix Runtime 147
Installing the OrbixCOMet Runtime 149
Minimizing the Client-Side Footprint 151
Using Handler DLLs 153
Creating and Registering Handler DLLs 153
Loading Handler DLLs at Runtime 154
Managing Handler DLLs 154

Chapter 13 Development Support Tools 157
The Central Role of the Type Store 158
The Caching Mechanism of the Type Store 159
The OrbixCOMet Tools GUI Screen 160
Location of the Command-Line Utilities 161
Adding New Information to the Type Store 161
Using the GUI Tool 162
Using the Command-Line Utilities 163
Deleting the Type Store Contents 166
Using the GUI Tool 166
Using the Command-Line Utilities 166

Rebuilding the Type Store	**167**
Using the GUI Tool	167
Using the Command-Line Utilities	167
Dumping the Type Store Contents	**167**
Creating an IDL File	**168**
Using the GUI Tool	168
Using the Command-Line Utilities	170
Creating a Type Library	**171**
Using the GUI Tool	172
Using the Command-Line Utilities	173
Generating a Handler DLL	**174**
Generating Server Stub Code and Support for Callbacks	**176**
Replacing an Existing DCOM Server	**177**

Part III Programmer's Reference

Chapter 14 OrbixCOMet API Reference	**181**
Automation Interfaces	**181**
DIOrbixServerAPI	181
DCollection	184
DICORBAAny	184
DICORBAFactory	189
DICORBAFactoryEx	191
DICORBAObject	192
DICORBAStruct	194
DICORBASystemException	195
DICORBATypeCode	196
DICORBAUnion	199
DICORBAUserException	200
DIForeignComplexType	200
DIForeignException	201
DIObject	201
DIObjectInfo	201
DIOrbixObject	202
DIOrbixORBObject	205
DIOrbixSSL	218
DIORBObject	220

IForeignObject 221
COM Interfaces 223
IOrbixServerAPI 223
ICORBA_Any 225
ICORBAFactory 227
ICORBAObject 228
ICORBA_TypeCode 230
ICORBA_TypeCodeExceptions 234
IForeignObject 235
IMonikerProvider 236
IOrbixObject 237
IOrbixORBObject 239
IOrbixSSL 250
IORBObject 252

Chapter 15 Introduction to OMG IDL 255
OMG IDL Interfaces 255
Oneway Operations 257
Context Clause 258
Modules 258
Exceptions 258
Inheritance 259
The Basic Types of OMG IDL 262
Constructed Types 263
Structures 263
Enumerated Types 264
Unions 264
Arrays 265
Template Types 265
Sequences 265
Strings 266
Constants 267
Typedef Declaration 267
Forward Declaration 268
Scoped Names 268
The Preprocessor 268
The Orb.idl Include File 269

Chapter 16 CORBA-to-Automation Mapping **271**
Basic Types **272**
Strings **273**
Interfaces **274**
Attributes 275
Operations 277
Inheritance 278
Complex Types **282**
Creating Constructed OMG IDL Types 283
Structs 283
Unions 285
Sequences 287
Arrays 290
Exceptions 291
The Any Type 293
Context Clause 293
Object References **293**
Modules **295**
Constants **296**
Enumerated Types **296**
Scoped Names **297**
Typedefs **298**

Chapter 17 Automation-to-CORBA Mapping **299**
Basic Types **300**
Strings **301**
Interfaces **301**
Properties and Methods 302
Inheritance 303
SafeArrays **304**
Exceptions **304**
Variant Types **305**
Object References **305**
Enumerated Types **306**
Typedefs **307**

Chapter 18 CORBA-to-COM Mapping **309**
- **Basic Types** **310**
- **Strings** **310**
- **Interfaces** **311**
  - Attributes 312
  - Operations 313
  - Inheritance 314
- **Complex Types** **317**
  - Creating Constructed OMG IDL Types 317
  - Structs 317
  - Unions 319
  - Sequences 320
  - Arrays 321
  - Exceptions 322
  - The Any Type 325
  - Context Clause 326
- **Object References** **326**
- **Modules** **327**
- **Constants** **328**
- **Enumerated Types** **328**
- **Scoped Names** **330**
- **Typedefs** **331**

Chapter 19 COM-to-CORBA Mapping **333**
- **Basic Types** **334**
- **Strings** **335**
- **Interfaces** **336**
  - Properties and Methods 336
  - Inheritance 338
- **Complex Types** **339**
  - Structs 339
  - Unions 340
  - Pointers 342
  - Arrays 342
  - Exceptions 343
  - Variant Types 345
- **Constants** **346**
- **Enumerated Types** **346**

Scoped Names 347
Typedefs 348

Chapter 20 System Exceptions 351
Exceptions Defined by CORBA 351
Orbix-Specific Exceptions 352

Chapter 21 OrbixCOMet Configuration 353
OrbixCOMet Keys 353
Common Keys 360
Orbix Keys 361

Chapter 22 OrbixCOMet Utility Options 363
Typeman Options 363
Ts2idl Options 365
Ts2tlb Options 366
Ts2sp Options 367
Aliassrv Options 368
Custsur Options 368
Tlibreg Options 369

Index 371

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 programmers—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 environments. It means COM programmers can use the tools familiar to them to build heterogenous systems that use both COM and CORBA components within a COM environment. OrbixCOMet is also designed to allow CORBA programmers to build, using COM programming tools, heterogenous systems that use both CORBA and COM components within a CORBA environment.

Audience

This guide is intended for use by CORBA and COM application programmers who wish to familiarise themselves with using OrbixCOMet to develop and deploy distibuted applications that combine CORBA and COM components within their own native object environment.

Contact Information

Orbix documentation is periodically updated. New versions between releases are available at this site:

`http://www.iona.com/docs/orbix/orbix33.html`

If you need assistance with Orbix or any other IONA products, contact IONA at `support@iona.com`. Comments on IONA documentation can be sent to `doc-feedback@iona.com`.

Organization of this Guide

This guide is divided into three main parts.

Part I, Introduction

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, "Usage Models and Bridge Locations"

You can use OrbixCOMet to develop and deploy distributed 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 or Automation client that calls objects in a CORBA server, and a CORBA client that calls objects in a COM or Automation server. This chapter explains how OrbixCOMet supports these usage models.

Part II, Programmer's Guide

Chapter 3, "Getting Started"

This chapter is provided as a quick means to getting started in application programming with OrbixCOMet. It explains the basics you need to know to develop a simple OrbixCOMet application, using PowerBuilder or Visual Basic, where an Automation client can invoke on an existing CORBA server. It also provides an introduction to writing COM clients, using OrbixCOMet.

Chapter 4, "Developing a Client in Automation"

This chapter expands on what you learned in Chapter 3, "Getting Started". It uses the example of a distributed telephone book application to show how to write Automation clients that can communicate with an existing CORBA C++ server, using PowerBuilder and Visual Basic.

Chapter 5, "Developing a Client in COM"

This chapter expands on what you learned in Chapter 3, "Getting Started". It uses the example of a distributed telephone book application to show how to write a COM C++ client that can communicate with an existing CORBA C++ server.

Chapter 6, "Implementing CORBA Clients"

This chapter is aimed at CORBA programmers who want to implement CORBA clients, using Automation-based tools such as Visual Basic and PowerBuilder, and COM-based tools such as C++.

Chapter 7, "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 8, "Implementing CORBA Servers"

You can use OrbixCOMet to implement CORBA servers, using Automation-based tools such as PowerBuilder or Visual Basic. These servers can accept requests from standard COM/Automation clients as well as from CORBA clients. This chapter explains how to use OrbixCOMet to implement a CORBA server.

Chapter 9, "Exception Handling"

Exception 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 10, "Implementing 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 11, "SSL Support"

SSL support with OrbixCOMet opens up the domain of SSL-secured CORBA programs to COM/Automation clients and servers. Using SSL with your OrbixCOMet applications means on-the-wire communication using IIOP is secure.

Chapter 12, "Deploying an OrbixCOMet Application"

This chapter provides examples of the various deployment models you can adopt when deploying a distributed application, using OrbixCOMet. It also describes the steps you must follow to deploy a distributed OrbixCOMet application.

Chapter 13, "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 type information, which is used by the dynamic bridge during runtime of your OrbixCOMet applications. This chapter describes the type store and the central role it plays in terms of the development support tools supplied with OrbixCOMet. It also describes the GUI and command-line versions of the development support tools that allow you to maintain the type store cache, and to create IDL files, type libraries, handler DLLs, and server stub code from existing type store information. Finally, it describes the tools that you can use to replace an existing COM or Automation server with a CORBA server.

Part III, Programmer's Reference

Chapter 14, "OrbixCOMet API Reference"

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

Chapter 15, "Introduction to OMG IDL"

This chapter describes the CORBA Interface Definition Language (OMG IDL) that is used to describe the interfaces to objects in Orbix

Chapter 16, "CORBA-to-Automation Mapping"

CORBA types are defined in OMG IDL. Automation types are defined in Microsoft IDL (MIDL). To allow interworking between Automation clients and CORBA servers, Automation clients must be presented with MIDL versions of the interfaces exposed by CORBA objects. Therefore, it must be possible to translate CORBA types to MIDL. This chapter outlines the CORBA-to-Automation mapping rules.

Chapter 17, "Automation-to-CORBA Mapping"

Automation types are defined in Microsoft IDL (MIDL). CORBA types are defined in OMG IDL. To allow interworking between CORBA clients and Automation servers, CORBA clients must be presented with OMG IDL versions of the interfaces exposed by Automation objects. Therefore, it must be possible to translate Automation types to OMG IDL. This chapter outlines the Automation-to-CORBA mapping rules.

Chapter 18, "CORBA-to-COM Mapping"

CORBA types are defined in OMG IDL. COM types are defined in Microsoft IDL (MIDL). To allow interworking between COM clients and CORBA servers, COM clients must be presented with MIDL versions of the interfaces exposed by CORBA objects. Therefore, it must be possible to translate CORBA types to MIDL. This chapter outlines the CORBA-to-COM mapping rules.

Chapter 19, "COM-to-CORBA Mapping"

COM types are defined in Microsoft IDL (MIDL). CORBA types are defined in OMG IDL. To allow interworking between CORBA clients and COM servers, CORBA clients must be presented with OMG IDL versions of the interfaces exposed by COM objects. Therefore, it must be possible to translate COM types to OMG IDL. This chapter outlines the COM-to-CORBA mapping rules.

Chapter 20, "System Exceptions"

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

Chapter 21, "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 22, "OrbixCOMet Utility Options"

This chapter describes the various options that are available with each of the OrbixCOMet command-line utilities.

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.

Bold — Bold text represents the names of GUI items, such as screens, fields, menu options, and buttons.

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 98 command prompt.
…	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

Introduction

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

Note: OrbixCOMet is not a CORBA C++ server-side implementation product. Any C++ server examples provided in this book are supplied for reference purposes only. It is assumed 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 or Automation client can call objects in a CORBA server. Because both COM and CORBA support distribution, the client and server can be on different machines.
- A CORBA client can call objects in a COM or Automation server. Again, the client and 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. Orbix supports a range of operating systems, such as Windows, UNIX, and OS/390. It also supports a range of programming languages, such as C++, Java, and (using OrbixCOMet) all COM and Automation-based languages.

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

OrbixCOMet supports both the Internet Inter-ORB Protocol (IIOP) and Microsoft DCOM protocol. 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 treat a COM or Automation object as if it were a CORBA object. This is because the object has an OMG IDL interface that the CORBA client can understand.

- A client working in the COM model can treat a CORBA object as if it were a COM or Automation object. This is because the object has a COM IDL interface that the COM or 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 is an overview of the interworking model.

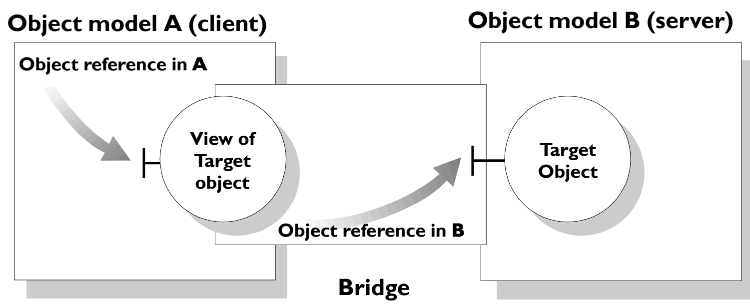

Figure 1.1: *The Interworking Model*

A client in one object system wants 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 the client can make requests in its familiar object model.

To implement the bridge, the interworking 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. Figure 1.3 on page 8 shows how this is implemented in OrbixCOMet.

The client makes requests on the view object. The bridge maps these 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

OrbixCOMet combines the best of both the OMG CORBA and Microsoft DCOM standards. It provides a high performance bi-directional dynamic bridge that enables two-way integration between COM/Automation and CORBA applications.

For a CORBA programmer, OrbixCOMet provides the expected development paradigm for ORB applications. The CORBA programmer starts with an OMG IDL specification. Using OrbixCOMet, a CORBA programmer can develop:

- CORBA clients, using COM-based tools such as C++, or Automation-based tools such as Visual Basic or PowerBuilder.
- CORBA servers, using Automation-based tools such as Visual Basic or PowerBuilder.

OrbixCOMet does not facilitate development of CORBA C++ servers. You can use the Orbix C++ product to implement CORBA C++ servers.

For a COM programmer, OrbixCOMet provides access to CORBA applications that are running on Windows, UNIX or OS/390 environments. Using OrbixCOMet, a COM programmer can use familiar COM-based and Automation-based tools to build heterogeneous systems that use both COM and CORBA components within a COM environment.

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 or 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 or Automation server. Refer to "Usage Models and Bridge Locations" on page 11 for more details of how you can combine the two object models.

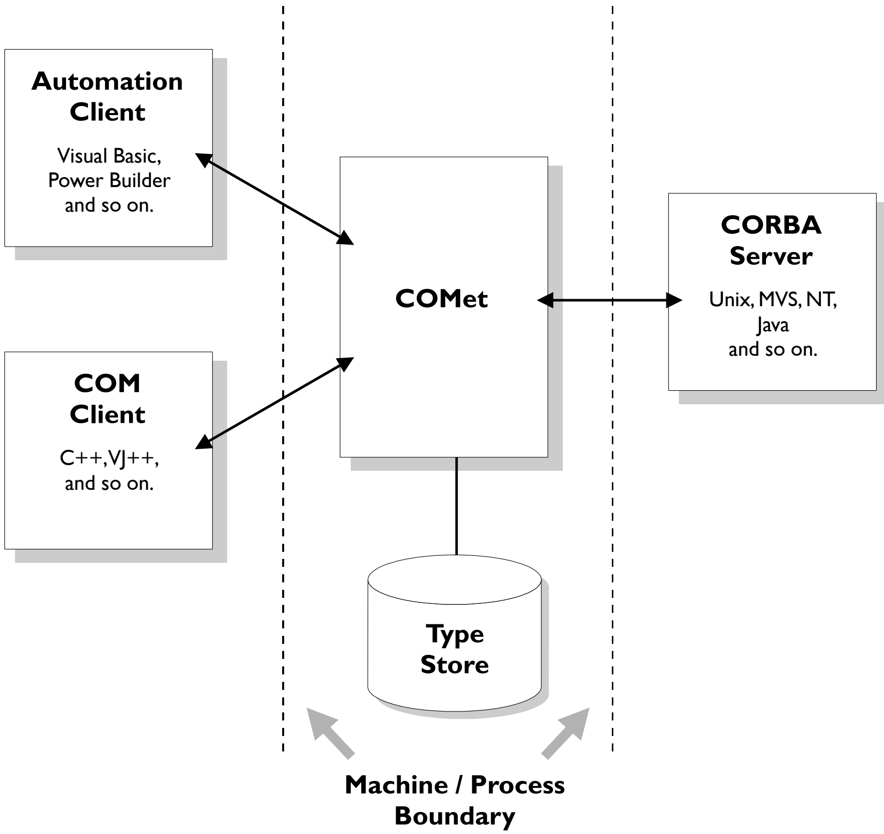

Figure 1.2: *OrbixCOMet's Implementation of the Interworking Model*

Bridge

The OrbixCOMet bridge is implemented as a set of DLLs that are capable of dynamically mapping requests between the two object models. Two-way interworking requires the bridge to provide the mappings and perform translation between CORBA and COM/Automation types.

The bridge uses another OrbixCOMet component, called the *type store*, as shown in Figure 1.2 on page 7. The type store provides information to the bridge about all the COM/Automation and CORBA types in your system. It holds a cache of all type information in a neutral binary format. Refer to "Managing the Type Store" on page 365 for more details about the workings of the type store.

As shown in Figure 1.3, a view object in the bridge contains both a COM/Automation object interface and an Orbix object interface. This means the bridge can expose an appropriate COM/Automation or CORBA interface to its clients. The bridge is not involved in requests sent between clients and servers of a single object model.

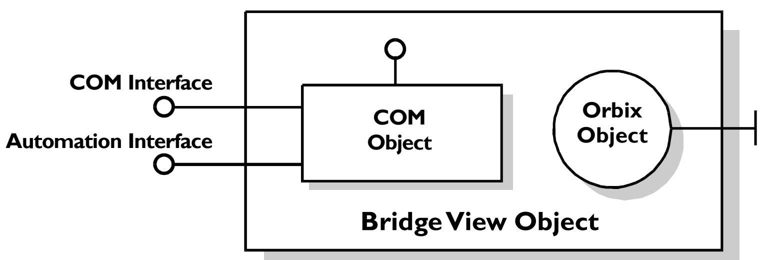

Figure 1.3: *An OrbixCOMet Bridge View Object*

Automation Client

An Automation client can use OrbixCOMet to communicate with a CORBA server. 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

A COM client can use OrbixCOMet to communicate with a CORBA server. 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

A CORBA server can be contacted by COM or Automation clients, using OrbixCOMet. This is a normal CORBA server written in any language and running on any platform supported by an ORB. (Depending on the location of the OrbixCOMet bridge in your system, the CORBA server might need to be running on Windows NT. Refer to "Usage Models and Bridge Locations" on page 11 for more details.)

If you use OrbixCOMet to develop a CORBA server, it must be written in an Automation-based language such as Visual Basic or PowerBuilder.

CORBA Client

A CORBA client can use OrbixCOMet to communicate with a COM or Automation server. This is a normal CORBA client written in any language and running on any platform supported by an ORB. (Depending on the location of the OrbixCOMet bridge in your system, the client platform might need to be running on Windows NT. Refer to "Usage Models and Bridge Locations" on page 11 for more details.)

If you use OrbixCOMet to develop a CORBA client, it must be written in a COM-based language such as C++, or an Automation-based language such as Visual Basic or PowerBuilder.

Automation Server

An Automation server can be contacted by CORBA clients, using OrbixCOMet. This is a regular Automation server written in Visual Basic, PowerBuilder, Excel, MFC, or any other Automation-compatible language.

COM Server

A COM server can be contacted by CORBA clients, using OrbixCOMet. This is a pure COM server written in C++ or any language that supports COM servers.

2

Usage Models and Bridge Locations

You can use OrbixCOMet to develop and deploy distributed 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 or Automation client that calls objects in a CORBA server, and a CORBA client that calls objects in a COM or Automation server. This chapter explains how OrbixCOMet supports these usage models.

Note: Refer to "Deploying an OrbixCOMet Application" on page 139 for more details and examples of the various ways you can use OrbixCOMet when deploying your applications.

Automation Client to CORBA Server

This section describes a usage model involving an Automation client and a CORBA server. Figure 2.1 shows a graphical overview of this usage model.

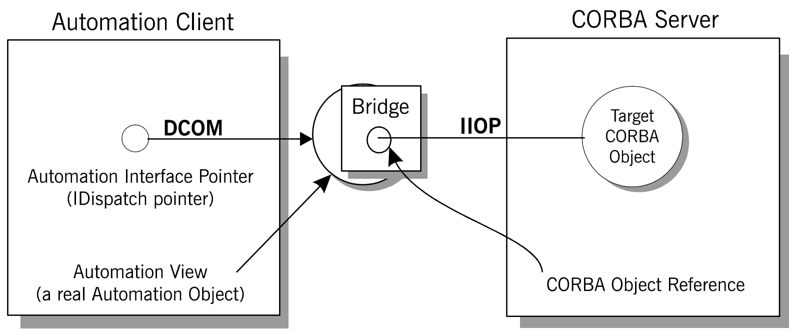

Figure 2.1: *Automation Client to CORBA Server*

The Automation Client

Using this model, an Automation client can use the DCOM protocol to communicate with a CORBA server. The client in Figure 2.1 can make method calls on an Automation view object in the bridge, using an `IDispatch` pointer. The bridge makes a corresponding operation call on the target object in the CORBA server, using a CORBA object reference.

An Automation client can use dual interfaces instead of straight `IDispatch` interfaces. The use of either of these determines whether early binding or late binding is allowed. (Refer to "Implementing CORBA Clients in Automation" on page 76 for more details.)

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

The client does not need to know that the target object is a CORBA object. An Automation client can be written in any Automation-based programming language, such as Visual Basic or PowerBuilder.

The CORBA Server

The CORBA server presents an OMG IDL interface to its objects. The server application can be developed (or already exist) on platforms other than Windows NT. However, if you choose to locate the bridge on the server machine, the server must be running on Windows NT. 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 (in this case, the server must be running on Windows NT), 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.

If the bridge is not located on the client machine, an Automation client always uses DCOM to communicate with the bridge. The bridge always uses IIOP to communicate with a CORBA server.

COM Client to CORBA Server

This section describes a usage model involving a COM client and a CORBA server. Figure 2.2 shows a graphical overview of this usage model.

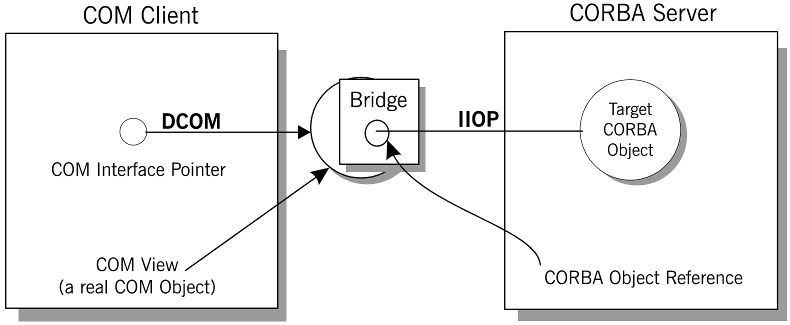

Figure 2.2: *COM Client to CORBA Server*

The COM Client

Using this model, a COM client can use the DCOM protocol to communicate with a CORBA server. The client in Figure 2.2 makes method calls on a COM view object in the bridge, using a COM interface pointer. The bridge makes a corresponding operation call on the target object in the CORBA server, using 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 does not need to 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 CORBA Server

The CORBA server presents an OMG IDL interface to its objects. The server application can be developed (or already exist) on platforms other than Windows NT. (However, if you choose to locate the bridge on the server machine, the server must be running on Windows NT.) 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 (in this case, the server must be running on Windows NT), 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.

If the bridge is not located on the client machine, a COM client always uses DCOM to communicate with the bridge. The bridge always uses IIOP to communicate with a CORBA server.

CORBA Client to COM or Automation Server

This section describes usage models involving a CORBA client and a COM or Automation server. Figure 2.3 and Figure 2.4 show a graphical overview of these usage models.

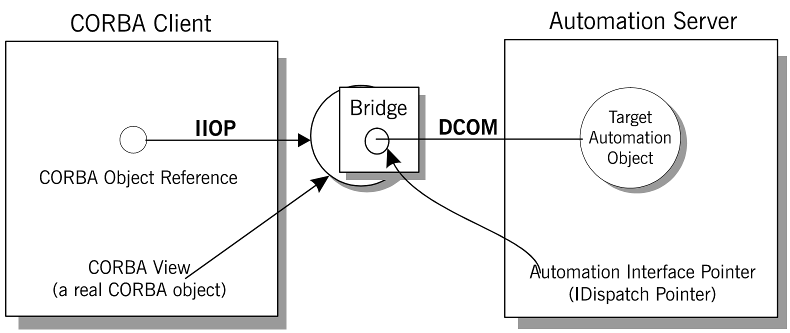

Figure 2.3: *CORBA Client to Automation Server*

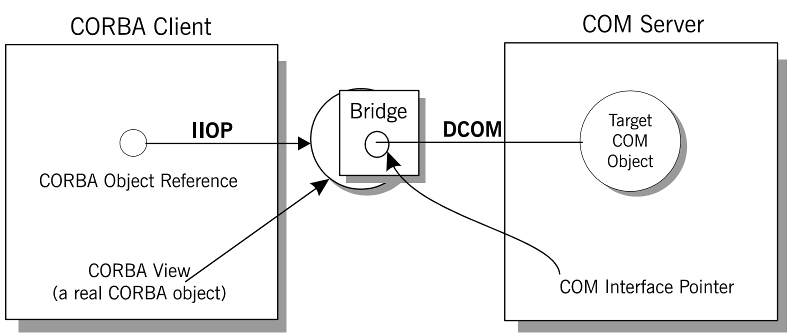

Figure 2.4: *CORBA Client to COM Server*

The CORBA Client

Using this model, a CORBA client can use the CORBA IIOP protocol to communicate with a COM or Automation server. The client makes method calls on a CORBA view object in the bridge, using a CORBA object reference. The bridge makes a corresponding operation call on the target object in the COM or Automation server, using 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 does not need to know that the target object is a COM or Automation object. A CORBA client can be developed on any platform including UNIX, Windows NT, and Windows 98. (However, if you choose to locate the bridge on the client machine, the client must be running on Windows NT). It can be written in any language supported by a CORBA implementation, such as C++, Java, or any Automation-based language.

The COM or Automation Server

The COM or Automation server presents a COM IDL 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 (in this case, however, the client must be running on Windows NT), on the COM or 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 or Automation client, because it translates CORBA operation calls into COM or Automation method calls on the server.

A CORBA client always uses IIOP to communicate with the bridge. The bridge always uses DCOM to communicate with a COM or Automation server.

Part II

Programmer's Guide

3

Getting Started

This chapter is provided as a quick means to getting started in application programming with OrbixCOMet. It explains the basics you need to know to develop a simple OrbixCOMet application, using PowerBuilder or Visual Basic, where an Automation client can invoke on an existing CORBA server. It also provides an introduction to writing COM clients, using OrbixCOMet.

Subsequent chapters provide further details about using OrbixCOMet for application development. Refer to “OrbixCOMet Configuration” on page 353 for details about how to configure your system.

As already explained in “How OrbixCOMet Implements the Interworking Model” on page 6, OrbixCOMet is a fully dynamic bridge that enables two-way integration between COM/Automation and CORBA applications. Using OrbixCOMet simply involves configuring the bridge to pick up the correct type information that you supply for each interface or complex type that your applications use. Refer to “Priming the OrbixCOMet Type Store Cache” on page 40 for details.

Server-Side Requirements

OrbixCOMet requires no code changes to existing CORBA servers. You can simply register the server executable with the Orbix Implementation Repository, using the `putit` command.

The following is an example of how to use `putit` to register the supplied `grid` demonstration server, where *`install-dir`* represents the Orbix installation directory:

```
putit grid install-dir\demos\COMet\corbasrv\grid\server.exe
```

You should also ensure that the Interface Repository (`IFR`) server (and the Naming Service, if you want to use it from your application) is registered in the Implementation Repository. This allows the daemon to launch these servers automatically, if necessary. Refer to the Orbix C++ documentation set for more details about registering servers.

Registering OMG IDL Type Information

OrbixCOMet is a purely dynamic bridge between COM/Automation and CORBA that is driven by type information derived from either a CORBA Interface Repository or Automation type libraries. The example in this chapter uses the Interface Repository. You must register your OMG IDL in the Interface Repository, using the `putidl` command. The following is an example of how to register the supplied `grid.idl` file that contains the `grid` interface:

```
putidl install-dir\demos\COMet\corbasrv\grid\grid.idl
```

The Orbix daemon can launch the `IFR` server automatically, if it is not already running when you run `putidl`. (This is assuming the `IFR` server has been registered with the Orbix daemon in the Implementation Repository.)

Note: This chapter assumes you are using Orbix as your server-side object request broker (ORB). Details about using other ORBs on your server side are provided later in this guide.

Implementing Automation Clients

This section describes how to develop a simple Automation client, using PowerBuilder and Visual Basic, that can communicate with a CORBA server. The supplied CORBA server implements a grid object, and the Automation client can communicate with the server to get and set values in the grid.

Writing a Client Using PowerBuilder

This section describes the development of a simple client application, using PowerBuilder with OrbixCOMet. You can find this example in *install-dir*\demos\COMet\PB\grid. The client interface is as shown in Figure 3.1.

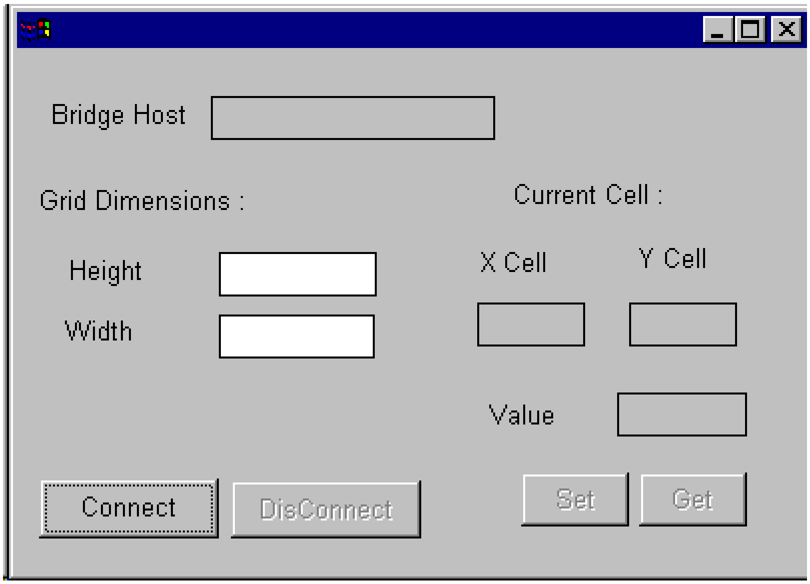

Figure 3.1: *PowerBuilder Client for the OrbixCOMet Grid Demonstration*

The following subsections describe the programming steps to develop this PowerBuilder client. Any filenames mentioned in this section refer to files contained in the *install-dir*\demos\COMet\PB\grid directory.

Global Data

Start by declaring the following global data:

```
// PowerBuilder
OleObject bridge
OleObject fact
OleObject grid_client
```

Connecting to the Orbix Grid Server from PowerBuilder

The following code is executed when you select the **Connect** button on the screen shown in Figure 3.1 on page 23:

```
// Powerscript
// create the CORBA factory object
fact = CREATE OleObject

//DCOM on the wire
//bridge = CREATE OleObject
//bridge.ConnectToNewObject("IT_CCIExWrap.IT_CCIExWrap.1")
//fact = bridge.IT_CreateRemoteFactory(server_name.Text)

// IIOP on the wire (requires bridge on client machine)
// the CORBA.Factory object may be created in the normal
// fashion
fact.ConnectToNewObject("CORBA.Factory")

// Exception parameter in case a CORBA exception occurs
OleObject ex
ex = CREATE OleObject

grid_client = CREATE OleObject
grid_client = fact.GetObject("grid:grid_marker:gridSvr" +
    server_name.Text, BYREF ex)

height_val.Text = string( grid_client.Height )
width_val.Text = string( grid_client.Width )

connect_button.Enabled = False
unplug_button.Enabled = True
set_button.Enabled = True
get_button.Enabled = True
```

The preceding code results in the creation of an instance of a `CORBA.Factory` object. After a `CORBA.Factory` object has been returned, a particular object is requested by calling the `GetObject()` method on the CORBA factory. (Refer to "DICORBAFactory" on page 189 for a description of `DICORBAFactory`. Alternatively, examine *`install-dir`*`\COMet\idl\ItStdAuto.idl`.)

Obtaining a Reference to a CORBA Object

The (D)ICORBAFactory interface contains a GetObject() method that allows a client to obtain references to CORBA objects. The OMG *COM/CORBA Interworking* specification at www.omg.org defines the (D)ICORBAFactory interface, and 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 GetObject() parameter string. In OrbixCOMet, the parameter to GetObject() can take either of the following formats:

- OrbixCOMet format:

  "*interface*:*marker*:*server*:*host*"

- Tagged format:

  "*interface*:*TAG*:*Tag data*"

  TAG can be either of the following:

IOR	In this case, *Tag data* is the hexadecimal string for the stringified IOR. For example: fact.GetObject("employee:IOR:123456789…")
NAME_SERVICE	In this case, *Tag data* is the Naming Service compound name separated by ".". For example: fact.GetObject("employee:NAME_SERVICE: IONA.employees.PD.Tom")

Note: If the interface is scoped (for example, "Module::Interface"), the interface token is "Module/Interface".

Disconnecting

When disconnecting, it is important to release all references to objects in the bridge, to allow the process to terminate. In the `grid` demonstration, this is performed by the following subroutine:

```
// PowerBuilder
grid_client.DisconnectObject()
DESTROY grid_client
fact.DisconnectObject()
DESTROY fact
bridge.DisconnectObject()
DESTROY bridge
```

Writing a Client Using Visual Basic

This section describes the development of a simple client application, using Visual Basic with OrbixCOMet. You can find this example in:

```
install-dir\demos\COMet\VB\grid
```

The client interface is as shown in Figure 3.2 on page 27.

The following subsections describe the programming steps to develop this Visual Basic client. Any filenames mentioned in this section refer to files contained in the *`install-dir`*`\demos\COMet\VB\grid` directory.

Global Data

Start by declaring the following global data:

```
' Visual Basic
Dim bridge As Object
Dim fact As Object
Dim gridDisp As Object
```

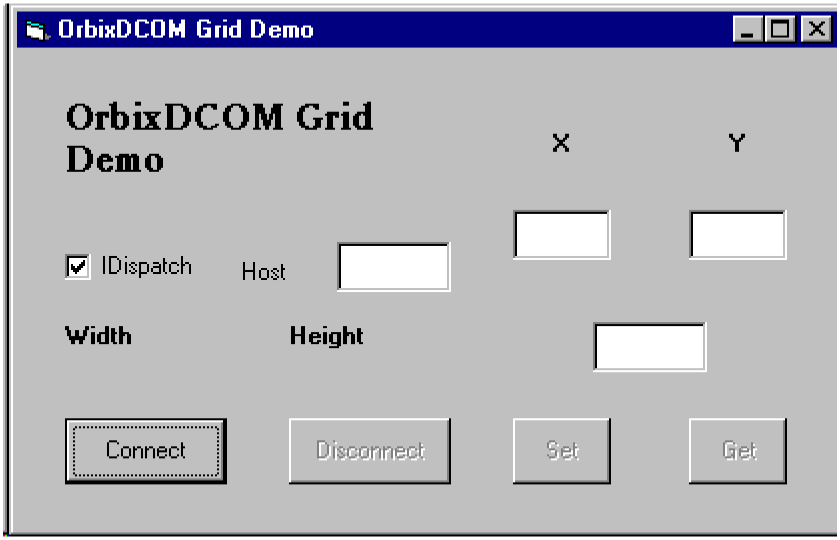

Figure 3.2: *Visual Basic Client for the OrbixCOMet Grid Demonstration*

Connecting to the Orbix Grid Server from Visual Basic

The following code is executed when you select the **Connect** button on the screen shown in Figure 3.2:

```
' Visual Basic
Private Sub Connect_Click()

      ' DCOM on the wire - see later
      ' Set bridge =
      ' CreateObject("IT_CCIExWrap.IT_CCIExWrap.1")
      ' Set fact =
      ' bridge.IT_CreateRemoteFactory(bridgeHost.Text)

      ' IIOP on the wire
      Set fact = CreateObject("CORBA.Factory")
      Set gridDisp = fact.GetObject("grid:grid_marker:
          gridSvr:" & server_name.Text)
```

```
        width_val.Caption = gridDisp.Width
        height_val.Caption = gridDisp.Height
        Command1.Enabled = False
        Command2.Enabled = True
        SetButton.Enabled = True
        GetButton.Enabled = True
End Sub
```

The preceding code results in the creation of an instance of a `CORBA.Factory` object. After a `CORBA.Factory` object has been returned, a particular object is requested by calling the `GetObject()` method on the factory. (Refer to "DICORBAFactory" on page 189 for a description of `DICORBAFactory`. Alternatively, examine *install-dir*`\COMet\idl\ItStdAuto.idl`.)

Obtaining a Reference to a CORBA Object

Refer to "Obtaining a Reference to a CORBA Object" on page 25 for details.

Disconnecting

When disconnecting, it is important to release all references to objects in the bridge, to allow the process to terminate. In the `grid` demonstration, this is performed by the following subroutine:

```
' Visual Basic
Private Sub Disconnect_Click()
    Set gridDisp = Nothing
    Set fact = Nothing
    Set bridge = Nothing
End Sub
```

Running the Client Application

To run the client application:

1. If you are using PowerBuilder, run `grid.exe`. If you are using Visual Basic, run `vbgrid.exe`. This opens the relevant client GUI interface shown in Figure 3.1 on page 23 or Figure 3.2 on page 27.
2. Specify the hostname in the appropriate field and select **Connect**. This contacts the supplied `grid` C++ server, and obtains the width and height of the grid.

3. Type x and y values for the grid coordinates.
4. Select **Set** to modify values in the grid, or **Get** to obtain values from the grid.
5. Select **Disconnect** when you are finished.

Using DCOM On-the-Wire with OrbixCOMet

The examples provided in "Implementing Automation Clients" on page 22 have all created an instance of the CORBA.Factory object in the client's address space (that is, in-process to the client). This section describes how you can use OrbixCOMet to write applications that launch the OrbixCOMet bridge out-of-process, either on the client machine or on a remote machine.

A DLL called CCIExWrapper.dll is provided with your OrbixCOMet installation. This DLL exposes the functionality of the CoCreateInstanceEx() DCOM method to PowerBuilder, Visual Basic, and Delphi programmers. The CoCreateInstanceEx() method allows you to specify the machine on which the OrbixCOMet bridge should be launched, thus allowing use of DCOM on-the-wire. You can of course use IIOP on-the-wire instead. Both configurations are equally easy to use from the client programmer's point of view. The decision about which protocol 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.

For example, consider the following Visual Basic code, which implements a check button (inprocess) to let the user decide whether to launch the bridge in-process to the client (and therefore use IIOP on-the-wire) or out-of-process (and therefore use 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
        Set srvObj = objFactory.GetObject("grid:
            grid_marker:gridSvr:" & 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 want 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 called `custsur.exe` (found in the *`install-dir`*`\COMet\bin` directory) 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 you are 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. Using OrbixCOMet in this manner requires no configuration changes on the client's machine. Refer to "Using OrbixCOMet with Internet Explorer" on page 31 for more details.

DCOM Security

Using DCOM on-the-wire to another machine requires that DCOM security issues are addressed. Security can be dealt with by using `DCOMCNFG.EXE`, or programmatically via API security functions, or using a combination of both approaches. Refer to "DCOM Trouble-Shooting" on page 40 for details of some DCOM-only applications shipped with OrbixCOMet that you can use to experiment with configuring DCOM. However, a full treatment of COM security is outside the scope of this guide. Refer to the COM security FAQ at `http://support.microsoft.com/support/kb/articles/q158/5/08.asp` for more details.

The Surrogate Executable

As already mentioned, when the bridge is launched out-of-process, the OrbixCOMet DLLs are not hosted by the default surrogate, `DLLHOST.exe`. Instead, they are hosted by a surrogate process, `custsur.exe`, which is found in the *`install-dir`*`\COMet\bin` directory. This is indicated by the following registry value that is set during installation:

```
HKEY_CLASSES_ROOT\AppID\{A8B553C5-3B72-11CF-BBFC-444553540000}
     [DllSurrogate] = install-dir\COMet\bin\custsur.exe
```

Using OrbixCOMet with Internet Explorer

Note: Before reading this section, ensure you have read "Using DCOM On-the-Wire with OrbixCOMet" on page 29.

The `CCIExWrapper.DLL` file supplied with OrbixCOMet wraps the DCOM `CoCreateInstanceEx()`method. This DLL can be referenced in HTML files, using the `OBJECT` tag. The reference supplies attribute values that specify the object name, object location, object 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 can work.) The `CLASS ID` attribute identifies the object implementation. The syntax for this attribute 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="\\machine-name\install-dir\COMet\bin\CIExWrapper.dll"
>
</OBJECT>
```

When the HTML file is first downloaded, the `CCIExWrapper.DLL` is also retrieved and registers itself on your machine (provided you agree, of course). This allows use of OrbixCOMet from client machines, with no configuration effort required on the client's part. The only requirement is that you must configure OrbixCOMet on the server side with respect to type information, access permissions, and so on, and place a HTML file on a server. This HTML file can contain VBScript or JavaScript for calling methods on the remote CORBA objects. DCOM is used on the wire. For example, the following VBScript example is used for connecting to the `grid` object on the `"advice.iona.com"` machine, 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:grid_marker:gridSvr:" &
    server_name.Text)
      lblStatus.Value = "Obtaining dimensions…"
      sleWidth.Value  = Grid.width
      sleHeight.Value = Grid.height
      lblStatus.Value = "Connected…"
End Sub

-->
</SCRIPT>
```

You can find the full version of the preceding example in *install-dir*`\demos\COMet\ie\grid\griddemo.htm`. To use this example, you must set your Internet Explorer security settings to "medium" in your Windows Control

Panel. That is all you need to do. A security setting of “medium” means that you are prompted whenever executable content is being downloaded. You do not need to have Orbix installed. You can now open the `griddemo.htm` file located in *`install-dir`*`\demos\COMet\IE`.

You must edit the following lines in the `griddemo.htm` file, to specify the name of the machine that you want to be contacted when the demonstration is downloaded:

```
CODEBASE="\\machine-name\install-dir\COMet\bin\CIExWrapper.dll"
```

and

```
Set fact = bridge.IT_CreateInstanceEx("{A8B553C5-3B72-11CF-BBFC-
444553540000}", "machine-name")
```

or

```
Set fact = bridge.IT_CreateRemoteFactory("machine-name")
```

`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. On the other hand, the CLSID for `CORBA.Factory` is hard-coded in the implementations of `IT_CreateRemoteFactory()`.

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. This is acceptable in this case. You should be presented with a simple GUI, similar to the Visual Basic or PowerBuilder GUI screens in Figure 3.1 on page 23 and Figure 3.2 on page 27. To use the demonstration:

1. Select **Connect**.
2. Type `x` and `y` values for the grid coordinates.
3. Select **Set** to modify values in the grid, or **Get** to obtain values from the grid.
4. Select **Disconnect** when you are finished.

Automation Dual Interface Support

Some Automation controllers (for example, Visual Basic) provide the option of using either straight `IDispatch` interfaces or dual interfaces for invoking on a server. OrbixCOMet supports the use of dual interfaces. The use of dual interfaces means that client invocations can be routed directly through the vtable.[1] This is known as *early binding*, because interfaces are known at compile time. The alternative to early binding is *late binding*, where client invocations are routed dynamically through `IDispatch` interfaces at runtime.

The advantage of using dual interfaces and early binding is that it helps to avoid the `IDispatch` marshalling overhead at runtime that can be associated with late binding. (Refer to “Implementing CORBA Clients in Automation” on page 76 for more details about early and late binding.) The use of dual interfaces requires the use of a type library. If you want to use dual interfaces in an Automation client that is to communicate with a CORBA server, you must create a type library, based on the OMG IDL type information implemented by the target CORBA server. OrbixCOMet provides a type library generation tool, `ts2tlb`, which produces type libraries, based on OMG IDL type information in the OrbixCOMet type store. In this way, Automation clients can be presented with an Automation view of the target CORBA objects.

The following `ts2tlb` command creates a `grid.tlb` type library in the `IT_grid` library, based on the OMG IDL `grid` interface:

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

Refer to “Development Support Tools” on page 157 for full details about `ts2tlb` and creating type libraries from OMG IDL.

Note: Ensure your OMG IDL is registered with the Interface Repository, using `putidl`, before you add it to the type store and use `ts2tlb` to create an Automation type library from it. Refer to “Development Support Tools” on page 157 for more details.

1. The vtable is a standard feature of object-oriented programming. It is a function table that contains entries corresponding to each operation defined in an interface.

The generated type library, based on the OMG IDL `grid` interface, appears as follows when viewed using oleview:

```
[odl,…]
interface DIgrid : IDispatch {
     [id(0x00000001)]
     HRESULT _stdcall get([in] short n, [in] short m,
         [out, optional] VARIANT* excep_OBJ,
         [out, retval] long* val);
     [id(0x00000002)]
     HRESULT _stdcall set([in] short n, [in] short m,
         [in] long value,
         [out, optional] VARIANT* excep_OBJ);
     [id(0x00000003), propget]
     HRESULT _stdcall height([out, retval] short* val);
     [id(0x00000004), propget]
     HRESULT _stdcall width([out, retval] short* val);
};
```

Note: All UUIDs are generated using the MD5 algorithm specified in the OMG *COM/CORBA Interworking* specification at `www.omg.org`.

Having created a reference to the type library, it can be used in Visual Basic, for example, as follows:

```
' Visual Basic
Dim custGrid As IT_grid.DIgrid
```

For more complicated OMG IDL interfaces (for example, those that pass user-defined types as parameters), `ts2tlb` attempts to resolve all those types from the disk cache, the Interface Repository, or both. It cannot produce a type library if any of the types it looks for are not found.

Finally, if you want to register the generated type library in the Windows registry, use the supplied `tlibreg` utility. You can also use `tlibreg` to unregister a type library. Refer to “OrbixCOMet Utility Options” on page 363 for more details about `tlibreg`.

Implementing COM Clients

OrbixCOMet provides support for COM customized interfaces. It adheres to the standards laid down in the OMG *COM/CORBA Interworking* specification at `www.omg.org` for mapping CORBA data types to COM. This support is aimed primarily at C++ programmers writing COM clients who want to make use of the full set of COM types, rather than being restricted to types that are compatible with Automation. Refer to "CORBA-to-COM Mapping" on page 309 for details of the mapping rules.

Generating COM IDL Definitions from OMG IDL

COM interfaces are defined in COM IDL (a derivative of DCE IDL), which is compiled to produce marshalling code for the interface. The first step in implementing a COM client that can communicate with a CORBA server is to generate the COM IDL definitions required by the COM client from the existing OMG IDL for the CORBA objects. OrbixCOMet provides a `ts2idl` utility that produces COM IDL, based on OMG IDL type information contained in the OrbixCOMet type store. In this way, COM clients can be presented with a COM view of the target CORBA objects.

The following `ts2idl` command creates a `grid.idl` COM IDL file, based on the OMG IDL `grid` interface:

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

For more complicated OMG IDL interfaces that employ user-defined types, you can specify a `-r` option with `ts2idl`, to completely resolve those types and to produce COM IDL for them also.

Refer to "Development Support Tools" on page 157 for full details about `ts2idl` and creating COM IDL definitions from OMG IDL.

Note: Ensure your OMG IDL is registered with the Interface Repository, using `putidl`, before you add it to the type store and use `ts2idl` to create COM IDL from it. Refer to "Development Support Tools" on page 157 for more details.

The generated COM IDL, based on the OMG IDL `grid` interface, is as follows:

```
[object,…]
     interface Igrid : IUnknown
     {
         HRESULT get([in] short n,
             [in] short m,
             [out] long *val);
         HRESULT set([in] short n,
             [in] short m,
             [in] long value);
             HRESULT _get_height([out] short *val);
             HRESULT _get_width([out] short *val);
     };
#endif
```

Writing COM Clients

After generating the required COM IDL definitions from OMG IDL, you must compile the COM IDL, using the MIDL compiler. This produces the C++ interface definitions to be used within the application, and a proxy/stub DLL to marshal the customized interface. This procedure is standard practice when writing COM applications. The `-p` option with `ts2idl` is a useful labor-saving device that can produce a makefile for building the proxy/stub DLL. For example, the following command produces a `grid.mk` file as well as the `grid.idl` file already shown in "Generating COM IDL Definitions from OMG IDL" on page 36:

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

The `grid.mk` file contains information on how to build and register the DLL. You need Visual C++ 6.0, to build this marshalling DLL.

You are now ready to write your COM client code. The basic operation of the client is to:

1. Create an instance of an object that implements `ICORBAFactory`, which is the COM version of the `DICORBAFactory` interface encountered already in "Implementing Automation Clients" on page 22.
2. Call `GetObject()` to get a pointer to the `IUnknown` interface of the COM view of the CORBA object.

3. Call `QueryInterface()` to get a pointer to the customized interface, which is `Igrid` in this example, and call the relevant methods.

The following subsections take each of these steps in turn and describe how to write a COM C++ client. You can find the complete client demonstration in *install-dir*\demos\COMet\com\grid.

Creating the CORBA Factory

You can get a pointer to `ICORBAFactory` by using `CoCreateInstanceEx()` as normal. You can load the OrbixCOMet bridge in-process to your COM client, launch it as a local server (out-of-process) on the client machine, or launch it on a remote machine. (This demonstration does not show how to launch the bridge remotely, but it simply involves passing a `COSERVERINFO` parameter to `CoCreateInstanceEx()`.) In this example, the choice is made at runtime, depending on how the client is started. The CORBA server to be contacted is called `grid`, and is registered on the `advice.iona.com` machine. For example:

```
HRESULT         hr = NOERROR;
IUnknown       *pUnk = NULL;
ICORBAFactory  *pCORBAFact = NULL;
DWORD           ctx;
// our custom interface
Igrid          *pIBasic = NULL;
MULTI_QI        mqi;
// Call to CoInitialize(), some error handling
// and so on omitted for clarity

memset (&mqi, 0x00, sizeof (MULTI_QI));
mqi.pIID = &IID_ICORBAFactory;

if(bOutOfProc)
    ctx = CLSCTX_LOCAL_SERVER;
else
    ctx = CLSCTX_INPROC_SERVER;
hr = CoCreateInstanceEx (IID_ICORBAFactory, NULL,
    ctx, NULL, 1, &mqi);
CheckHRESULT("CoCreateInstanceEx()", hr, FALSE);
pCORBAFact = (ICORBAFactory*)mqi.pItf;
```

Calling GetObject()

The call to `GetObject()` looks similar to the Visual Basic example:

```
hr = pCORBAFact->GetObject("grid:grid_marker:gridSvr:
    advice.iona.com",&pUnk);
if(!CheckErrInfo(hr, pCORBAFact, IID_ICORBAFactory))
{
      pCORBAFact->Release();
      return;
}
pCORBAFact->Release();
```

Note: `CheckErrorInfo()` is a utility function used by the demonstrations to check the thread's `ErrorInfo` object after each call. This is useful for obtaining information about, for example, a CORBA system exception raised during the course of a call.

Calling QueryInterface() and Relevant Methods

Finally, you can obtain a pointer to the customized `Igrid` interface, using a call to `QueryInterface()`, and then make calls to set or get values in the grid. For example:

```
short width, height;
Igrid *pIF= 0;
hr = pUnk->QueryInterface(IID_Igrid, (PPVOID)& pIF);
if(!CheckErrInfo(hr, pUnk, IID_Igrid))
{
    pUnk->Release();
    return;
}
hr = pIF->_get_width(&width);
CheckErrInfo(hr, pIF, IID_Igrid);
cout << "width is " << width << endl;
hr = pIF->_get_height(&height);
CheckErrInfo(hr, pIF, IID_Igrid);
cout << "height is " << height << endl;
pIF->Release();
```

Priming the OrbixCOMet Type Store Cache

When you are ready to run your application for the first time, you have the option of improving the runtime performance by adding the type information required by the application to the OrbixCOMet type store. This is also called *priming* the type store cache. Priming the cache means the type store already holds the required type information in memory before you run your application. Therefore, the application does not have to contact the Interface Repository for each IDL type required, or COM type libraries for each COM IDL type required.

Priming the type store cache is a useful but optional step that is only relevant before the first run of an application that will be using type information previously unseen by the type store. On exiting an application, new entries in the memory cache are written to persistent storage and are automatically reloaded the next time the application is executed. Therefore, the cache can satisfy all subsequent queries for previously obtained type information.

Refer to "Development Support Tools" on page 157 for details about the workings of the OrbixCOMet type store cache and how to prime it.

DCOM Trouble-Shooting

The `install-dir\COMet\dcomapp` directory contains two subdirectories, called `testDll` and `testExe`. These contain pure DCOM applications that are completely independent of Orbix and OrbixCOMet. Their purpose is to allow verification of a DCOM installation on a given machine. Because they are pure DCOM only, they remove one variable from the equation when trouble-shooting is in operation. Each application has a simple server, written using ATL (active template library), and an associated Visual Basic client.

The testExe Application

The `install-dir\COMet\dcomapp\testExe` directory should look something like the following:

```
20/02/98  20:01        <DIR>    client
21/02/98  16:29        <DIR>    server
20/02/98  20:01        <DIR>    vbclient
```

The subdirectories can be described as follows:

- The `server` subdirectory contains an ATL server, the binary for which can be found in *install-dir*\COMet\bin\IT_DcomApp.exe. You can build the server from scratch in the `server` directory, if you wish. (The source is provided.) Register the server, using the following command:

  install-dir\COMet\bin:\> IT_DcomApp /regserver

- The `vbclient` subdirectory contains a simple Visual Basic client for the application. When you run the client, the test has completed successfully if the window shown in Figure 3.3 on page 41 appears. If, as is likely, you intend to use OrbixCOMet with clients and servers on different machines, you should run these tests between those machines.
- The `client` subdirectory contains a simple COM C++ client for the application.

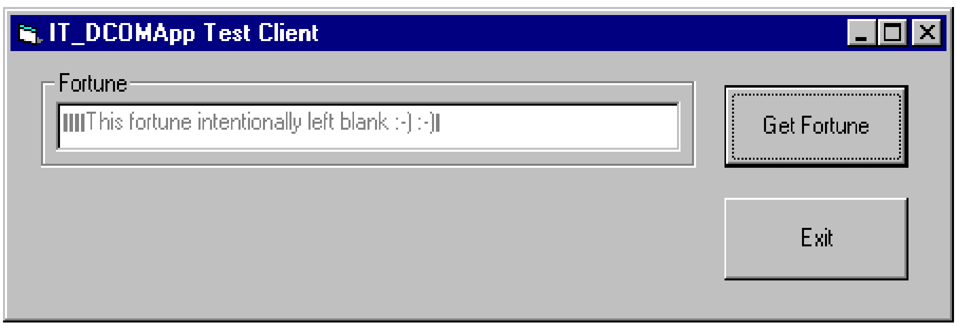

Figure 3.3: *IT_DCOMApp Test Client—Successful Operation*

If the window shown in Figure 3.3 does not appear, or if an error occurs as shown in Figure 3.4 on page 42, refer to "Miscellaneous Configuration Tips" on page 43.

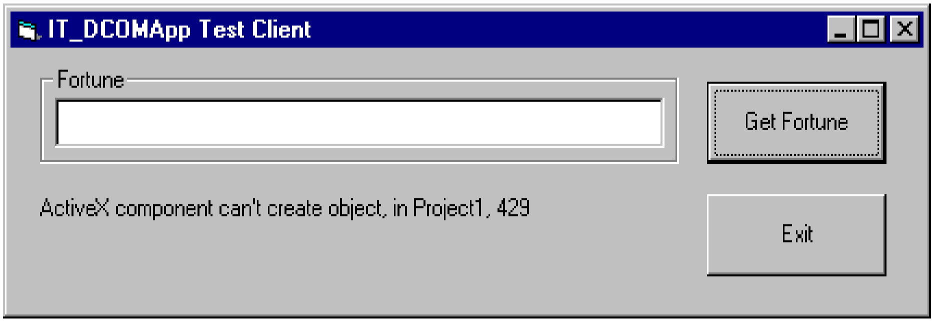

Figure 3.4: *IT_DCOMApp Test Client—Error Launching Server*

The testDll Application

The `testDll` application verifies that surrogates work correctly on your machine. You should test this if you want to use OrbixCOMet out-of-process.

To do this:

1. Use `OLEVIEW` to launch the `IT_DcomTestDLL` class. This opens the **OLE/COM Object Viewer** screen.
2. From the **Object** pulldown menu, select `CoCreateInstance` flags of `CLXCTX_INPROC_SERVER`.
3. If this test fails, refer to "Miscellaneous Configuration Tips" next.

Miscellaneous Configuration Tips

This section outlines the steps you should follow if your test does not complete successfully:

1. Verify that the server is actually registered, using `OLEVIEW` if possible.
2. If `OLEVIEW` is available, try launching the application from within `OLEVIEW`, and specify `CoCreateInstance` flags of `CLSCTX_LOCAL_SERVER`.
3. If you are using the surrogate process, use `dcomcnfg` to ensure that the **Default Authentication Level** is set to `Connect`, and the **Default Impersonation Level** is set to `Identify`.
4. On Windows NT, use the `\winnt\system32\eventvwr.exe` event viewer to look for logged DCOM events. Figure 3.5 on page 44 shows a typical example of a logged error.
5. Consult the OrbixCOMet Knowledge Base at:

   `http://www.iona.com/support/kb`
6. Consult the DCOM mailing list archive at:

   `http://microsoft.ease.lsoft.com/archives.index.html`
7. Consult the frequently asked questions about COM security at:

   `http://support.microsoft.com/support/kb/articles/`
   `q158/5/08.asp`

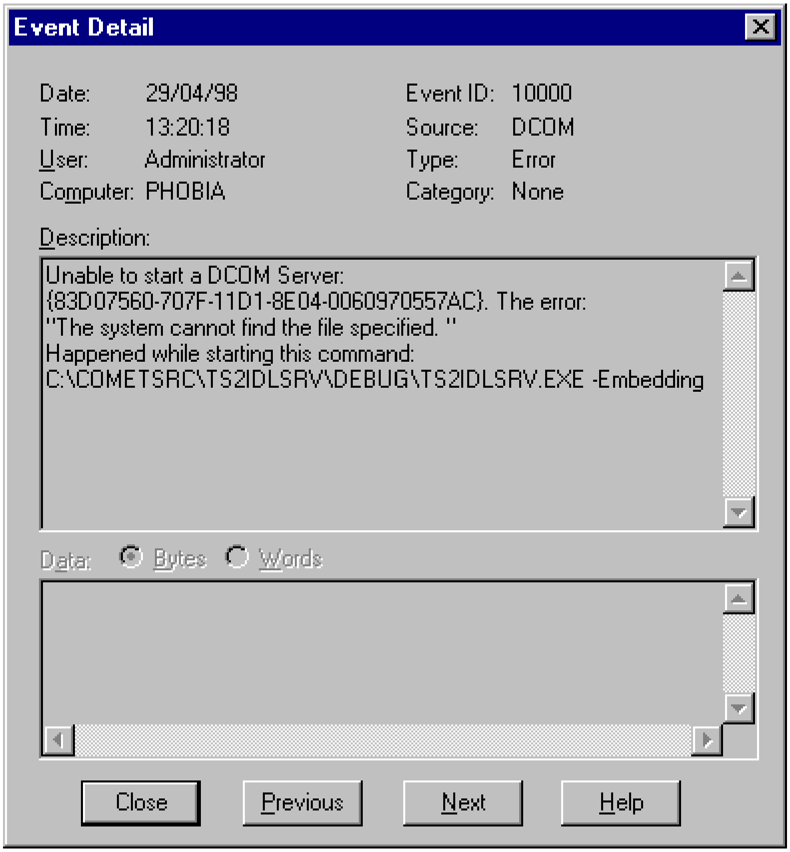

Figure 3.5: *Typical Example of a Logged Error*

4

Developing a Client in Automation

This chapter expands on what you learned in "Getting Started" on page 21. It uses the example of a distributed telephone book application to show how to write Automation clients that can communicate with an existing CORBA C++ server, using PowerBuilder and Visual Basic.

You can find versions of the Automation client application described in this chapter at the following locations, where `install-dir` represents the Orbix installation directory:

Visual Basic	`install-dir\demos\COMet\VB\PhoneBook`
PowerBuilder	`install-dir\demos\COMet\PB\PhoneBook`
Internet Explorer	`install-dir\demos\COMet\IE\PhoneBook`

The server application is implemented in C++ and its code is located in the `install-dir\demos\COMet\corbasrv\phonebook` directory. You do not need to understand how the server is implemented, to follow the examples 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 255 for more details.

The Telephone Book Example

Figure 4.1 illustrates the components of a telephone book application. The CORBA server contains an object that supports the PhoneBook interface. Your task is to implement the Automation client that will make requests on the PhoneBook object.

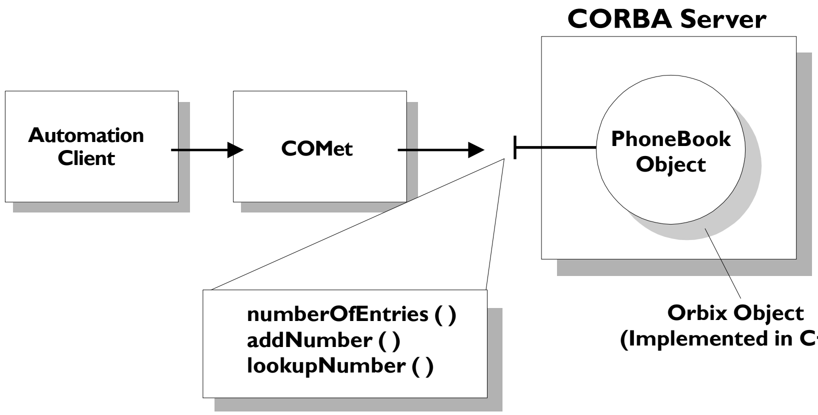

Figure 4.1: *Telephone Book Example*

"How OrbixCOMet Implements the Interworking Model" on page 6 explained that a client makes method calls on a view object in the OrbixCOMet bridge. The principal task of the Automation client in this example is to obtain a reference to an Automation PhoneBook view object in the bridge. The PhoneBook view object exposes an Automation DIPhoneBook interface, generated from the OMG IDL PhoneBook interface. (Refer to "CORBA-to-Automation Mapping" on page 271 for details of how CORBA types are mapped to Automation.) When the client makes method calls on the PhoneBook view object, the bridge forwards the client requests to the target CORBA PhoneBook object.

Creating a Type Library

"Automation Dual Interface Support" on page 34 has already explained that when using an Automation client, you have the option in some controllers (for example, Visual Basic) of using straight `IDispatch` interfaces or dual interfaces, which determines whether your application can use early or late binding. If you want to use dual interfaces, you must create a type library. In this case, you want to create an Automation client that can communicate with a CORBA server, so you must create a type library that is based on the OMG IDL interfaces exposed by the CORBA server. You can create a type library, based on existing OMG IDL information in the type store, using either the GUI or command-line version of the OrbixCOMet `ts2tlb` utility. Refer to "Development Support Tools" on page 157 for more details.

Implementing the Client

This section describes how to implement the client, using Visual Basic and PowerBuilder. The client presents the interface shown in Figure 4.2.

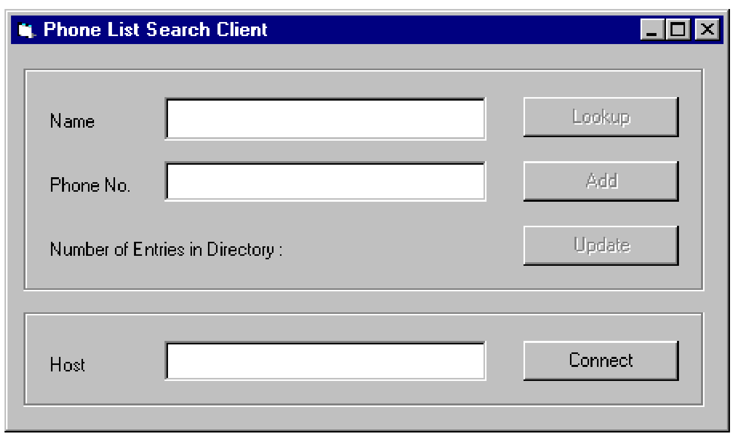

Figure 4.2: *Using the Phone List Search Client Application*

Obtaining a Reference to a CORBA Object

This section includes Visual Basic and PowerBuilder examples of the client code used to obtain 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:
    PhoneBook_marker:PhoneBookSrv:" & host.Text)
```

PowerBuilder

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

In the preceding Visual Basic and PowerBuilder examples:

1. The client first instantiates a CORBA object factory in the bridge. The CORBA object factory is a factory for creating view objects. It is assigned the `CORBA.Factory` ProgID.
2. The client then calls `GetObject()` on the CORBA object factory. It passes the name of the `PhoneBook` object in the CORBA server in the parameter for `GetObject()`. In this case, the parameter for `GetObject()` takes the following format:

   `interface:marker:server:host`

   Refer to "Obtaining a Reference to a CORBA Object" on page 25 for full details of the format of the parameter for `GetObject()`.

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. Figure 4.3 on page 49 shows how the call to `GetObject()` achieves this.

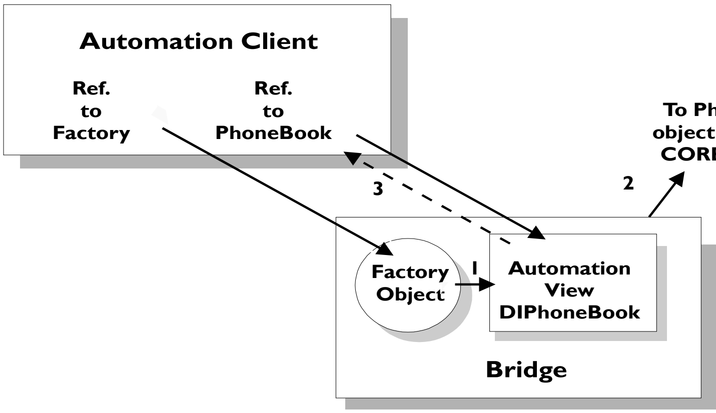

Figure 4.3: *Binding to the Phone Book Object*

In Figure 4.3, `GetObject()`:

1. Creates an Automation view object in the OrbixCOMet bridge that implements the `DIPhoneBook` dual interface.
2. Binds the Automation view object to the CORBA implementation object named in the string parameter for `GetObject()`.
3. Returns a reference to the view object.

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

The Visual Basic Client Code in Detail

This section provides a more detailed Visual Basic example of the client application. It shows how the Visual Basic code extracts shown in "Obtaining a Reference to a CORBA Object" on page 48 fit into the following steps to implement the Visual Basic client.

- General declarations.
- Creating the form.
- Connecting to the CORBA server.
- Invoking operations on the `PhoneBook` object.
- Unloading the form.

General Declarations

Declare a reference to the object factory and to the `phonebookObj` Automation view object:

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

Creating the Form

Create an instance of the CORBA object factory when the Visual Basic form is created, and assign it the `CORBA.Factory` ProgID:

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

Connecting to the CORBA Server

Implement the **Connect** button, call `GetObject()` on the CORBA object factory, and pass the name of the `PhoneBook` object as the parameter to `GetObject()`:

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

In the preceding code, the implementation of the **Connect** button connects to the PhoneBook object in the CORBA server. After the call to `GetObject()`, the client can use the `phoneBookObj` object reference to invoke operations on the target PhoneBook object in the server. This is illustrated next in "Invoking Operations on the PhoneBook Object".

Invoking Operations on the PhoneBook Object

Implement the **Add**, **LookUp**, and **Update** buttons, which call the 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

Release the CORBA object factory and the Automation view object, using the `Form_Unload()` subroutine:

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

The PowerBuilder Client Code in Detail

This section provides a more detailed PowerBuilder example of the client application. It shows how the PowerBuilder code extracts shown in "Obtaining a Reference to a CORBA Object" on page 48 fit into the following steps to implement the PowerBuilder client.

- General declarations.
- Loading the window.
- Connecting to the CORBA server.
- Invoking operations on the `PhoneBook` object.
- Unloading the window.

General Declarations

Declare global variables for the object factory and the `phonebookObj` Automation view object:

```
OleObject ObjFactory
OleObject phoneBookObj
```

Loading the Window

Create an instance of the CORBA object factory within the open event for the **Phone List Search Client** window, and assign it the `CORBA.Factory` ProgID:

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

Connecting to the CORBA Server

Implement the clicked event for the **Connect** button, call `GetObject()` on the CORBA object factory, and pass the name of the PhoneBook object as the parameter to `GetObject()`:

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

In the preceding code, the clicked event for the **Connect** button connects to the PhoneBook object in the CORBA server. After the call to `GetObject()`, the client can use the phoneBookObj object reference to invoke operations on the target PhoneBook object in the server. This is illustrated next in "Invoking Operations on the PhoneBook Object".

Invoking Operations on the PhoneBook Object

Implement the clicked event for the **Add**, **LookUp**, and **Update** buttons, which call the OMG IDL 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 object factory 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:

1. Ensure 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 Programs** group on the Windows **Start** menu.
2. Register the CORBA server with the Implementation Repository on the server's host, using `putit`. (Usually, it is not necessary to register a server, if the server has been written and registered by someone else.)

   You can use `putit` as follows:

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

   In this case, *`your_path`* represents 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.
3. Run the client.

   On the **Phone List Search Client** screen, type the server's hostname in the **Host** textbox, and select **Connect**. You can now add and look up telephone book entries.

   If your client is inactive for some time, the `PhoneBookSrv` server is timed-out and exits. It is reactivated automatically if the client issues another request.

5

Developing a Client in COM

This chapter expands on what you learned in "Getting Started" on page 21. It uses the example of a distributed telephone book application to show how to write a COM C++ client that can communicate with an existing CORBA C++ server.

You can find a version of the COM client application described in this chapter in `install-dir\demos\COMet\com\phonebook`, where `install-dir` represents the Orbix installation directory. This directory contains Visual C++ COM client code.

The CORBA server application is implemented in C++ and its code is located in the `install-dir\demos\COMet\corbasrv\phonebook` directory of your OrbixCOMet installation. You do not need to understand how the CORBA server is implemented, 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 255 for more details.

The Telephone Book Example

Figure 5.1 illustrates the components of a telephone book application. The CORBA server contains an object that supports the PhoneBook interface. Your task is to implement the COM client that will make requests on the PhoneBook object.

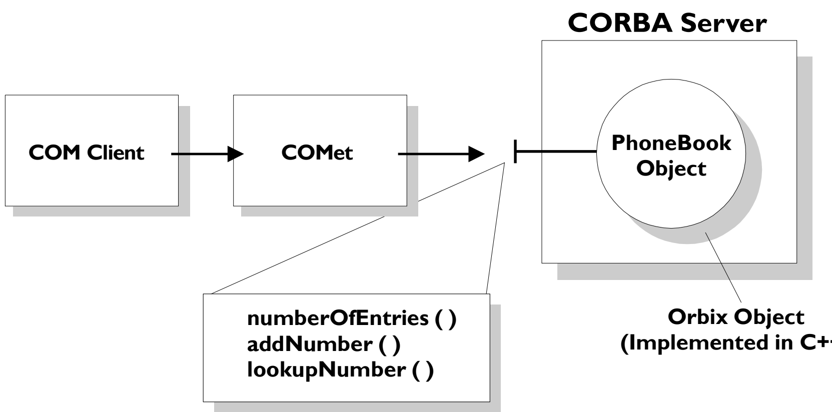

Figure 5.1: *Telephone Book Example*

"How OrbixCOMet Implements the Interworking Model" on page 6 explained that a client makes method calls on a view object in the OrbixCOMet bridge. The principal task of the COM client in this example is to obtain a reference to a COM PhoneBook view object in the bridge. The PhoneBook view object exposes the COM IPhoneBook interface, generated from the OMG IDL PhoneBook interface. (Refer to "CORBA-to-COM Mapping" on page 309 for details of how CORBA types are mapped to COM.) When the client makes method calls on the PhoneBook view object, the bridge forwards the client requests to the target CORBA PhoneBook object.

Obtaining a COM IDL Interface

"Generating COM IDL Definitions from OMG IDL" on page 36 has already explained that the normal procedure for writing a client in COM is to first obtain a COM IDL definition for the object interface. In this case, you want to create a COM client that can communicate with a CORBA server, so you must create COM IDL definitions that are based on the OMG IDL interfaces exposed by the CORBA server. You can generate COM IDL, based on existing OMG IDL information in the type store, using either the GUI or command-line version of the OrbixCOMet `ts2idl` utility. Refer to "Development Support Tools" on page 157 for details.

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 the DCOM protocol to correctly make a connection to the remote OrbixCOMet bridge from the client. You can use the supplied `ts2idl` utility to create the sources for the proxy/stub DLL. For this example, use the following command:

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

When you are generating a COM IDL 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 `Phonebookps.MK` and is located in the *`install-dir`*`\demos\COMet\COM\PhoneBook` directory.

Refer to "Development Support Tools" on page 157 to find out more about generating smart proxy DLLs and server stub code.

Implementing the Client

This section describes how to implement the client, using COM C++.

Obtaining a Reference to a CORBA Object

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

```
//General Declarations
IUnknown *pUnk=NULL;
IPhoneBook *pIPhoneBook=NULL;

//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:PhoneBook_marker:
    PhoneBookSrv:%s", hostname);

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

In the preceding example:

1. The client first instantiates a CORBA object factory in the bridge. The CORBA object factory is a factory for creating view objects. It is assigned the `IID_ICORBAFactory` IID.
2. The client then calls `GetObject()` on the CORBA object factory. It passes the name of the `PhoneBook` object in the CORBA server in the parameter for `GetObject()`. In this case, the parameter for `GetObject()` takes the following format:

   `interface:marker:server:host`

   Refer to "Obtaining a Reference to a CORBA Object" on page 25 for full details of the format of the parameter for `GetObject()`.

Note: If the interface is scoped (for example, "`Module::Interface`"), the interface token is "`Module/Interface`".

The purpose of the call to `GetObject()` is to get a pointer to the `IUnknown` interface (`pUnk`) of the COM view of the target `PhoneBook` object. Figure 5.2 shows how the call to `GetObject()` achieves this.

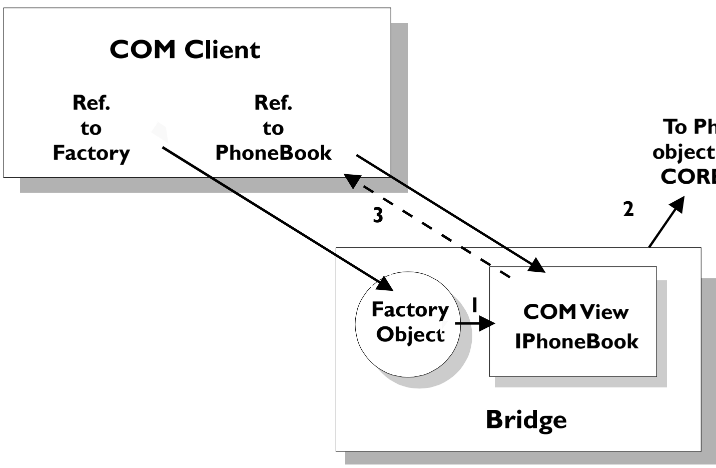

Figure 5.2: *Binding to the Phone Book Object*

In Figure 5.2, `GetObject()`:

1. Creates a COM view object in the OrbixCOMet bridge that implements the COM `IPhoneBook` interface.
2. Binds the COM view object to the CORBA `PhoneBook` implementation object named in the parameter for `GetObject()`.
3. Sets the pointer specified by the second parameter (`pUnk`) to point to the `IUnknown` interface of the COM view object.

After the call to GetObject(), the client in this example can obtain a pointer to the IPhoneBook interface (pIPhoneBook) by performing a QueryInterface() on the pointer to the IUnknown interface of the COM view object. The client can then use the pIPhoneBook object reference to invoke operations on the target PhoneBook object in the server.

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 in COM” on page 82 for more details.

The COM C++ Client Code in Detail

This section provides a more detailed example of the COM C++ client application, using Visual C++ 6.0. It shows how the code extracts shown in “Obtaining a Reference to a CORBA Object” on page 58 fit into the following steps to implement the COM C++ client:

- Include statements.
- General declarations.
- Connecting to the CORBA factory.
- Connecting to the CORBA server.
- Invoking operations on the PhoneBook object.

Includes

Include the phoneBook.h header file created from the COM IDL file, which was generated from the OMG IDL for the CORBA object in the type store:

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

General Declarations

Declare a reference to the CORBA object factory and to a PhoneBook COM view object:

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

Connecting to the CORBA Factory

Use the DCOM `CoCreateInstanceEx()` call to create a remote instance of the CORBA object factory on the client machine, and assign it the `IID_ICORBAFactory` IID:

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

Connecting to the CORBA Server

Call `GetObject()` on the CORBA object factory, and pass the name of the PhoneBook object as the parameter:

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

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

After the call to `GetObject()`, the client in this example can obtain a pointer to the `IPhoneBook` interface (`pIPhoneBook`) by performing a `QueryInterface()` on the pointer to the `IUnknown` interface of the COM view object. The client can then use the `pIPhoneBook` object reference to invoke operations on the target PhoneBook object in the server. This is illustrated next in “Invoking Operations on the PhoneBook Object”.

Invoking Operations on the PhoneBook Object

The following code shows how to invoke operations on the PhoneBook object in the CORBA server, to add a number to the telephone book, and look up entries:

```
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

To run the client:

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 Programs** group on the Windows **Start** menu.
2. Register the CORBA server with the Implementation Repository on the server's host, using `putit`. (Usually, it is not necessary to register a server, if the server has been written and registered by someone else.)

   You can use `putit` as follows:

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

   In this case, *your_path* represents 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.
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
   ```

6

Implementing CORBA Clients

This chapter is aimed at CORBA programmers who want to implement CORBA clients, using Automation-based tools such as Visual Basic and PowerBuilder, and COM-based tools such as C++.

The topics covered in this chapter include:

- How programs communicate with the ORB to obtain services or to modify the ORB's default behavior.
- Obtaining object references.
- The interworking 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 shows how to implement Visual Basic, PowerBuilder and COM C++ client examples for the bank server that is developed in "Implementing CORBA Servers" on page 99.

Interfaces to the ORB

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

- `(D)IORBObject`

  These interfaces contain a set of methods defined by the *COM/CORBA Interworking* specification. These methods provide clients with access to the operations on the ORB pseudo-object, and allow a client to request the ORB to perform some action.

  `(D)IORBObject` include 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. It also contains methods that allow a client to obtain an object reference through which a component of the ORB (for example, the Interface Repository or one of the CORBA services) can be used.

- `(D)IOrbixORBObject`

  These interfaces contain all the methods contained in the compliant `(D)IORBObject` interfaces along with a set of methods that provide access to OrbixCOMet-specific features for controlling the ORB and requesting the ORB to perform some action.

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

Refer to "OrbixCOMet API Reference" on page 181 for a full description of `(D)IORBObject` and `(D)IOrbixORBObject`.

The ORB has the `CORBA.ORB.2` ProgID. The code examples in the following subsections show 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(False)
```

COM C++

```
// Access to  IOrbixORBObject is via IORBObject
IORBObject* poOrb = NULL;
IOrbixORBObject *poOrbixOrb = NULL;

mqi.pIID = &IID_IORBObject;

hr = CoCreateInstanceEx(IID_IORBObject, NULL, ctx,
    NULL, 1, &mqi);
CheckHRESULT("CoCreateInstanceEx IID_IORBObject", hr, FALSE);

poOrb = (IORBObject*)mqi.pItf;

hr = poOrb->QueryInterface(IID_IOrbixORBObject,
    (void**) &poOrbixOrb);
CheckHRESULT("QueryInterface IORBObject for IID_IOrbixORBObject",
    hr, FALSE);
poOrb -> Release ();

BOOLEAN bRetVal = FALSE ;
hr = poOrbixOrb -> PingDuringBind (bRetVal, &bRetVal);
CheckHRESULT("PingDuringBind", hr, FALSE);
```

Obtaining 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 DICORBAFactory and ICORBAFactory interfaces, which provide the GetObject() and CreateObject() methods to allow a client to obtain references to CORBA objects.

GetObject()

The COM IDL definition for GetObject() is as follows:

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

As explained in "Developing a Client in Automation" on page 45 and "Developing a Client in COM" on page 55, 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. The parameter string can take either of the following formats:

- "*interface*:*marker*:*server*:*host*"
- "*interface*:*TAG*:*Tag data*"

The components of the string can be described 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.
server	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.
host	This is the Internet hostname 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.
TAG	Two types of *TAG* are allowed. Each type has a different form of *Tag data*. Valid *TAG* types are: • `IOR`—In this case, the *Tag data* is the hexadecimal string for the stringified IOR. For example: `fact.GetObject("employee:IOR:123456789…")` • `NAME_SERVICE`—In this case, the *Tag data* is the Naming Service compound name separated by "`.`". For example: `fact.GetObject("employee:NAME_SERVICE: IONA.employees.PD.Tom")`

CreateObject()

The COM IDL defintion for `CreateObject()` is as follows:

```
// COM IDL
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 examples, assume that the client wants to bind to a `Bank` object that is registered with the name `Commercial.Trust`.

Note: The following code examples create an IDL sequence of `NameComponents` to construct a `CosNaming::Name`. Refer to "CORBA-to-Automation Mapping" on page 271 and "CORBA-to-COM Mapping" on page 309 for more details of how to create an OMG IDL sequence in an Automation or COM application.

Visual Basic

The following is a Visual Basic example:

```
' Visual Basic
Dim objFactory as DICORBA_Orbix.DICORBAFactory
Set objFactory = CreateObject("CORBA.Factory")

'Create a CosNaming::Name sequence of Name Components
Dim bankName as Object
Set bankName = objFactory.CreateType(Nothing, "CosNaming/Name")

'Init the CosNaming::Name sequence to store 2 Name Components
bankName.Count = 2

'Populate each Name Component in the sequence
bankName(0).id = "Commercial"
bankName(0).kind = ""
bankName(1).id = "Trust"
bankName(1).kind = ""
```

PowerBuilder The following is a PowerBuilder example:

```
//PowerBuilder
//Create an empty CosNaming::Name sequence
bankName = CREATE OleObject
bankName = ObjFactory.CreateType(Nothing,"CosNaming/Name")

//Initialize the sequence, to store 2 Name Components
bankName.Count = 2

//Populate each NameComponent in the sequence
bankName.getitem(0).id = "Commercial"
bankName.getitem(0).kind = ""
bankName.getitem(1).id = "Trust"
bankName.getitem(1).kind = ""
```

Refer to "Creating Constructed OMG IDL Types" on page 283 for details of how to use `CreateType()`.

COM C++ The following is a COM C++ example:

```
// COM C++
// Create an empty sequence of CosNaming::NameComponents
CosNaming_Name bankName;
CosNaming_NameComponent BankNameComp;

// Initialize the sequence, to store 2 Name Components
bankName.cbMaxSize = 2;
bankName.cbLengthUsed = 2;
bankName.pValue = new CosNaming_NameComponent
    [bankName.cbLengthUsed];

// Populate each Name Component in the sequence
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 of the object in the Naming Service. This section provides code examples showing how to do this.

Visual Basic The following is a Visual Basic example:

```
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(".NameService")

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 77 for details.)

PowerBuilder The following is a PowerBuilder example:

```
OleObject ObjFactory
ObjFactory = CREATE OleObject
OleObject theORB
theORB = CREATE OleObject

myNS = CREATE OleObject
myNS = ObjFactory.GetObject(".NameService")

NSObj = myNS.resolve(bankName)

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

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

COM C++

The following is a COM C++ example. In this case, the desired interface is obtained, using `QueryInterface()`, after you have called `Resolve()`:

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

hr = pCORBAFact->GetObject(".NameService", &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 first obtains a reference to an object by binding to the object via `(D)ICORBAFactory::GetObject()` or `(D)ICORBAFactory::CreateObject()`, or by using the Naming Service. This object is known as a root object. 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 a reference to the Automation or COM view is returned to the client. The following example, taken from a Visual Basic client of the bank server, illustrates this method:

```
'Visual Basic
Dim fact As CORBA_Orbix.DICORBAFactoryEx
Dim bankObj As Object
Dim accountDisp As Object
…
Set fact = CreateObject("CORBA.Factory")
Set bankObj = fact.GetObject("bank:bank_marker:bank:" +
    CorbaServer.Text)
…
'Get an object reference as a return value
Set accountDisp = bankObj.newAccount(Namebox.Text, ex)
…
'Use the object reference
accountDisp.makeLodgement(Amount.Text)
Balance.Caption = accountDisp.Balance
```

A more complete version of the code is provided in "A Visual Basic Client Program" on page 78.

Interworking Interfaces on Objects

Orbix objects support the interface defined in their IDL file. All Orbix objects also support the following interfaces:

(D)`ICORBAObject`	Support for these interfaces is mandated by the *COM/CORBA Interworking* specification. 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, find the object's marker name, close the underlying communications connection to the server, and determine whether the communications channel between the client and server is open.

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

Refer to "OrbixCOMet API Reference" on page 181 for details of all interfaces supported in OrbixCOMet.

Implementing CORBA Clients in Automation

This section provides further details of how to use Automation to implement a client program that can act like a CORBA client of a CORBA server.

Late Binding

Late (or dynamic) binding is the assignment of types to variables at runtime. It involves the use of the IDispatch interface on an Automation object. Late binding means that all invocations through the object require the parameters to be marshalled through IDispatch, and then to CORBA.

Early Binding

Early (or static) binding is the assignment of types to variables at compile time. If you make a call on an early bound object, you avoid the IDispatch marshalling overhead. This improves performance, most notably when the bridge is loaded in-process to your client application.

The code examples in "Developing a Client in Automation" on page 45 use late binding (via the IDispatch interface) and declare 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 the DIAccount type, declare a reference, accountObj, as follows:

```
' Visual Basic
Dim accountObj As IT_Library_bank.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`. This section shows how clients can obtain a `CheckingAccount` interface pointer.

Visual Basic The following is a Visual Basic example of how to narrow object references:

```
Dim currentAccountDisp As IT_Library_bank.DIcurrentAccount
Dim accountDisp As IT_Library_bank.DIaccount
Dim orb As CORBA_Orbix.DIOrbixORBObject
…
'Obtain an account ref.
Set accountDisp = …
…
'Is it actually a current account ?
Set currentAccountDisp = accountDisp

If currentAccountDisp Is Nothing Then
    ' Narrow Failed
EndIf
```

PowerBuilder The following is a PowerBuilder example of how to narrow object references:

```
// Example of explicit narrow in a late bound IDispatch client
OleObject orb
orb = CREATE oleObject

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

OleObject ObjAccount

//Get Account object
ObjAccount = …

OleObject ObjCurrentAccount
ObjCurrentAccount = orb.Narrow ("currentAccount",ObjAccount)
If isNull ( ObjCurrentAccount ) Then
    // Narrow failed
    …
End If
```

Note: Refer to the entry for `DIOrbixObject::Narrow()` in "OrbixCOMet API Reference" on page 181 for an alternative way of narrowing an object reference.

A Visual Basic Client Program

This section shows the code for a Visual Basic client of the `bank` server that is developed in "Implementing CORBA Servers" on page 99. The code in this section is based on the **Bank** form in Figure 6.1 on page 79.

The `bank` server presents the following interface to its clients:

```
interface account {
    readonly attribute float balance;

    void makeLodgement (in float f);
    void makeWithdrawal (in float f);
};

interface currentAccount : account {
    readonly attribute float overdraftLimit;
};

interface bank {
    exception reject {string reason;};

    account newAccount (in string name) raises (reject);
    currentAccount newCurrentAccount(in string name,
        in float limit) raises (reject);
    void deleteAccount (in account a);

};
```

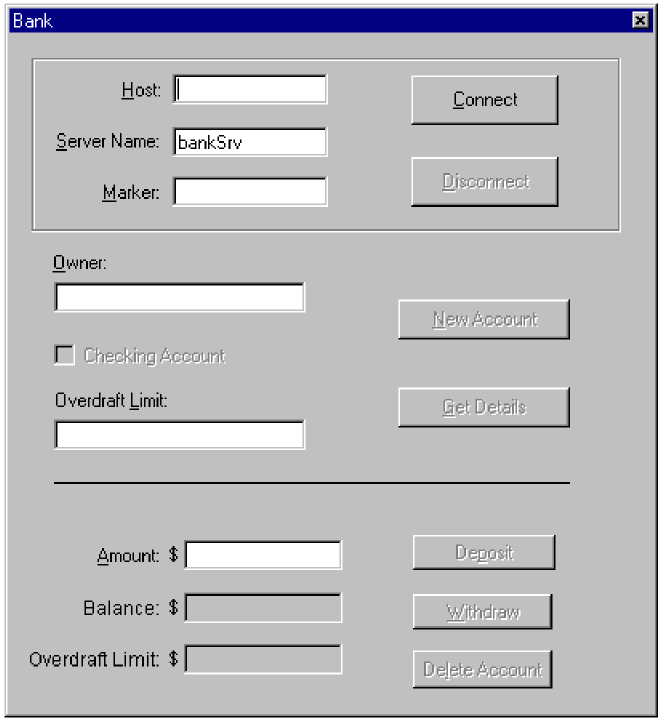

Figure 6.1: *Bank Form Presenting the User's View of the Bank Service*

General Declarations

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

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. Refer to "Creating a Type Library" on page 171 for more details of how to create a type library.

Creating the Form

The `Form_Load()` subroutine, which is called when the Bank form is loaded, creates a CORBA object factory in the bridge, which is used to create Automation views.

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

Connecting to the CORBA Server

```
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
```

In this case, when a user selects the **Connect** button in Figure 6.1 on page 79, the client connects to the bank server on the host named in the **Host** textbox, and uses the `DICORBAFactory::GetObject()` method to bind to the `Bank` object whose marker is specified in the **Marker** textbox.

It is important to handle errors that might be raised by the call to `GetObject()`. A call to `GetObject()`, or any other remote call, might 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 to Automation exceptions by the bridge. (Refer to "Exceptions" on page 291 for more details.) In Visual Basic, these exceptions can be trapped, using the `On Error` statement, and they can be handled, using the standard `Visual Basic Err` object. "Exception Handling" on page 109 explains CORBA exceptions, and alternative ways of handling them in a client.

Invoking Operations on Remote CORBA Objects

The following example shows a `newAcc_Click()`subroutine that responds to user requests to create bank accounts. The IDL definitions specify that the `Bank::newAccount()` operation can raise the `Bank::Reject` user exception, if the bank fails to create an account. In the following code, this exception is trapped using the `On Error` statement:

```
Private Sub newAcc_Click()
    On Error GoTo errorTrap

    If Namebox.Text = "" Then
        MsgBox("Enter Account Owner's Name")
        Exit Sub
    End If

    Set accountDisp = bankObj.newAccount(Namebox.Text)
    makeD.Enabled = True
    deleteAcc.Enabled = True
    newAcc.Enabled = False

    Exit Sub

errorTrap:
        MsgBox("Error: " & Err.Description & " in " & Err.Source)
        Err.Clear
        Resume Next

End Sub
```

"Exception Handling" on page 109 shows a better way to handle this exception that provides more information to the user.

Disconnecting from the CORBA Server

Release the views in the bridge when the user disconnects from the bank server:

```
Private Sub cmdDisconnect_Click()
    …
    Set bankObj = Nothing
    Set bankAccount = Nothing
End Sub
```

Exiting the Application

Release the CORBA object factory when the user exits the application:

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

Implementing CORBA Clients in COM

This section provides further details of how to use COM C++ to implement a client program that can act like a CORBA client of a CORBA server.

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 C-style casts, which are type unsafe, 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, it is sufficient to make a call to `QueryInterface()`, using the IID of the derived interface. If `QueryInterface()` fails, this object cannot be validly converted to an instance of the derived type.

A COM C++ Client Program

This section shows the code for a COM C++ client of the bank server that is developed in "Implementing CORBA Servers" on page 99.

The bank server presents the following interface to its clients:

```
interface account {
    readonly attribute float balance;

    void makeLodgement (in float f);
    void makeWithdrawal (in float f);
};

interface currentAccount : account {
    readonly attribute float overdraftLimit;
};

interface bank {
    exception reject {string reason;};

    account newAccount (in string name) raises (reject);
    currentAccount newCurrentAccount(in string name,
        in float limit) raises (reject);
   void deleteAccount (in account a);
};
```

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_marker:bankSvr:" &
    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

```
bool doOperations(Ibank *pIF)
{
    HRESULT hr = NOERROR;
    Iaccount *pAcc = 0;
    IcurrentAccount *pCurrAcc = 0;
    LPSTR firstName = "Ronan", secondName = "John";
    bool bExit=false;

    cout << "---------------- doOperations begin ---------------"
<< endl;

    hr = pIF->newAccount(firstName, &pAcc, NULL);
    bExit=CheckErrInfo(hr, pIF, IID_Ibank);
    printAccountInfo(pAcc);
    hr = pIF->deleteAccount(pAcc);
    bExit=CheckErrInfo(hr, pIF, IID_Ibank);
    pAcc->Release();

    hr = pIF->newCurrentAccount(secondName, 1000, &pCurrAcc,
        NULL);
    bExit=CheckErrInfo(hr, pIF, IID_Ibank);
    printAccountInfo(pCurrAcc);
    hr = pIF->deleteAccount(pCurrAcc);
    bExit=CheckErrInfo(hr, pIF, IID_Ibank);
    pCurrAcc->Release();
    cout << "---------------- doOperations end  ---------------" <<
            endl;
    return bExit;
}

void printAccountInfo(Iaccount *pAcc)
{
```

```
HRESULT hr = NOERROR;
IcurrentAccount *pCurrAcc = 0;
IOrbixObject *pOrbixObj = 0;
float balance = 0, overdraft = 0, deposit = 1000000;

cout << "---------- printAccountInfo begin ----------" << endl;

if(SUCCEEDED(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 CurrentAccout (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;
    }
    cout << "---------------- printAccountInfo end ---------------
" << endl;
}
```

Disconnecting from the CORBA Server

```
hr = pIBasic->deleteAccount(pAcc);
CheckErrInfo(hr, pIBasic, IID_Ibank);
pAcc->Release();
pIBasic->Release();
```

Exiting the Application

```
CoUninitialize();
```

7

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 programmers 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 main steps to expose DCOM servers to CORBA clients are:

- Build and register the DCOM server and any proxy/stub DLLs.
- Prime the OrbixCOMet type store with the correct type library.
- Register the supplied surrogate server executable (`custsur.exe`) in the Implementation Repository, under a given server name.
- Generate OMG IDL definitions from COM IDL, using `ts2idl`.
- Write a CORBA client to bind to the server and call operations.

This chapter describes how to perform each of these steps.

The Supplied DCOM Server

IONA ships some pure DCOM applications with OrbixCOMet in the *install-dir*\COMet\dcomapp directory, where *install-dir* represents the Orbix installation directory. 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 *install-dir*\COMet\dcomapp\testExe\server directory. This server is written using ATL and exposes objects supporting the following COM IDL 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 you run the COM C++ client supplied in the *install-dir*\COMet\dcomapp\testexe\client directory, the output is as follows:

```
[install-dir\COMet\dcomapp\testexe\client]client advice

Your fortune is :

     This fortune intentionally left blank :-) :-)
```

Building the DCOM Server and Proxy Stub DLLs

Build the supplied DCOM server executable, using the following command in the *install-dir*\COMet\dcomapp\testexe\server directory:

```
nmake -f IT_DcomApp.mak
```

Build the supplied proxy stub DLLs, using the following command in the *install-dir*\COMet\dcomapp\testexe\server directory:

```
nmake -f IT_DcomAppps.mk
```

At this point, you might wish to check the server's operation, using the DCOM client supplied in *install-dir*\COMet\dcomapp\testexe\client.

Priming the Type Store

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. Alternatively, you can populate the type store in advance, which is also known as *priming* the type store. You can use the following command to prime the type store:

```
typeman -e typename
```

Because you want to contact a DCOM server, all the marshalling code is 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 install-dir\COMet\dcomapp\testexe\server\IT_DcomApp.tlb
```

Note: You must supply the full path to the type library. Refer to "Development Support Tools" on page 157 for full details about the type store and how to prime it.

Registering the Server

The next step is to decide on a CORBA server name, and to create an entry in the Orbix 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. You must specify custsur.exe as the server executable 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.

Enter the following in the *install-dir*\COMet\bin directory, to register the DCOM fortune server in the Implementation Repository.

```
putit fortune "install-dir\COMet\bin\custsur.exe -t 10000"
```

In the preceding example, the -t option with custsur is specified as a parameter to the command, to provide a default timeout (in milliseconds) for the server. Refer to "OrbixCOMet Utility Options" on page 363 for more details about the options available with custsur.

To expose the server to CORBA, you simply need to:

- Register the type library.
- Register custsur.exe in the Implementation Repository under a server name.

Generating OMG IDL

For a CORBA client to invoke requests on a DCOM server, the CORBA client must be presented with a CORBA view of the server objects. This means that you must generate the OMG IDL definitions required by the CORBA client from the existing COM IDL for the DCOM server objects. You can use the ts2idl utility supplied with OrbixCOMet to create OMG IDL from existing COM IDL type information held in the OrbixCOMet type store. The ts2idl utility generates OMG IDL from COM IDL, by applying the standard mapping rules described in "COM-to-CORBA Mapping" on page 333.

The following command creates an OMG IDL file, fortune.idl, from the COM IDL interface shown in "The Supplied DCOM Server" on page 90:

```
ts2idl -i -r -f fortune.idl IT_DCOMAPPLib::IT_DcomTest
```

The generated OMG IDL file, fortune.idl, has two interfaces in this case (that is, IT_DCOMAPPLib::IIT_DcomTest and a coclass pseudo interface called IT_DCOMAPPLib::IT_DcomTest). Both of these interfaces are scoped within a module called IT_DCOMAPPLib, which is the internal type library name. You can check this using oleview if you wish.

The generated OMG IDL for `fortune.idl` is as follows:

```
// OMG IDL
// 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 the `BSTR` type maps to a readonly attribute of the `string` type. 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 the different ways that DCOM and CORBA handle reference counting.

  DCOM uses distributed reference counting. This means that when all outstanding references to an object are released (even for references held by remote clients), the server object's reference count falls to zero and the object is destroyed. When all objects in a DCOM server have been destroyed, the server shuts down.

  CORBA uses a different approach. 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 that launches DCOM servers in response to requests from CORBA clients, because the bridge does not know when to release DCOM interface pointers. The solution to this problem lies in the lifecycle interfaces, especially the `CosLifeCycle::LifeCycleObject::remove()` method. When a CORBA client has 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.

- A coclass pseudo interface is generated. Coclasses are a COM IDL feature that 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 COM IDL example in this chapter, `IIT_DcomTest` is the default interface for the `IT_DcomTest` coclass, which is the object that serves up fortune strings. `IIT_DcomTest` 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) are also 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 Interworking* 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. First, you should obtain an initial object reference. The following example uses `_bind()` to do this, but you can also use `custsur.exe` to generate IORs for CORBA clients. (Refer to "Connection and Usage with the Custsur Executable" on page 97 for more details). After obtaining an IOR, a client can then invoke operations on the server. 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 examine the task list while running this client, you can see that `IT_DcomApp.exe` appears briefly and disappears after the second call to `remove()`. This means that the DCOM server is correctly shut down, because of the lifecycle support.

CORBA Client Example Using Composable Support

This section provides an example of a CORBA client of the DCOM `fortune` server that uses `composable` support (rather than the pseudo coclass object support described in the preceding example):

```
#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 with the Custsur Executable

You can use `custsur.exe` to generate IORs for CORBA clients. The following options are available with `custsur`:

`-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 generates an IOR for the `IT_DcomTest` interface in the `fortune` server, and writes 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 CORBA client using the IOR generated in the preceding command:

```
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();
```

8

Implementing CORBA Servers

You can use OrbixCOMet to implement CORBA servers, using Automation-based tools such as PowerBuilder or Visual Basic. These servers can accept requests from standard COM or Automation clients as well as from CORBA clients. This chapter explains how to use OrbixCOMet to implement a CORBA server.

Note: OrbixCOMet is designed to support development of CORBA servers, using PowerBuilder or Visual Basic only. It does not facilitate automatic generation of C++ server skeleton code. If you want to implement a CORBA C++ server, use the Orbix C++ product.

Steps to Implementing a CORBA Server

The steps to implement a CORBA server, using OrbixCOMet, are:

1. Define and register the OMG IDL interfaces for the objects in your system.
2. Generate a corresponding type library or COM IDL definitions, using the supplied OrbixCOMet development tools.
3. Generate PowerBuilder or Visual Basic server skeleton code, using the supplied OrbixCOMet development tools.

4. Implement the OMG IDL interfaces by implementing a class in your chosen development language, exactly as you would for a normal Automation server.
5. Register your server with OrbixCOMet to make it appear as a CORBA server to CORBA clients.
6. Register your server in the Implementation Repository, so that it can be activated by the Orbix daemon (if necessary) when a CORBA client invokes on it.

This chapter describes how to perform each of these steps. You can find a Visual Basic version of the server in the `install-dir\demos\COMet\VB\bankSrv` directory, where *`install-dir`* represents the Orbix installation directory.

Defining and Registering OMG IDL Interfaces

A CORBA server presents an OMG IDL interface to its clients. The first step in implementing a CORBA server is to define the OMG IDL interfaces for the objects required in your system.

The OMG IDL example provided with your OrbixCOMet installation represents a bank and its accounts, as follows:

```
interface account {
   readonly attribute float balance;

   void makeLodgement(in float f);
   void makeWithdrawal(in float f);
};

interface currentAccount : account{
   readonly attribute float overdraftLimit;
};
```

The next step is to register your OMG IDL interfaces with the Interface Repository. This is necessary to allow OMG IDL type information to be added to the OrbixCOMet type store cache. The type store manager utility, `typeman`, searches the Interface Repository whenever it encounters any OMG IDL type information not currently held in the type store cache. Regardless of how new

OMG IDL type information is added to the cache (that is, manually before running an application, or automatically at application runtime), that type information must be obtained from the Interface Repository.

Use the `putidl` utility to register your OMG IDL with the Interface Repository. For example, the following command registers a `bank.idl` file contained in a `c:\bank` directory:

```
c:\bank> putidl bank.idl
```

Generating a Type Library or COM IDL

An Automation implementation for each interface in your OMG IDL file must be provided in your server. Each object that you implement must have methods and properties that correspond exactly to those in the OMG IDL interface definition, according to the standard mapping rules. Refer to “CORBA-to-Automation Mapping” on page 271 and “CORBA-to-COM Mapping” on page 309 for details of these rules.

To determine what the signature of each method in your server implementation should be, you must generate one of the following from your OMG IDL type information in the type store:

- A type library, created using `ts2tlb`.
- A COM IDL file, created using `ts2idl`.

Refer to “Development Support Tools” on page 157 for details about using `ts2tlb` and `ts2idl`.

Generating Server 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 server skeleton code, refer to “Development Support Tools” on page 157.

Implementing the Server Interfaces

To implement the OMG IDL interfaces, you implement a class in your chosen implementation language (that is, Visual Basic or PowerBuilder), 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 `Accounts` and `CurrentAccounts` collections are needed to maintain a collection of `Account` and `CurrentAccount` objects owned by the bank.

In the code examples in the following subsections, the additions to the generated code are shown in bold text.

Implementing the Account Interface

```
Private accBalance As Single
Private accOwner As String

Public Property Let balance(ByVal var_balance As Single)
    accBalance = var_balance
End Property

Public Property Get balance() As Single
    balance = accBalance
End Property

Public Property Let owner(ByVal var_owner As String)
    accOwner = var_owner
End Property

Public Property Get owner() As String
    owner = accOwner
End Property
```

```
Public Sub makeLodgement(ByVal var_amount As Single, Optional
    IT_Ex As Variant)
    accBalance = accBalance + var_amount
    frmBankSrv.Details.AddItem "made lodgement : balance
        is" & accBalance
End Sub

Public Sub makeWithdrawal(ByVal var_amount As Single,
    Optional IT_Ex As Variant)
    ' Check that the withdrawal does not
    ' exceed the balance:
    If ((accBalance - var_amount) >= 0) Then _
        accBalance = accBalance - var_amount

End Sub

Private Sub Class_Initialize()
    accBalance = 0
End Sub
```

Implementing the CurrentAccount Interface

The CurrentAccount interface inherits from Account. To implement the CurrentAccount interface, you must reimplement the properties and methods inherited from Account. You must also implement the overdraftLimit property that the CurrentAccount interface adds.

```
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 makeLodgement(ByVal amount As Single, Optional IT_Ex As
Variant)
    parentAcc.makeLodgement amount
End Sub

Public Sub makeWithdrawal(ByVal amount As Single,
    Optional IT_Ex As Variant)
    ' Check that the withdrawal does not exceed
    ' the balance including overdraftlimit:
    If ((parentAcc.balance - (amount - overdraftLimit))>= 0)
        Then _
        parentAcc.balance = parentAcc.balance - amount
End Sub
```

Implementing the Bank Interface

The newAccount() and newCurrentAccount() operations on the Bank interface raise an exception if the bank fails to create an account. The code to raise an exception is not included in this example. "Exception Handling" on page 109 deals with this topic in detail.

```
Private Accounts As New Accounts
Private CurrentAccounts As New CurrentAccounts
Private objFactory As CORBA_Orbix.DICORBAFactoryEx

Public Function newAccount(ByVal var_owner As String, Optional
IT_Ex As Variant) As Object
Dim excp As BankSrv.DIbank_reject

    If frmBankSrv.AccountAlreadyExists(var_owner) Then
        frmBankSrv.Details.AddItem "Account already exists for
            Customer : " & var_owner
```

```
        Set excp = objFactory.CreateType(Nothing,
            "BankSrv.Bank_Reject")
        excp.reason = "Account already exists!!!"
        IT_Ex = excp
    Else
        Set newAccount = Accounts.Add(var_owner)
        frmBankSrv.Details.AddItem "Created new account for
            Customer : " & newAccount.owner
    End If

End Function

Public Sub deleteAccount(ByVal var_owner As Object, Optional IT_Ex
    As Variant)

    Accounts.Remove var_owner.owner
    CurrentAccounts.Remove var_owner.owner
    frmBankSrv.RemoveAccount (var_owner.owner)
    frmBankSrv.Details.AddItem "Account deleted for : " &
        var_owner.owner

End Sub

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 the Server 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

This section shows how to use Visual Basic to tranform an Automation server to a CORBA server.

```
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 `pbgenreg.exe` tool. From PowerBuilder 6.0, `pbgenreg` is accessible from the PowerBuilder menu.

```
// 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 'bank.bankImplObject'
OleObject bankObj
bankObj = CREATE OleObject
bankObj.ConnectToNewObject("bank.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 preceding Visual Basic and PowerBuilder examples instantiate a `bank` object, and register 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 chooses the marker for the `bank` object.

The next step is to activate the server, so that any objects registered with the bridge can receive incoming requests from CORBA clients. In this case, the call to `Activate()` gives the server the name `bank`. This is also the name with which the server is to be registered in the Implementation Repository. (Refer to "Registering the CORBA Server in the Implementation Repository" on page 108 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 is 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 Implementation Repository. This means the Orbix daemon can know how to activate the server, if the server is not already running 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 "Registering the Server with OrbixCOMet" on page 105. In this example, the server must therefore be registered with the name bank.

You can register your server as follows, using `putit`, where *`executable_file`* is the full path to the server program:

```
putit bank executable_file
```

9

Exception Handling

Exception 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 from reporting network problems to 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. Refer to the Orbix C++ documentation set for more details.

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 through calls to OrbixCOMet. OrbixCOMet might raise a system exception if, for example, it encounters a problem with the network.

Example of a User Exception

Recall the `Bank` interface defined in "Implementing CORBA Servers" on page 99:

```
// OMG IDL
interface Bank {

    exception Reject {
        string reason;
    };

    Account newAccount(in string owner) raises (Reject);
    …
};
```

In this case, the `newAccount` operation raises a single `Reject` exception. 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 `newAccount()` operation raises the `Reject` user exception. The `Reject` exception contains one member, of the `string` type, 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 OrbixCOMet encounters some problem 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:

```
// COM IDL
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 "CORBA-to-Automation Mapping" on page 271 for details of how OMG IDL interfaces and exceptions map to Automation. Refer to "CORBA-to-COM Mapping" on page 309 for details of how OMG IDL interfaces and exceptions map 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 `(D)IForeignException` interface, which 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 methods can be described as follows:

`EX_majorCode()`	This indicates the category of exception raised. It can be any of the following, defined in the `ITStdInterfaces.tlb` file:
	`EXCEPTION_NO`
	`EXCEPTION_USER`
	`EXCEPTION_SYSTEM`

`EX_Id()`	This 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 a User Exception" on page 110).

System exceptions also have the following additional properties, which are defined in the `(D)ICORBASystemException` interface:

```
interface DICORBASystemException : DIForeignException {
    [propget] HRESULT EX_minorCode(
        [retval,out] long* IT_retval);
    [propget] HRESULT EX_completionStatus(
        [retval,out] long* IT_retval);
};
```

The methods can be described as follows:

`EX_completionStatus()`	This indicates the status of the operation at the time the system exception is 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 had not completed before the exception was raised. `COMPLETION_MAYBE` — This means the operation was initiated, but it cannot be determined if it completed.
`EX_minorCode()`	This returns a code describing the type of system exception that has occurred. 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 to Automation exceptions by the bridge. This allows exceptions that are 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. For example, in “Example of a User Exception” on page 110, the `Reject` exception contains one member, which is called `reason` and is of the `string` type. However, using Automation’s native exception handling, exception 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 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. As already described in “Exceptions” on page 291, an OMG IDL operation maps to an Automation method that has an additional optional parameter.

For example:

```
interface Account {
    …
    void makeDeposit(in float amount,
        out float balance);
};
```

This maps to:

```
// COM IDL
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 to see if it contains an exception after the call. To use exceptions in this manner, however, the `IT_Ex` parameter must first be initialized to `Nothing` in the client code, as follows:

```
…
Dim IT_Ex As Object
Set IT_Ex = Nothing
…
```

If this optional exception parameter is used, OrbixCOMet does not translate any CORBA exceptions that might occur during the call into an Automation exception. Instead, the optional exception parameter is populated with rich error information relating to any CORBA exception that occurs. 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 COM IDL methods that can be used by the caller to extract this additional information. (Refer to "CORBA-to-Automation Mapping" on page 271 and "Automation-to-CORBA Mapping" on page 299 for details of the mapping between OMG IDL and COM IDL user exceptions.)

Standard Automation exception handling is disabled when the exception parameter is passed in a method. This allows the value of the exception to be examined inline.

Assume that `newAccount()` can raise the user exception, `Reject`, defined as follows:

```
// OMG IDL
interface Bank {
   exception Reject {
      string reason;
   };
...
};
```

You can use type information to check the type of exception that occurred:

```
' Visual Basic
Dim ex As Variant
Set ex = Nothing

'Optional exception param passed, therefore COMet will not convert
'a CORBA Exception into an Automation exception
Set accountDisp = bankObj.newAccount(Namebox.Text, ex)

'Did any exception occur ?
If ex.EX_majorCode <> CORBA_ORBIX.EXCEPTION_NO Then
    'Is it a user exception occur ?
    If TypeOf ex Is CORBA_ORBIX.DICORBAUserException Then

        ' Which user exception ?
        If TypeOf ex Is IT_Library_bank.DIbank_reject Then
            Dim exReject As IT_Library_bank.DIbank_reject
            Set exReject = ex
            MsgBox exReject.EX_Id, , "User Exception Ex_Id :"
            MsgBox exReject.INSTANCE_repositoryId, , "User
                Exception INSTANCE_repositoryId :"
```

```
            MsgBox exReject.reason, , "User Exception reason :"
        End If

    'Is it a system exception ?
    ElseIf TypeOf ex Is CORBA_ORBIX.DICORBASystemException Then
       Dim exSystemException As CORBA_ORBIX.DICORBASystemException
        Set exSystemException = ex

        MsgBox "System exception has occurred : " &
            exSystemException.EX_Id

        Select Case exSystemException.EX_completionStatus
            Case CORBA_ORBIX.COMPLETION_MAYBE
               MsgBox "System exception Completion Status : Maybe "
            Case CORBA_ORBIX.COMPLETION_NO
                MsgBox "System exception Completion Status : No "
            Case CORBA_ORBIX.COMPLETION_YES
                MsgBox "System exception Completion Status : Yes "
            Case Else
               MsgBox "Unknown System exception Completion Status"
        End Select
    End If
End If
```

Note: In the preceding example, `ex` is declared as a `Variant` type, and it is initalized to `Nothing`. This sets up a variant that contains an object equal to nothing. This is the correct way to interface from Visual Basic to OrbixCOMet when using late binding in an Automation client.

Exception Handling in COM

This section describes exception handling in COM. As already explained in "Exceptions" on page 322, a CORBA exception maps to a COM IDL interface and an exception structure that appears as the last parameter of any mapped operation.

Catching COM Exceptions

The bridge translates 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++

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_marker:bankSvr:", 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 `newAccount()` operation 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 "Creating Constructed OMG IDL Types" on page 283 and page 317 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 must then assign the exception to the `IT_ex` parameter, which transmits system and user exceptions back to the caller. It is good practice to exit the function immediately after raising an exception.

Automation Exceptions

The following is a Visual Basic example of how to raise an exception:

```
' 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

The following is a COM C++ example of how to raise an exception:

```
// COM ++
try
{
    short h, w;
    DIbankPtr bank;
    DIaccountPtr acc;
    DICORBAFactoryPtr fact;

    fact.CreateInstance("CORBA.Factory");
    bank = fact->GetObject("bank:bank_marker:bankSvr", 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;
}
```

10

Implementing 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 to, 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 or 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.

To implement callbacks, you must:

- Define the OMG IDL interfaces for the server objects and client objects.
- Generate the skeleton code for the callback objects.
- Write a client.
- Write a server.
- Register the server in the Implementation Repository.

The following sections describe each of these steps in turn.

Defining OMG IDL Interfaces

The client implements an interface that the server uses 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 `registerClient()` operation registers a client with the server. The parameter to `registerClient()` is of the `NotifyCallback` type, so that the client can pass a reference to itself to the server. The server can maintain this reference in a list of clients that should be notified of events of interest.

The `unregisterClient()` operation tells the server that the client is no longer interested in receiving callbacks. The server can then remove the client from its list of interested clients.

Generating Skeleton Code for Callback Objects

As in the case of creating a server, you should generate the skeleton code for the callback objects. Refer to “Generating Server Stub Code and Support for Callbacks” on page 176 for details of how to do this. Generating the skeleton code for the callback objects ensures that your interfaces have the correct parameters, in the correct order, and so on.

Writing a Client

To write a client, you must implement the `NotifyCallback` interface for the client objects. You can use the generated skeleton code for the callback objects as a template, as if the client were a CORBA server.

Visual Basic

The following is an example of a Visual Basic client:

```
Dim clientObj as New NotifyCallback

Dim ObjFactory As Object
Set ObjFactory = CreateObject("CORBA.Factory")
…
Dim serverObj as clientBridge.DIRegisterCallback
Set serverObj = ObjFactory.GetObject("CallBack:CallBack_marker:
    callbackSvr:" & Hostname.Text)
serverObj.registerClient clientObj
… 'Your code goes here
Public Sub notifyClient(Optional ByRef IT_Ex As Variant)

End Sub
…
```

In the preceding example, the client creates an implementation object, `clientObj`, of the `NotifyCallback` type. It binds to an object of the `RegisterCallback` type in the server. At this point, the client holds an implementation object for the `NotifyCallback` type, and a reference to an Automation view object, `serverObj`, for an object of the `RegisterCallback` type.

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 registerClient() operation on the serverObj view object, 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 cannot bind to the client's implementation object.

PowerBuilder

The following is an example of a PowerBuilder client:

```
OleObject NotifyCallback
OleObject ObjFactory

ObjFactory = CREATE OleObject
serverObj = CREATE OleObject

serverObj = ObjFactory.GetObject("CallBack:CallBack_marker:
    callbackSvr:HostName")
serverObj.Register(NotifyCallback)
```

In the preceding example, NotifyCallback and ObjFactory are global variables.

COM C++

The following is an example of a COM C++ client:

```
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_marker:
    callbackSvr:%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();
```

Writing the Server

The server application is implemented as a normal OrbixCOMet server, as described in "Implementing CORBA Servers" on page 99. In particular, you must:

- Implement the `RegisterCallback` interface.
- Invoke the `notifyClient()` operation.

The following subsections describe each of these steps in turn.

Implementing the RegisterCallback Interface

You must provide an implementation class for the `RegisterCallback` interface, using the skeleton code generated for the callback objects 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 or 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

The following is a Visual Basic example of how to implement the `RegisterCallback` interface:

```
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

The following is a PowerBuilder example of how to implement the `RegisterCallback` interface:

```
// Create two functions passing a user object
registerClient (…
unregisterClient (…
```

COM C++

The following is a COM C++ example of how to implement the `RegisterCallback` interface:

```
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 to Notify the Client

After the view is created in the server address space, the server can invoke the `notifyClient()` operation 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 to, or withdrawal from, a bank account).

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 Implementation Repository. Therefore, the server cannot bind to the client's implementation object.

Visual Basic

The following is a Visual Basic example of how to invoke `notifyClient()`:

```
Dim callbackObj as serverBridge.DINotifyCallback

' Get the reference to the client from the server's stored data
Set callbackObj = …

' Call back to client
callbackObj.notifyClient
```

PowerBuilder

The following is a PowerBuilder example of how to invoke `notifyClient()`:

```
// Get the reference to the client from the server's stored data
OleObject CallbackObj
…
CallbackObj.ConnectToNewObject(…)
…
CallbackObj.notify()
```

COM C++

The following is a COM C++ example of how to invoke `notifyClient()`:

```
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;
```

Registering the Callback Object Server

Finally, the server instantiates an object of the `RegisterCallback` type, registers the object with the bridge, and activates itself as a CORBA server.

Visual Basic

The following is a Visual Basic example:

```
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

The following is a PowerBuilder example:

```
// 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++

The following is a COM C++ example:

```
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;
```

11

SSL Support

SSL support with OrbixCOMet opens up the domain of SSL-secured CORBA programs to COM/Automation clients and servers. Using SSL with your OrbixCOMet applications means on-the-wire communication using IIOP is secure.

The recommended OrbixCOMet deployment scenario for COM or Automation clients is to use OrbixCOMet in-process and connect to secure CORBA servers. In this scenario, all on-the-wire communication is performed using SSL-secured IIOP. This means OrbixCOMet applications using SSL can avail of the cornerstone security attributes of authentication, privacy and integrity, with the need for little or no extra code.

The use of secure IIOP between OrbixCOMet and CORBA clients and servers does not require any changes to deployment scenarios where DCOM on-the-wire is also used (that is, where OrbixCOMet is used out-of-process by COM or Automation clients).

The key attribute of an SSL-secured application is its association with an X.509 certificate. This association is established for each application by specifying an X.509 certificate file and supplying the password of the private key stored within the certificate. Therefore, an OrbixCOMet application can be initalized as an SSL application if it has access to a certificate, and the password for the certificate's private key.

Note: OrbixCOMet SSL is only available with OrbixOTM 3.0 and later versions. This chapter assumes you have a prior knowledge of OrbixSSL. (Refer to the *OrbixSSL C++ Programmer's and Administrator's Guide* for full details.)

Enabling SSL in an OrbixCOMet Application

SSL support is added to OrbixCOMet applications, using a combination of API calls to the OrbixCOMet (D)`IOrbixSSL` interface, and by specifying configuration information within the `OrbixCOMetSSL` configuration scope in the OrbixSSL configuration file (`OrbixSSL.cfg`). In the case of CORBA clients talking to existing DCOM servers, SSL support can be added simply by using `OrbixCOMetSSL` configuration scope settings.

The OrbixCOMet (D)`IOrbixSSL` interface can be used by OrbixCOMet applications to specify the password and certificate combination that is used to enable SSL for the application. A reference to the (D)`IOrbixSSL` interface is obtained by a call to `GetOrbixSSL()` on the (D)`IOrbixORBObject` interface. The following is an example of how to do this in a Visual Basic client:

```
Dim objSSL As CORBA_Orbix.DIOrbixSSL
Dim strPassword as String
     Set objSSL = orb.GetOrbixSSL()
     objSSL.InitSSL
     strPassword = InputBox("Enter Password ", "", "")
     objSSL.SetPrivateKeyPassword strPassword
     objSSL.SetSecurityName
     "C:\Iona\OrbixSSL\certificates\demos\democlient"
     Set objSSL = Nothing
```

The process of specifying the private key password and X.509 certificate is more usually effected by calling (D)`IOrbixSSL::SetPrivateKeyPassword`, and then specifying a configuration scope parameter to (D)`IOrbixSSL:InitScopeSSL`. For example:

```
     …
     objSSL.InitScope("OrbixCOMetSSL.Demos")
     …
```

The parameter specified in the call to InitScope() identifies a scope in the OrbixSSL.cfg file, which contains a specification of the SSL security policy settings to be implemented by OrbixSSL on behalf of users of this policy. (Refer to the *OrbixSSL C++ Programmer's and Administrator's Guide* for more details about configuration scopes.)

The configuration scope specified usually contains a value for IT_CERTIFICATE_FILE, which determines the certificate that SSL associates with the application after the call to InitScopeSSL. Scopes and policies can be specified on a per-application basis in the OrbixSSL.cfg file.

The following example taken from OrbixSSL.cfg shows the specification of the certificate associated with the scope OrbixCOMetSSL.Demos:

```
OrbixCOMetSSL
{
     # This cert is used by OrbixCOMet SSL clients and
     # servers
     Demos
     {
        # Password for the demoserver cert is
        # "demopassword"
        IT_CERTIFICATE_FILE= _demos_cert_path +
           "demo_server";
     };
};
```

Regardless of whether SetSecurityName or InitScopeSSL is used to specify the certificate, the password must be specified first, using SetPrivateKeyPassword. You should ensure that private key passwords are never hard-coded in your applications. Instead, where necessary, users should be prompted to enter private key passwords at runtime. As with all OrbixSSL applications, OrbixCOMet clients and servers are required to perform their SSL initialization before they bind to secure CORBA servers or register themselves as secure CORBA servers.

OrbixCOMet SSL Handler DLLs

OrbixCOMet handler DLLs can be used to inject extra SSL functionality into OrbixCOMet applications. OrbixSSL provides the facility to register a C++ callback function that is invoked by OrbixSSL during client and server authentication. This callback function is passed details of the X.509 certificate of the application being connected to. Customized checks can then be made on the certificate in the implementation of the callback function. If the callback function returns TRUE, it is signifying to SSL that the certificate is acceptable. If it returns FALSE, OrbixSSL aborts the connection attempt and throws an authorization failure exception.

The SSLHandler demonstration in the *install-dir*\demos\COMet\corbasrv directory (where *install-dir* represents the Orbix installation directory) contains an example of an OrbixCOMet SSL handler DLL. This handler DLL is used by the secure COM grid client and the secure Visual Basic Bank client, to perform customized validation of the certificates being presented by their respective servers. (Refer to "Using Handler DLLs" on page 153 for details.)

Handler DLLs are activated from within Automation clients simply by calling the LoadHandler method on (D)IOrbixORBObject. For example:

```
orb.LoadHandler "mySSL"
```

Secure CORBA Clients Accessing Existing DCOM Servers

OrbixCOMet SSL support also enables secure CORBA clients to connect to existing DCOM servers (that is, third-party DCOM servers). Currently, OrbixCOMet facilitates CORBA clients connecting to third-party DCOM servers, using the OrbixCOMet generic custom surrogate program (custsur.exe), which acts in this scenario like a CORBA server. If the CORBA client is secure, OrbixCOMet (hosted by custsur) must initialize itself as a secure CORBA server.

The current fortune demonstration CORBA client makefile registers custsur as a CORBA server with the following command:

```
putit fortune "install-dir\COMet\bin\custsur.exe -t 20000"
(-t : Specify server time out in milliseconds )
```

To instruct an instance of custsur to initialize itself as a secure CORBA server, the -l switch must be passed to custsur as a command-line parameter. For example, to make the fortune server secure, add the -l switch as follows:

```
putit fortune "install-dir\COMet\bin\custsur.exe -l -t 20000"
```

To run the fortune server persistently, add the -l switch to the command line in addition to the -s switch, which is required to specify the server name:

```
C:\IONA\OrbixCOMet_3.0\custsur.exe -l -s “fortune” -t 20000
```

Specifying the Custsur.exe Certificate

In secure mode, custsur is like any other CORBA server, in that it must be associated with an X.509 certificate and have access to the private-key password for the certificate.

The association between a certificate and an instance of custsur (acting as a particular CORBA server) is made via the entries in the OrbixCOMetSSL.CustSur scope within the OrbixSSL configuration scope (OrbixSSL.cfg). This scope has a separate entry for each CORBA server that custsur is configured to impersonate. Each configuration scope is identified by the CORBA server name. Therefore, in the preceding example, when custsur is asked to initialize itself securely as the fortune CORBA server, it looks for a certificate and other SSL security policy configuration information within the OrbixCOMetSSL.CustSur.fortune configuration scope.

```
OrbixCOMetSSL
{
      # This policy is used by CustSur (hosting COMet)
      # in its role as a generic Secure CORBA Server to
      # CORBA Clients. The nested entries at this scope
      # detail the SSL configuration for each CORBA
      # Server CustSur has been registered to impersonate.
      CustSur
      {
         # This Policy is used by CustSur registered as
         # the fortune CORBA Server fortune
         {
            IT_CERTIFICATE_FILE= _demos_cert_path +
               \"demoserver\";"
         };
      };
};
```

Specifying the Corresponding Private-Key Password

Secure CORBA servers have two options for retrieving passwords at runtime. The first is that the server prompts the user to specify the private-key password for its certificate when it is launched. The other option is that the server receives a password from the OrbixSSL key distribution mechanism (KDM) at server start-up. The `custsur` utility supports both of these methods.

The KDM is a persistent CORBA server supplied with OrbixSSL. (Refer to the *OrbixSSL C++ Programmer's and Administrator's Guide* for more details about the KDM.) It provides a mechanism whereby passwords can be securely supplied to servers at start-up. This negates the requirement for user intervention (that is, prompting the user to enter a password). Entries can be added to the KDM database, using the OrbixSSL `putKDM` utility. For example:

```
putKDM fortune demopassword
```

The KDM maintains an encrypted database that stores server names and private key password pairs. It must be started persistently before the Orbix daemon is started. When the Orbix daemon launches a server, it checks with the KDM to see if an entry exists for that server. If an entry does exist, the password is passed to the OrbixSSL runtime in the server. The SSL initialization code in an SSL-enabled application determines whether or not a password has been received from the KDM via the Orbix daemon at start-up. The `(D)IOrbixSSL:HasPassword` method returns `false` if no password has been obtained from the KDM. The start-up code for `custsur` uses the return value of `HasPassword` to determine whether it needs to prompt the user for the private-key password of the certificate.

OrbixCOMet Type Store Manager and the Secure IFR

In a secure CORBA environment, the Interface Repository is configured to run as an SSL-secured CORBA server. The OrbixCOMet type store manager (`typeman.exe`) retrieves type information from the Interface Repository. Therefore, in an SSL-secured environment, `typeman` must also be configured to run securely. You can use the `COMet.TypeMan.TYPEMAN_SSL_ENABLED` configuration variable to make `typeman` SSL-enabled. Refer to "OrbixCOMet Configuration" on page 353 for more details.

12

Deploying an OrbixCOMet Application

This chapter provides examples of the various deployment models you can adopt when deploying a distributed application, using OrbixCOMet. It also describes the steps you must follow to deploy a distributed OrbixCOMet application.

Deployment Models

OrbixCOMet supports communication using the DCOM protocol and the CORBA IIOP protocol. You therefore have a great degree of flexibility in terms of how you can combine your COM/Automation and CORBA applications. "Usage Models and Bridge Locations" on page 11 has already described the various ways you can combine COM/Automation and CORBA clients and servers. It has also introduced the concept of being able to install the OrbixCOMet bridge anywhere in your system.

This section provides further examples of the various OrbixCOMet deployment models and bridge locations available. As a general rule of thumb, remember:

- The machine(s) on which the OrbixCOMet bridge is located must be running on Windows.
- A client uses its associated protocol to communicate with the bridge (that is, IIOP for CORBA, DCOM for COM/Automation). The bridge uses the server's associated protocol to communicate with the server.

Bridge on Each Client Machine

In this model, the OrbixCOMet bridge is installed on each client machine. Figure 12.1 shows COM or Automation clients communicating with a CORBA server. In this case, the recommended deployment scenario is to load the bridge in-process to each client. Alternatively, the bridge could be loaded out-of-process on each client machine, but in this case you should use straight `IDispatch` interfaces instead of dual interfaces, because there are limitations to using OrbixCOMet out-of-process with dual interfaces.

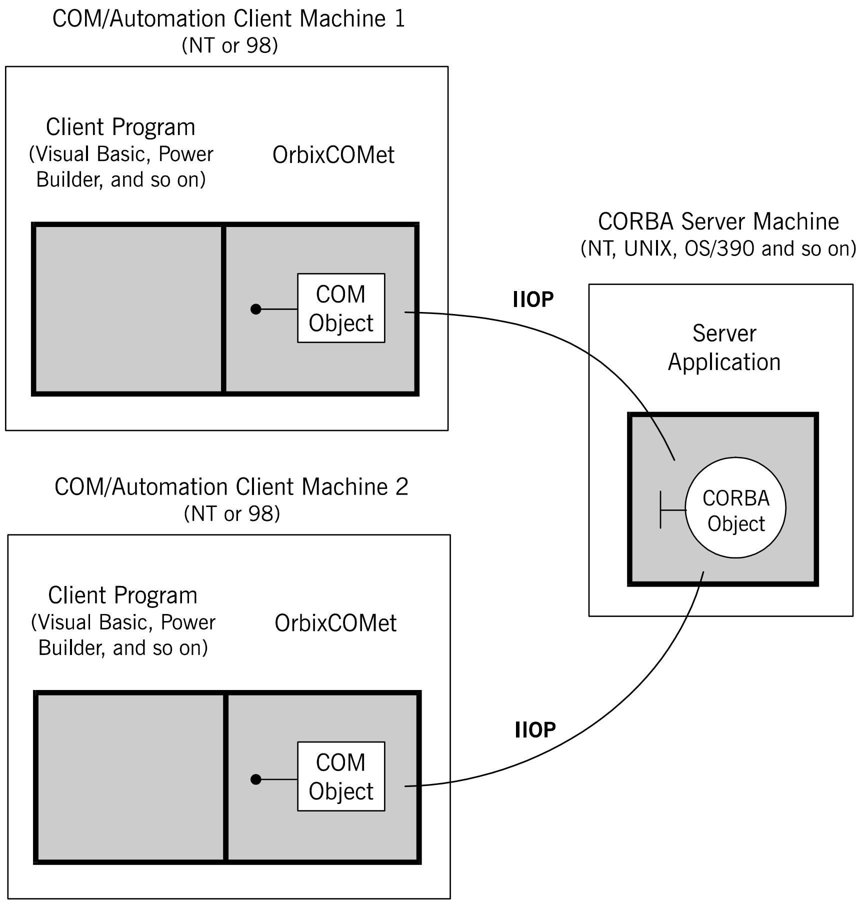

Figure 12.1: *COM/Automation to CORBA with Bridge on Each Client Machine*

Figure 12.2 shows CORBA clients communicating with a COM or Automation server. In this case, each CORBA client must be running on Windows, and the bridge is always loaded out-of-process. Each CORBA client uses IIOP to communicate with the bridge, and the bridge uses DCOM to communicate with the COM or Automation server.

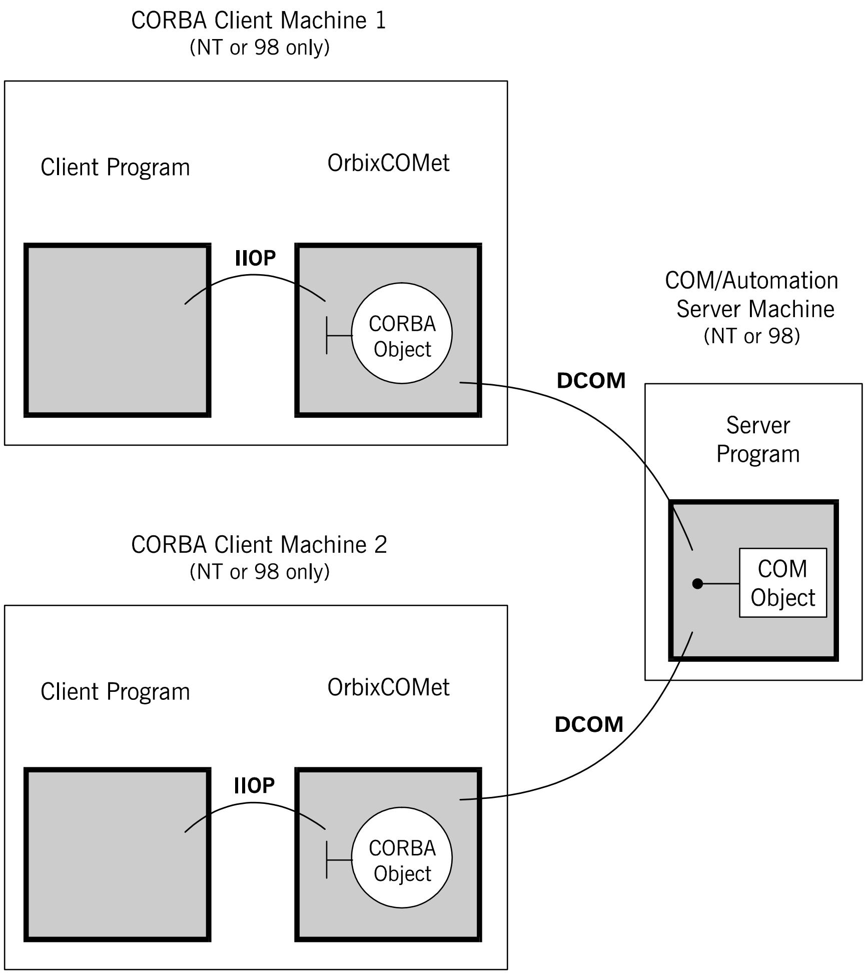

Figure 12.2: *CORBA to COM/Automation with Bridge on Each Client Machine*

Bridge on Server Machine

In this model, OrbixCOMet is only installed on your server machine. In Figure 12.3, COM or Automation clients use DCOM to communicate with the bridge on the server machine.

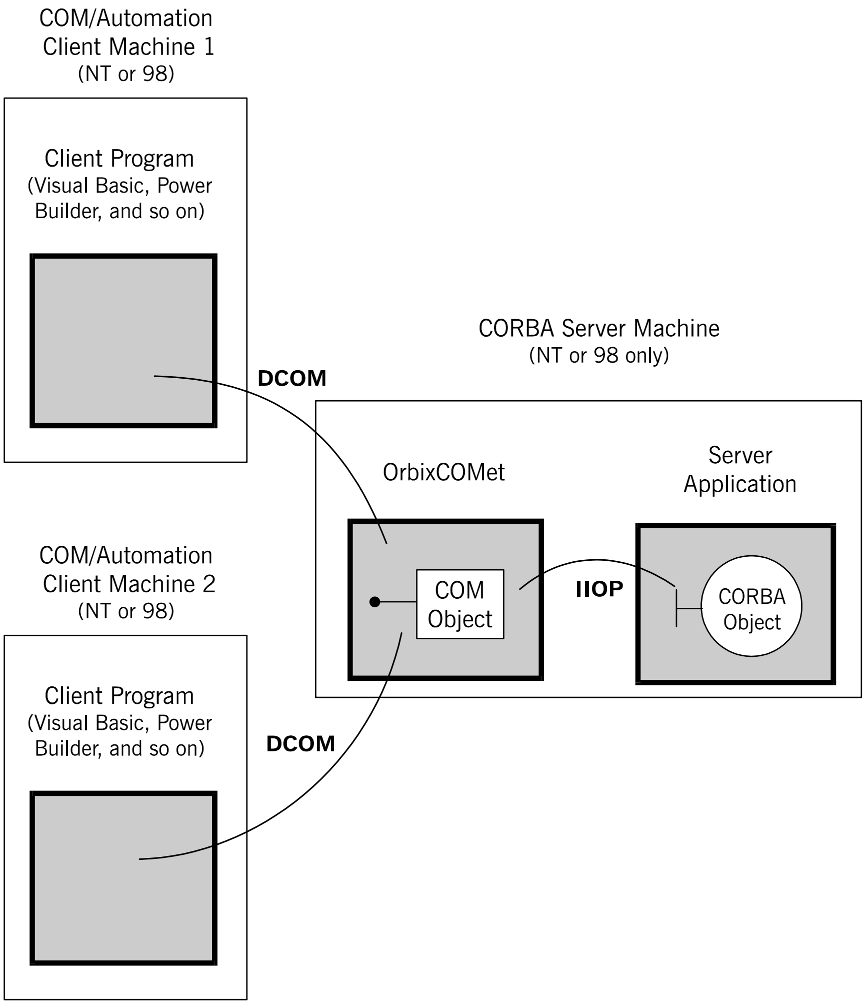

Figure 12.3: *COM/Automation to CORBA with Bridge on Server Machine*

In Figure 12.4, CORBA clients use IIOP to communicate with the bridge on the server machine. The bridge uses DCOM to communicate with the COM or Automation server program. The number of DCOM clients that can be supported in the model shown in Figure 12.3 is considerably less than the number of CORBA clients that can be supported in the model shown in Figure 12.4.

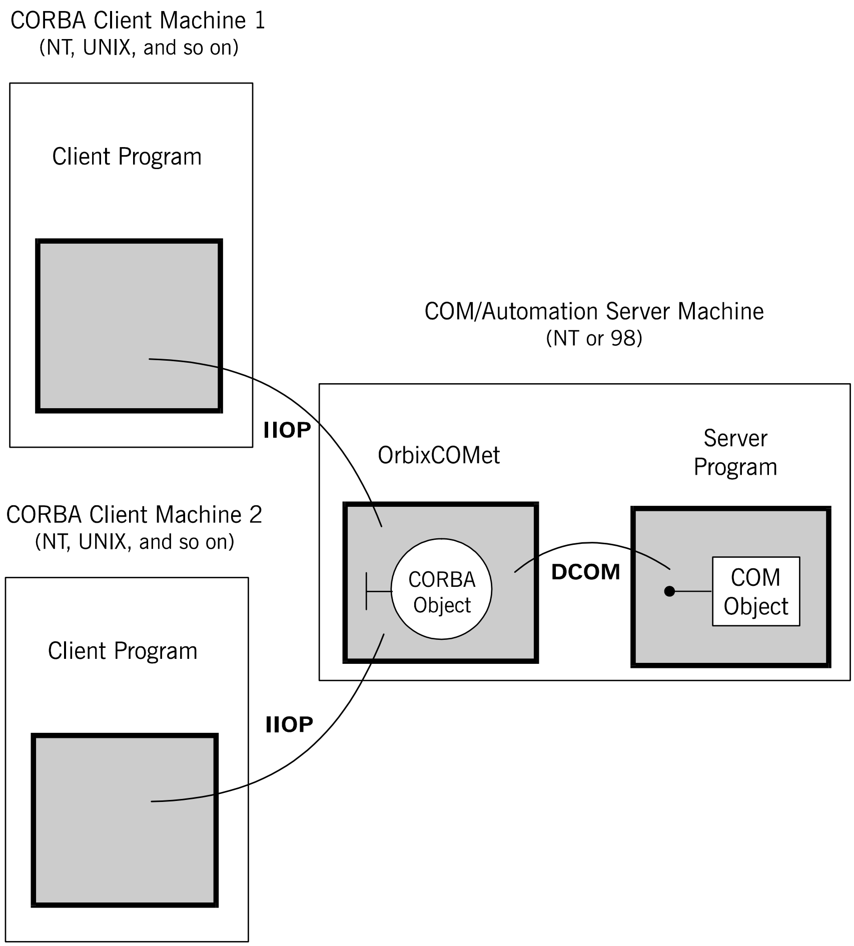

Figure 12.4: *CORBA to COM/Automation with Bridge on Server Machine*

Bridge on Intermediary Machine

In this model, the bridge is shared by multiple clients. It is installed on a single separate machine that must be running on Windows.

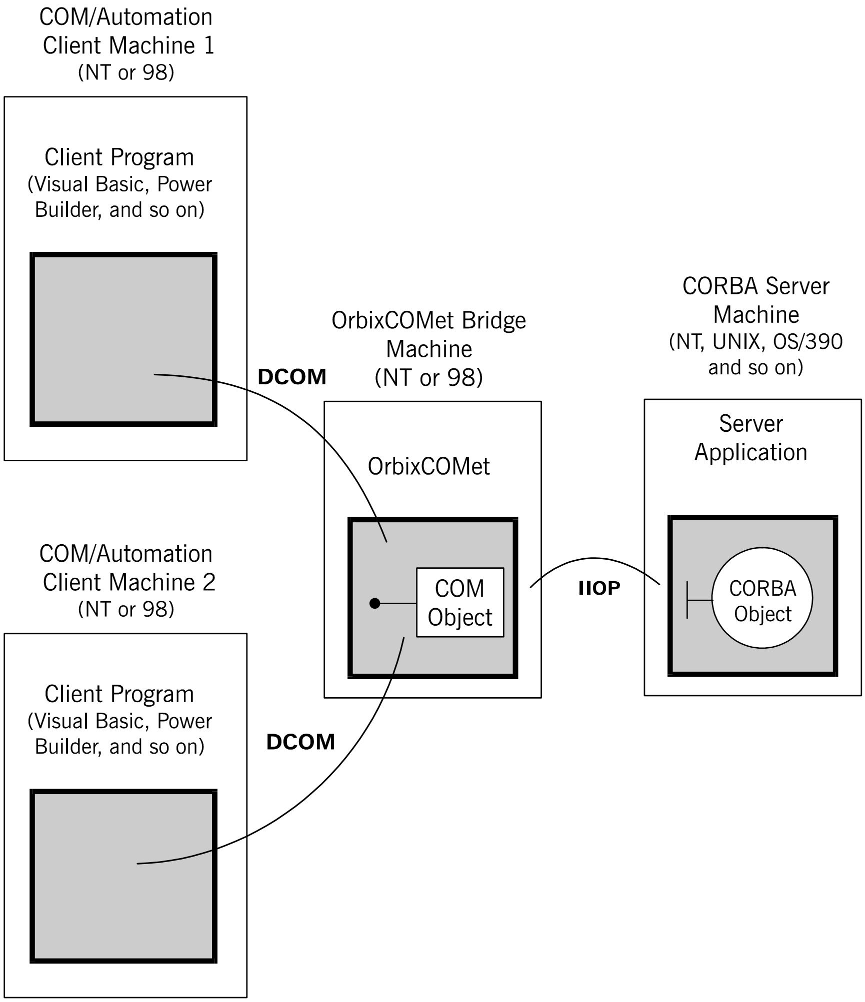

Figure 12.5: *COM/Automation to CORBA with Bridge on Intermediary Machine*

In Figure 12.5, COM or Automation clients use DCOM to communicate with the bridge, and the bridge uses IIOP to communicate with the CORBA server.

In Figure 12.6, CORBA clients use IIOP to communicate with the bridge, and the bridge uses DCOM to communicate with the COM or Automation server.

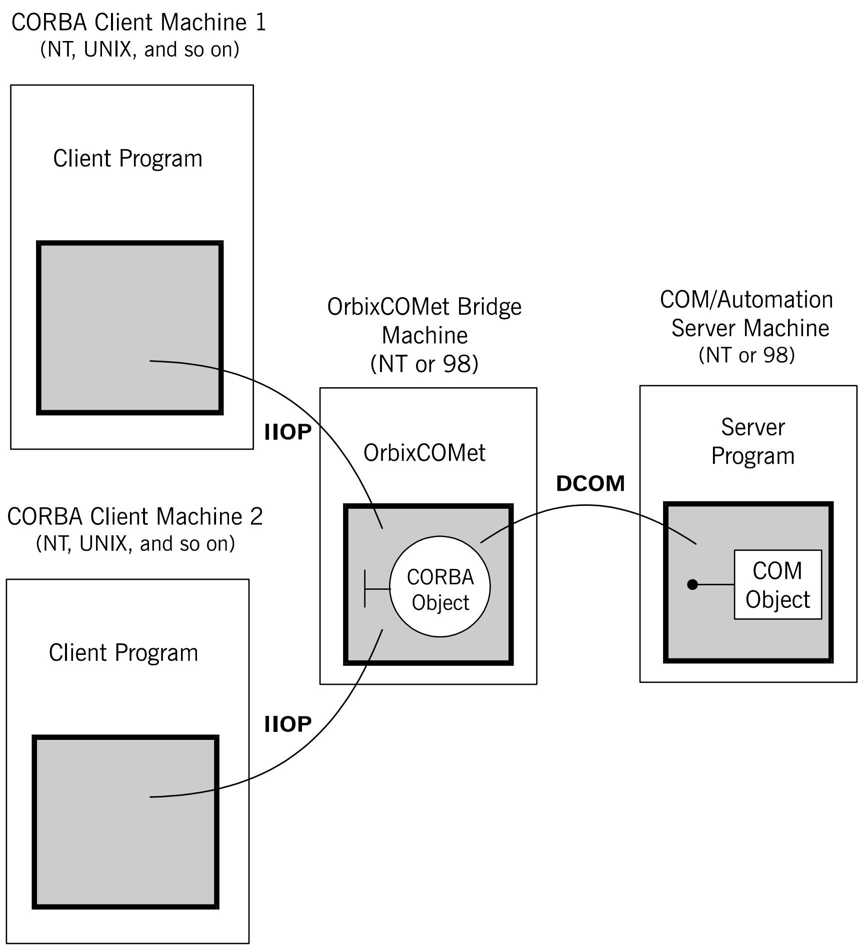

Figure 12.6: *CORBA to COM/Automation with Bridge on Intermediary Machine*

In Figure 12.5 on page 144, you only need to be able to create a remote instance of the CORBA object factory on your client machines. This is normally done using the DCOM `CoCreateInstanceEx()` method. OrbixCOMet provides a simple wrapper (called `CCIExWrapper.dll`) for this function for any languages, such as Visual Basic Script or PowerBuilder, that do not directly support this DCOM call. When using multiple DCOM clients with a single bridge, as shown in Figure 12.5, the setting of the `COMet.Typeman.TYPEMAN_READONLY` configuration variable is particularly important. Refer to "OrbixCOMet Configuration" on page 353 for details.

Internet Deployment

When deploying an OrbixCOMet application on the Internet, you can choose from the following options:

- Download the entire OrbixCOMet bridge to the client machine. To do this, you can bundle the bridge files, for example, in a single CAB file. In this case, your ActiveX control uses IIOP to communicate with your Internet server.
- Download only the DLLs and leave the bridge on the Internet server. In this case, your ActiveX control uses DCOM to communicate with your Internet server.

The setting of the `COMet.Typeman.TYPEMAN_READONLY` configuration variable is particularly important for internet deployment. Refer to "OrbixCOMet Configuration" on page 353 for details.

Deployment Steps

To install an application developed with OrbixCOMet you must install:

- Your application's runtime.
- The development language's runtime.
- The Orbix 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 353.

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 runtime requirements for your development language normally consist of:

- Runtime libraries (such as Visual Basic or PowerBuilder runtime libraries).
- Support libraries (such as Roguewave tools or extra libraries).

Details of the runtime requirements of your development language can be found in the documentation set for the specific development language.

Installing the Orbix Runtime

Regardless of the model you adopt in deploying your OrbixCOMet applications, the Orbix runtime requirements remain the same. This section describes the Orbix-specific files, libraries, and executables required to run your OrbixCOMet applications.

Orbix Daemon

The Orbix daemon, `orbixd`, is always required on the server host. Ensure the daemon is running on your server before you try to run your application. Refer to the Orbix C++ documentation set for details of how to start the daemon.

Orbix Configuration Files

The following Orbix configuration files are held in the *install-dir*\config directory, where *install-dir* represents the Orbix installation directory:

- IONA.cfg
- Orbix.cfg
- common.cfg

The configuration files are required both on the client and server hosts. You must modify the configuration entries in these files appropriately for your system. When specifying a pathname for a specific directory, you must provide the full pathname and ensure that it is valid. The Orbix daemon must have read/write permissions on the directories specified in these pathnames. (Refer to "OrbixCOMet Configuration" on page 353 for details about the various configuration files and entries.)

You can make these available to OrbixCOMet by placing them in a config subdirectory under the directory that is pointed to by the registry entry IONA Technologies\IT_INSTALLATION_DIR. You must also set the IT_CONFIG_PATH environment variable to point to this config subdirectory.

Environment Variables

You must set the following environment variables:

IT_CONFIG_PATH	Set this to the directory containing your configuration files.
PATH	Set this to include the Orbix \bin directory.

Error Messages File

You can use the IT_ERRORS configuration variable to specify the location of the Orbix ErrorMsgs file.

Orbix Executables

If you are using the Interface Repository, the IFR.exe is required. You can locate your IFR server anywhere in your system: on the client machine, on its own dedicated machine, on the dedicated OrbixCOMet bridge machine (if applicable), or on the server machine. The OrbixCOMet bridge does not care where the Interface Repository is located, as long as it can be accessed using IIOP. You can use the TYPEMAN_IFR_HOST configuration variable to specify the location of the Interface Repository in your system.

Depending on your system requirements, other Orbix executables (for example, lsit, putit, rmit, and so on) might also be required. These utilities are normally only used for system administration purposes after application setup. Refer to the *Orbix Administrator's Guide C++ Edition* for more details of these utilities.

Orbix Runtime Libraries

The *install-dir*\bin directory includes the following required Orbix runtime libraries:

- ITM_M*xx*.DLL
- ITGLM*xx*.DLL
- ITCDRM*xx*.DLL
- ITOLM*xx*.DLL
- ITLIM*xx*.DLL
- ITLMM*xx*.DLL

Installing the OrbixCOMet Runtime

The OrbixCOMet runtime environment requires considerably less disk space than a full installation of OrbixCOMet on a development machine. This section describes the requirements for installing the OrbixCOMet runtime.

OrbixCOMet Runtime Libraries and Support Files

The *install-dir*\COMet\bin directory includes the following required OrbixCOMet runtime libraries and support files:

- CCIExWrapper.dll*
- custsur.exe
- ITCplx.dll*
- ITGeneric.dll*

- ITMisc.dll
- ITStdObjs.dll *
- ITStdPS.dll *
- ITts2tlb.dll
- ITUnknown.dll
- TSM.dll
- TSMlog.dll
- itpwd.dll
- ITSSLWrapper.DLL

Note: The files marked with an asterisk (*) in the preceding list must be explicitly registered with COM. You can register all the files simultaneously, by running the regcomet.bat file in the *install-dir*\COMet\bin directory. Alternatively, you can register each file individually, using the regsvr32 *dllname* command.

The custsur.exe file is only required if OrbixCOMet is being used out-of-process from DCOM clients. However, you should distribute the entire contents of the *install-dir*\COMet\bin directory, to ensure that you have all the required files.

OrbixCOMet Configuration File

The *install-dir*\config\OrbixCOMet.cfg is required both on the client and server hosts. This is placed in the same \config directory as the Orbix configuration files, as already described in "Orbix Configuration Files" on page 148.

You must modify the configuration entries in this file appropriately for your system. When specifying a pathname for a specific directory, you must provide the full pathname and ensure that it is valid. The Orbix daemon must have read/write permissions on the directories specified in these pathnames. Refer to "OrbixCOMet Configuration" on page 353 for details about the various configuration entries in this file.

Type Libraries

If your client references any type libraries, they must be installed on the client machine, and registered in the Windows registry. You can register a type library, using the supplied `tlibreg` utility. Refer to "Creating a Type Library" on page 171 for more details.

Handler DLLs

If you have built any handler DLLs, you must deploy and register them on each machine where you have installed OrbixCOMet. Refer to "Using Handler DLLs" on page 153 for more details of how to register your handler DLLs.

DCOM

If the OrbixCOMet bridge is on a separate machine, your client must be able to make the DCOM `CoCreateInstanceEx()` call to create a remote instance of the CORBA object factory on your client machine. To do this, however, some Automation controllers (for example, Visual Basic 5.0) require that the `CCIExWrapper.dll` supplied with OrbixCOMet is installed and registered on the client machine. The `CCIExWrapper.dll` file wraps the call to `CoCreateInstanceEx()` and allows Automation clients to communicate with a remote OrbixCOMet bridge, using DCOM. You can register this DLL as follows:

```
c:\> regsvr32 CCIExWrapper.dll
```

Note: Visual Basic 6.0 allows you to pass an extra host parameter to `CreateObject()`, which bypasses the need for `CCIExWrapper.dll`.

Minimizing the 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 that you can use a centralized installation of the OrbixCOMet bridge for your clients, which provides the deployment option of using DCOM as the wire protocol for communication between the client and the bridge.

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 then forwards 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. Currently, 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`, which is a small (less than 40K) ActiveX that that can be automatically downloaded with your web page, and allows connection to a remote instance of the OrbixCOMet bridge. The examples provided in the *`install-dir`*`\demos\COMet\IE` directory 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 can choose 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).

As in the case of Internet-based deployment, you can use the supplied `CCIExWrapper.DLL` to limit the OrbixCOMet footprint to less than 40K.

If you are using early binding (that is, dual interfaces), you must include the Automation type library that you created, using the `cometcfg` GUI tool or the `ts2tlb` command-line utility. This allows 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 must deploy and register a proxy/stub DLL for all the COM interfaces that your client uses. OrbixCOMet can automatically generate the MIDL definitions and makefile, which are needed to create this DLL, using the `cometcfg` GUI tool or the `ts2idl` command-line utility.

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 *install-dir*\COMet\bin directory.

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, using the OrbixCOMet SrvAlias tool. For deployment purposes, an additional tool AliasSrv, can be used to restore these settings during installation. See the *install-dir*\demos\COMet\COM\coCreate demonstration for details. (Refer to "Replacing an Existing DCOM Server" on page 177 for more details.))

Using Handler DLLs

An important feature of OrbixCOMet is the way it facilitates the use of customized handlers to inject extra functionality into your applications at runtime. Handlers normally implement functionality such as smart proxies, filters, transformers, and iocallbacks for connection events. You can use the LoadHandler() method on the (D)IOrbixORBObject API to load additional handlers into memory during runtime of your OrbixCOMet applications, and thus avail of the extra functionality that those handlers provide.

Creating and Registering Handler DLLs

Before you can load a hander into an OrbixCOMet application, you must first generate some code that wraps the handler file and encapsulates it into a Windows DLL. OrbixCOMet provides a ts2sp command line utility that generates and builds handler DLLs. This utility can also register handler DLLs with the Windows registry and OrbixCOMet. Refer to "Development Support Tools" on page 157 for full details about using the ts2sp utility to create handler DLLs.

Loading Handler DLLs at Runtime

After a handler DLL has been built and registered, it can be loaded for use by a COM or Automation client. The following code example is taken from a Visual Basic client that loads a handler DLL called `myCOMetFilter`:

```
Dim objORB As DIOrbixORBObject
Dim objFactory As DICORBAFactoryEx
Set objORB = CreateObject("CORBA.ORB.2")

'explicitly load the handler
objORB.LoadHandler ("myCOMetFilter")
```

This adds the functionality implemented by `myCOMetFilter` into the application process for use by the ORB runtime.

Managing Handler DLLs

When a DLL file is being referenced by a process, a read-only restriction is placed on the DLL, to prevent it from being deleted or modified by another process. This restriction can also prevent you from rebuilding a DLL until it is released by the process that has loaded it. A typical scenario for this is when you use Active Server Page (ASP) scripts to access your CORBA server. Some Microsoft applications, such as Internet Explorer and Excel, tend to hold onto any DLLs that have been loaded using calls to `CreateObject()`.

To release a handler DLL, you must shut down and restart the application that is holding a reference to the DLL. If you cannot shut down the application itself, you should restart the machine on which it is running. If you are using Visual Basic, it is possible to force a handler DLL to be released by using code that forces OrbixCOMet to shut down. For example:

```
Dim objORB As DIOrbixORBObject
Dim objFactory As DICORBAFactoryEx

Set objORB = CreateObject("CORBA.ORB.2")
objORB.StartUp
objORB.SetConfigValue "COMET_SHUTDOWN_POLICY", "Explicit"

Set objFactory = CreateObject("CORBA.Factory")
objORB.LoadHandler ("YourHandler")
myObj = objFactory.GetObject(…)
```

```
…

objORB.ShutDown
Set objFactory = Nothing
Set objORB = Nothing
```

In the preceding example, the OrbixCOMet runtime is stopped after the call to `ShutDown`, and you cannot make any more calls through OrbixCOMet for the rest of the application run.

Handler DLLs generated by OrbixCOMet include a `DllMain()` method that you can use to perform initialization or deletion procedures when the DLL is loaded or unloaded. `DllMain()` contains a `reason` parameter. When Windows attaches your handler DLL to a process, it calls `DllMain()` and passes a value of `DLL_PROCESS_ATTACH` to the `reason` parameter. Similarly, when Windows releases a DLL, it calls `DllMain()` and passes a value of `DLL_PROCESS_DETACH` to the `reason` parameter. The following is an example of how `DllMain()` is implemented:

```
BOOL APIENTRY DllMain(HANDLE hInst, ULONG reason, LPVOID) {
    if (reason == DLL_PROCESS_ATTACH) {
    // perform your handler-specific initialization
    // here
    }
    else if (reason == DLL_PROCESS_DETACH) {
    // perform your handler-specific destruction or
    // garbage collection here
    }
    return TRUE;
}
```

You should avoid any thread-specific calls within `DLLMain()`. This is because Windows suspends all threads, except the currently running thread, on entry to that function. You do not have to use `DLLMain()` to perform initialization and deletion procedures. You could instead use a static object within the DLL, where the constructor of the static object performs initialization, and the destructor of the static object performs termination. Refer to the demonstrations in the `demo\corbasrv` directory for an example.

13

Development Support Tools

OrbixCOMet is a high-performance bridge that stores OMG IDL and type library information at the bridging location in an ORB-neutral binary format. The OrbixCOMet type store holds a cache of this type information, which is used by the dynamic bridge during runtime of your OrbixCOMet applications. This chapter describes the type store and the central role it plays in terms of the development support tools supplied with OrbixCOMet. It also describes the GUI and command-line versions of the development support tools that allow you to maintain the type store cache, and to create IDL files, type libraries, handler DLLs, and server stub code from existing type store information. Finally, it describes the tools that you can use to replace an existing COM or Automation server with a CORBA server.

Both a GUI version and command-line version of the development support tools are supplied with OrbixCOMet. The GUI tool and command-line utilities 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, changes made to the type store via the command line are not automatically reflected on the GUI interface. Refer to “The OrbixCOMet Tools GUI Screen” on page 160 for details of how to refresh the GUI interface to see command-line changes.

The Central Role of the Type Store

Figure 13.1 is a graphical overview of the central role played by the type store in the use of the OrbixCOMet development utilities. As shown in Figure 13.1, the `typeman` utility manages the information in the type store, while other utilities use the type store information to generate the new type definitions and code that are required for the development of distributed COM/CORBA applications.

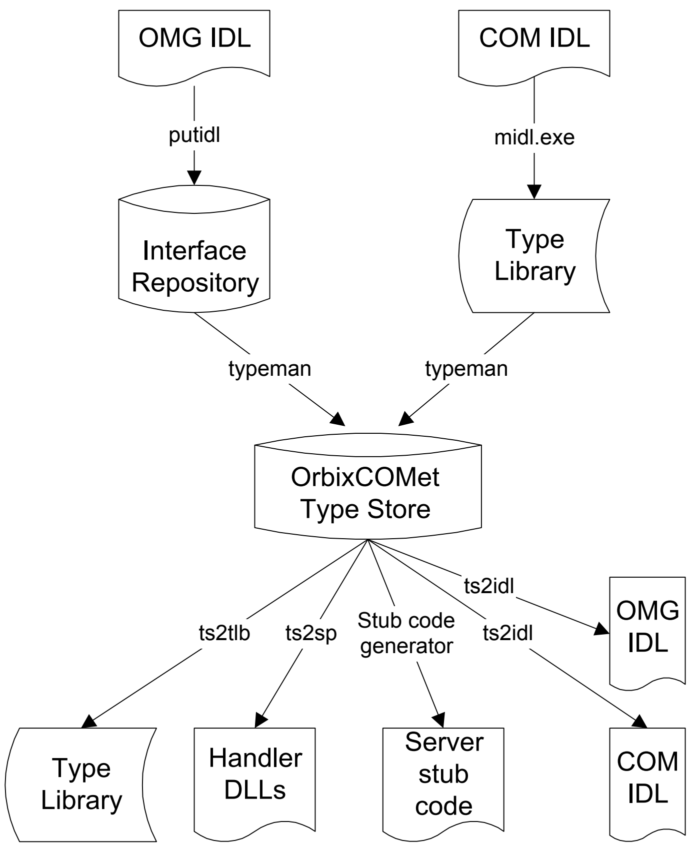

Figure 13.1: *OrbixCOMet Type Store and the Development Utilities*

The Caching Mechanism of the Type Store

As shown in Figure 13.1 on page 158, OMG IDL files define the IDL interfaces for CORBA objects. You can store OMG IDL in the Interface Repository in binary format, using the `putidl` command. Similarly, COM IDL files define the IDL interfaces for COM or Automation objects. When you run the COM IDL compiler, `midl.exe`, it automatically creates a type library that stores the COM IDL in binary format.

OrbixCOMet uses the type information available in the Interface Repository and type libraries. However, a possible performance bottleneck might result at application runtime if OrbixCOMet had to contact the Interface Repository for each OMG IDL definition, and contact type libraries for each COM IDL definition. This is because every query might involve multiple remote invocations. To avoid any bottleneck, OrbixCOMet uses a memory and disk cache of type information. This means it only has to query the Interface Repository once for each OMG IDL definition, and query the type library once for each COM IDL definition.

The `typeman` utility converts OMG IDL and COM IDL type information into an ORB-neutral binary format, and caches it in memory. The type information can consist of module names, interface names, or data types. At application 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 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 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 351.) When lookups are performed, if the required type information is not already in the memory cache, `typeman` pulls it out of the disk cache. If the required type information is not already in the memory or disk cache, `typeman` pulls it out of the Interface Repository or type library (depending on whether it is an OMG IDL or COM IDL type definition). The related type information then becomes the most recent item in the queue in the type store memory cache.

The OrbixCOMet Tools GUI Screen

Note: You can ignore this section if you intend using the command-line utilities only. However, you must use the GUI tool if you want to generate server stub code from existing type store information.

If you are using the GUI tool, the **OrbixCOMet tools** screen in Figure 13.2 is always your initial starting point. To open the **OrbixCOMet tools** screen, enter `cometcfg` on the command line, or select the **COMet tools** option on the Windows **Start-IONA-Orbix3.3** menu path.

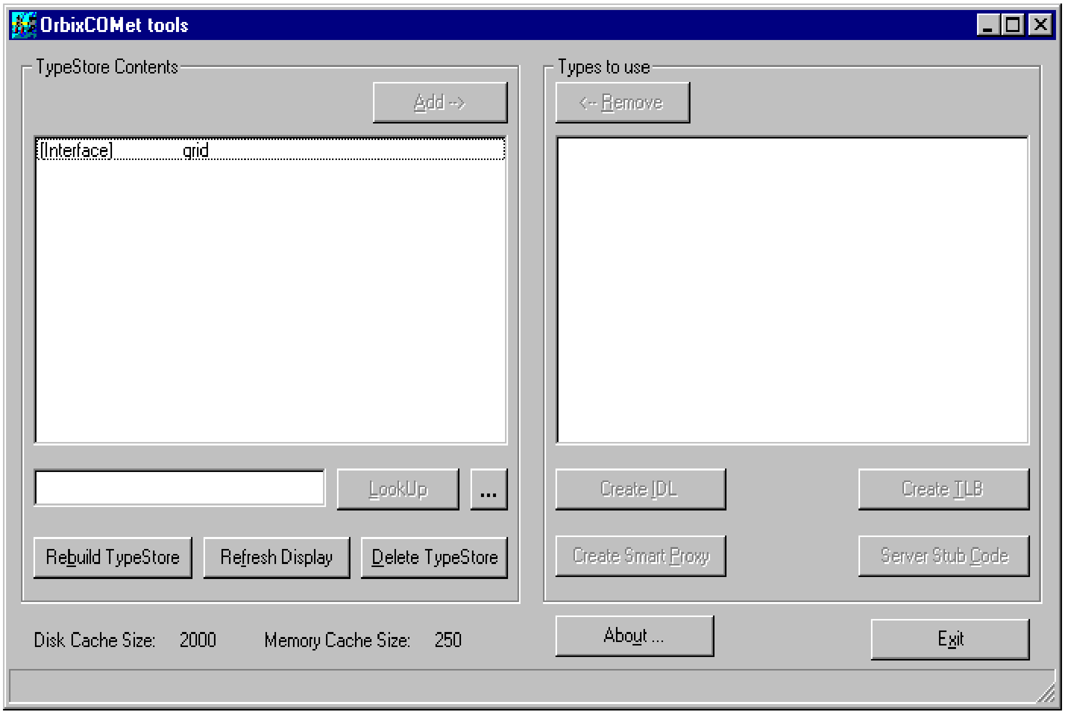

Figure 13.2: *OrbixCOMet Tools Screen*

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 an ORB-neutral binary format, regardless of whether it has originated from OMG IDL files or type libraries. 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 file from cached type library information.
- Create a COM IDL file or type library from cached OMG IDL information.
- Create PowerBuilder or Visual Basic server stub code.

If you are using both the GUI tool and the command-line utilities, changes made to the type store cache via the `typeman` command-line utility do not appear automatically in the **TypeStore Contents** panel on the **OrbixCOMet tools** screen, shown in Figure 13.2 on page 160. In this case, select the **Refresh Display** button to reflect any changes that you have made via the command line.

Location of the Command-Line Utilities

The command-line utilities described in this chapter are located in the *`install-dir`*`\COMet\bin` directory, where *`install-dir`* represents the Orbix installation directory.

Adding New Information to the Type Store

"The Caching Mechanism of the Type Store" on page 159 has described how the type store cache can obtain its information on an as-needed basis at application runtime. However, users can choose to add the required type information to the cache before the first run of an application. This is known as *priming* the cache, and it can lead to a notable performance improvement.

Priming the cache is a useful but optional step that helps to optimize the first run of an OrbixCOMet application that is using previously unseen OMG IDL or COM IDL 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 for that type information are satisfied by the cache.

As shown in Figure 13.1 on page 158, you can add both OMG IDL and type library information to the type store, using either the GUI tool or the command-line utilities.

Note: An OMG IDL interface must be registered in the Interface Repository, using `putit`, before you can add it to the OrbixCOMet type store. This is because `typeman` queries the Interface Repository for OMG IDL type information not currently held in the type store cache.

Using the GUI Tool

Use the **OrbixCOMet tools** screen shown in Figure 13.2 on page 160. 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 an OMG IDL interface that you want to add.
   - Select the browse button, which is marked by an ellipsis (that is, ...). This provides you with a dialog box containing a list of type library names. Select a type library name to return it to the field.
2. Select the **LookUp** button. If you have entered an OMG IDL interface name, OrbixCOMet searches both the type store cache and the Interface Repository for the specified name. If the relevant name is not already in the cache, and it is found in the Interface Repository, it is then automatically added to the cache. Similarly, if you have selected a type library name, OrbixCOMet searches the type store cache for the specified name. If the relevant name is not already in the cache, it is then automatically added to the cache.

Using the Command-Line Utilities

The typeman utility adds information to the type store cache. For example, the following command adds the grid interface to the type store:

```
typeman -e grid
```

You can call up the usage string for typeman as follows:

```
typeman -?
```

The usage string for typeman is:

```
TypeMan [filename | -e name|uuid|TLBName] [-v[s[i] method]]
        [options]

        filename: Name of input text file.
       -e:  Look up entry (name, {uuid} or type library pathname).
        -c[n][u]: List disk cache contents, n: Natural order,
                  u: display uuid.
        -w[m]: Delete (wipe) cache contents. [m]: Delete uuid-
               mapper contents.
        -f: List type store data files.
        -r: Resolve all references (use to generate static
           bridge compatible names for CORBA sequences).
        -i:  Always connect to IFR (for performance comparisons).
        -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[+|tlb|union]: Log TS basic contents ['+' shows new's/
                       delete's]. tlb: TypeLib, union: Logs OMG IDL
                        for unions.

        -?2: Priming input file format info.
```

Refer to “OrbixCOMet Utility Options” on page 361 for details of each of the options available with typeman.

Priming the Type Store with an Individual Entry

To prime the type store with the type information for an individual entry, specify one of the following with the `typeman` command:

- An OMG IDL typename.
- A fully qualified type library pathname.
- The UUID of a COM IDL interface.

For example, to prime the cache with the OMG IDL `mymodule::mygrid` interface, enter:

```
typeman -e mymodule::mygrid
```

In this case, the `-e` option instructs `typeman` to query the Interface Repository for the specified `mygrid` interface, and then add it to the type store. Ensure that you enter the fully scoped name of the OMG IDL type, as shown.

Note: Remember, OMG IDL interfaces must be registered in the Interface Repository, using `putit`, before you can add them to the OrbixCOMet type store. If `typeman` cannot find the relevant interfaces in the Interface Repository, it cannot add the relevant type information to the cache.

To prime the cache with the `mytypelib` type library, held in `c:\temp`, enter:

```
typeman -e c:\temp\mytypelib
```

In this case, the `-e` option instructs `typeman` to prime the cache with the type information for `mytypelib`. The full path to the type library must be entered.

To prime the cache with the UUID of a COM IDL interface, enter:

```
typeman -e {UUID}
```

In this case, replace *`UUID`* with the actual UUID. Remember to enclose the UUID in opening and closing braces, as shown.

Priming the Type Store with Multiple Entries

To prime the type store with multiple entries simultaneously, create a text file that lists any number and combination of the following:

- OMG IDL typenames.
- Fully qualified type library pathnames.
- COM IDL UUIDs.

You can call the text file any name you want (for example, `prime.txt`). Each entry in the text file must be on a separate line. For example:

```
MyAccount
// This is a comment about mytypelib
c:\temp\mytypelib
// This is a comment about the UUID
{00020813-0000-0000-C000-00000000046}
Chat::ChatClient
Chat::ChatServer
```

As shown in the preceding example, OMG IDL typenames must be fully scoped, type library pathnames must be supplied in full, and UUIDs must be enclosed in opening and closing braces. You can comment out a line by putting // at the start of it. If you insert a double blank line, it is treated as the end of the text file. The `-?2` option with `typeman` allows you to view the format that the text file entries should take.

After you have created the text file, enter the following command (assuming you have called the file `prime.txt`), to prime the cache with the type information relating to the text file entries:

```
typeman prime.txt
```

This can be a convenient way of managing the cache, and repriming it with a modified list of types. Refer to "Rebuilding the Type Store" on page 167 for more details.

Deleting the Type Store Contents

You can delete the entire contents of the type store, using either the GUI tool or the command-line utilities. It is not possible to selectively delete only some type store entries. To delete entries, you must delete the entire cache.

Using the GUI Tool

To delete the entire contents of the type store, select the **Delete TypeStore** button on the **OrbixCOMet tools** screen shown in Figure 13.2 on page 160.

Using the Command-Line Utilities

Either of the following commands deletes the entire contents of the type store:

```
typeman -wm
```

or

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

In this case, the second command assumes the `typeman` data files are held in `c:\temp`. (The `COMet.TypeMan.TYPEMAN_CACHE_FILE` configuration variable determines where the data files are stored. Refer to "OrbixCOMet Configuration" on page 351 for more details.) The `typeman` data files include:

`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.

Note: The `typeman -w` command does not delete the `typeman.map` file. You must enter `typeman -wm` to ensure that this file is also deleted.

Rebuilding the Type Store

You can rebuild the type store from a record of existing entries, using either the GUI tool or the command-line utilities.

Using the GUI Tool

To automatically rebuild the type store from a record of existing entries, select the **Rebuild TypeStore** button on the **OrbixCOMet tools** screen shown in Figure 13.2 on page 160.

Using the Command-Line Utilities

Rebuilding the type store from the command line involves first deleting the type store contents as described in "Deleting the Type Store Contents" on page 166, and then re-priming the cache, using the `typeman` utility. If you wish, you can create a single text file that contains all the Interface Repository and type library entries that you want to add. Refer to "Priming the Type Store with Multiple Entries" on page 165 for more details.

Dumping the Type Store Contents

The `typeman` utility is also a useful diagnostic utility, in that it allows for dumping the contents of the type store cache. For example, the following command prints the methods of the `grid` interface in alphabetical order and also in vtable order (this order is determined by the *COM/CORBA Interworking* specification at `www.omg.org`):

```
[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 the preceding example denotes attribute `get` operations. The absence of `height set` and `width set` implies that these are read-only attributes.

Creating an IDL File

The normal procedure for writing a CORBA client to contact a COM or Automation server is to first obtain an OMG IDL definition of the target COM or Automation interface, which the CORBA client can understand. Similarly, the normal procedure for writing a COM or Automation client to contact a CORBA server is to first obtain a COM IDL definition of the target CORBA interface, which the COM or Automation client can understand. As shown in Figure 13.1 on page 158, you can generate OMG IDL definitions from existing type library information in the type store, and you can generate COM IDL definitions from existing OMG IDL information in the type store. You should ensure that each IDL file contains a module, to minimize manual lookups.

Note: Creating COM IDL in this way allows you to create a standard DCOM proxy/stub DLL that can be installed with a COM or Automation 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 or Automation client and a proxy/stub DLL.

Using the GUI Tool

To create an IDL file from the **OrbixCOMet tools** screen in Figure 13.2 on page 160:

1. If you want to create an OMG IDL file, select an item of COM IDL type information from the **TypeStore Contents** panel. If you want to create a COM IDL file, select an item of OMG IDL type information from the **TypeStore Contents** panel.

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 13.3.

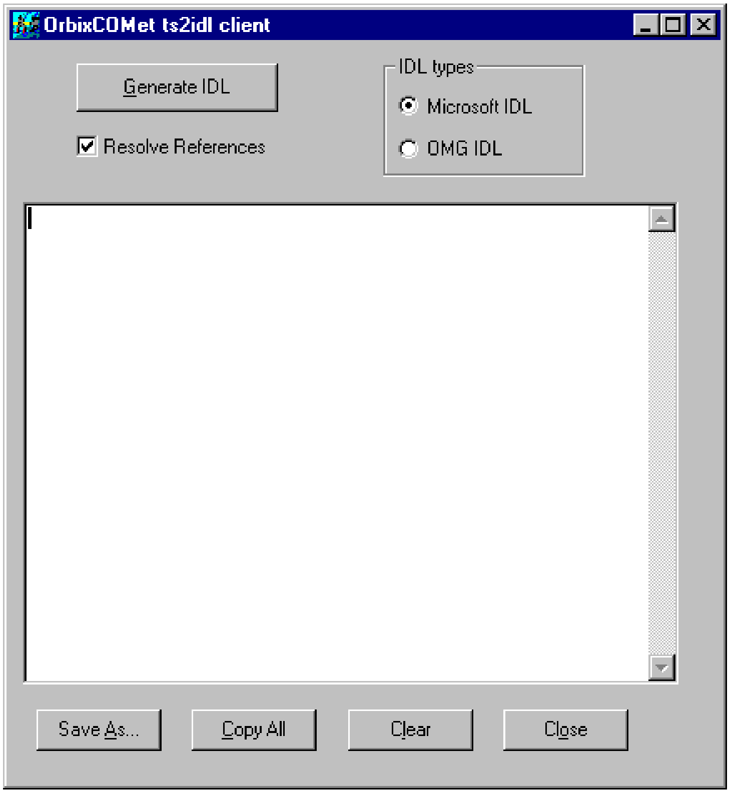

Figure 13.3: *Creating an IDL File*

4. If you are creating a COM IDL file from OMG IDL type information, select the **Microsoft IDL** radio button. If you are creating an OMG IDL file from COM IDL type information, select the **OMG IDL** radio button.

5. If you want to:
   - 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.
6. Select the **Generate IDL** button. This creates the IDL file.

Using the Command-Line Utilities

The `ts2idl` utility creates an IDL file from existing type information in the type store. For example, the following command creates a `grid.idl` file, based on the `grid` interface:

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

You can call up the usage string for `ts2idl` as follows:

```
ts2idl -v
```

The usage string for `ts2idl` is:

```
Usage:
ts2idl [options] <type name | type library 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.
   -i : Generate OMG IDL.
   -m : Generate COM IDL (default).
   -p : Generate makefile for proxy/stub DLL.
   -s : Force inclusion of standard types (ITStdcon.idl / orb.idl).
   -f : <filename>.
   -v : Print this message.

   Tip : Use -p to generate a makefile for the marshalling DLL.
```

Refer to "OrbixCOMet Utility Options" on page 361 for details of each of the options available with `ts2idl`.

For more complicated interfaces that use user-defined types, you can use the `-r` option with `ts2idl`, to completely resolve those user-defined types and produce COM IDL for them also.

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

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 want to develop an Automation client to contact a CORBA server, and the Automation client will only use straight `IDispatch` interfaces, there is no need to create a type library from existing OMG IDL information in the type store. This is because OrbixCOMet automatically copies the related type information into the type store when it performs a lookup on the target CORBA object, using `GetObject()`.

The following is a Visual Basic example of how an Automation client can use `GetObject()` to get an object reference to a CORBA object:

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

However, if you want to develop an Automation client that uses dual interfaces, you must create a type library from existing OMG IDL information in the type store, using either the GUI tool or the command-line utilities.

Using the GUI Tool

To create a type library from the **OrbixCOMet tools** screen in Figure 13.2 on page 160:

1. From the **TypeStore Contents** panel, select an item of OMG IDL type information on which you want to base 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 13.4.

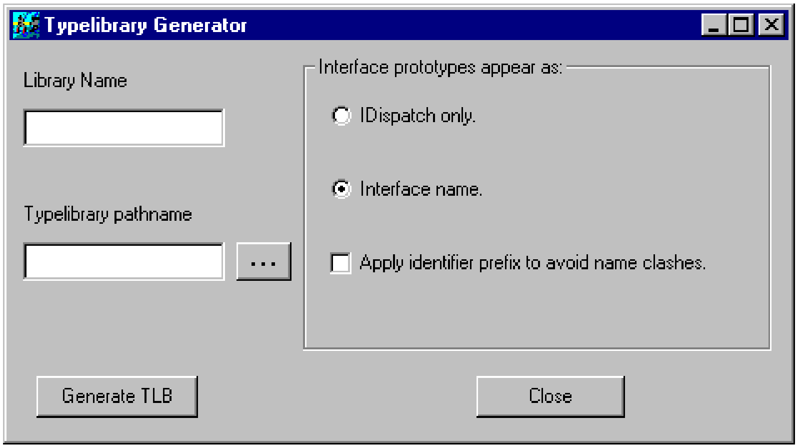

Figure 13.4: *Creating a Type Library from OMG IDL*

4. In the **Library Name** field, type the internal library name. This can be the same as the type library pathname if you wish, but ensure that the library does not have the same name as any of the types that it contains.
5. In the **Typelibrary pathname** field, type 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 COM IDL keywords.
8. Select the **Generate TLB** button. This creates the type library.

Using the Command-Line Utilities

The ts2tlb utility creates a type library from existing OMG IDL type information in the type store. For example, the following command creates a grid.tlb file in the IT_grid library, based on the OMG IDL grid interface:

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

You can call up the usage string for ts2tlb as follows:

```
ts2tlb -v
```

The usage string for ts2tlb is:

```
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.
```

Refer to “OrbixCOMet Utility Options” on page 361 for details of each of the options available with ts2tlb.

Generating a Handler DLL

OrbixCOMet is shipped with a set of pre-built DLLs that act as a dynamic bridge between CORBA and COM environments. OrbixCOMet also allows you to generate additional DLLs to encapsulate any extra handler code that you might have developed and want to load into your OrbixCOMet applications at runtime, to provide extra functionality. Handlers can implement functionality such as smart proxies, filters, transformers, and iocallbacks for connection events. The `(D)IOrbixORBObject` interface contains a `LoadHandler()` method that can load handler DLLs into memory when you are running an OrbixCOMet application. (Refer to "DIOrbixORBObject" on page 205 or "IOrbixORBObject" on page 239 for more details about `LoadHandler()`.)

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.

The `ts2sp` command-line utility generates handler DLLs from existing type information in the type store. For example, the following command generates a handler DLL called `ClientFilterH.dll`, based on the original handler code contained in `MyFilter.cpp`:

```
ts2sp -n myCOMetFilter -p ClientFilter -f MyFilter.cpp
```

In the preceding example, the `-p` option instructs `ts2sp` to generate a makefile called `ClientFilter.mak`. You can then use this makefile to build the handler DLL, as follows:

```
nmake -f ClientFilter.mak
```

The preceding command builds the handler DLL, assigns it the filename `ClientFilterH.dll`, and registers it in the Windows registry. (Remember, to build a handler DLL, you must have an Orbix development system and Visual C++ installed.)

The `-p` option also creates some support code in `ClientFilter.cpp` and `ClientFilter.h`. This support code is used by the generated handler DLL to register itself with OrbixCOMet. The handler DLL registers itself, using the name you supply with the `-n` option to `ts2sp` (in this case, `myCOMetFilter`) and the full path to the DLL. This allows OrbixCOMet to subsequently recognize it as a valid handler.

You can call up the usage string for `ts2sp` as follows:

```
ts2sp -v
```

The usage string for `ts2sp` is:

```
Usage:
ts2sp <options> interface1 […interfaceN]
   -v : Show this screen.
   -n <keyname> : Keyname of handler DLL.
   -m : Do not overwrite the makefile.
   -p <project> : Specify project name (name of .mak file etc.).
   -f <file> : Specify additional source file (files
               implementing the smart proxies).
   -d <output dir> : (optional) Specify output directory
                     (default to current directory).

   NOTE: Any additional source files are assumed to be in the
         directory indicated by the -d option.
```

Refer to “OrbixCOMet Utility Options” on page 361 for details of each of the options available with `ts2sp`.

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 objects or callback objects. You can use the GUI tool to generate stub code for Visual Basic and PowerBuilder servers. (Refer to the *OrbixCOMet Release Notes* for details of the programming language versions supported by this release).

To create server stub code from the **OrbixCOMet tools** GUI screen in Figure 13.2 on page 160:

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 13.5.

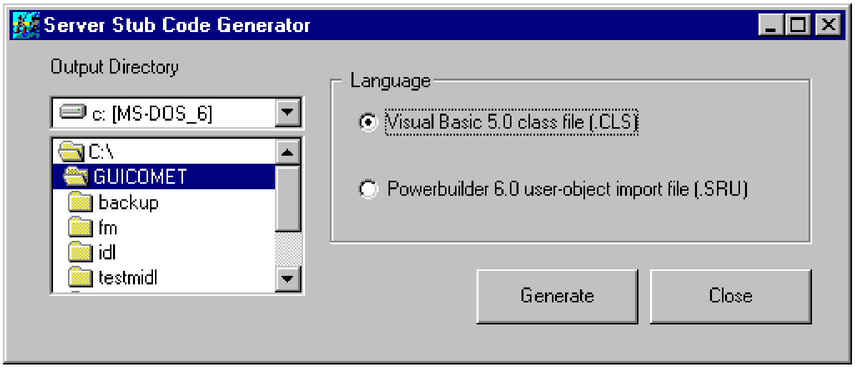

Figure 13.5: *Generating Server Stub Code*

4. Select the radio button corresponding to the language you are using.
5. Select the target directory where you want the code to be saved.
6. Select the **Generate** button. This generates the stub code.

Replacing an Existing DCOM Server

At some stage, it might become necessary to replace an existing COM or Automation server with a CORBA server, without the opportunity to modify existing COM or Automation clients. However, such clients are not 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 COM or Automation server. This ensures that when the client is subsequently run, the bridge is activated in response to the client's CoCreateInstanceEx() or CreateObject() calls.

OrbixCOMet supplies a srvAlias utility, which you can enter at the command line, to open the **Server Aliasing Registry Editor** screen shown in Figure 13.6 on page 178. The screen in Figure 13.6 allows you to place some entries in the registry, to allow server 'aliasing'. 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 utility allows users to save the new registry entries in binary format, so that an accompanying aliassrv utility can be used at application deployment time to restore the entries on the destination machine. For example, given a file called replace.reg, which contains the modified registry entries, the following command aliases 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 is used instead. See the *install-dir*\demos\COMet\corbasrv\replace directory for an example of srvalias and aliassrv in action.

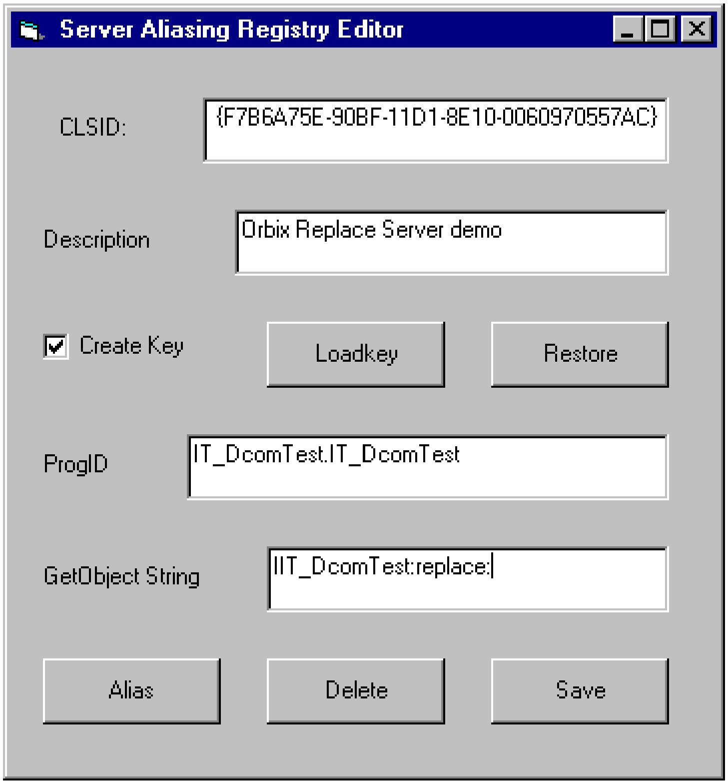

Figure 13.6: *Aliasing the Bridge*

Part III

Programmer's Reference

14

OrbixCOMet API Reference

This chapter describes the application programming interface (API) for OrbixCOMet, which is defined in COM IDL. It is divided into two main sections. The first section provides the API reference for Automation. The second section provides the API reference for COM.

Automation Interfaces

This section describes the Automation API 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 89 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 A bridge exposes an Automation interface, which allows the bridge to act as a CORBA server. This interface can be obtained, using the `ServerAPI` ProgID.

The Automation server should instantiate an object of this type and use it to control the Automation server's behavior 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, using `putit`. After `Activate()` is called, your server is ready to receive incoming requests from CORBA clients. You should register all your implementation objects, using `SetObjectImpl()`, before calling `Activate()`.

`Deactivate()`	This deactivates your application as a CORBA server. After `Deactivate()` is called, your application can no longer process incoming requests from CORBA clients. The `cServerName` parameter contains the name of the CORBA server. The server must be registered with this name in the 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 depends primarily on your development environment.
`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 contained in the `CIFace` parameter, with the Orbix marker contained in the `cMarker` parameter. (Markers are used to uniquely identify different instances of the same interface.) If no marker is passed, Orbix automatically selects 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 Orbix daemon.
`SetObjectImplPersistent()`	See `SetObjectImpl()`. The `CIORFileName` parameter indicates where to write the IOR for the object.

DCollection

Synopsis

```
[oleautomation,dual,uuid(...)]
interface DCollection : DIForeignComplexType {
    [propget,id(100)] HRESULT Count([retval,out] long* IT_retval);
    [propput,id(100)] HRESULT Count([in] long val);
    [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 `DCollection` interface.

Methods

`Count()`	This sets or gets 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, using `propget`, or inserts an item into the collection at the specified index, using `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

Synopsis

```
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);
};
```

Description The OMG IDL any type translates to the `DICORBAAny` Automation interface. 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 it contains. The kind of value that the `DICORBAAny` contains can be found, using the `kind()` method. This method returns an enumerated value of the `CORBATCKind` type. For example, a

DICORBAAny containing a struct is of the tk_struct kind; a DICORBAAny containing an object is of the tk_objref kind; a DICORBAAny containing a typedef is of the tk_alias kind.

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 the CORBATCKind type. For example, an any that contains a sequence is of the tk_sequence kind. 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 of the tk_objref, tk_struct, tk_union, tk_enum, tk_alias, or tk_except kind. 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 runtime access to the Interface Repository.

`name()`

This can be called on a DICORBAAny of the tk_objref, tk_struct, tk_union, tk_enum, tk_alias, or tk_except kind. If called on a DICORBAAny of a different kind, it raises a BadKind exception.

It returns the name that identifies the type. The returned name does not contain any scoping information.

`member_count()`

This can be called on a DICORBAAny of the 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 of the tk_struct, tk_union, tk_enum, or tk_except. If called on a DICORBAAny of a different kind, it raises a BadKind exception.

The member_name() method returns the name of the member specified in the index parameter. The returned name 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. The index starts at 0.

member_type()	This can be called on a DICORBAAny of the tk_struct, tk_union, or tk_except kind. 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. The index starts at 0.
member_label()	This can be called on a DICORBAAny of the tk_union kind. If called on a DICORBAAny of a different kind, it raises a BadKind exception. The member_label() method returns the case label of the union member identified by the index parameter. (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. The index starts at 0.
discriminator_type()	This can be called on a DICORBAAny of the tk_union kind. 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 of the tk_union kind. If called on a DICORBAAny of a different kind, it raises a BadKind exception. The default_index() method returns the index of the default member; it returns -1 if there is no default member.

`length()`	This can be called on a `DICORBAAny` of the `tk_string`, `tk_sequence`, or `tk_array` kind. For a bounded string or sequence, `length()` returns the value of 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` of the `tk_sequence`, `tk_array`, or `tk_alias` kind. 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, `content_type()` 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* specification 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 parameter to `GetObject()` can take either of the following formats:

- *interface*:*marker*:*server*:*host*
- *interface*:*TAG*:*Tag data*

The components of the string can be described as follows:

interface—This is the IDL interface that the target object supports. If the interface is scoped (for example, `"Module::Interface"`) the interface token is `"Module/Interface"`.

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.

server—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.

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

TAG—Two type of *TAG* are allowed. Each type has a different form of *Tag data*. Valid *TAG* types are:

- *IOR*—In this case, the *Tag data* is the hexadecimal string for the stringified IOR. For example: `fact.GetObject("employee:IOR:123456789…")`
- *NAME_SERVICE*—In this case, the *Tag data* is the Naming Service compound name separated by ".". For example: `fact.GetObject("employee:NAME_SERVICE: IONA.employees.PD.Tom")`

UUID `{204F6241-3AEC-11CF-BBFC-444553540000}`

Notes Automation/CORBA-compliant.

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 creation of Automation objects, which represent the OMG IDL `struct`, `union`, and `exception` complex types.

You can create an object representing an OMG IDL complex type in a client, 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, 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 `DICORBAFactory` interface.

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 contained in the `typeName` parameter should be interpreted. Global scope is indicated by passing the `Nothing` parameter.
`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 runtime 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 `DICORBAObject` interface. 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 runtime 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, it is that server's name, if it is known. For an object created in a client program, it is 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 contained in the `repositoryID` parameter. 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 `obj` parameter. 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, 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 `maximum` parameter specifies the maximum value that is to be returned by the `Hash()` method. For example, by setting `maximum` to `7`, the object reference space is partitioned into a maximum of eight sub-spaces (because 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 supports the `DICORBAstruct` interface. 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 `DICORBASystemException` interface. (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 `COMPLETION_YES` value indicates that the operation had completed before the exception was raised.
	The `COMPLETION_NO` value indicates that the operation had not completed before the exception was raised.
	The value `COMPLETION_MAYBE` indicates that the operation was initiated, but it cannot be determined at what stage the exception occurred.

UUID `{A8B553C9-3B72-11CF-BBFC-444553540000}`

Notes Automation/CORBA-compliant.

DICORBATypeCode

Synopsis

```
[oleautomation,dual,uuid(...)]
interface DICORBATypeCode : DIForeignComplexType {
[propget] HRESULT kind ([retval,out] CORBA_TCKind * val);
// 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 `DICORBATypeCode` interface.

Methods

`kind()`	This can be called on all typecodes. It finds the type of OMG IDL definition described by a typecode. It returns an enumerated value.

`id()`	This can be called on a `DICORBATypeCode` of the `tk_objref`, `tk_struct`, `tk_union`, `tk_enum`, `tk_alias`, or `tk_except` kind.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` of the `tk_objref`, `tk_struct`, `tk_union`, `tk_enum`, `tk_alias`, or `tk_except` kind. If called on a `DICORBATypeCode` of a different kind, it raises a `BadKind` exception. It returns the name that identifies the type. The returned name does not contain any scoping information.
`member_count()`	This can be called on a `DICORBATypeCode` of the `tk_struct`, `tk_union`, `tk_enum`, or `tk_except` kind. 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` of the `tk_struct`, `tk_union`, `tk_enum`, or `tk_except` kind. If called on a `DICORBATypeCode` of a different kind, it raises a `BadKind` exception. The `member_name()` method returns the name of the member identified by the `index` parameter. The returned name 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. The index starts at `0`.

`member_type()`

This can be called on a DICORBATypeCode of the tk_struct, tk_union, or tk_except kind. If called on a DICORBATypeCode 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. The index starts at 0.

`member_label()`

This can be called on a DICORBATypeCode of the tk_union kind. If called on a DICORBATypeCode of a different kind, it raises a BadKind exception.

The member_label() method returns the case label of the union member identified by the index parameter. (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. The index starts at 0.

`discriminator_type`

This can be called on a DICORBATypeCode of the tk_union kind. 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 of the tk_union kind. If called on a DICORBATypeCode of a different kind, it raises a BadKind exception.

The default_index() method returns the index of the default member; it returns -1 if there is no default member.

length	This can be called on a DICORBATypeCode of the tk_string, tk_sequence, or tk_array kind. 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 DICORBATypeCode of the tk_sequence, tk_array, or tk_alias kind. 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 {
[id(400)] HRESULT Union_d ([retval,out] VARIANT * val);
};
```

Description An Automation interface that results from the translation of an OMG IDL union definition supports the DICORBAUnion interface.

Methods

Union_d()	This returns the current value of the union's discriminant.

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 `DICORBAUserException` interface. 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 (for example, `struct`, `union`, or `exception`) supports the `DIForeignComplexType` interface.

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 since CORBA 2.2. The approved way to get a repository ID is through `DIObjectInfo::unique_id()`, and then use `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 `DIForeignException` interface.

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);
    HRESULT clone ([in] IDispatch * target,
        [optional,in,out] VARIANT * IT_Ex,
        [retval,out] IDispatch ** resultObj);
};
```

Description This allows you to retrieve 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 a new instance that is identical to the target instance.

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);
    [propget] HRESULT InterfaceName([retval,out] BSTR * name);
}
```

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 called `m` (which supports the `A` interface) in the `s` server on the `h` host:

```
' 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(…)
```

`Marker()`	The `propget` method finds the object's marker name. The `propput` method sets the object's marker name. When setting the object's marker, if you choose a marker that is already in use for an object of the same interface within the server, OrbixCOMet assigns a different marker to the object. (The object with the original marker is not affected.) You might want 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, based on the marker included in the request. Thus, if an object is made known to a remote client (refer to "Obtaining Object References" on page 68 for details of the various ways you can do this), and the object's marker is subsequently changed within the server by a call to `Marker()`, a subsequent request from the remote client fails because the client is using 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 it 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 optimization, so that a connection between a client and server that is rarely used is not kept open for long periods when not in use. The channel is automatically reopened when an invocation is made on the object. 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 retrieves 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.)
`InterfaceName`	This returns the interface name of 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);
    HRESULT Narrow([in] IDispatch * poObj,
        [in] BSTR cNewIFaceName,
        [optional,in,out] VARIANT *IT_Ex,
        [retval,out] IDispatch ** poDerivedObj);
    HRESULT GetOrbixSSL([optional,in,out] VARIANT *IT_Ex,
        [retval,out] IDispatch ** IT_retval);
    HRESULT ReleaseCORBAView([in] IDispatch * poObj,
        [in] VARIANT_BOOL 1ToDestruction,
        [optional,in,out] VARIANT *IT_Ex,
        [retval,out] VARIANT_BOOL * IT_retval);
    HRESULT UseTransientPort([in] VARIANT_BOOL OnOff,
        [optional,in,out] VARIANT *IT_Ex,
        [retval,out] VARIANT_BOOL * IT_retval);
};
```

Description `DIOrbixORBObject` provides Orbix-specific methods that allow programmers 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 `DIORBObject` interface.

The ORB has the ProgID, `CORBA.ORB.2`, which 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 most cases. The value set by this method 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 the `AbortSlowConnect()` method, 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 favored over new connection attempts. The `EagerListeners()` method 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 retries the attempt every two seconds until either the call can be made or there have been too many retries. The `MaxConnectRetries()` method 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 is only used if it is set to zero.

`ReSizeObjectTable()`

All Orbix implementation objects in an address space are registered in its object table, which is 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 following value in the `CORBA.h` file:

`CORBA_OBJECT_TABLE_SIZE_DEFAULT`

If you anticipate that a program will handle a much larger number of objects than the default size (which is about 1,000), you can use this function to resize the table.

`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 has no affect on the target object. The pinging causes the target server process to be activated, if necessary, and confirms that this server recognizes the target object. Pinging can be enabled, using `PingDuringBind()`, by passing `1` to the `pingOn` parameter. Pinging can be disabled by passing `0` to `pingOn`. The previous setting is returned in the `IT_retval` parameter.

You might wish to disable pinging to improve efficiency by reducing the overall number of remote invocations. In this case, Orbix checks the object's availability only when a method is invoked on the object, and not when the bind attempt is made.

If `PingDuringBind(false)` is called:

- A `_bind()` to an unavailable object does not immediately raise an exception, but subsequent requests using the object reference returned from `_bind()` do fail and raise a `CORBA::INV_OBJREF` system exception.
- If a hostname is specified to `_bind()`, the `_bind()` itself does not make any remote calls; it simply sets up a proxy with the required fields.
- If a hostname 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.

`NoReconnectOnFailure()`	When an Orbix client first contacts a server, a single communications channel is established between the client and server. This connection is used for all subsequent communications between the client and server. The connection is closed only when the client or the server exits. When a server exits while a client is still connected, the next invocation by the client raises a system exception of the `CORBA::COMM_FAILURE` type. If the client attempts another invocation, Orbix automatically tries to re-establish the connection. This default behavior can be changed by passing the value `0` (false) to `NoReconnectOnFailure()`. In this case, all client attempts to contact a server, after the communications channel has been closed, 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 connection can block for a considerable time if a node, known to the local node, is inactive or unreachable. Set `OnOff` to `1` to abort slow connection attempts.

`ActivateCVHandler()`	Thia activates the configuration value handler specified in the `identifier` parameter. You must call `ReinitialiseConfig()` before modifications by this function can take effect. Refer to the Orbix documentation set for information on configuration handlers.
`DeactivateCVHandler()`	This deactivates the configuration value handler specified in the `identifier` parameter. You must call `ReinitialiseConfig()` before modifications by this function can take effect. Refer to the Orbix documentation set for information on configuration handlers.
`ActivateOutputHandler()`	This activates the output handler specified in the `identifier` parameter. 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 can take effect. Refer to the Orbix documentation set for information on configuration handlers.
`PlaceCVHandlerBefore()`	See `PlaceCVHandlerAfter()`.
`DeactivateOutputHandler()`	This deactivates the output handler specified in the `identifier` parameter. 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 specified in the `derived` parameter. The interface specified in the `derived` parameter is included in the list, because it is considered a base interface of itself.
`IsBaseInterfaceOf()`	This determines whether the interface specified in the `maybeBase` parameter is a base interface of the interface specified in the `derived` parameter. `IsBaseInterfaceOf()` returns `1` if the interface specified in the `maybeBase` parameter is a base interface of the interface specified in the `derived` parameter (or if the same interface is specified in both the `derived` and `maybeBase` parameter). Otherwise, it returns `0`.
`CloseChannel()`	This requests Orbix to close the underlying communications connection to the server. This method is intended as an optimization so that a connection between a client and server that is rarely used is not kept open for long periods when not in use. The channel is automatically reopened when an invocation is made on the object. 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 method is ignored when making a connection between a client and a server. It comes into effect only when the connection has been established.
`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.)
`Output()`	This outputs application's diagnostic and other output via the active output handlers. Unless overridden by an implementation of the `CORBA::ORB::UserOutput` C++ class, 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.

`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; 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 multithreaded 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.
`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. The level values are: 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. This is done 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 initializes the bridge. Invoking this method is optional. If StartUp() is not invoked, the bridge is automatically initialized when the first object is created. However, it is a CORBA guideline that an ORB should be initialized before being used. Therefore, you should call this method before doing anything else (that is, before you make any calls to `GetObject()` or `CreateType()` on `DICORBAFactory`.
`ShutDown()`	This shuts down the bridge. Invoking this method might be necessary, if you are experiencing hang-on-exit problems. After this method is called, no more invocations can be made using CORBA.
`LoadHandler()`	This forces OrbixCOMet to load the specified handler DLL into memory. Handlers can contain smart proxies, filters, transformers, and so on.

`Narrow()`	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:
	
	`Set objFact =`
	`    CreateObject("CORBA.Factory")`
	`Set orb =`
	`    CreateObject("CORBA.ORB.2")`
	`Set aObj = objFact.GetObject("A:" +`
	`    ior)`
	`Set cObj = orb.Narrow(aObj, "C")`
	`If cObj Is Nothing Then`
	`    MsgBox "Error: narrow failed"`
	`End If`

`GetOrbixSSL()`	This obtains a pointer to the `DIOrbixSSL` interface when Orbix SSL support is being used.
`ReleaseCORBAView()`	This is used by clients to free the CORBA view of a DCOM callback object when receipt of callbacks in no longer required.
`UseTransientPort()`	This is a wrapper around the Orbix call of the same name. It places a transient port number, instead of the Orbix daemon's port number, in any exported IORs.

UUID `{036A6A33-0BB3-CF47-1DCB-A2C4E4C6417A}`

Notes Orbix-specific.

DIOrbixSSL

Synopsis

```
[oleautomation,dual,uuid(...)]
interface DIOrbixSSL : IDispatch {
    HRESULT InitSSL([optional,in,out]  VARIANT *IT_Ex,
        [retval,out] int* nRet);
    HRESULT InitScopeSSL([in] BSTR cPolicyName,
        [optional,in,out]  VARIANT *IT_Ex,
        [retval,out] int* nRet);
    HRESULT SetSecurityName([in] BSTR cCertName,
        [optional,in,out] VARIANT *IT_Ex,
        [retval,out] int* nRet);
    HRESULT GetSecurityName([optional,in,out] VARIANT *IT_Ex,
        [retval,out] BSTR* cCertName);
    HRESULT SetPrivateKeyPassword([in] BSTR cPassword,
        [optional,in,out] VARIANT *IT_Ex,
        [retval,out] int* nRet);
    HRESULT HasPassword([optional,in,out] VARIANT *IT_Ex,
        [retval,out] VARIANT_BOOL *IT_retval);
};
```

Description `DIOrbixSSL` provides support for integrating SSL support into OrbixCOMet applications. A reference to this interface is retrieved, using a call to the `GetOrbixSSL()` method on the (D)`IOrbixORBObject` interface.

Methods

`InitSSL()`	This initializes the SSL library. It must be called by each OrbixCOMet SSL-enabled application before any attempts are made to bind to CORBA clients or servers, and before any calls to other `DIOrbixSSL` methods.

`InitScopeSSL()`	This instructs SSL to implement the SSL policies included in the configuration scope (`OrbixSSL.cfg`) specified in the `cPolicyName` parameter. (For further details, refer to `IT_SSL::initScope` in the *OrbixSSL C++ Programmer's and Administrator's Guide*.) The specified configuration scope can contain a value for `IT_CERTIFICATE_FILE`, which specifies the location of an X.509 certificate file. As a result of the call to `InitScopeSSL`, the identified certificate is initialized by the SSL runtime, and is associated with the application.
`SetSecurityName()`	This method is passed the location of a file containing an X.509 certificate and private key to be associated with an OrbixCOMet SSL-enabled application. (For further details, refer to `IT_SSL::setSecurityName` in the *OrbixSSL C++ Programmer's and Administrator's Guide*.)
`GetSecurityName()`	This retrieves the security name of the certificate being used by an OrbixCOMet SSL-enabled application.
`SetPrivateKeyPassword()`	This specifies the pass phrase to be used to unlock the private key of an X.509 certificate. The private key is stored in an X.509 certificate in encrypted PEM format with a secret pass phase. The private-key pass phrase is required to unlock the private key. The private-key pass phrase is generally chosen by the system administrator when creating the application certificate signing request.
`HasPassword`	This determines whether the server has received a private-key pass phrase from the server key distribution mechanism (KDM). If the server has not received a pass phrase, a valid password must be supplied, using `SetPrivateKeyPassword`.

UUID `{57fl303l-fe22-lld2-af83-00a024d8995c}`

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 `DIORBObject` interface provides Automation/CORBA-compliant methods that request the ORB to perform actions.

The ORB has the `CORBA.ORB.2` ProgID.

In OrbixCOMet, the `CORBA.ORB.Orbix` name 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 an IOR.
`StringToObject()`	This accepts a string produced by `ObjectToString()` and returns the corresponding object reference.

`GetInitialReferences()`	The IFR and the CORBA services can only be used by first obtaining a reference to an object through which the service can be used. The Automation/CORBA standard defines `GetInitialReferences()` as a way to list the available services. (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 `name` parameter specifies 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 `DICORBAObject` interface.

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 `systemIDs` parameter 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`. If the proxy is a proxy for an object in more than one object system, the order of IDs in the `systemIDs` array indicates the caller's order of preference. 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).
`GetRepositoryId()`	This returns an IFR identifier for the object. This method requires runtime access to the IFR.

UUID `{204f6242-3aec-11cf-bbfc-444553540000}`

Notes Automation/CORBA-compliant.

COM Interfaces

This section describes the COM API 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 89 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

A bridge exposes a COM interface, which allows the bridge to act as a CORBA server. This interface can be obtained, using the `ServerAPI` ProgID.

The COM server should instantiate an object of this type and use it to control the COM server’s behavior 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, using `putit`.

After `Activate()` is called, your server is ready to receive incoming requests from CORBA clients.

You should register all your implementation objects, using `SetObjectImpl()`, before calling `Activate()`.

`Deactivate()`

This deactivates your application as a CORBA server. After `Deactivate()` is called, your application can no longer process incoming requests from CORBA clients.

The `cServerName` parameter contains the name of the CORBA server. The server must be registered with this name in the 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 depends 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 contained in the `CIFace` parameter with the Orbix marker contained in the `cMarker` parameter. (Markers are used to uniquely identify different instances of the same interface.) If no marker is passed, Orbix automatically selects 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 Orbix daemon.
`SetObjectImplPersistent()`	See `SetObjectImpl()`. The `CIORFileName` parameter indicates where to write the IOR for the object.

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;
```

```
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 The OMG IDL any type translates to the `ICORBAAny` COM interface.

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* specification 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:

- *`interface:marker:server:host`*
- *`interface`*`:`*`TAG`*`:`*`Tag data`*

The components of the string can be described as follows:

`interface`—This is the IDL interface that the target object supports. If the interface is scoped (for example, "`Module::Interface`"), the interface token is "`Module/Interface`".

`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.

server—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.

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

TAG—Two types of *TAG* are allowed. Each type has a different form of *Tag data*. Valid *TAG* types are:

- *IOR*—In this case, the *Tag data* is the hexadecimal string for the stringified IOR. For example: `fact.GetObject("employee:IOR:123456789…")`
- *NAME_SERVICE*—In this case, the *Tag data* is the Naming Service compound name separated by ".". For example: `fact.GetObject("employee:NAME_SERVICE: IONA.employees.PD.Tom")`

`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 runtime 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, it is that server's name, if it is known. For an object created in a client program, it is 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` parameter. 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 `obj` parameter. 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, these object references are for different objects.

The `Hash()` method allows you to partition the space of object references into sub-spaces of potentially equivalent object references.

The `maximum` parameter specifies the maximum value that is to be returned from the `Hash()` method. For example, by setting `maximum` to `7`, the object reference space is partitioned into a maximum of eight sub-spaces (because 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 typecodes are equal. Otherwise, it returns `false`.
`kind()`	This can be called on all typecodes. It finds the type of OMG IDL definition described by the typecode. It returns an enumerated value of the `CORBATCKind` type. For example, a typecode that contains a sequence is of the `tk_sequence` kind. Once the kind of value stored by the typecode is known, the methods that can be called on the typecode are determined.

`id()`	This can be called on an `ICORBA_TypeCode` of the `tk_objref`, `tk_struct`, `tk_union`, `tk_enum`, `tk_alias`, or `tk_except` kind. 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 runtime access to the IFR.
`name()`	This can be called on an `ICORBA_TypeCode` of the `tk_objref`, `tk_struct`, `tk_union`, `tk_enum`, `tk_alias`, or `tk_except` kind. If called on an `ICORBA_TypeCode` of a different kind, it raises a `BadKind` exception. It returns the name that identifies the type. The returned name does not contain any scoping information.
`member_count()`	This can be called on an `ICORBA_TypeCode` of the `tk_struct`, `tk_union`, `tk_enum`, or `tk_except` kind. 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` of the `tk_struct`, `tk_union`, `tk_enum`, or `tk_except` kind. If called on an `ICORBA_TypeCode` of a different kind, it raises a `BadKind` exception. The `member_name()` method returns the name of the member specified in the `nIndex` parameter. The returned name does not contain any scoping information. A `Bounds` exception is raised if the `nIndex` is greater than or equal to the number of members that make up the type. The index starts at `0`.

`member_type()`	This can be called on an `ICORBA_TypeCode` of the `tk_struct`, `tk_union`, or `tk_except` kind. If called on an `ICORBA_TypeCode` of a different kind, it raises a `BadKind` exception. It returns the type of the member specified in the `nIndex` parameter. A `Bounds` exception is raised if the `nIndex` parameter is greater than or equal to the number of members that make up the type. The index starts at `0`.
`member_label()`	This can be called on an `ICORBA_TypeCode` of the `tk_union` kind. If called on an `ICORBA_TypeCode` of a different kind, it raises a `BadKind` exception. The `member_label()` method returns the case label of the union member specified in the `nIndex` parameter. (The case label is an integer, char, boolean, or enum type.) A `Bounds` exception is raised if the `nIndex` is greater than or equal to the number of members that make up the type. The index starts at `0`.
`discriminator_type()`	This can be called on an `ICORBA_TypeCode` of the `tk_union` kind. 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` of the `tk_union` kind. If called on an `ICORBA_TypeCode` of a different kind, it raises a `BadKind` exception. The `default_index()` method returns the index of the default member; it returns `-1` if there is no default member.

`length()`	This can be called on an `ICORBA_TypeCode` of the `tk_string`, `tk_sequence`, or `tk_array` kind. For a bounded string or sequence, `length()` returns the bound value. 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` of the `tk_sequence`, `tk_array`, or `tk_alias` kind. 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 l;} TypeCodeBounds;
typedef struct tagTypeCodeBadKind {long l;} 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 exceptions that can occur with an ICORBA_TypeCode at runtime to be raised.

Methods

_get_Bounds()	This returns a Bounds exception, which results if the nIndex parameter is greater than or equal to the number of members that make up the type.
_get_BadKind()	This returns a BadKind exception, which results from performing 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 in string form from a proxy. The `systemIDs` parameter 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`. If the proxy is a proxy for an object in more than one object system, the order of IDs in the `systemIDs` array indicates the caller's order of preference. 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);
    HRESULT _get_InterfaceName ([out] LPSTR * name);
};
```

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 227 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 method is intended as an optimization, so that a rarely used connection between a client and server is not kept open for long periods while not in use. The channel is automatically reopened when an invocation is made on the object. If the client holds proxies for other objects in the same server, the channel is closed for all such proxies. The channel 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 who are using libraries or systems that depend on the UNIX `select()` system call might need to know which file descriptors are scanned by Orbix. This method 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.)
`_get_InterfaceName()`	This returns the interface name of 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);
    HRESULT GetOrbixSSL([optional,in,out] VARIANT *IT_Ex,
        [retval,out] IDispatch ** IT_retval);
    HRESULT ReleaseCORBAView([in] IDispatch * poObj,
        [in] VARIANT_BOOL 1ToDestruction,
        [optional,in,out] VARIANT *IT_Ex,
        [retval,out] VARIANT_BOOL * IT_retval);
    HRESULT UseTransientPort([in] VARIANT_BOOL OnOff,
        [optional,in,out] VARIANT *IT_Ex,
        [retval,out] VARIANT_BOOL * IT_retval);
};
```

Description The `IOrbixORBObject` interface provides Orbix-specific methods that allow programmers to control some aspects of the ORB (Orbix) or request the ORB to perform actions.

The ORB has the ProgID, `CORBA.ORB.2`, which is the COM/CORBA-compliant name. In OrbixCOMet, the `CORBA.ORB.Orbix` name 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 most cases.

The value set by this method 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 favored over new connection attempts. The `EagerListeners()` method 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 retries the attempt every two seconds until either the call can be made or there have been too many retries. The `MaxConnectRetries()` method 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 is only 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 has no effect on the target object. The pinging causes the target server process to be activated, if necessary, and confirms that this server recognizes the target object. Pinging can be enabled, using `PingDuringBind()`, by passing `1` to the `pingOn` parameter. Pinging can be disabled by passing `0` in `pingOn`. The previous setting is returned in the `IT_retval` parameter.

You might wish to disable pinging to improve efficiency by reducing the overall number of remote invocations. In this case, Orbix checks the object's availability only when a method is invoked on the object, and not when the bind attempt is made.

If `PingDuringBind(false)` is called:

- A `_bind()` to an unavailable object does not immediately raise an exception, but subsequent requests using the object reference returned from `_bind()` do fail and raise a `CORBA::INV_OBJREF` system exception.
- If a hostname is specified to `_bind()`, the `_bind()` itself does not make any remote calls; it simply sets up a proxy with the required fields.
- If a hostname 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 following value in the `CORBA.h` file:

```
CORBA_OBJECT_TABLE_SIZE_DEFAULT
```

If you anticipate that a program will handle a much larger number of objects than the default size (which is about 1,000), 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 and server. This connection is used for all subsequent communications between the client and server. The connection is closed only when the client or the server exits.

When a server exits while a client is still connected, the next invocation by the client raises a system exception of `CORBA::COMM_FAILURE` type. If the client attempts another invocation, Orbix automatically tries to re-establish the connection.

This default behavior can be changed by passing the value `0` (false) to `NoReconnectOnFailure()`. In this case, all client attempts to contact a server, after the communications channel has been closed, raise a `CORBA::COMM_FAILURE` system exception.

`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 inactive or unreachable. Set `OnOff` to `1` to abort slow connection attempts.
`ActivateCVHandler()`	This activates the configuration value handler specified in the `identifier` parameter. You must call `ReinitialiseConfig()` before modifications by this method can take effect. Refer to the Orbix documentation set for information on configuration handlers.
`DeactivateCVHandler()`	This deactivates the configuration value handler specified in the `identifier` parameter. You must call `ReinitialiseConfig()` before modifications by this method can take effect. Refer to the Orbix documentation set for information on configuration handlers.
`ActivateOutputHandler()`	This activates the output handler specified in the `identifier` parameter. 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 method can take effect. Refer to the Orbix documentation set for information on configuration handlers.
`PlaceCVHandlerBefore()`	See `PlaceCVHandlerAfter`.
`DeactivateOutputHandler()`	This deactivates the output handler specified in the `identifier` parameter. 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 specified in the `derived` parameter. The interface specified in the `derived` parameter is included in the list, because it is considered a base interface of itself.
`IsBaseInterfaceOf()`	This determines whether the `maybeABase` interface is a base interface of the interface specified in the `derived` parameter. `IsBaseInterfaceOf()` returns `1` if `maybeABase` is a base interface of the interface specified in the `derived` parameter (or if the same interface is specified in both the `derived` and `maybeABase` parameter). Otherwise, it returns `0`.

`CloseChannel()`	This requests Orbix to close the underlying communications connection to the server. This function is intended as an optimization, so that a rarely used connection between a client and server is not kept open for long periods while not in use. 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 setting.
`DefaultTxTimeout()`	This sets the timeout for all remote calls and returns the previous setting. By default, there is no timeout value set for remote calls; that is, the default timeout is infinite. The value set by this method 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; 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 multithreaded 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 specified in the `name` parameter. Refer to the Orbix documentation set for information on configuration values.
`SetConfigValue()`	This sets the value of the configuration entry specified in the `name` parameter for this process only. (It does not set the configuration entry in the Orbix configuration file.)

`Output()`	This outputs the application's diagnostic and other output via the active output handlers. Unless overridden by an implementation of the `CORBA::ORB::UserOutput` C++ class, 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 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. The level values are: 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 outpu. This is done 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 initializes the ORB. Invoking this method is optional. If `StartUp` is not invoked, the ORB is automatically initialized when the first object is created. However, it is a CORBA guideline that an ORB should be initialized before being used. Therefore, you should call this method before doing anything else (that is, before you make any calls to `GetObject` or `CreateType` on `ICORBAFactory`).
`ShutDown()`	This shuts down the bridge. Invoking this method might be necessary if you are experiencing hang-on-exit problems. After this method 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.
`GetOrbixSSL()`	This obtains a pointer to the `IOrbixSSL` interface when Orbix SSL support is being used.
`ReleaseCORBAView()`	This is used by clients to free the CORBA view of a DCOM callback object when receipt of callbacks in no longer required.
`UseTransientPort()`	This is a wrapper around the Orbix call of the same name. It places a transient port number, instead of the Orbix daemon's port number, in any exported IORs.

UUID `{4ea7b110-1a93-f447-1dc7-c8c8b25be06f}`

Notes Orbix-specific.

IOrbixSSL

Synopsis

```
[object, uuid(adecd691-fd88-11d2-af83-00a024d8995c)]
interface IOrbixSSL : IUnknown {
    HRESULT InitSSL([out] int* nRet);
    HRESULT InitScopeSSL([in] LPSTR cPolicyName,
        [out] int* nRet);
    HRESULT SetSecurityName([in] LPSTR cCertName,
        [out] int* nRet);
    HRESULT GetSecurityName([out] LPSTR* cCertName);
    HRESULT SetPrivateKeyPassword([in] LPSTR cPassword,
        [out] int* nRet);
    HRESULT HasPassword([out] BOOLEAN* bRet);
};
```

Description `IOrbixSSL` provides support for integrating SSL support into OrbixCOMet COM applications. A reference to this interface is retrieved, using a call to `GetOrbixSSL()` on the `IOrbixORBObject` interface.

Methods

`InitSSL()`	This initializes the SSL library. It must be called by each OrbixCOMet SSL-enabled application before any attempts are made to bind to CORBA clients or servers, and before any calls to other `IOrbixSSL` methods.
`InitScopeSSL()`	This instructs SSL to implement the SSL policies included in the configuration scope (`OrbixSSL.cfg`) specified in the `cPolicyName` parameter. (For further details, refer to `IT_SSL::initScope` in the *OrbixSSL C++ Programmer's and Administrator's Guide*.) The specified configuration scope can contain a value for `IT_CERTIFICATE_FILE`, which specifies the location of an X.509 certificate file. As a result of the call to `InitScopeSSL`, the identified certificate is initialized by the SSL runtime, and is associated with the application.

`SetSecurityName()`	This method is passed the location of a file that contains an X.509 certificate and private key to be associated with an OrbixCOMet SSL-enabled application. (For further details, refer to `IT_SSL::setSecurityName` in the *OrbixSSL C++ Programmer's and Administrator's Guide*.)
`GetSecurityName()`	This retrieves the security name of the certificate being used by an OrbixCOMet SSL-enabled application.
`SetPrivateKeyPassword()`	This specifies the pass phrase to be used to unlock the private key of an X.509 certificate. The private key is stored in an X.509 certificate in encrypted PEM format with a secret pass phase. The private-key pass phrase is required to unlock the private key. The private-key pass phrase is generally chosen by the system administrator when creating the application certificate signing request.
`HasPassword`	This determines whether the server has received a private-key pass phrase from the server key distribution mechanism (KDM). If the server has not received a pass phrase, a valid password must be supplied, using `SetPrivateKeyPassword`.

UUID `{adecd691-fd88-lld2-af83-00a024d8995c}`

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 with 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 following format:

`:\`*`host`*`:`*`serverName`*`:`*`marker`*`:`*`IFR_host`*`:`
`IFR_server``:`*`interfaceMarker`*

The fields can be described as follows:

- `host`—This is the hostname of the target.

ObjectToString()
(continued)

- serverName—This is the name of the target object's server. This is the name used to register the server in the Implementation Repository. It is also the name specified to CORBA::BOA::impl_is_ready(), CORBA::BOA::object_is_ready(), or set by setServerName(). For a local object in a server, this is the server's name (if it is known); otherwise, it is the identifier of the process. The server name is known if the server is launched by the Orbix daemon, or if the server is launched manually and the server name is passed to impl_is_ready(), or if 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 is 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 "IFR" string.
- 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 `name` parameter specifies the desired service. A list of supported services can be obtained using `DIORBObject::GetInitialReferences()`.

UUID `{204F6245-3AEC-11CF-BBFC-444553540000}`

Notes COM/CORBA-compliant.

15

Introduction to OMG IDL

This chapter describes the CORBA Interface Definition Language (OMG IDL), which is used to describe the interfaces to CORBA objects.

The OMG IDL language 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 manages bank accounts. A user of an account object wants to make deposits and withdrawals. An account object also needs 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 the `balance` and `owner` attributes; these are properties of an `Account` object. The `balance` attribute can take values of the `float` type, which is one of the basic types of OMG IDL and represents a floating point type (such as `102.31`). The `owner` attribute is of the `string` type and is defined as `readonly`.[1]

Two operations, `makeDeposit()` and `makeWithdrawal()`, are provided. Each of these has two parameters of the `float` type. Each parameter must specify the direction in which the parameter is passed. The possible parameter-passing modes are:

`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 does not have to 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 following 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 can 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 block until the operation request has been performed.

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 is now 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 the account.

The newAccount() operation uses the raises expression to specify that it can raise two exceptions, called Reject and TooMany. The Reject and TooMany exceptions are defined within the Bank interface. The Reject exception defines a member of the string type, which is used to specify the reason why 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 runtime errors that can occur during the execution of a request. Refer to "System Exceptions" on page 351 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 behavior. They can also have additional properties and operations.

The relationships among these interfaces can be described in a type hierarchy as shown in Figure 15.1. The `Account` interface is called a *base interface* of `CheckingAccount` and `SavingsAccount`. The `CheckingAccount` and `SavingsAccount` interfaces are called *derived interfaces* of `Account`.

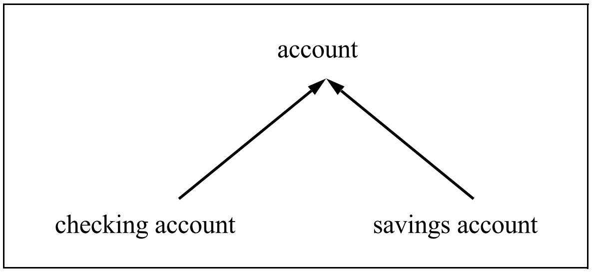

Figure 15.1: *Inheritance*

You can define `CheckingAccount` interface as follows:

```
// OMG IDL
interface CheckingAccount : Account {
    readonly attribute overdraftLimit;
    boolean orderChequeBook();
};
```

It defines one attribute, `overdraftLimit`, but it inherits the `balance` and `owner` attributes defined in its base interface, `Account`. Similarly, it inherits the `makeDeposit()` and `makeWithdrawal()` operations from `Account`, and defines a new `orderChequebook()` operation. An implementation of the `CheckingAccount` interface can provide code that is different to an implementation of the `Account` interface.

You can define the `SavingsAccount` interface 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 check book. Thus, the multiple inheritance hierarchy is as shown in Figure 15.2.

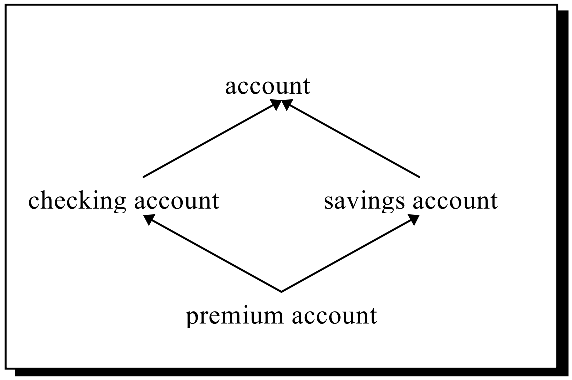

Figure 15.2: *Multiple Inheritance*

The `SavingsAccount` interface is defined as follows:

```
// OMG IDL
interface SavingsAccount : Account {
    float calculateInterest();
};
```

The `PremiumAccount` interface can then be defined 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 the attributes and operations on base interfaces are available on instances of a derived interface.

The Basic Types of OMG IDL

Table 15.1 lists the basic types supported in OMG IDL.

Type	OMG IDL Identifier	Description
Floating point	`float` `double`	IEEE single-precision floating point numbers. IEEE double-precision numbers.

Integer	`long`
	`short`
	`unsigned long`
	`unsigned short`
		`–231...231-1 (32bit)`
		`–215...215-1 (16bit)`
		`0...232-1 (32bit)`
		`0...216-1 (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 15.1: *OMG IDL Basic Types*

Note: There is no `int` type in OMG IDL, and `char` cannot be qualified by `unsigned`.

The any type allows an interface to specify that a parameter or result type can contain an arbitrary type of value that is to be determined at runtime. For example:

```
// OMG IDL
interface I {
    void op(in any a);
};
```

A process that receives an any type must determine what type of value it contains, and then extract the value.

Constructed Types

As well as the basic types listed in Table 15.1 on page 262, 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 PersonalDetails struct has two members: name of the string type, and age of the short type. The getPersonalDetails() operation 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 color { red, green, blue, yellow, white };
```

This is more readable than defining color as a short type. 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 the long type. 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];
```

The `bankAccounts` and `gridArr` types can be used, for example, to define parameters to an operation.

Template Types

OMG IDL provides two template types, sequence and `string`, which are described in the following subsections.

Sequences

An OMG IDL sequence 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, because a sequence is not of fixed size (although a bounded sequence has a fixed maximum size). Therefore, a sequence is a more flexible data type, and should be used instead of 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 defines a bounded sequence, vectorTen, of size 10:

```
// OMG IDL
sequence<long, 10> vectorTen;
```

This means that the `vectorTen` sequence can be of any length up to the bound (that is, 10).

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 types. 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 the `long`, `unsigned long`, `short`, `unsigned short`, `char`, `boolean`, `float`, `double`, and `string` type can be declared. Constants of the `octet` type cannot be declared.

Typedef Declaration

A typedef declaration can be used to define a meaningful or simpler name for a basic or user-defined type. For example, the following defines `size` as a synonym for `short`:

```
// OMG IDL
typedef short size;
```

The following is a parameter declaration, using `size`:

```
// OMG IDL
in size i
```

The following is a parameter declaration, using `short`, which is equivalent to the preceding declaration:

```
// OMG IDL
in short i
```

Similarly, assume you make the following typedef declaration:

```
// OMG IDL
typedef Account Accounts[100];
```

This 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 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, enums, exceptions, interfaces, attributes, and operations.

A qualified or *scoped name* has the format *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 is also the case with a C++ include file, the following directives should be used in an OMG IDL file that might potentially be 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 `NamedValue`, `Principal`, and `TypeCode` pseudo types are available in an OMG IDL file, only if it includes the following directive:

```
#include <orb.idl>
```

The `Object` interface name, which is the implicit base interface of all interfaces, is available in all files.

16

CORBA-to-Automation Mapping

CORBA types are defined in OMG IDL. Automation types are defined in Microsoft IDL (COM IDL). To allow interworking between Automation clients and CORBA servers, Automation clients must be presented with COM IDL versions of the interfaces exposed by CORBA objects. Therefore, it must be possible to translate CORBA types to COM IDL. This chapter outlines the CORBA-to-Automation mapping rules.

For the purposes of illustration, this chapter describes a textual mapping between OMG IDL and COM 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.

Basic Types

OMG IDL basic types map to compatible types in Automation. Table 16.1 shows the mapping rules for each basic type.

OMG IDL	Description	COM IDL	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 16.1: *CORBA-to-Automation Mapping Rules for Basic Types*

a. `UI1` is supported in Windows 32-bit programs.

The types supported by OMG IDL and Automation do not correspond exactly, because Automation offers a more limited support for basic types. For example, Automation does not support unsigned types (that is, `unsigned short` or `unsigned long`). In some cases, the mapping rules involve a type promotion, to avoid data loss (for example, translating OMG IDL `unsigned short` to Automation `long`.) In other cases, the mapping rules involve a type demotion (for example, translating OMG IDL `unsigned long` to Automation `long`.)

An Automation view interface provides an Automation client with an Automation view of a CORBA object. An operation of an Automation view interface uses the mappings shown in Table 16.1 on page 275, to perform bidirectional translation of parameters and return types between Automation and CORBA. It translates `in` parameters from Automation to CORBA, and translates `out` parameters from CORBA back to Automation. Because there is not an exact correspondence between the types supported by Automation and CORBA, the following translations performed by an Automation view operation result in a runtime error:

- Translating an `in` parameter of the Automation `long` type to the OMG IDL `unsigned long` type, if the value of the Automation `long` parameter is a negative number.
- Demoting an `in` parameter of the Automation `long` type to the OMG IDL `unsigned short` type, if the value of the Automation `long` parameter is either negative or greater than the maximum value of the OMG IDL `unsigned short` type.
- Demoting an `out` parameter of the OMG IDL `unsigned long` type back to the Automation `long` type, if the value of the OMG IDL `unsigned long` parameter is greater than the maximum value of the Automation `long` type.

Strings

OMG IDL bounded and unbounded strings map to an Automation BSTR. For example:

```
// OMG IDL
// This definition might appear within a struct
// definition.
string name<20>;
string address;
```

This maps to:

```
// COM IDL
BSTR name;
BSTR address;
```

A runtime error occurs if a BSTR exceeds the full length of a bounded string.

Interfaces

An OMG IDL interface maps to an Automation view interface. The following is an example of an OMG IDL interface, Bank:

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

This maps to the Automation view interface, DIBank:

```
// COM IDL
// Definitions that are not of interest here.

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

As shown in Figure 16.1 on page 275, the Automation view in the bridge supports the DIBank interface. 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 DIBank interface 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.

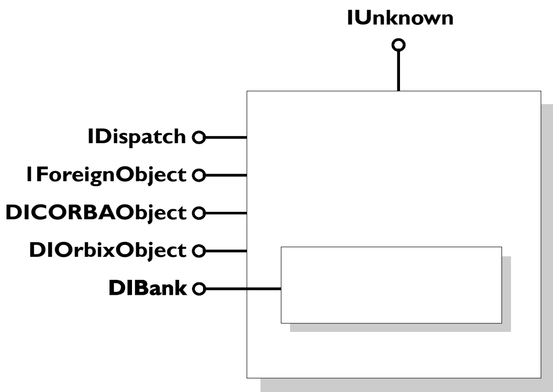

Figure 16.1: *Automation View of the Bank Interface*

Attributes

An OMG IDL attribute maps to an Automation property, as follows:

- A normal attribute maps to a property that has a method to set the value and a method to get the value.
- A `readonly` attribute maps to a property that only has a method to get the value.

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);
};
```

This maps to:

```
// COM IDL
[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);
}
```

The `get` method returns the attribute value contained in the `[retval,out]` parameter.

Visual Basic The following is a Visual Basic example of how to `set` and `get` the balance of an account object, `accountObj`:

```
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 The following is a PowerBuilder example of how to `set` and `get` the balance of an account object, `accountObj`:

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

integer myBalance

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

Operations

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

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

This maps to:

```
// COM IDL
[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 );
}
```

The following mapping rules apply for the parameter-passing modes:

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

The following mapping rules apply for return types:

- An OMG IDL `void` return type does not need any translation.
- An OMG IDL return type that is not `void` maps 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.
- All operations on Automation view interface have an optional `out` parameter of the `VARIANT` type. This parameter appears before the return type and is used to return exception information. Refer to "Exceptions" on page 291 for more information.

- If the CORBA operation has no return value, the optional out parameter of the VARIANT type 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.

The following is a Visual Basic example, based on the generated definitions in the preceding COM IDL example:

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

Inheritance

This section describes the CORBA-to-Automation mapping rules for single inheritance and multiple inheritance.

Single Inheritance

A hierarchy of singly-inherited OMG IDL interfaces maps to an identical hierarchy of Automation view interfaces. The following is an example of an 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();
    };
};
```

This maps to the following Automation view interfaces:

```
// COM IDL
[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 16.2 on page 280 is an example of a CORBA interface hierarchy.

In Figure 16.2, the hierarchy can be read as follows:

- `Account` and `Simple` derive from `Bank`.
- `CheckingDetails` derives from `Account` and `Simple`.
- `Miscellaneous` derives from `CheckingDetails`.

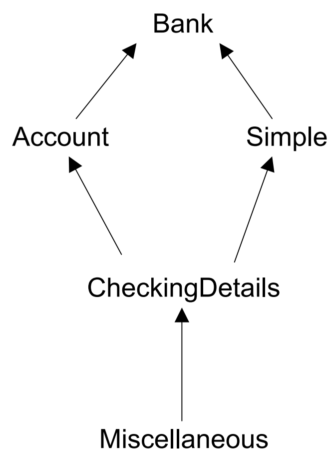

Figure 16.2: *Example of a CORBA Interface Hierarchy*

In this example, `CheckingDetails` is the point of multiple inheritance. The CORBA hierarchy maps to two Automation single inheritance hierarchies (that is, `Bank-Account-CheckingDetails` and `Bank-Simple`). The leftmost strand is considered the main strand, which 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. 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` can access all the methods of the hierarchy without having to call `QueryInterface`.

The following OMG IDL represents a hierarchy based on the example shown in Figure 16.2 on page 280:

```
// OMG IDL
{
    interface Bank
    {
        void OpBank();
    };
    interface Account : Bank
    {
        void OpAccount();
    };
    interface Simple : Bank
    {
        void OpSimple();
    };
    interface CheckingDetails : Account, Simple
    {
        void OpCheckingDetails();
    };
    interface Miscellaneous : CheckingDetails
    {
        void OpMiscellaneous();
    };
};
```

This maps to the following two Automation view hierarchies:

```
// COM IDL
// strand 1:Bank-Account-CheckingDetails
[oleautomation, dual, uuid(…)]
interface Bank:IDispatch
{
    HRESULT OpBank([optional, out] VARIANT * excep_OBJ);
}
[oleautomation, dual, uuid(…)]
interface Account:Bank
{
    HRESULT OpAccount([optional, out] VARIANT * excep_OBJ);
}
```

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

Complex Types

Translation is straightforward where there is a direct Automation counterpart for a CORBA type. However, Automation has no data type corresponding to a user-defined complex type. CORBA complex types are therefore mapped to Automation view interfaces. Each element in the complex type maps to a property in the Automation view, with a `get` method to retrieve its value, and a `set` method to alter its value. This section describes the CORBA-to-Automation mapping rules for the following complex types:

- Structs
- Unions
- Sequences
- Arrays
- Exceptions
- The any type

Creating Constructed OMG IDL Types

OMG IDL constructed types such as `struct`, `union` and `exception` map 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 `DIForeignComplexType` interface.

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

The prototype for `CreateType()` is:

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

The parameters for `CreateType()` can be explained as follows:

- 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 want to create.

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

Refer to “Structs” on page 283, “Unions” on page 285, and “Exceptions” on page 291 for examples of how to use `CreateType()` to create structs, unions, and exceptions.

Structs

An OMG IDL struct maps 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 mapped from a struct.

For example:

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

This is mapped as if it were defined as:

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

Figure 16.3 shows the Automation view of the translation.

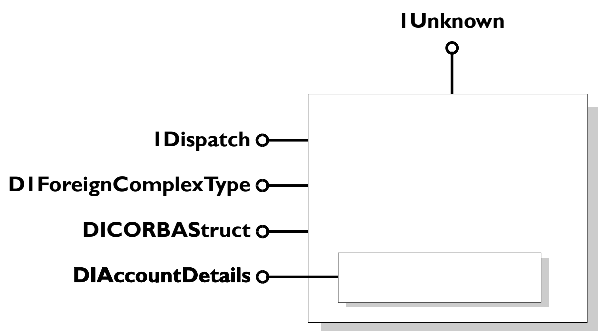

Figure 16.3: *Automation View of the OMG IDL AccountDetails Struct*

The following is a Visual Basic example, based on the preceding IDL definition:

```
Dim ObjFactory As CORBA_Orbix.DICORBAFactoryEx
Dim details As BankBridge.DIAccountDetails
…
Set details = ObjFactory.CreateType(Nothing, "AccountDetails")
details.balance = 1297.66
details.number = 109784
```

Unions

An OMG IDL union maps 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 mapped from a union.

The DICORBAUnion2 interface is defined, to describe CORBA union types that support multiple case labels for each union branch. This provides two additional accessors, as follows:

```
// COM IDL
[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 uses the current discriminant value to initialize the VARIANT with the union element. All mapped unions should support the DICORBAUnion2 interface.

The following is an example of an OMG IDL union type:

```
// OMG IDL
interface A {…};

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

This maps to the following Automation pseudo-union:

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

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

In this case, the mapped Automation dual interface derives from the DICORBAUnion2 interface. The UNION_d property 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 COM IDL 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. Do not try 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 is 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 16.4 shows the Automation view of the translation of the OMG IDL U union.

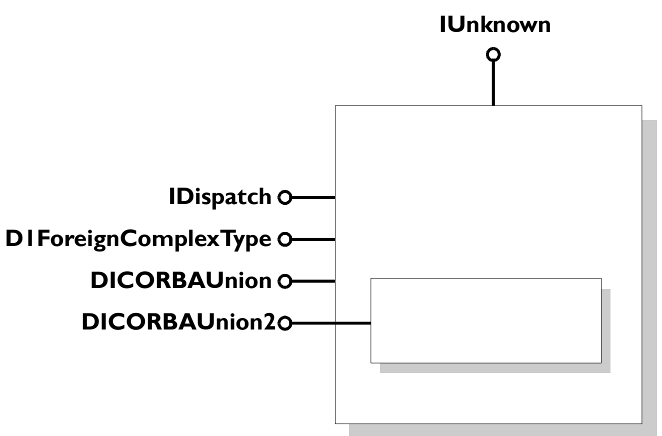

Figure 16.4: *Automation View of an OMG IDL Union*

The following Visual Basic example is based on the preceding COM IDL:

```
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$(myUnion.l)
    Case 2: MsgBox ("Union (float):" & Str$(myUnion.f)
    Case Else : MsgBox ("Union contains object reference")
End Select
```

Sequences

An OMG IDL sequence maps to either an Automation SafeArray or an OLE collection. The `COMet.Mapping.UseSAFEARRAYMapping` configuration value determines the type of mapping in effect. It is set to "`yes`" by default, which means sequences map to SafeArrays. If it is set to "`no`", sequences map to OLE collections. You should set this configuration value only once in your application.

For example, the following is the `Form-Load` of a Visual Basic application:

```
' Visual Basic
Dim orb as object
…
Set orb = CreateObject("CORBA.ORB.2")
orb.SetConfigValue("COMet.Mapping.UseSAFEARRAYMapping", "yes")
```

SafeArrays

If the `COMet.Mapping.UseSAFEARRAYMapping` configuration value is set to "`yes`", an OMG IDL sequence maps to a `VARIANT` type containing an Automation SafeArray. An OMG IDL bounded sequence maps 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 maps to an empty SafeArray that can grow or shrink to any size. The `COMet.Mapping.SAFEARRAYS_CONTAIN_VARIANTS` configuration value maps a sequence of any type to a SafeArray of `VARIANT` types containing the real type.

OLE Collections

If the `COMet.Mapping.UseSAFEARRAYMapping` configuration value is set to "no", an OMG IDL bounded or unbounded sequence maps to a `VARIANT` type 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 type	This gets or sets the number of elements in the collection.
`Item`	Read/Write Parameterized Property type	This gets or sets 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 parameterized properties:

Method	Type	Description
`getItem`	Method	This gets or sets the number of elements in the collection.
`setItem`	Method	This gets or sets access to individual elements in the collection.

Refer to "OrbixCOMet API Reference" on page 181 for a full description of the COM IDL definitions for the `DCollection` interface.

The following is an example of an OMG IDL bounded and unbounded sequence:

```
// 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)
        …
    };
};
```

This maps to:

```
// COM IDL
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);
};

[oleautomation, 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);
};
```

The following Visual Basic example is based on the preceding COM IDL:

```
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
```

Arrays

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

SafeArrays

Multidimensional OMG IDL arrays map to `VARIANT` types containing multidimensional SafeArrays. The order of dimensions in the OMG IDL array from left to right corresponds to the ascending order of dimensions in the SafeArray. An error occurs if the number of dimensions in an input SafeArray does not match the CORBA type.

OLE Collections

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

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 351.
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 maps 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;
};
```

This maps to:

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

Figure 16.5 on page 292 provides an Automation view of the translation of the `Bank::Reject` exception. Refer to "System Exceptions" on page 351 for more details about exceptions.

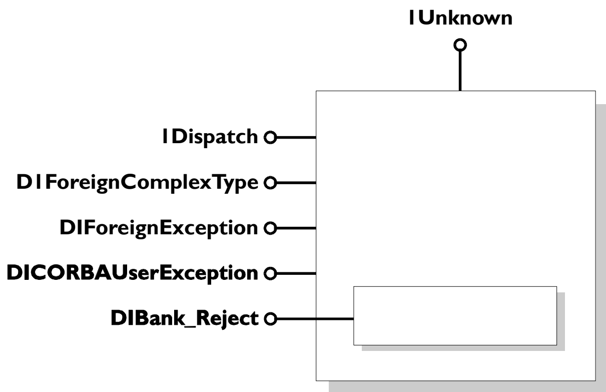

Figure 16.5: *Automation View of Bank_Reject*

System Exceptions

A CORBA system exception maps to the `DICORBASystemException` Automation interface, which derives from `DIForeignException`. For example:

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

The `EX_minorCode` attribute 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, as follows:

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

This interface is explained in "DICORBASystemException" on page 195.

The Any Type

The OMG IDL any type maps 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 16.1 on page 272. If the any contains a complex type, the VARIANT contains an IDispatch view of the CORBA type. If the any contains a CORBA sequence or array type, the VARIANT contains either an Automation SafeArray or an OLE Collection, depending on the setting of the COMet.Mapping.UseSAFEARRAYMapping configuration value.

Context Clause

There is no standard CORBA-to-Automation mapping specified for OMG IDL contexts.

Object References

When an OMG IDL operation returns an object reference, or passes an object reference as an operation parameter, this is mapped as a reference to an IDispatch interface in COM IDL. 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);
};
```

This maps to:

```
// COM IDL
[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);
};
```

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 The following is a Visual Basic example, based on the preceding mapping rules for object references:

```
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
```

Modules

An OMG IDL definition contained within the scope of an OMG IDL module maps 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 {
    …
    };
};
```

This maps to:

```
// COM IDL
[oleautomation, dual, uuid(…), helpstring("Finance_Bank")]
interface DIFinance_Bank : IDispatch {
    …
}
```

Visual Basic The following Visual Basic example shows how the mapped definition is subsequently used:

```
Dim bankObj As DIFinance_Bank
```

Constants

There is no Automation definition 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 "Scoped Names" on page 297 for more details.) For example, the following is an example of an OMG IDL constant definition:

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

Visual Basic The preceding constant definition can be represented in Visual Basic as follows:

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

PowerBuilder The preceding constant definition can be represented in PowerBuilder as follows:

```
// PowerBuilder
CONSTANT long Max=100
```

Enumerated Types

A CORBA enum maps to an Automation enum. For example:

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

This maps to:

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

Because Automation maps enum parameters to the platform's integer type, a runtime error occurs 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 with the name of the interface or module in the Automation controller language. (Refer to "Scoped Names" on page 297 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.

Scoped Names

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

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

This yields the scoped name, `Finance::Bank::PersonnelRecord`, which maps to the Automation identifier, `Finance_Bank_PersonnelRecord`.

Typedefs

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

There is no translation provided for typedefs for the basic OMG IDL types listed in Table 16.1 on page 272. Therefore, a Visual Basic programmer cannot make use of these typedef definitions for basic types. For example:

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

This can be used as follows in Visual Basic:

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

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

17

Automation-to-CORBA Mapping

Automation types are defined in Microsoft IDL (COM IDL). CORBA types are defined in OMG IDL. To allow interworking between CORBA clients and Automation servers, CORBA clients must be presented with OMG IDL versions of the interfaces exposed by Automation objects. Therefore, it must be possible to translate Automation types to OMG IDL. This chapter outlines the Automation-to-CORBA mapping rules.

For the purposes of illustration, this chapter describes a textual mapping between COM IDL 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.

Basic Types

Automation basic types map to compatible types in OMG IDL. Table 17.1 shows the mapping rules for each basic type.

COM IDL	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 17.1: *Automation-to-CORBA Mapping Rules for Basic Types*

A CORBA view interface provides a CORBA client with a CORBA view of an Automation object. An operation of a CORBA view interface uses the mappings shown in Table 17.1, to perform bidirectional translation of parameters and return types between CORBA and Automation. It translates in parameters from CORBA to Automation, and translates out parameters from Automation back to CORBA.

Because there is not an exact correspondence between the types supported by CORBA and Automation, the following translations performed by a CORBA view operation result in a runtime error:

- Translating an `in` parameter of the OMG IDL `COM::Currency` type to the Automation `CURRENCY` type, if the supplied `COM::Currency` value does not translate to a meaningful Automation `CURRENCY` value.
- Translating an `in` parameter of the OMG IDL `double` type to the Automation `DATE` type, if the OMG IDL double value is negative or converts to an impossible date.

Strings

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

```
// COM IDL
BSTR address;
```

This maps to:

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

Interfaces

An Automation interface maps to an OMG IDL interface. For example:

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

This maps to:

```
// 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 maps to the `long` type. This allows COM `SCODE` values to be passed through to the CORBA client.

Properties and Methods

The following mapping rules apply for Automation properties and methods:

- An Automation method maps to an OMG IDL operation.
- An Automation property that has a method to get the value, and a method to set the value, maps to a normal OMG IDL attribute.
- An Automation property that only has a method to get the value maps to a `readonly` OMG IDL attribute.

For example:

```
// COM IDL
[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);
}
```

This maps 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);
};
```

The following mapping rules apply for the parameter-passing modes:

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

Inheritance

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

```
// COM IDL
[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);
};
```

This maps 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 theBalance);
};
interface checkingAccount: account
{
    readonly attribute short overdraftLimit;
    short orderChequeBook();
};
```

SafeArrays

The following Automation-to-CORBA mapping rules apply for SafeArrays[1]:

- An Automation SafeArray maps to an OMG IDL unbounded sequence.
- 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.

Exceptions

The following Automation-to-CORBA mapping rules apply for exceptions:

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

1. An Automation SafeArray is an array of other types that contains (in addition to the data) information about the size of each element, the number of dimensions, and the size of each dimension.

- 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.

Variant Types

An Automation VARIANT type maps to the OMG IDL any type. If the VARIANT type contains a DATE or CURRENCY element, these are mapped as shown in "Basic Types" on page 300.

Object References

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

```
// COM IDL
[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 maps to:

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

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

Enumerated Types

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

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

This maps to:

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

Automation enumerators can have explicitly assigned values. In CORBA, rather than being explicitly assigned, enum values start at zero and increase in increments of one. 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 enumerators. Therefore, Automation enums with explicitly assigned values map to OMG IDL enums in ascending order of their value, to preserve the order of the enumerators.

Typedefs

An Automation typedef maps to an OMG IDL typedef. For example:

```
// COM IDL
typedef [public] BSTR custName;
```

This maps to:

```
// OMG IDL
typedef string custName;
```

The only exception to this is an Automation enum that is required to be a typedef. For example:

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

This maps to:

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

18

CORBA-to-COM Mapping

CORBA types are defined in OMG IDL. COM types are defined in Microsoft IDL (COM IDL). To allow interworking between COM clients and CORBA servers, COM clients must be presented with COM IDL versions of the interfaces exposed by CORBA objects. Therefore, it must be possible to translate CORBA types to COM IDL. This chapter outlines the CORBA-to-COM mapping rules.

For the purposes of illustration, this chapter describes a textual mapping between OMG IDL and COM 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 runtime.

Basic Types

OMG IDL basic types map to compatible types in COM. Table 18.1 shows the mapping rules for each basic type.

OMG IDL	Description	COM IDL	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 18.1: *CORBA-to-COM Mapping Rules for Basic Types*

Strings

An OMG IDL string maps to a COM IDL LPSTR, which is a null-terminated 8-bit character string. The following subsections describe the mappings for bounded and unbounded strings.

Unbounded Strings

The following is a definition for an OMG IDL unbounded string:

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

This maps to:

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

Bounded Strings

The following is a definition for an OMG IDL bounded string:

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

This maps to:

```
// COM IDL
const long N = …;
typdef [string, unique] char (*BOUNDED_STRING) [N];
```

Interfaces

An OMG IDL `RepositoryId` maps to a COM IDL `IID` (Interface ID) that is similar to the `DCE UUID` (or identical in the case of Windows32-bit programs).

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.

Attributes

An OMG IDL attribute maps to a COM IDL attribute, as follows:

- A normal attribute maps to a property that has a method to set the value and a method to get the value.
- A `readonly` attribute maps to a property that only has a method to get the value.

For example:

```
// 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:
};
```

In this case, the read-write `Profile` attribute maps to the following COM IDL:

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

The readonly Balance attribute maps to the following COM IDL:

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

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

Operations

An OMG IDL operation maps to a COM IDL 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);
};
```

This maps to:

```
// COM IDL
[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);
};
```

The following mapping rules apply for parameters and return types:

- An OMG IDL in parameter maps to a COM IDL [in] parameter.
- An OMG IDL out parameter maps to a COM IDL [out] parameter.

- An OMG IDL `inout` parameter maps to a COM IDL `[in,out]` parameter.
- An OMG IDL return type maps to a COM IDL `[out]` parameter as the last parameter in the signature.

Inheritance

CORBA and COM have different models for inheritance. CORBA interfaces can be multiply inherited, but COM does not support multiple interface inheritance. The CORBA-to-COM mapping rules for an interface hierarchy are as follows:

- Each OMG IDL interface that does not have a parent maps to a COM IDL interface derived from the `IUnknown` interface.
- Each OMG IDL interface that inherits from a single parent maps to a COM IDL interface deriving from the mapping for the parent interface.
- Each OMG IDL interface that inherits from multiple parents is mapped to a COM IDL interface derived from the `IUnknown` interface. This COM IDL interface then aggregates both base interfaces.
- For each CORBA interface, the mapping for operations precedes the mapping for attributes.

Figure 18.1 on page 315 shows a CORBA interface hierarchy. In this hierarchy:

- `Account` and `Simple` derive from `Bank`.
- `CheckingDetails` derives from `Account` and `Simple`.
- `Miscellaneous` derives from `CheckingDetails`.

The following OMG IDL represents the interface hierarchy in Figure 18.1:

```
// 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();
};
```

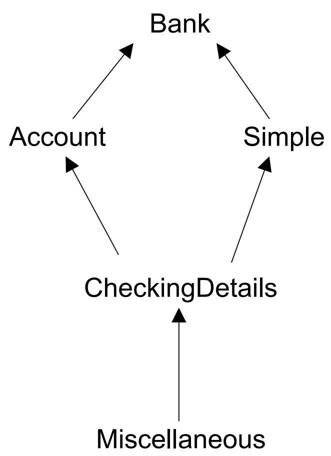

Figure 18.1: *Example of a CORBA Interface Hierarchy*

This maps to the following COM IDL:

```
// COM IDL
[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 COM IDL, the name of the interface is preceded by the letter `I`. If the interface is scoped by OMG IDL modules, using `::`, this is replaced by an underscore (for example, `foo::bar` maps to `Ifoo_bar`).

Complex Types

OMG IDL includes a number of types that do not have counterparts in COM IDL. This section describes the CORBA-to-COM mapping rules for the following complex types:

- Structs
- Unions
- Sequences
- Arrays
- Anys

Creating Constructed OMG IDL Types

OMG IDL constructed types such as `struct`, `union`, `sequence` and `exception` map to corresponding struct types in COM IDL.

To create a complex OMG IDL type, you should simply instantiate an instance of its COM IDL `struct` type. You must create an object representing an OMG IDL constructed type in a client, 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, to return it as an `out` or `inout` parameter, or return value, from an OMG IDL operation.

Structs

An OMG IDL struct maps to a COM IDL struct. For example:

```
// OMG IDL
typedef … T0;
typedef … T1;
typedef … T2;
…
typedef … Tn;
struct STRUCTURE
{
    T0 m0;
    T1 m1;
```

```
    T2 m2;
…
   Tn mN;
};
```

This maps to:

```
// COM IDL
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;
};
```

This maps to:

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

Unions

A discriminated union in OMG IDL maps to an encapsulated union in COM IDL. 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];
    };
```

This maps to:

```
// COM IDL
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
```

Sequences

CORBA sequences have no direct corresponding type in COM. A CORBA sequence can be bounded (that is, of fixed length) or unbounded (that is, of variable length). A CORBA sequence maps to a COM structure. This section describes the CORBA-to-COM mapping rules for bounded and unbounded sequences.

Unbounded Sequences

The following is an OMG IDL unbounded sequence of some type, T:

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

This maps to:

```
// COM IDL
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 COM IDL struct containing a unique pointer to a conformant array of type U, where U is the COM IDL mapping of T. The enclosing struct in the COM IDL mapping is necessary, to provide a scope in which extent and data bounds can be defined.

Bounded Sequences

The following is an OMG IDL bounded sequence of some T type, which can grow to be N size):

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

This maps to:

```
// COM IDL
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.

Arrays

OMG IDL arrays map to corresponding COM arrays. The array element types follow their standard mapping rules. The following is an OMG IDL array of some type, `T`:

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

This maps to a COM IDL array of type `U`:

```
// COM IDL
const long N = …;
typedef … U;
typedef U ARRAY_OF_U[N];
```

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

Note: If the ellipsis (that is, …) in the OMG IDL example represents `octet`, the ellipsis in the COM IDL example must be `byte`. That is why the types of the array elements have different names in the two texts.

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 351.
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 maps to a COM IDL interface and an exception structure that appears as the last parameter of any operation mapped from OMG IDL to COM IDL.

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

The following OMG IDL 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 NotAuthorized{};
        float Deposit(in float Amount) raises(InvalidAmount);
        float Withdraw(in float Amount) raises(InvalidAmount,
            NotAuthorized);
    };
};
```

This maps to:

```
// COM IDL
struct BANK_InsufficientFunds
{
    float balance;
};
struct BANK_InvalidAmount
{
    float amount;
};
struct BANK_Account_NotAuthorized
{
};

interface IBANK_AccountUserExceptions: IUnknown
{
    HRESULT get_InsufficientFunds([out] BANK_InsufficientFunds *
        exceptionBody);
    HRESULT get_InvalidAmount([out] BANK_InvalidAmount *
        exceptionBody);
    HRESULT get_NotAuthorized([out] BANK_Account_NotAuthorized *
        exceptionBody);
};

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

System Exceptions

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

```
// COM IDL
SetErrorInfo(OL,NULL); //Initialize 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 accesses the error object as follows:

```
// COM C++
// 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(…);
    }
}
```

The Any Type

The OMG IDL any type does not map directly to COM. It maps to the following interface definition:

```
// COM IDL
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

There is no standard CORBA-to-COM mapping specified for OMG IDL contexts.

Object References

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

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

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

This maps to:

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

The following COM C++ code is based on the preceding COM IDL definition:

```
// 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();
```

Modules

An OMG IDL definition contained within the scope of an OMG IDL module maps to its corresponding COM IDL definition, by prefixing the name of the COM IDL type definition with the name of the module. For example:

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

This maps to:

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

Constants

An OMG IDL const type maps to a COM IDL const type. 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 = "…";
```

This maps to:

```
// COM 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 LPSTR STR = "…";
```

Enumerated Types

A CORBA enum maps to a COM enum. For example:

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

This maps to:

```
// COM IDL
[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, COM IDL 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.

CORBA observes scoping of enum declarations, but COM ignores such scoping and always treats an enum declaration as though it were globally defined. To avoid potential name clashes, translated enums in COM IDL are prefixed with the enclosing type in which they are declared. Therefore, in the preceding example, the OMG IDL `A_or_B_or_C` enum is mapped to `MyIntf_A_or_B_or_C`.

The COM IDL keyword, `v1_enum`, is required 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 COM IDL only supports 2^{16} identifiers. Truncation might therefore result.

Scoped Names

An OMG IDL scoped name must be fully qualified in COM IDL 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);
};
```

This maps to:

```
COM IDL
[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);
};
```

Typedefs

A CORBA typedef maps to a COM IDL typedef. A typedef is most often used for array and sequence definitions. For example:

```
// OMG IDL
interface Account {…};

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

This maps to:

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

19

COM-to-CORBA Mapping

COM types are defined in Microsoft IDL (COM IDL). CORBA types are defined in OMG IDL. To allow interworking between CORBA clients and COM servers, CORBA clients must be presented with OMG IDL versions of the interfaces exposed by COM objects. Therefore, it must be possible to translate COM types to OMG IDL. This chapter outlines the COM-to-CORBA mapping rules.

For the purposes of illustration, this chapter describes a textual mapping between COM IDL 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.

Basic Types

COM basic types map to corresponding types in CORBA. Table 19.1 shows the mapping rules for each basic type.

COM IDL	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 19.1: *COM-to-CORBA Mapping Rules for Basic Types*

Strings

Table 19.2 shows the COM-to-CORBA mapping rules for strings.

COM IDL	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 19.2: *COM IDL to OMG IDL String Mappings*

An error occurs if a COM server returns a BSTR containing embedded nulls to a CORBA client.

Unbounded Strings

The following is a COM IDL statement for an unbounded string:

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

This maps to:

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

Bounded Strings

The following is a COM IDL statement for a bounded string:

```
// COM IDL
const long N = …;
typedef [string,unique] char (* BOUNDED_STRING) [N];
```

This maps to:

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

Unicode Unbounded Strings

The following is a COM IDL statement for a unicode unbounded string:

```
// COM IDL
typedef [string,unique] LPWSTR UNBOUNDED_UNICODE_STRING;
```

This maps to:

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

Unicode Bounded String

The following is a COM IDL statement for a unicode bounded string:

```
// COM IDL
const long N = …;
typedef [string,unique] wchar t*(BOUNDED_UNICODE_STRING) [N];
```

This maps to:

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

Interfaces

This section describes the COM-to-CORBA mapping rules for interfaces. A COM interface maps to an OMG IDL interface.

Properties and Methods

The following mapping rules apply for COM properties and methods:

- A COM method maps to an OMG IDL operation.
- A COM property that has a method to get the value, and a method to set the value, maps to a normal OMG IDL attribute.
- A COM property that only has a method to get the value maps to a readonly OMG IDL attribute.

For example:

```
// COM IDL
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);
    // f is a propget/propput pair
    [propget] HRESULT f([out] float* val);
    [propput] HRESULT f([in] float val);
    // l is a propget only
    HRESULT l([out,retval] long* val);
    // b is a propput only
    [propput] HRESULT b([in] boolean val);
};
```

This maps 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);
    attribute float f;
    readonly attribute long l;
    attribute boolean b;
};
```

The following mapping rules apply for parameters and return types:

- A COM IDL `[in]` parameter maps to an OMG IDL `in` parameter.
- A COM IDL `[out]` parameter maps to an OMG IDL `out` parameter.
- A COM IDL `[inout]` parameter maps to an OMG IDL `in,out` parameter.
- A COM IDL `[retval,out]` parameter maps to an OMG IDL `return` value.

All COM interfaces must have a HRESULT return type (which is essentially a typedef to a long 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.

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 that is used to provide a standard mechanism for accessing multiple interfaces from a client, even though those interfaces are not related by inheritance. It is defined in CORBA as follows:

```
// 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 the CORBA::Object type.

The following is a COM IDL definiton for an interface, IFoo:

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

This maps to:

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

Complex Types

This section describes the COM-to-CORBA mapping rules for the following complex types:

- Structs
- Unions
- Pointers
- Arrays
- Exceptions
- Variants

COM constructed types such as `struct`, `union`, and `array` map to the corresponding CORBA constructed types. This is outlined in the following subsections.

Structs

A COM IDL struct maps to a corresponding struct in OMG IDL. Each field in the struct is mapped according to the mapping rules for its type. For example:

```
// COM IDL
struct foo {
    long l;
    LPTSTR s;
};
```

This maps to:

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

Unions

This section describes the COM-to-CORBA mapping rules for encapsulated and non-encapsulated unions.

Encapsulated Unions

The following is an example of a COM IDL encapsulated union:

```
// COM IDL
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;
```

This maps 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

COM IDL non-encapsulated unions (and COM IDL encapsulated unions with non-constant discriminators) map to the OMG IDL any type. For example:

```
// COM IDL
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;
```

This maps 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 COM IDL union.

Pointers

The following mapping rules apply for pointers:

- A COM IDL reference pointer maps to a CORBA sequence containing one element.
- A COM IDL unique pointer (with no aliases or cycles) maps to a CORBA sequence containing zero or one elements.
- A COM IDL full pointer (with no aliases or cycles) maps to a CORBA sequence containing zero or one elements.

A runtime error occurs 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.

Arrays

This section describes the COM-to-CORBA mapping rules for arrays.

Fixed Arrays

A COM IDL fixed-length array maps to an OMG IDL fixed-length array. The type of the array element is mapped according to the mapping rules for that type. For example:

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

This maps to:

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

Non-Fixed Arrays

A COM IDL non-fixed-length array maps to an OMG IDL sequence. For example:

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

This maps to:

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

Exceptions

This section describes the COM-to-CORBA mapping rules for exceptions.

System Exceptions

COM system exception codes are defined with the FACILITY_NULL and FACILITY_RPC facility codes, which map to CORBA standard exceptions. Table 19.3 lists the mappings from COM FACILITY_NULL exceptions to CORBA.

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

Table 19.3: *Mapping from COM FACILITY_NULL to CORBA Standard Exceptions*

COM	CORBA
E_HANDLE	BAD_PARAM

Table 19.3: *Mapping from COM FACILITY_NULL to CORBA Standard Exceptions*

Table 19.4 list the mappings from COM FACILITY_RPC exceptions to CORBA. (All FACILITY_RPC exceptions not listed in Table 19.4 map to the 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
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

Table 19.4: *Mapping from COM FACILITY_RPC to CORBA Standard Exceptions*

COM	CORBA
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 19.4: *Mapping from COM FACILITY_RPC to CORBA Standard Exceptions*

User Exceptions

COM user-defined exception codes map to CORBA user exceptions and 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;};
```

Variant Types

A COM `VARIANT` type maps to the OMG IDL `any` type. The allowable `VARIANT` 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 occurs at runtime if a CORBA client returns an inconvertible any type to a COM server.

Constants

A COM IDL constant maps to a corresponding OMG IDL constant. For example:

```
// COM 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 = "…";
```

This maps 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 = "…";
```

Enumerated Types

A COM IDL enum maps to an OMG IDL enum. For example:

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

This maps to:

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

COM enumerators can have explicitly assigned values. In CORBA, rather than being explicitly assigned, enum values start at zero and increase in increments of one. Because OMG IDL does not support explicitly tagged enumerators, the CORBA view of a COM object must maintain the mapping of the values of the enumerators. Therefore, COM enums with explicitly assigned values are mapped to OMG IDL enums in ascending order of their value, to preserve the order of the enumerators.

Scoped Names

COM IDL considers all definitions to be at global scope. Therefore, to avoid collisions across interfaces when translating from COM IDL 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); }
```

This 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);
}
```

Typedefs

A COM IDL typedef maps to an OMG IDL typedef. For example:

```
// COM IDL
interface IAccount : IUnknown {…};
Typedef struct {
…
} AccountList;
```

This maps to:

```
// OMG IDL
interface Account {…};

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

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 initialization 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*

21

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 specific to OrbixCOMet, which are held in the `install-dir\config\orbixcomet.cfg` file (where `install-dir` represents the Orbix installation directory). The configuration variables are declared within various scopes within the `COMet{...}` scope. As shown in this section, you can also use the `COMet.Scope name.` prefix to refer to individual entries in the configuration file.

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, which indicates where these DLLs are located. The default value is `"COMet\Handler DLLs"`. It is placed in HKEY_CLASSES_ROOT, so that users without administrative privileges can read and modify the values. The values look like the following:

```
[HKEY_CLASSES_ROOT\COMet\Handler DLLs]
grid="c:\foo\bar.dll"
bank="c:\foo\bar2.dll"
```

Key `COMet.Config.COMET_ROOT="`*`install-dir`*`\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. In this example, *`install-dir`* represents the Orbix installation directory.

Key `COMet.Config.COMET_SHUTDOWN_POLICY="implicit"`

Description Valid values are:

`"implicit"`	This is the default setting. It means that OrbixCOMet shuts down the first time `DllCanUnloadNow` is about to return a `yes` value.
`"explicit"`	This means that you must make a call to `ORB::ShutDown()`, to force OrbixCOMet to shut down.
`"none"`	This means that OrbixComet does 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 can arise, however, if the DLL is re-used, because Orbix has already been shut down.
`"atExit"`	This means that the OrbixCOMet bridge only shuts down at process-exit time. This is the recommended setting when running applications in the Visual Basic development environment.

Key `COMet.Config.COMET_UPDATE_LEVEL="3-3-00"`

Description This includes information about the version and patch level of OrbixCOMet. You should quote this value whenever posting to `support@iona.com`.

Key `COMet.Config.CALL_BACK_TIMER_DELAY="10"`

Description This sets the interval, in milliseconds, for the OrbixCOMet event timer. The default value is `10`, but this can be reduced to improve performance on faster machines. This value is only used with client callbacks, or when CORBA clients are communicating with DCOM servers.

Key `COMet.Config.FORCE_PROXY="yes"`

Description This specifies whether OrbixCOMet should perform a `CoUnMarshalInterface` on interface pointers for DCOM callback objects. If this value is set to "yes", it forces the creation of a proxy for unmarshalling the callback object. You should only set this value to "no" if problems are being experienced with callbacks.

Key `COMet.Mapping.UseSAFEARRAYMapping="yes"`

Description The Automation mapping for OMG IDL sequences and arrays is to Automation compatible SafeArrays, as described in the *COM/CORBA Interworking* specification. 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, set this value to "no".

Key `COMet.Mapping.SAFEARRAYS_CONTAIN_VARIANTS="yes"`

Description There is a problem in Visual Basic when dealing with SafeArrays as `out` parameters. Visual Basic does not correctly check the `V_VT` type of the SafeArray contents, and automatically assumes that they are of the `VARIANT` type. When constructing the `out` parameter, OrbixCOMet cannot tell if the parameter type has been declared (using the `dim` statement) as the real type from the type library or simply as `SAFEARRAY`. This key determines whether OrbixCOMet should treat, for example, a sequence of `long` types as mapping to a SafeArray of `long` types, or to a SafeArray of `VARIANT` types where each `VARIANT` contains a `long` type.

Key `COMet.Mapping.KEYWORDS="grid, DialogBox, bar, Foobar, height"`

Description This allows you to enter a list of words that you want prefixed with `IT_clash`, to avoid clashes when using `ts2idl` to generate IDL definitions.

Key `COMet.Mapping.AUTOEVENTS="No"`

Description This allows you to choose the method of handling and dispatching incoming Orbix events to COM or Automation clients and servers. In this context, an Orbix event is a call from a CORBA client to a COM or Automation server, or a callback from a CORBA server to a COM or Automation client. In both cases, the incoming call to the COM or Automation client or server must be retrieved and dispatched to the appropriate COM or Automation object. Valid values are

"No"	This means it is the COM or Automation programmer's responsibility to ensure processing of incoming Orbix events. This is achieved by periodically calling the `DispatchEvents()` method in the (D)`IOrbixServerAPI` interface, usually via a timer on the application's main thread.
"Yes"	This means OrbixCOMet automatically retrieves and dispatches the events, using a dedicated thread.
"WinMode"	This means OrbixComet automatically converts incoming Orbix events into messages that are sent to the message queue of the COM or Automation application. The application's message pump then dispatches these requests. This setting is a hybrid of the "yes" and "no" values, in that Orbix events are dispatched automatically by the application thread that is processing the application's message pump.

Key `COMet.Mapping.ALLOW_ANON_MARKERS="no"`

Description In the case of When you have a CORBA client communicating with a DCOM server, anonymous binds to CORBA wrappers have been deprecated. Instead, `ts2idl` generates a string of the `const` type, which takes the following form:

```
#ifndef _IT_COMET_ANON_
#define _IT_COMET_ANON_
const string IT_ANON = "IT_COMET_ANON";
#endif
```

Markers used in calls to `_bind()` should begin with this string. The following are examples of valid markers:

```
"IT_COMET_ANON"
"IT_COMET_ANON1"
"IT_COMET_ANON_excelObj"
```

Because of this, the default for the `COMet.Mapping.EXTRA_REF_CORBAVIEW` configuration variable is now "no", in contrast to previous releases. For backwards compatibility, anonymous binds are allowed if the

`COMet.Mapping.ALLOW_ANON_MARKERS` configuration variable is set to `"yes"`, but this is not recommended in most cases. (A possible exception to this might be with the use of the (D)`IOrbixServerAPI` interface.)

Key `COMet.Debug.MessageLevel="255, c:\temp\comet.log"`

Description This can take any value in the range `0`–`255`. The higher the value, the more logging information is available. In the preceding example, a value of `255` means that all messages are logged, in the specified `comet.log` file.

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 optimizes 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 `advice.iona.com` machine, the `orbix.hst` file looks 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 typeman has read-only rights. This setting is particularly important when there are multiple DCOM clients of OrbixCOMet sharing the bridge on a single intermediary machine. It is also important for internet deployment. The `typeman -e *` command instructs `typeman` to read the entire contents of the Interface Repository into the type store. You should set this configuration variable to "`no`" before priming the type store. You should set it to "`yes`" after priming the type store.

Key	`COMet.TypeMan.TYPEMAN_LOGGING="none"`
Description	This key determines how the OrbixCOMet type store manager, `typeman`, logs information about the type store contents. Valid values are:

`"None"`	This is the default value.
`"stdout"`	This sends output to the screen. Use this option only with `typeman`.
`"DBMon"`	This sends output to `DBMon.exe`.
`"file"`	This sends output to the file specified by the `COMet.Typeman.TYPEMAN_LOG_FILE` configuration variable.

Key	`COMet.TypeMan.TYPEMAN_LOG_FILE="c:\temp\typeman.log"`
Description	If the value of the `TYPEMAN_LOGGING` configuration variable is set to `"file"`, this key specifies the full path to that output file for `typeman` logging instructions.
Key	`COMet.TypeMan.TYPEMAN_SSL_ENABLED="yes"`
Description	In a secure CORBA environment, the Interface Repository is configured to run as an SSL-secured CORBA server. The OrbixCOMet type store manager (`typeman.exe`) retrieves type information from the Interface Repository. Therefore, in an SSL-secured environment, `typeman` must also be configured to run securely. You can use this configuration variable to indicate that typeman is SSL-enabled.
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. You can use this in conjunction with the `COMet.TypeMan.TYPEMAN_IFR_NS_NAME` configuration variable.

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.

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="`*`install-dir`*`\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 + "install-dir\bin\bongo.zip;
install-dir\bin\marimba.zip;install-dir\bin\NSclasses.zip;
install-dir\bin\utils.zip;install-dir\bin\rt.jar;
install-dir\bin\orbixweb.jar;install-
dir\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 Orbix{...} scope. 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.

22

OrbixCOMet Utility Options

This chapter describes the various options that are available with each of the OrbixCOMet command-line utilities.

Typeman Options

The `typeman` utility manages the OrbixCOMet type store. The options available with `typeman` are:

`-b` This allows you to view the bucket sizes in the memory cache hash table.

`-c` This allows you to view the contents of the type store disk cache. You can specify `-cn` to view the contents in the order in which they have been added to the cache. You can specify `-cu` to view the UUID of each type listed. (Every type in the type store has an associated UUID, regardless of whether it has originated from a type library or the Interface Repository. OrbixCOMet generates UUIDs for OMG IDL types, using the MD5 algorithm, as specified by the OMG.)

`-d` This allows you to set the number of allowable entries in the disk cache. You must qualify `-d` with a number, which indicates the number of allowable entries. The default is `2000`. This value should normally be ten times larger than the value specified with the `-m` option, which sets the number of allowable entries in the memory cache.

`-e` This instructs `typeman` to search the Interface Repository or a type library for a specific item of type information, and then add it to the type store cache. You must qualify `-e` with an OMG IDL interface name, a full type library pathname, the UUID of a COM IDL interface, or the name of a text file that lists the aforementioned in any combination. Refer to "Adding New Information to the Type Store" on page 161 for details of how to specify each.

If you specify an OMG IDL interface name that is not already in the cache, `typeman` looks up the Interface Repository. If you specify a type library pathname or UUID that is not already in the cache, `typeman` looks up the relevant type library. Regardless of where the type information originates, `typeman` then copies it to the type store cache.

`-f` This allows you to view the type store data files. These include the disk cache data file (`typeman._dc`), the disk cache index file (`typeman.idc`), the disk cache empty record index file (`typeman.edc`), and the UUID name mapper file (`typeman.map`).

`-h` This instructs `typeman` to display `"Cache miss"` on the screen, if a type it is looking for is not already in the cache. If the type is already in the cache, typeman displays `"Mem cache hit"` on the screen.

`-i` This instructs `typeman` to always query the Interface Repository for an item of OMG IDL type information. This can be used to compare the performance of different ORBs and so on.

`-l` This logs the type store basic contents to the screen. Enter `-l+` to log newly added and deleted entries. Enter `-l tlb` to log type library information. Enter `-l union` to log OMG IDL information for unions.

`-r` This generates static bridge compatible names for OMG IDL sequences.

`-v` This allows you to view the v-table contents for an interface or struct. This option provides output such as the following:

```
Name sorted                V-table        DispId   Offset
balance           get       makeLodgement   1        0
makeLodgement               makeWithdrawal  2        1
makeWithdrawal              balance         3        2
overdraftLimit    get       overdraftLimit  4        3
```

-w This deletes the type store contents. This means it deletes the disk cache data file (typeman._dc), the disk cache index file (typeman.idc), and the disk cache empty record index file (typeman.edc). If you also want to delete the UUID name mapper file (typeman.map), you must enter -wm instead. Deleting the type store contents is useful when you want to reprime the cache. You might want to reprime the cache, for example, if it contains type information for an interface that has subsequently been modified.

-z This allows you to view the actual size to which the memory cache temporarily grows when typeman is loading in a containing type (such as a module) to retrieve a contained type (such as an interface within that module).

-? This outputs the usage string for typeman.

-?2 This allows you to view the format of the entries that you can include in a text file, which you can specify with the -e option, if you want to prime the cache with any number and combination of type names, type library pathnames, and COM IDL UUIDs simultaneously.

Ts2idl Options

The ts2idl utility allows you to create OMG IDL definitions, based on existing type library information in the type store. Similarly, it allows you to create COM IDL definitions, based on existing OMG IDL type information in the type store. The options available with ts2idl are:

-b You can use this option when generating OMG IDL, based on type library information in the type store. It specifies that interface pointers that are passed as parameters to operations described in the type library are to be mapped as the CORBA::Object type in the generated OMG IDL, rather than as their dynamic type. Use -b in conjunction with -r.

-c You can use this option when generating COM IDL, based on OMG IDL information in the type store. It instructs ts2idl not to query the Interface Repository for the specified OMG IDL interface. In this case, ts2idl only searches the type store for the relevant information.

`-f` Use this to specify the name of the IDL file to be created. You must qualify this option with the filename (for example, `grid.idl`). In turn, you must qualify the filename with the name of the item of type information on which it is being based. For example:

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

`-i` This instructs `ts2idl` to generate an OMG IDL file, based on type library information in the type store.

`-m` This instructs `ts2idl` to generate a COM IDL file, based on OMG IDL information in the type store. This is a default option. You do not have to specify `-m`, to create a COM IDL file. Unless you explicitly specify `-i`, to create OMG IDL, `ts2idl` assumes you want to create a COM IDL file.

`-p` You can use this option when generating COM IDL, based on OMG IDL information in the type store. It is a useful labor-saving device that produces a makefile for building the proxy/stub DLL, which subsequently marshals requests from the COM client to CORBA objects.

`-r` You can use this option when generating COM IDL, based on OMG IDL information in the type store. It can be used for generating COM IDL, based on complicated OMG IDL interfaces that employ user-defined types. The `-r` option completely resolves those types and produces COM IDL for them.

`-s` This forces inclusion of standard types from `ITStdcon.idl` and `orb.idl`.

`-v` This outputs the usage string for `ts2idl`. You can also use `-?` for this.

Ts2tlb Options

The `ts2tlb` utility allows you to create a type library, based on existing OMG IDL type information in the type store. The options available with `ts2tlb` are:

`-f` Use this to specify the name of the type library to be created. You must qualify this option with the type library filename. The default is to use the type name on which the type library is based, with a `.tlb` suffix (for example, `grid.tlb`).

-i This indicates that interface prototypes are to appear as IDispatch, instead of using the specific interface name. If you do not specify this option, the specific interface name is used.

-l Use this to specify the internal library name in which the type library is to be created. You must qualify this option with the library name. The default is to use the type name on which the type library is based, with an IT_Library_ prefix (for example IT_Library_grid).

-p This prefixes parameter names with it_.

-v This outputs the usage string for ts2tlb. You can also use -? for this.

Ts2sp Options

The ts2sp utility allows you to create handler DLLs to encapsulate any extra handler code that you might have developed and want to use at runtime, to inject extra functionality into your applications. The options available with ts2sp are:

-d You can use this option to specify the output directory to which you want the generated handler DLL to be saved. If you specify -d, you must qualify it with the full path to the directory you want to use. If you do not specify -d, the generated handler DLL is saved to the current directory in which you run the ts2sp command.

-f This allows you to specify the original source file on which the handler DLL is to be based. You must qualify this option with the filename (including extension) of the original source file.

-m Use this to specify the internal library name in which the type library is to be created. You must qualify this option with the library name. The default is to use the type name on which the type library is based, with an IT_Library_ prefix (for example IT_Library_grid).

-n This specifies the keyname of the handler DLL. You must qualify -n with the keyname.

-p This instructs ts2sp to generate a makefile, which can then be used to be build the handler DLL. You must qualify -p with the name of the makefile. (For example, if you enter -p ClientFilter, the ts2sp utility generates a makefile called ClientFilter.mak.)

`-v` This outputs the usage string for `ts2sp`. You can also use `-?` for this.

Aliassrv Options

The `aliassrv` utility is used in association with the `srvAlias` GUI tool to allow you to replace a legacy DCOM server with a CORBA server. Refer to "Replacing an Existing DCOM Server" on page 177 for more details. The options available with `aliassrv` are:

`-c` This indicates the CLSID of the legacy DCOM server that is being replaced. You must qualify this option with the actual CLSID enclosed in opening and closing braces (that is, `{` and `}`).

`-d` This deletes the registry key denoted by the specified CLSID. You must qualify `-d` with the `-c` option and CLSID.

`-r` This aliases the specified CLSID to OrbixCOMet, so the next time you run a DCOM client of the legacy server whose CLSID is specified, OrbixCOMet is used instead of the legacy server. You must qualify `-r` with the name of the file that contains the modified registry entries, to restore the registry entries on the destination machine. For example:

```
aliassrv -r replace.reg -c {CLSID}
```

`-v` This outputs the usage string for `ts2sp`. You can also use `-?` for this.

Custsur Options

The `custsur.exe` is a generic surrogate program that hosts the OrbixCOMet DLLs when the bridge is loaded out-of-process. You can use `custsur` to generate IORs for non-Orbix clients. The options available with `custsur` are:

`-f` This specifies the filename to which the IOR is to be written.

`-g` This instructs `custsur` to generate an IOR.

`-i` This specifies the interface name for which the IOR is to be created.

`-m` This specifies the marker name.

`-s` This specifies the name of the server.

-t	This specifies a time-out value, in milliseconds, for the server being implemented by `custsur`.

-v	This outputs the usage string for `ts2sp`. You can also use `-?` for this.

Tlibreg Options

The `ttlibreg.exe` utility allows you to register and unregister a type library that you have generated from OMG IDL, using `ts2tlb`. The tlibreg utility registers the type library with the Windows registry. The options available with `tlibreg` are:

-u	This unregisters a type library. You must qualify this option with the full type library pathname.

-v	This outputs the usage string for `ts2sp`. You can also use `-?` for this.

Index

A

AbortSlowConnects() 211, 244
Activate() 182, 224
ActivateCVHandler() 212, 244
ActivateOutputHandler() 212, 244
ActivatePersistent() 183, 225
activating CORBA servers 107
adding information to type store 163
algorithm, MD5 35
aliassrv 177
  options 368
anys 185, 226
  mapping from CORBA to Automation 293
  mapping from CORBA to COM 325
API 181
arrays
  mapping from COM to CORBA 342
  mapping from CORBA to Automation 290
  mapping from CORBA to COM 321
  OMG IDL definition 265
attributes 256
  mapping from CORBA to Automation 275
  mapping from CORBA to COM 312
Automation clients
  implementing in PowerBuilder 23, 52
  implementing in Visual Basic 26, 50
  introduction to 9
Automation interfaces
  DCollection 184
  DICORBAAny 184
  DICORBAFactory 189
  DICORBAFactoryEx 191
  DICORBAObject 192
  DICORBAStruct 194
  DICORBASystemException 195
  DICORBATypeCode 196
  DICORBAUnion 199
  DICORBAUserException 200
  DIForeignComplexType 200
  DIForeignException 201
  DIObject 201
  DIObjectInfo 201
  DIOrbixObject 202
  DIOrbixORBObject 205
  DIOrbixServerAPI 181
  DIOrbixSSL 218
  DIORBObject 220
  IForeignObject 221
  IOrbixSSL 250
Automation servers 10

B

base interfaces, finding 213
BaseInterfacesOf() 213, 245
basic types 262
  mapping from Automation to CORBA 300
  mapping from COM to CORBA 334
  mapping from CORBA to Automation 272
  mapping from CORBA to COM 310
Bind() 203
binding
  early 76
  late 76
  PingDuringBind() 210, 242
  to objects 203
  View to target 48, 49, 59
bridge 5, 8

C

caching mechanism 159
callbacks 123–132
  implementing 123
  ReclaimCallbackStore() 211
catching COM exceptions 118
clients
  Automation, using PowerBuilder 23, 52
  Automation, using Visual Basic 26, 50
  collocation 214, 246
  COM, using C++ 36, 55
  CORBA, using COM C++ 65
  CORBA, using PowerBuilder 65
  CORBA, using Visual Basic 65
  implementing in Automation 45
  implementing in COM C++ 55
  implementing in CORBA 65
  writing 94, 125
client-side footprint 151
clone() 202

CloseChannel() 205, 213, 238, 246
CoCreateInstance() 60
Collocated() 214, 246
collocation 214, 246
COM apartments and threading 82
COM clients
  implementing 36
  implementing in C++ 36, 55
  introduction to 9
COM interfaces
  ICORBA_Any 225
  ICORBA_TypeCode 230
  ICORBA_TypeCodeExceptions 234
  ICORBAFactory 227
  ICORBAObject 228
  IForeignObject 235
  IMonikerProvider 236
  IOrbixObject 237
  IOrbixORBObject 239
  IOrbixServerAPI 223
  IORBObject 252
COM library 9
COM servers 10
cometcfg 160
command-line utilities
  locating 161
  options 363
commands
  aliassrv 177
  custsur 97
  srvAlias 177
  tlibreg 35
  ts2idl 170
  ts2sp 174
  ts2tlb 173
  typeman 163
configuration
  GetConfigValue() 214, 247
  ReinitialiseConfig() 215, 248
  SetConfigValue() 214, 247
configuration handlers
  activating 212, 244
  deactivating 212, 244
  order of 212, 245
configuration keys
  common 360
  Orbix 361
  OrbixCOMet 353
connecting
  AbortSlowConnects() 211, 244
  ConnectionTimeout() 208, 241
  EagerListeners() 215, 247
  MaxConnectRetries() 209, 241
  NoReconnectOnFailure() 211, 243
ConnectionTimeout() 208, 241
constants 267
  mapping from COM to CORBA 346
  mapping from CORBA to Automation 296
  mapping from CORBA to COM 328
constructed types 263
  creating 191, 317
  mapping from CORBA to COM 317
content_type() 189, 199, 234
context 293, 326
CORBA clients
  implementing in Automation 76
  implementing in COM 82
  introduction to 10
CORBA exceptions
  handling in Automation 113
  handling in COM 118
  properties of 112
  raising in a server 120
CORBA servers
  implementing 102
  introduction to 9
CORBA.ORB.2 208
CORBA.ORB.Orbix 208
Count() 184
CreateObject() 70, 190, 228
CreateType() 192, 283
CreateTypeById() 192
creating
  constructed types 191, 317
  exceptions 191, 317
  handler DLLs 153
  IDL files 57, 168
  structs 191, 317
  type libraries 47, 171
  unions 191, 317
custsur 31, 89
  options 368

D

DCollection 184
Deactivate() 183, 224
DeactivateCVHandler() 212, 244
DeactivateOutputHandler() 212, 245
deactivating CORBA servers 107
default_index() 188, 198, 233

DefaultTxTimeout() 214, 246
deleting type store contents 166
deploying applications 139–155
deployment models 139–146
  bridge on each client machine 140
  bridge on server machine 142
  bridge shared by multiple clients 144
  internet 146
Developing 45, 55
development utilities 157
diagnostics
  output() 248
  SetDiagnostics() 216, 248
DICORBAAny 184
DICORBAFactory 68, 189
DICORBAFactoryEx 191, 283
DICORBAObject 75, 192
DICORBAStruct 194, 283
DICORBASystemException 112, 195, 292
DICORBATypeCode 196
DICORBAUnion 199, 285
DICORBAUserException 200
DIForeignComplexType 200, 283
DIForeignException 112, 201
DIObject 201
DIObjectInfo 201
DIOrbixObject 75, 202
DIOrbixORBObject 66, 205
DIOrbixServerAPI 89, 181
DIOrbixSSL 218
DIORBObject 66, 220
discriminator_type() 188, 198, 233
DispatchEvents() 183, 224
dual interfaces 34
Dumping 167

E

EagerListeners() 215, 247
early binding 76
enums 264
  mapping from Automation to CORBA 306
  mapping from COM to CORBA 346
  mapping from CORBA to Automation 296
  mapping from CORBA to COM 328
equal() 231
equivalence, of object references 193
Err object 114
EX_completionStatus() 195
EX_Id() 201
EX_majorCode() 201
EX_minorCode() 195
exception handling 109–120
  inline 115
exceptions 109–120, 258
  creating 191, 317
  handling in Automation 113
  handling in COM 118
  mapping from COM to CORBA 343
  mapping from CORBA to Automation 291
  mapping from CORBA to COM 322
  properties of 112
  raising in a server 120
exposing DCOM servers to CORBA clients 89

F

factory
  See object factory
FileDescriptor() 205, 238
forward declaration 268

G

generating handler DLLs 174
generating skeleton code 176
  for callbacks 125
  for servers 101
generating UUIDs 35
get_BadKind() 235
get_Bounds() 235
get_CORBAAnyData() 226
get_Host() 238
get_InterfaceName() 238
get_Marker() 238
get_moniker() 237
get_typeCode() 226
get_value() 226
GetConfigValue() 214, 247
GetCORBAObject() 221
GetForeignReference() 222, 236
GetImplementation() 193, 229
GetInitialReferences() 221, 254
GetInterface() 193, 229
GetItem() 184
GetObject() 68, 190, 227
  example 48, 125
  parameter to 69
GetOrbixSSL() 217, 249
GetRepositoryId() 222
GetSecurityName() 219, 251
GetServerAPI() 249

GetUniqueId() 236

H
handler DLLs
  creating 153
  generating 174
  loading at runtime 154
  managing 154
  registering 153
handling exceptions
  in Automation 113
  in COM 118
Hash() 194, 230
HasPassword() 219, 251
HasValidOpenChannel() 205, 238
Host() 204

I
ICORBA_Any 225
ICORBA_TypeCode 230
ICORBA_TypeCodeExceptions 234
ICORBAFactory 68, 227
ICORBAObject 75, 228
id() 186, 197, 232
IDL files
  creating from type store 36
IDL operations 74
IDL, creating from type store 168
IForeignObject 221, 235
IMonikerProvider 236
implementation repository 54, 63
  registering CORBA servers 108
Implementing 22, 36, 65
implementing
  Automation clients 45–54
  callbacks 123
  COM client 55–63
  CORBA clients in Automation 76
  CORBA clients in COM 82
  CORBA servers 102
  interfaces 102
  server for client callbacks 128
inheritance 259
  mapping from CORBA to COM 314
InitScopeSSL() 219, 250
InitSSL() 218, 250
inline exception handling
  in Automation 115
installing
  application runtime 147
  development language runtime 147
  Orbix runtime 147
  OrbixCOMet runtime 149
INSTANCE_clone() 200
INSTANCE_repositoryId() 200
InterfaceName() 205
interfaces 255
  finding base interfaces 213, 245
  IDL, implementing 102
  mapping from Automation to CORBA 301
  mapping from COM to CORBA 336
  mapping from CORBA to Automation 274
  mapping from CORBA to COM 311
  to CORBA objects 75
  to ORB 66
internet deployment 146
Internet Explorer 31
interworking
  concepts 4
  interfaces on objects 75
  model 5
interworking model
  implementation of 6
IOrbixObject 75, 237
IOrbixORBObject 66, 239
IOrbixServerAPI 89, 223
IOrbixSSL 250
IORBObject 66, 252
IsA() 193, 229
IsBaseInterfaceOf() 213, 245
IsEquivalent() 193, 229
IsNil() 193, 229
Item() 184

K
kind() 186, 196, 231

L
late binding 76
length() 189, 199, 234
libraries, Orbix runtime 147
libreg
  options 369
LoadHandler() 216, 249
loading handler DLLs at runtime 154

M
managing handler DLLs 154

mapping from Automation to CORBA
  basic types 300
  enums 306
  interfaces 301
  methods 302
  object references 305
  properties 302
  safearrays 304
  strings 301
  typedefs 307
  variants 305
mapping from COM to CORBA
  arrays 342
  basic types 334
  constants 346
  enums 346
  exceptions 343
  interfaces 336
  methods 336
  pointers 342
  properties 336
  scoped names 347
  strings 335
  structs 339
  typedefs 348
  unions 340
  variants 345
mapping from CORBA to Automation
  anys 293
  arrays 290
  attributes 275
  basic types 272
  constants 296
  enums 296
  exceptions 291
  interfaces 274
  modules 295
  object references 293
  operations 277
  scoped names 297
  sequences 287
  strings 273
  structs 283
  typedefs 298
  unions 285
mapping from CORBA to COM
  anys 325
  arrays 321
  attributes 312
  basic types 310
  constants 328
  constructed types 317
  enums 328
  exceptions 322
  inheritance 314
  interfaces 311
  modules 327
  object references 326
  operations 313
  scoped names 330
  sequences 320
  strings 310
  structs 317
  typedefs 331
  unions 319
marker() 204
markers, setting 204
MaxConnectRetries() 209, 241
MD5 algorithm 35
member_count() 187, 197, 232
member_label() 188, 198, 233
member_name() 187, 197, 232
member_type() 188, 198, 233
methods
  mapping from Automation to CORBA 302
  mapping from COM to CORBA 336
minimizing client-side footprint 151
modules 258
  mapping from CORBA to Automation 295
  mapping from CORBA to COM 327

N

name() 187, 197, 232
Naming Service 70
Narrow() 78, 217
narrowing object references 77, 82
nil object references 193
NonExistent() 193, 229
NoReconnectOnFailure() 211, 243

O

object factory 48, 58
object references
  binding 203
  converting to strings 220, 252
  equivalent 193
  finding 68–75
  getting foreign 222, 236
  mapping from Automation to CORBA 305

mapping from CORBA to Automation 293
mapping from CORBA to COM 326
narrowing 77, 82
nil 193
obtaining 68–75
object table, resizing 209, 243
objects
instantiating in bridge 107
interface to CORBA 75
registering with OrbixCOMet 105
ObjectToString() 220, 252
obtaining a reference to a CORBA object
in Automation 48
in COM 58
obtaining a reference to the ORB 66
obtaining object references 68
OMG IDL 255–269
arrays 265
attributes 256
basic types 262
constants 267
constructed types 263
enums 264
exceptions 258
forward declaration 268
inheritance 259
interfaces 255
modules 258
oneway operations 257
operations 256
orb.idl 269
scoped names 268
sequences 265
strings 266
structs 263
typedefs 267
unions 264
OMG IDL preprocessor 268
OMG IDL template types
sequences 265
strings 266
operations 256
mapping from CORBA to Automation 277
mapping from CORBA to COM 313
oneway 257
ORB
interface to 66
obtaining reference to 66
orb.idl 269
Orbix
interface to 66
runtime 147
Orbix object name
specifying 69
output handlers
activating 212, 244
deactivating 212, 245
Output() 214, 248

P

parameter to GetObject() 69
PingDuringBind() 210, 242
PlaceCVHandlerAfter() 212, 245
PlaceCVHandlerBefore() 212, 245
pointers
mapping from COM to CORBA 342
preprocessor 268
priming the type store 163
properties
mapping from Automation to CORBA 302
mapping from COM to CORBA 336
of exceptions 112
put_CORBAAnyData() 226
put_Host() 238
put_Marker() 238
put_value() 226
putit 54, 63
putit command 108

Q

qualified names 268

R

raising an exception in a server 120
rebuilding the type store 167
ReclaimCallbackStore() 211
references
See object references 193
registering a type library 35, 369
registering CORBA servers 54, 63, 108
registering handler DLLs 153
registering objects 105
ReinitialiseConfig() 215, 248
ReleaseCORBAView() 217, 249
replacing DCOM servers with CORBA servers 177, 368
ReSizeObjectTable() 209, 243
ResolveInitialReference() 221, 254
running a server 108

runtime
application 147
language 147
Orbix 147
OrbixCOMet 149

S

safearrays
mapping from Automation to CORBA 304
scoped names 268
mapping from COM to CORBA 347
mapping from CORBA to Automation 297
mapping from CORBA to COM 330
scoped_name() 202
sequences 265
mapping from CORBA to Automation 287
mapping from CORBA to COM 320
servers
activating 107, 182, 224
collocation 214, 246
deactivating 107, 183, 224
implementing for client callbacks 128
implementing in CORBA 99–108
registering 54, 63
SetConfigValue() 214, 247
SetDiagnostics() 216, 248
SetItem() 184
SetObjectImpl() 183, 225
SetObjectImplPersistent() 183, 225
SetPrivateKeyPassword() 219, 251
SetSecurityName() 219, 251
ShutDown() 216, 249
specifying the Orbix object name 69
srvAlias 177, 368
SSL
enabling 134
handler DLLs 135
StartUp() 216, 249
stringified object references 220, 252
strings 266
mapping from Automation to CORBA 301
mapping from COM to CORBA 335
mapping from CORBA to Automation 273
mapping from CORBA to COM 310
StringToObject() 220, 254
structs 263
creating 191, 317
mapping from COM to CORBA 339
mapping from CORBA to Automation 283
mapping from CORBA to COM 317
stub code
generating 176
surrogateSee custsur
system exception properties 112
system exceptions
defined by CORBA 351
mapping from CORBA to Automation 292
mapping from CORBA to COM 323
Orbix-specific 352

T

tag field 264
template types
sequences 265
strings 266
timeouts
for remote calls 214, 246
tlibreg 35
toolsSee commands
ts2idl 170
options 365
ts2sp 174
options 367
ts2tlb 173
options 366
type 8
type libraries
creating from type store 34, 171
type store
adding new information to 161
caching mechanism 159
central role of 158
creating IDL files from 168
deleting contents of 166
dumping contents of 167
introduction to 8
rebuilding 167
type_name() 202
typedefs 267
mapping from Automation to CORBA 307
mapping from COM to CORBA 348
mapping from CORBA to Automation 298
mapping from CORBA to COM 331
typeman 163
options 363

U

Union_d() 199
unions 264

creating 191, 317
discriminated 264
mapping from COM to CORBA 340
mapping from CORBA to Automation 285
mapping from CORBA to COM 319
unique_id() 202
usage models 11–17
Automation client to CORBA server 12
COM client to CORBA server 14
CORBA client to COM/Automation server 16
user exceptions
mapping from CORBA to Automation 291
mapping from CORBA to COM 322
UseTransientPort() 217, 249
using OrbixCOMet with Internet Explorer 31
utilitiesSee commands
utility options 363
UUIDs, generating 35

V

value() 186
variants
mapping from Automation to CORBA 305
mapping from COM to CORBA 345
views 5
obtaining reference to in Automation 46
obtaining reference to in COM 56

W

writing a client 94, 125