IONA®

Orbix®

COMet Programmer's Guide and Reference

Version 6.2, December 2004

Making Software Work Together™

While the information in this publication is believed to be accurate, IONA Technologies PLC makes no warranty of any kind to this material including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. IONA Technologies PLC shall not be liable for errors contained herein, or for incidental or consequential damages in connection with the furnishing, performance or use of this material.

Updated: 04-Jul-2005

Contents

Part 1 Introduction

Part 2 Programmer’s Guide

Part 3 Programmer's Reference

List of Figures

List of Tables

Preface

COMet combines the best of both the object management group (OMG) common object request broker architecture (CORBA) and Microsoft component object model (COM) standards. It provides a high performance dynamic bridge, which enables transparent communication between COM clients and CORBA servers.

COMet is designed to allow COM programmers—who use tools such as Visual C++, Visual Basic, PowerBuilder, Delphi, or Active Server Pages on the Windows desktop—to easily access CORBA applications running in Windows, UNIX, or OS/390 environments. It means that COM programmers can use the tools familar to them to build heterogeneous systems that use both COM and CORBA components within a COM environment.

The interworking model and mapping standards described in this guide are based on chapters 17, 18, and 19 of the OMG *Common Object Request Broker: Architecture and Specification* at `ftp://ftp.omg.org/pub/docs/formal/01-12-35.pdf`.

If you need help with this or any other IONA products, contact IONA at `support@iona.com`. Comments on IONA documentation can be sent to `docs-support@iona.com`.

Audience

This guide is intended for COM application programmers who want to use COMet to develop and deploy distributed applications that combine CORBA and COM components within a COM environment. This guide assumes that the reader already has a working knowledge of COM-based and Automation-based tools, such as Visual Basic, PowerBuilder, and Visual C++. (See "COM Overview" on page 9 for a distinction between COM and Automation.)

Organization of this guide

This guide is divided as follows:

Part 1 "Introduction"

This part first provides an introductory overview of the main principles of both COM and CORBA. It then provides an introduction to COMet and an overview of the various ways you can use it in a distributed system.

Part 2 "Programmer's Guide"

This part describes how to:

- Use COMet to develop COM and Automation clients that can communicate with a CORBA server.
- Implement exception handling and client callbacks in your COMet applications.
- Deploy a distributed COMet application.
- Use the various development utilities that are supplied with COMet.

Part 3 "Programmer's Reference"

This part describes:

- The application programming interfaces (APIs) supplied with COMet.
- The semantics of CORBA IDL for defining interfaces to CORBA applications.
- The rules for mapping CORBA IDL types to COM and Automation.
- The configuration variables associated with COMet.
- The arguments available with each COMet utility.

Related reading

The following related reading material is recommended:

- The Common Object Request Broker: Architecture and Specification at ftp://ftp.omg.org/pub/docs/formal/01-12-35.pdf.
- COM-CORBA Interoperability, Ronan Geraghty et al., (Prentice Hall, 1999).

Additional resources

The IONA knowledge base contains helpful articles, written by IONA experts, about the Orbix and other products. You can access the knowledge base at the following location:

`http://www.iona.com/support/knowledge_base/`

The IONA update center contains the latest releases and patches for IONA products:

`http://www.iona.com/support/updates/`

Typographical 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 path names 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.

Keying conventions

This guide may use the following keying conventions:

No prompt
When a command's format is the same for multiple platforms, a prompt is not 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, Windows 95, or Windows 98 command prompt.

```
. . .
.
.
.
```
Horizontal or vertical ellipses in format and syntax descriptions indicate that material has been eliminated to simplify a discussion.

[ ]
Brackets enclose optional items in format and syntax descriptions.

{ }
Braces enclose a list from which you must choose an item in format and syntax descriptions.

|
A vertical bar separates items in a list of choices enclosed in { } (braces) in format and syntax descriptions.

Part 1

Introduction

In This Part

This part contains the following chapters:

CHAPTER 1

COM and CORBA Principles

COMet is an implementation of the Object Management Group (OMG) Interworking Architecture specification for allowing component object model (COM) clients to communicate with common object request broker architecture (CORBA) servers.[1] *Both CORBA and COM are standards for distributed object technology. This chapter provides an introductory overview of the main principles of both COM and CORBA.*

In This Chapter

This chapter discusses the following topics:

Note: A more in-depth study of COM and CORBA is outside the scope of this guide.

[1] The Interworking Architecture specification is part of the CORBA Specification available at `ftp://ftp.omg.org/pub/docs/formal/01-12-35.pdf`. COMet is not a full implementation of the Interworking Architecture specification, because it does not also allow CORBA clients to communicate with COM servers.

Main Similarities and Differences

Overview

This section outlines the main similarities and differences between COM and CORBA. The following topics are discussed:

- "Similarities" on page 4.
- "Differences" on page 4.

Similarities

COM and CORBA share the following principles:

- The system architecture is based around the concept of objects.
- An object is a discrete unit of functionality.
- An object exposes its behavior through a set of well defined interfaces.
- The details of an object's implementation are hidden from the clients that want to make requests on it.

Differences

Table 1 summarizes the main differences between COM and CORBA.

Table 1: *Main Differences between COM and CORBA (Sheet 1 of 2)*

COM	CORBA
An object is typically a subcomponent of an application that represents a point of exposure to other components of that application, or to other applications.	An object is an independent component with a related set of behaviors, transparently available to any CORBA client, regardless of where the object or client are implemented in the system.
The domain of an object is typically a single-user, multitasking visual desktop environment, such as Microsoft Windows.	The domain of an object is typically an arbitrarily scalable distributed network.

Table 1: *Main Differences between COM and CORBA (Sheet 2 of 2)*

COM	CORBA
The purpose of COM is to expedite collaboration and information sharing among applications using the same desktop, by allowing a user to manipulate visual elements on the screen.	The purpose of CORBA is to allow independent components of a distributed system to be shared among a wide variety of possibly unrelated applications and objects in that distributed system.

CORBA Overview

Overview

CORBA is a standard for distributed object technology from the OMG. This section provides a brief overview of the fundamental principles of a CORBA object management system. The following topics are discussed:

- "CORBA Objects" on page 6.
- "Object IDs and References" on page 6.
- "CORBA Object Interfaces" on page 6.
- "CORBA Client Requests" on page 7.
- "CORBA Object Lifetime" on page 7.
- "Object Request Broker" on page 7.
- "Multiple Inheritance" on page 8.

CORBA Objects

A CORBA object is a discrete, independent unit of functionality, comprising a related set of behaviors. A particular CORBA object can be described as an entity that exhibits a consistency of interface, behavior (or functionality), and state over its lifetime.

CORBA uses the concept of a portable object adapter (POA), which is used to map abstract CORBA objects to their actual implementations. A CORBA object can be implemented in any programming language that CORBA supports, such as C++ or Java.

Object IDs and References

A CORBA object has both an object ID and an object reference. An object ID identifies an object with respect to a particular POA instance. An object reference contains unique details about an object, including its object ID and POA identifier, which can be used by clients to locate and invoke on that object. See "CORBA Client Requests" on page 7 for more details about the use of object references.

CORBA Object Interfaces

A CORBA object presents itself to its clients through a published interface, defined in OMG interface definition language (IDL). The concept of keeping an object's interface separate from its implementation means that a client can make requests on an object without needing to know how or where that object is implemented.

The IDL interfaces for CORBA objects can be stored (registered) in an interface repository. CORBA identifies an interface by means of an interface repository ID. Even if you update a particular interface in some way, its repository ID can remain the same.

CORBA Client Requests

In CORBA, a client can access an object's interface and its underlying functionality by making one or more requests on that object. Each client request is made on a specific instance of an object, which is identifiable and contactable via an object reference that is unique to that object instance. An object reference is a name that is used to consistently identify a particular object during that object's lifetime. An object reference in CORBA is roughly equivalent to the concept of an interface pointer in COM.

CORBA client requests can contain parameters consisting of object references or data values that correspond to particular types of data supported by the system. A client request can be dynamically created at runtime (rather than simply being statically defined at compile time) on any object whose interfaces are stored in an interface repository.

CORBA Object Lifetime

The in-memory lifetime of a CORBA object is independent of the lifetime of any clients that hold a reference to it. This means that a client that is no longer running can continue to maintain object references. It also means that a server object can deactivate and remove itself from memory when it becomes idle (although this does consequently mean that the server application must be made to explicitly decide when this should happen).

Object Request Broker

A CORBA system is based on an architectural abstraction called the object request broker (ORB). An ORB allows for:

- Interception and transfer of client requests to servers across the network, and the return of output from the server back to the client.
- Registration of data types and their interfaces, defined in OMG IDL.
- Registration of object instance identities, from which the ORB can construct appropriate object references for use by clients that want to make requests on those object instances.
- Location (and activation, if necessary) of objects.

Orbix is IONA's implementation of an ORB.

Figure 1 provides an overview of the role of the ORB in CORBA client-server communication.

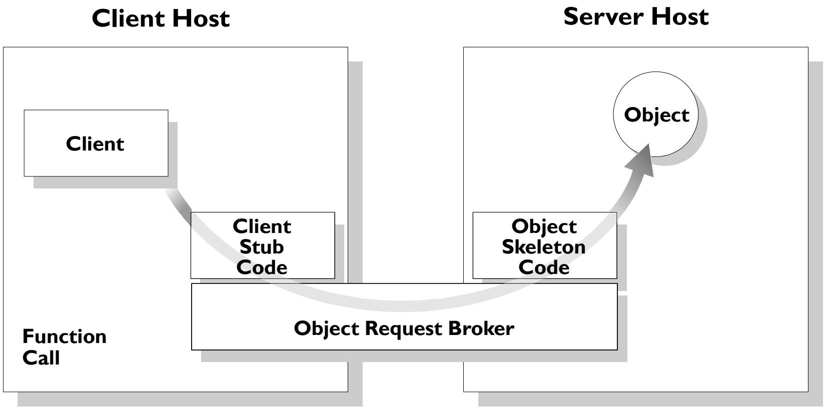


Figure 1: *Role of the ORB in Client-Server Communication*

Multiple Inheritance

CORBA supports the concept of multiple interface inheritance. This basically means that a CORBA object interface can be extended by making it derive from one or more other interfaces. The derived interface ends up having not only its own defined functionality, but also the functionality of the interface(s) from which it derives. Interfaces can also be evolved dynamically at runtime, by having new interfaces derive from existing interfaces.

A CORBA object reference refers to a CORBA object that exposes a single, most-derived interface in which any and all parent interfaces are joined. CORBA does not support the concept of objects with multiple, disjoint interfaces. See "Introduction to OMG IDL" on page 269 for more details of multiple inheritance.

COM Overview

Overview

For the purposes of clarity, this overview of COM is divided into two subsections. The first provides an overview of COM itself, and the second provides an overview of Automation, which is an extension of COM.

In This Section

This section discusses the following topics:

COM

Overview

COM is a standard for distributed object technology from Microsoft Corporation. This subsection provides a brief overview of the fundamental principles of a COM object management system. The following topics are discussed:

- "Background" on page 10.
- "COM Objects" on page 10.
- "COM Class" on page 10.
- "COM Object Interfaces" on page 11.
- "COM Client Requests" on page 11.
- "COM Object Lifetime" on page 11.
- "Multiple Inheritance" on page 12.

Background

COM is an object programming standard that evolved from the object linking and embedding (OLE) standard, which specifies how an object created with one end-user application could be linked or embedded within another end-user application (for example, an Excel spreadsheet within a Word document). This subsection provides a brief overview of the fundamental principles of a COM object management system.

COM Objects

A COM object is typically a subcomponent of an application, representing a point of exposure to other components of the same application, or to other applications. A particular COM object can be described as an active instance of an implementation; an instance in this case can be described as an entity whose interface (or one of whose interfaces) is returned by calling the COM `IClassFactory::CreateInstance` method.

COM Class

COM supports an implementation typing mechanism that is centered around the concept of a COM class. A COM class implements an interface and has a well-defined identity. Implementations are identified by class IDs. An implementation repository, called the Windows system registry, maps implementations to specific units of executable code that embody their actual code realizations. A single instance of a COM class can be registered in COM's active object registry. The only inherently available reference for a COM instance is its `Unknown` pointer.

The identity and management of object state are generally kept separate from the identity and lifecycle of COM class instances. For example, files that contain the state of a document object are persistent. A single COM instance of a document type could load, manipulate, and store several different document files over its lifetime; similarly, multiple COM instances of different object types could load and use the the same file.

COM Object Interfaces

A COM object exposes its interfaces in a virtual function table (also called a vtable), which contains entries corresponding to each operation defined in an interface. COM interfaces are usually described in Microsoft interface definition language (IDL). COM identifies an interface by means of a COM interface ID (IID). If you update a COM interface in some way, it is normal practice to use a different IID for the updated interface.

COM Client Requests

In COM, a client can make a request on an object if it has both compile-time knowledge of the object's interface structure and a reference to an instance offering that interface. A COM client can call the COM `GetActiveObject` function to obtain an `IUnknown` pointer for an active object.

A COM client can use a COM interface pointer to make requests on an object. Interface pointers in COM are roughly equivalent to the concept of object references in CORBA. COM interfaces cannot be invoked by a client that does not have compile-time knowledge of them.

COM Object Lifetime

The in-memory lifetime of a COM object is linked to the lifetime of the clients that hold a reference to it. This means that the object is destroyed when no more clients are attached to it. This can lead to problems, however, if a client crashes without releasing its references to the object. To avoid this, COM provides support for clients to ping servers, so that if a client ping is not received within a designated timeframe, the references it held can then be released.

As an alternative to having clients ping servers, an alternative form of binding can be used in COM, through the use of monikers (that is, persistent interface references). Monikers are conceptually equivalent to CORBA object references. Although the use of monikers can help in determining when deactivation should occur, it does, however, mean that a COM client must be explicitly set up to use this alternative form of binding, to allow the server to release its references if necessary.

Multiple Inheritance

Unlike CORBA, COM does not support the concept of multiple interface inheritance. This has consequences for the way in which multiply-inherited CORBA interfaces are mapped to COM—see "Mapping for Interface Inheritance" on page 368 for more details. You can use the COM `QueryInterface()` method to find out and explore the interfaces that a particular COM object supports.

Automation

Overview

This subsection provides a brief overview of the fundamental principles of Automation. The following topics are discussed:

- "Extension of COM" on page 13.
- "Automation Object Interfaces" on page 13.
- "Automation Client Requests" on page 13.
- "Dual Interfaces" on page 14.
- "Automation Object Lifetime" on page 14.
- "Multiple Inheritance" on page 14.
- "Summary of Differences between COM and Automation" on page 14.

Extension of COM

Automation is an extension of COM and is implemented through it. Automation provides a mechanism for dynamic operation invocation at runtime (unlike a pure COM call that relies on static information known at compile time). However, the data types that Automation supports are only a subset of the types supported by COM (for example, Automation does not support complex, user-defined constructed types, such as structs or unions). Microsoft Excel is an example of a typical Automation application.

Automation Object Interfaces

Automation interfaces can be described in Microsoft object definition language (ODL). Automation interfaces can be registered in a binary type library, which allows for runtime checking of client requests.

Automation Client Requests

Unlike COM interfaces, Automation interfaces can be invoked dynamically at runtime, through a special COM interface, called `IDispatch`. This is also known as *late binding*. An Automation client can use the Automation `GetObject` function (equivalent to the COM `GetActiveObject` function) to obtain an `IUnknown` pointer for an active object in COM's active object registry.

Dual Interfaces

Some Automation controllers (for example, Visual Basic) provide the option of using either straight `IDispatch` interfaces or *dual interfaces* for invoking on a server. An Automation dual interface is a COM vtable-based interface that derives from the `IDispatch` interface. It is therefore a hybrid form of interface, which supports both an Automation and a COM-like interface.

The use of dual interfaces means that client invocations can be routed directly through the vtable. This is known as *early binding*, because interfaces are known at compile time. One advantage to early binding is that it removes the performance overhead associated with late binding at runtime.

Automation Object Lifetime

As for COM objects, the in-memory lifetime of an Automation object is linked to the lifetime of the clients that hold a reference to it. See "COM Object Lifetime" on page 11 for more details.

Multiple Inheritance

Because COM does not support the concept of multiple interface inheritance, neither does Automation. This has consequences for the way in which multiply-inherited CORBA interfaces are mapped to Automation—see "Mapping for Interface Inheritance" on page 325 for more details.

Automation objects typically provide all Automation operations in a single `IDispatch` interface, in a flat format. In an Automation controller that provides the option of using dual interfaces, you can use dual interfaces to expose multiple `IDispatch` interfaces for a particular COM co-class. For example, a `Dim X as new Y` statement in Visual Basic can be used to invoke a `QueryInterface()` on the `Y` interface.

Summary of Differences between COM and Automation

The following is a summary of the main differences between COM and Automation interfaces:

Table 2: *Differences between COM and Automation Interfaces*

COM Interfaces	Automation Interfaces
Support a full range of COM types, including user-defined constructed types such as unions or structs.	Support only a subset of COM types. Automation does not, for example, support user-defined constructed types.

Table 2: *Differences between COM and Automation Interfaces*

COM Interfaces	Automation Interfaces
Can only be invoked by clients with compile-time knowledge of them.	Can be invoked at runtime (if required) through a special COM interface, called `IDispatch`.
Define methods only.	Define both properties and methods.

Note: The interface syntax and semantics for COM and Automation are not the same. The OMG therefore presents separate sets of rules for mapping CORBA types to COM and for mapping CORBA types to Automation. See “Mapping CORBA to COM” on page 357 and “Mapping CORBA to Automation” on page 313 for more details of these rules.

CHAPTER 2

Introduction to COMet

COMet enables transparent communication between clients that are running in a Microsoft COM environment and servers that are running in a CORBA environment. This chapter introduces COMet, first by outlining the concepts of the standard interworking model on which it is based, and then by describing how COMet implements these concepts.

In This Chapter

This chapter discusses the following topics:

Note: COMet supports development and deployment of COM or Automation clients that can communicate with CORBA servers. Any CORBA C++ server examples provided in this guide are supplied for reference purposes only. It is assumed that you already have a CORBA server implementation product. The examples provided are for use with Orbix 6.1.

The Interworking Model

Overview

This section describes the principles of the interworking model on which COMet is based. The following topics are discussed:

- "Interworking Architecture Specification" on page 18.
- "Overview of Interworking Model" on page 18.
- "Bridge" on page 19.
- "Bridge View of Target Object" on page 19.

Interworking Architecture Specification

The *Interworking Architecture* specification, which is part of the OMG *Common Object Request Broker: Architecture and Specification* at `ftp://ftp.omg.org/pub/docs/formal/01-12-35.pdf`, defines the standard interworking model that specifies how the integration between COM or Automation clients and CORBA object models is achieved.

Overview of Interworking Model

Figure 2 provides an overview of the interworking model, which involves a client in one object system (in this case, COM or Automation) that wants to send a request to an object in another object system (in this case, CORBA).

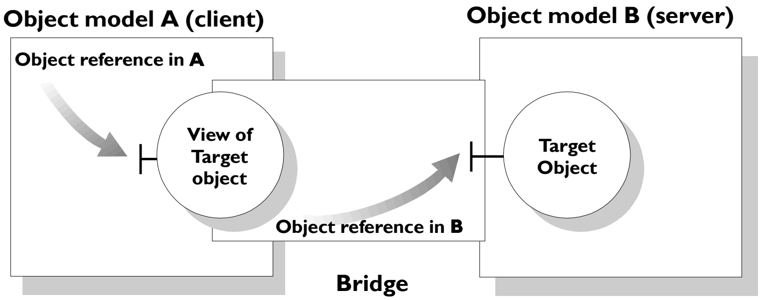


Figure 2: *The Standard Interworking Model*

Bridge

The interworking model shown in Figure 2 on page 18 provides a *bridge* that acts as an intermediary between the two object systems. The bridge provides the mappings that are required between the object systems. It provides these mappings transparently, so that the client can make requests in its familiar object model.

Bridge View of Target Object

To effect 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* foreign object in the model that is understood by the client. See Figure 4 on page 22 for an overview of how the view object is implemented in COMet.

The client makes requests on the view object's interface in the bridge. The bridge then maps these requests into requests on the target object's interface, and forwards them to the target object across the system boundary. The workings of the bridge are transparent to the client, so the client does not have to know that the objects it is using belong to another object system.

The bridge can consist of multiple view objects. Each view object in the bridge is bound to an Orbix object reference that corresponds to a real target object across the system boundary. See Figure 4 on page 22 for more details.

How COMet Implements the Model

Overview

This section describes how COMet implements the interworking model. The following topics are discussed:

- "Role of COMet" on page 20.
- "Graphical Overview of Role" on page 21.
- "COM View of CORBA Objects" on page 21.
- "Graphical Overview of View" on page 22.
- "Creating a View" on page 22.
- "Advantages for the COM Programmer" on page 23.
- "Supported Protocols" on page 23.

Role of COMet

COMet supports application integration across network boundaries, different operating systems, and different programming languages. It provides a high performance dynamic bridge that enables integration between COM or Automation and CORBA objects. It allows you to develop and deploy COM or Automation client applications that can interact with existing CORBA server applications that might be running on Windows or another platform.

Graphical Overview of Role

Figure 3 provides a conceptual overview of how COMet implements the interworking model.

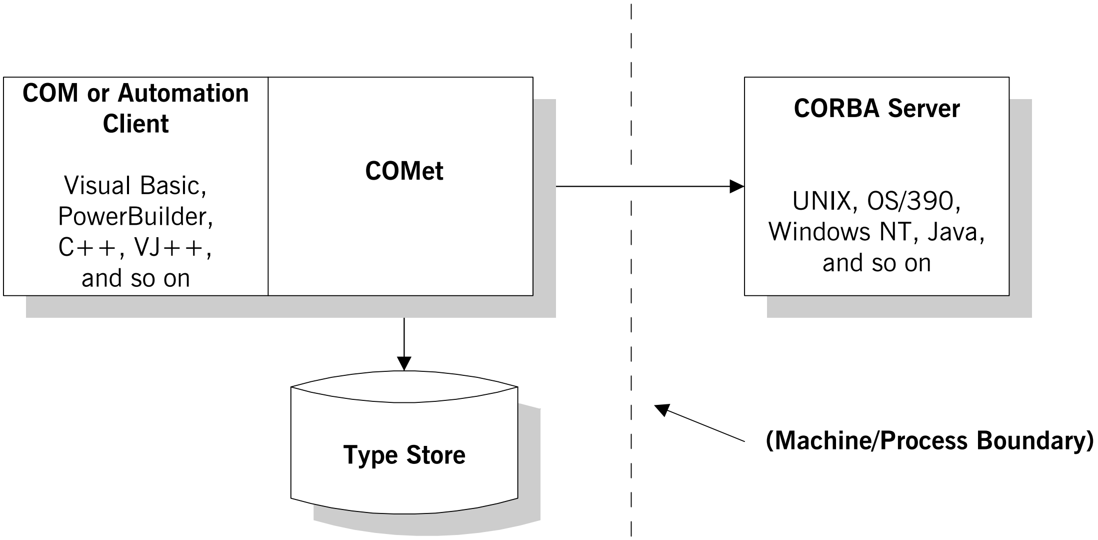

COM or Automation Client
Visual Basic, PowerBuilder, C++, VJ++, and so on
COMet
CORBA Server
UNIX, OS/390, Windows NT, Java, and so on
Type Store
(Machine/Process Boundary)

Figure 3: *COMet's Implementation of the Interworking Model*

Figure 3 shows no process boundary between the client and COMet, which is the only supported scenario for COM clients. In the case of Automation clients, however, you can choose to have a process and machine boundary between the client and COMet, or to have no machine boundary between COMet and the server. See "Usage Models and Bridge Locations" on page 27 for more details.

COM View of CORBA Objects

As explained in "Bridge View of Target Object" on page 19, the interworking model provides the concept of a view object in the bridge, which allows a client to make requests on an object in a foreign object system as if that object were in the client's own native system. It follows that COMet supports the concept of COM or Automation views of CORBA objects.

This in turn means that a corresponding COM or Automation interface must be generated for each CORBA interface that is implemented by the CORBA objects a client wants to invoke. (COMet supplies utilities that allow you to generate such COM or Automation interfaces from CORBA interfaces, and these are described in more detail in "Development Support Tools" on page 171.) At application runtime, a client can create and subsequently invoke on view objects that implement and expose these COM or Automation interfaces (see "Creating a View" on page 22 for more details).

Graphical Overview of View

Figure 4 provides a graphical overview of how a view object is implemented in COMet.

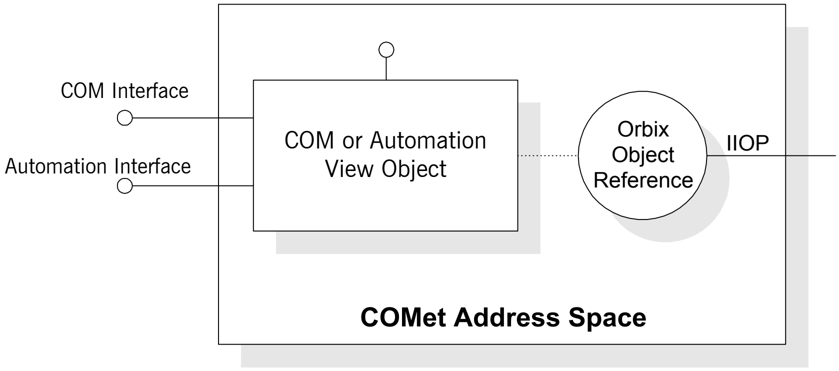


Figure 4: *View Object in COMet*

Creating a View

A view object is created in the COMet bridge when a client calls the COMet-supplied (D)`ICORBAFactory::GetObject()` method on a particular CORBA object. As shown in Figure 4 on page 22, a view exposes COM or Automation interfaces, which correspond to the CORBA interfaces on the object that the client wants to invoke. The view object is automatically bound on creation to an Orbix object reference for the target object. This object reference is returned to the client, to allow it to invoke operations on the target object. See Part 2 “Programmer’s Guide” and “COMet API Reference” on page 217 for more details of how to use `DICORBAFActory::GetObject()`.

Note: All COM views that are mapped from a particular OMG IDL interface must share the same COM IIDs. See “Mapping Interface Identifiers” on page 362 for more details.

Advantages for the COM Programmer

COMet provides two main advantages to COM programmers:

1. COMet provides access to existing CORBA servers, which can be implemented 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 different programming languages, including C++ and Java.
2. Using COMet, 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. COMet, therefore, presents a programming model that is familiar to the COM programmer.

Supported Protocols

COMet supports both the internet inter-ORB protocol (IIOP) and Microsoft’s distributed component object model (DCOM) protocol. This means that any IIOP-compliant ORB can interact with a COMet application.

Note: There are some restrictions in the use of DCOM with COMet. These are explained in more detail in “Usage Models and Bridge Locations” on page 27. The recommended approach is to load the bridge in-process to the client (that is, in the client’s address space) and hence allow the client machine to use IIOP to communicate with the server.

COMet System Components

Overview

This section describes the various components that comprise a COMet system. The following topics are discussed:

- "Bridge" on page 24.
- "Type Store" on page 24.
- "Automation Client" on page 24.
- "COM Client" on page 25.
- "COM Library" on page 25.
- "CORBA Server" on page 25.

Bridge

The bridge is a synonym for COMet itself. It is implemented as a set of DLLs that are capable of dynamically mapping requests from a COM or Automation environment to a CORBA environment. The bridge provides the mappings and performs the necessary translation between COM or Automation and CORBA types.

As shown in Figure 4 on page 22, a view object in the bridge contains both a COM/Automation object interface and an Orbix object interface. This means that the bridge can expose an appropriate COM or Automation interface to its clients.

Type Store

As shown in Figure 3 on page 21, COMet uses a component called the type store. The type store is used to hold a cache of information about all the CORBA types in your system. COMet can retrieve this information from the Interface Repository at application runtime, and then automatically update the type store with this information for subsequent use, instead of having to query the Interface Repository for it again. The type store holds its cache of type information in a neutral binary format. See "Development Support Tools" on page 171 for more details about the workings of the type store.

Automation Client

An Automation client can use COMet to communicate with a CORBA server. This is a regular Automation client written in a language such as Visual Basic, PowerBuilder, or any other Automation-compatible language.

COM Client

A COM client can use COMet to communicate with a CORBA server. This is a pure COM client (that is, not an Automation-based 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 COMet. 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 COMet bridge in your system, the CORBA server might need to be running on Windows (if so, preferably Windows 2000, for reasons of scalability). See "Usage Models and Bridge Locations" on page 27 for more details.

CHAPTER 3

Usage Models and Bridge Locations

You can use COMet to develop and deploy distributed applications consisting of COM or Automation clients that can call objects in a CORBA server. This chapter explains how COMet supports this usage model for both COM and Automation.

In This Chapter

This chapter discusses the following topics:

Note: See "Deploying a COMet Application" on page 151 for more details and examples of the various ways you can use COMet when deploying your applications.

Automation Client to CORBA Server

Overview

This section describes a usage model involving an Automation client and a CORBA server. The following topics are discussed:

- "Graphical Overview" on page 28.
- "Automation Client" on page 28.
- "Automation Client with Bridge In-Process" on page 29.
- "Automation Client with Bridge Out-of-Process" on page 29.
- "CORBA Server" on page 29.
- "Bridge" on page 30.

Graphical Overview

Figure 5 shows a graphical overview of this usage model.

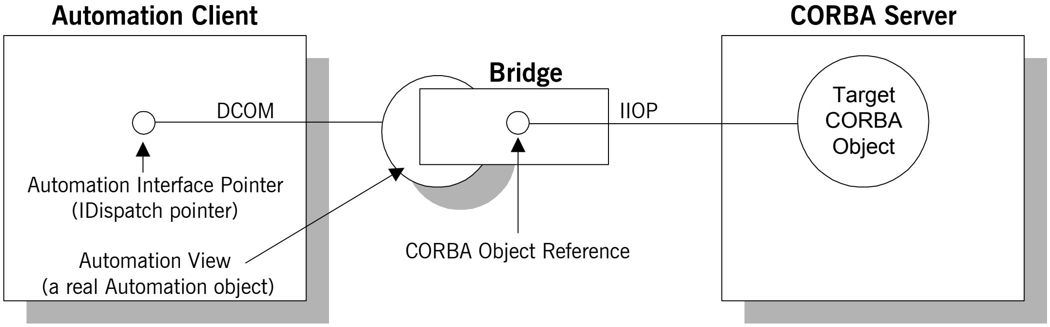

Automation Client
CORBA Server
Bridge
DCOM
IIOP
Target CORBA Object
Automation Interface Pointer (IDispatch pointer)
Automation View (a real Automation object)
CORBA Object Reference

Figure 5: *Automation Client to CORBA Server*

Automation Client

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

An Automation client can have the bridge loaded in any of the following ways:

- In-process (that is, in the client's address space).

- Out-of-process on the client machine.
- Out-of-process on a separate machine.

Automation Client with Bridge In-Process

The recommended deployment scenario for an Automation client with COMet is to load the bridge in-process (that is, in the client's address space). This involves the use of IIOP as the wire protocol for communication between the Automation client machine and CORBA server.

When the bridge is loaded in-process, an Automation client can use dual interfaces instead of `IDispatch` interfaces. COMet does not support the use of dual interfaces when the bridge is loaded out-of-process. The use of either dual interfaces or `IDispatch` interfaces determines whether early binding or late binding is allowed. (See "Automation Client Requests" on page 13 and "Dual Interfaces" on page 14 for a definition of early and late binding.)

Automation Client with Bridge Out-of-Process

Figure 5 on page 28 shows a scenario where the Automation client is using DCOM to communicate with the bridge, which means the bridge is loaded out-of-process on a separate machine. Although this is a supported deployment scenario for Automation clients, it is not recommended unless the bridge machine is running on Windows 2000, because it otherwise limits the number of clients that can be handled.

Note: If you want to load the bridge out-of-process, your Automation client must use `IDispatch` interfaces instead of dual interfaces.

As shown in Figure 5 on page 28, the Automation client uses an `IDispatch` pointer to make method calls on an Automation view object in the bridge. The bridge uses a CORBA object reference to make a corresponding operation call on the target object in the CORBA server.

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

CORBA Server

The CORBA server presents an OMG IDL interface to its objects. The server application can exist on platforms other than Windows. However, if you choose to locate the bridge on the server machine, the server must be running on Windows (preferably Windows 2000 for reasons of scalability). It can be written in any language supported by a CORBA implementation, such as C++ or Java.

Bridge

The bridge can be located on the Automation client machine, on an intermediary machine, or on the CORBA server machine. If the bridge is not located on the client machine, the bridge machine must be running on Windows (preferably Windows 2000 for reasons of scalability).

The bridge 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, the Automation client uses DCOM to communicate with it. The bridge uses IIOP to communicate with the CORBA server.

COM Client to CORBA Server

Overview

This section describes a usage model involving a COM client and a CORBA server. The following topics are discussed:

- "Graphical Overview" on page 31.
- "COM Client" on page 31.
- "CORBA Server" on page 32.
- "Bridge" on page 32

Graphical Overview

Figure 6 shows a graphical overview of this usage model.

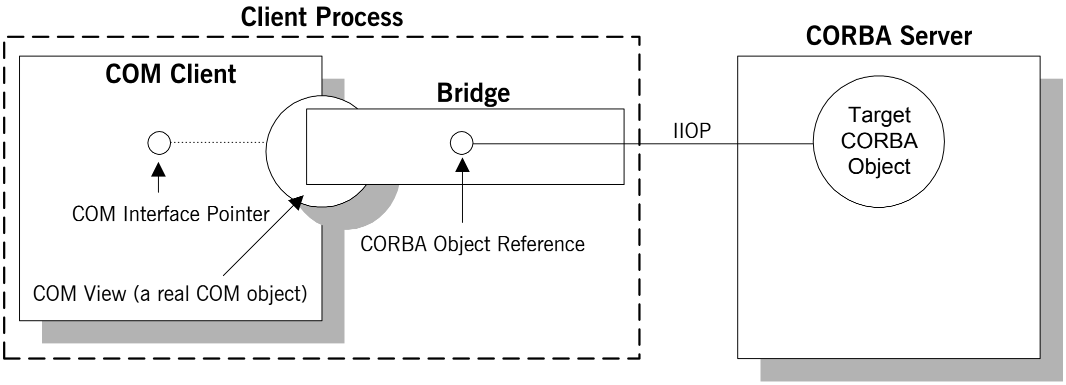


Figure 6: *COM Client to CORBA Server*

COM Client

The only supported deployment scenario for a COM client with COMet is to load the bridge in-process (that is, in the client's address space). This involves the use of IIOP as the wire protocol for communication between the COM client machine and CORBA server. Figure 6 provides a graphical overview of this scenario.

The COM client can use a COM interface pointer to make method calls on a COM view object in the bridge. The bridge uses a CORBA object reference to make a corresponding operation call on the target object in the CORBA server.

The dynamic marshalling engine of COMet 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.

CORBA Server

The CORBA server presents an OMG IDL interface to its objects. The server application can exist on platforms other than Windows. It can be written in any language supported by a CORBA implementation, such as C++ or Java.

Bridge

The bridge must be located in-process to the COM client. The bridge 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.

Part 2

Programmer's Guide

In This Part

This part contains the following chapters:

CHAPTER 4

Getting Started

This chapter is provided as a means to getting started quickly in application programming with COMet. It explains the basics you need to know to develop a simple COMet application that consists of a COM or Automation client, written in PowerBuilder, Visual Basic, or COM C++, which can call objects in an existing CORBA C++ server.

In This Chapter

This chapter discusses the following topics:

Prerequisites

Overview

This section describes the prerequisites to starting application development with COMet. The following topics are discussed:

- "Client-Side Requirements" on page 36.
- "Server-Side Requirements" on page 36.
- "Registering OMG IDL Type Information" on page 36.
- "Priming the Type Store" on page 37.

Client-Side Requirements

Ensure that both Orbix and COMet are installed and configured correctly. See the Orbix 6.1 *Installation Guide* for more details about installation. See the Orbix 6.1 *Deployment Guide* and *Configuration Reference* for details about configuring both Orbix and COMet.

Server-Side Requirements

COMet requires no changes to existing CORBA servers. See the Orbix documentation set for details of how to manage servers. This chapter assumes that you are using Orbix as your server-side object request broker (ORB).

Registering OMG IDL Type Information

As explained in "How COMet Implements the Model" on page 20, COMet is a fully dynamic bridge that enables integration between COM or Automation clients and CORBA servers. The bridge is driven by OMG IDL type information derived from a CORBA Interface Repository.

Before you run an application, ensure that your OMG IDL is registered in the Interface Repository. This is because COMet is designed to automatically retrieve the required type information from the Interface Repository at application runtime. COMet then saves this information to the type store for subsequent use. See the Orbix documentation set for details of how to register OMG IDL.

Priming the Type Store

As an alternative to having COMet retrieve the type information from the Interface Repository at application runtime, you can manually configure the type store with the required type information before the first run of an application. This is also known as priming the cache and is described in more detail in "Priming the COMet Type Store Cache" on page 78. This also requires that the OMG IDL is registered in the Interface Repository.

Developing Automation Clients

Overview

You can use COMet to develop Automation client applications, using any Automation-based tool. This section describes how to use COMet to develop Automation clients in Visual Basic and PowerBuilder.

In This Section

This section discusses the following topics:

Introduction

Overview

This subsection provides an introduction to the Automation client demonstrations provided. The following topics are discussed:

- "The Grid Demonstration" on page 39.
- "OMG IDL grid Interface" on page 39.
- "Automation DIgrid Interface" on page 40.
- "Visual Basic Client GUI Interface" on page 41.
- "Location of Visual Basic Source Files" on page 41.
- "PowerBuilder Client GUI Interface" on page 42.
- "Location of PowerBuilder Source Files" on page 42.

The Grid Demonstration

The examples developed in this section are Automation clients, written in Visual Basic and PowerBuilder, which can access and modify values that are assigned to cells within a grid that is implemented as an object in a supplied CORBA server.

OMG IDL grid Interface

The `grid` object in the CORBA server implements the following OMG IDL `grid` interface:

```
// OMG IDL
interface grid {
     readonly attribute short height;
     readonly attribute short width;
     void set(in short n, in short m, in long value);
     long get(in short n, in short m);
};
```

Automation DIgrid Interface

The corresponding Automation interface for the preceding OMG IDL interface is called `DIgrid`, and is defined as follows:

```
[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);
};
```

The Automation view of the target CORBA object must implement the `DIgrid` interface.

Visual Basic Client GUI Interface

Figure 7 shows the Visual Basic client GUI interface implemented in this section.

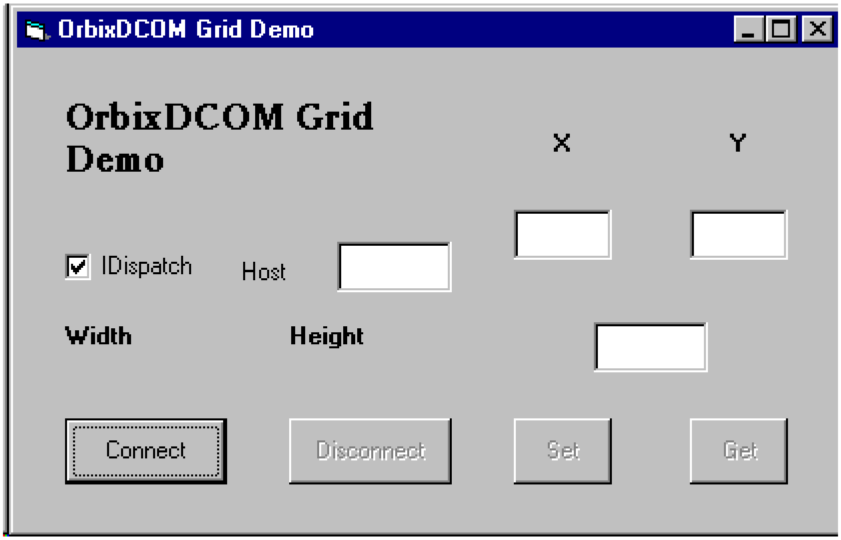


Figure 7: *Visual Basic Client GUI for the COMet Grid Demonstration*

Location of Visual Basic Source Files

The source for the Visual Basic demonstration is in *`install-dir`*`\demos\comet\grid\vb_client`, where *`install-dir`* represents the Orbix installation directory.

PowerBuilder Client GUI Interface

Figure 8 shows the PowerBuilder client GUI interface implemented in this section.

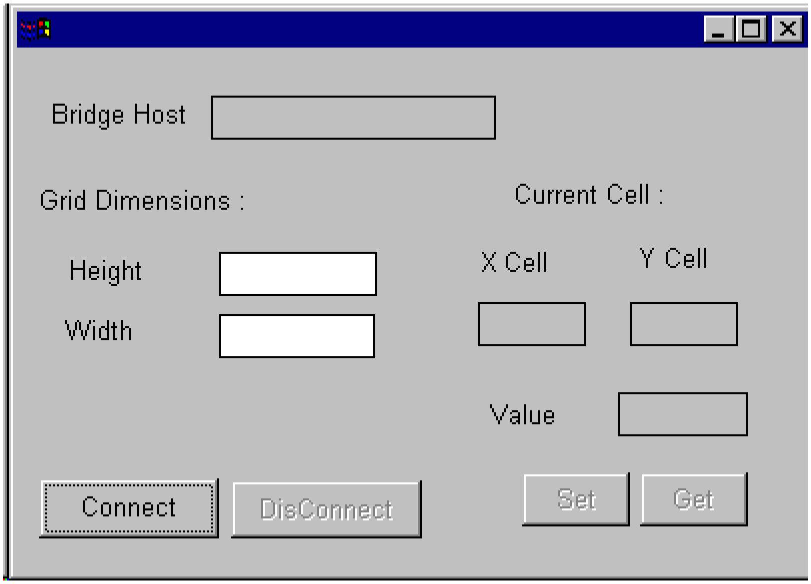


Figure 8: *PowerBuilder Client GUI for the COMet Grid Demonstration*

Location of PowerBuilder Source Files

The source for the PowerBuilder demonstration is in `install-dir\demos\comet\grid\pb_client`, where *`install-dir`* represents the Orbix installation directory.

Using the Visual Basic Genie

Overview

This subsection provides an introduction to using the supplied Visual Basic genie for development of Automation clients. The following topics are discussed:

- "Visual Basic Genie" on page 43.
- "C++ Genie" on page 43.
- "Overview of Client Development Process" on page 44.
- "Explanation of Client Development Process" on page 44.
- "Development Steps Using Code Generation" on page 45.
- "Files Generated by the Visual Basic Genie" on page 45.

Visual Basic Genie

COMet is shipped with a Visual Basic code generation genie that can automatically generate the bulk of the application code for a Visual Basic client, based on OMG IDL definitions. Both a GUI and command-line version of the genie are supplied. The use of the Visual Basic genie is not compulsory for creating Visual Basic clients, using COMet. However, using the genie makes the development of Visual Basic clients much faster and easier.

C++ Genie

The Visual Basic genie is designed to create Visual Basic clients that can communicate with C++ servers that have been created using the C++ genie supplied with the CORBA Code Generation Toolkit. (See the *CORBA Code Generation Toolkit Guide* for details about the C++ genie.)

Overview of Client Development Process

Figure 9 provides an overview of how the client development process works with the genie.

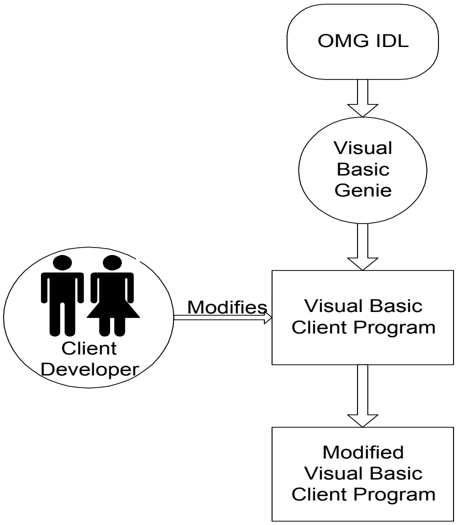

OMG IDL
Visual Basic Genie
Client Developer
Modifies
Visual Basic Client Program
Modified Visual Basic Client Program

Figure 9: *Development Overview Using Code Generation*

Explanation of Client Development Process

Figure 9 on page 44 can be explained as follows:

1. The code generation genie takes the OMG IDL file as input and generates a complete client program. See "Files Generated by the Visual Basic Genie" on page 45 for details of the Visual Basic files that the genie generates.

   Note: The generated client is a dummy implementation that invokes every operation on each interface in the IDL file exactly once. The dummy client is a working application that can be built and run immediately.

2. The client developer can then modify the client to complete the application.

Development Steps Using Code Generation

The main steps to develop a client-server application, using code generation, are as follows:

Step	Action
1	Generate the CORBA server code, using the C++ genie supplied with the CORBA Code Generation Toolkit. See the *CORBA Code Generation Toolkit Guide* for more details.
2	Generate the Visual Basic client, using the Visual Basic genie supplied with COMet. See "Generating Visual Basic Client Code" on page 199 for details of how to use the genie.
3	Register the appropriate OMG IDL file with the Orbix Interface Repository.[a] See the *CORBA Administrator's Guide* for details.
4	Load the `client.vbp` file into the Visual Basic IDE. Then build the client as normal.

[a] You only need to perform this step if you are using the command-line version of the genie. The GUI version of the genie automatically registers the OMG IDL, if it has not already been registered.

Files Generated by the Visual Basic Genie

The Visual Basic genie creates the following files:

`client.vbp`	This is the Visual Basic project file for the client.
`client.frm`	This is the main Visual Basic form for the client.
`FindIOR.frm`	This form contains the functions needed by the client to select a `.ref` file. The `.ref` file is written by the server and contains the server object's IOR.
`Call_Funcs.bas`	This contains Visual Basic code for implementing the operations defined in the selected interface(s).
`Print_Funcs.bas`	This contains functions for printing the values of all the CORBA simple types supported by COMet. It also contains functions for printing any user-defined types declared in the IDL file.
`Random_Funcs.bas`	This contains functions for generating random values for all the CORBA simple types supported by COMet. It also contains functions for generating random values for any user-defined types declared in the IDL file.

`IT_Random.cls` This class is a random number generator that is used in the generated `Random_Funcs.bas` file.

Writing a Visual Basic Client without the Genie

Overview

This subsection describes the steps to use COMet to develop a simple Visual Basic client of a CORBA server, if you are not using the code generation genie. The steps are:

Step	Action
1	Declare global data.
2	Connect to the Orbix `grid` server, and obtain an object reference for the `grid` object.
3	Invoke operations on the `grid` object.
4	Disconnect.

Any filenames mentioned in this subsection refer to files contained in the *`install-dir`*`\demos\comet\grid\vb_client` directory.

Step 1—Declaring Global Data

Start by declaring global variables for the bridge (`bridge`), the CORBA object factory (`fact`), and the Automation view object (`gridDisp`).

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

Step 2—Connecting to Server and Obtaining Object Reference

The following code is executed when you click **Connect** on the Visual Basic client window shown in Figure 7 on page 41:

Example 1:

```
   ' Visual Basic
   Private Sub Connect_Click()

1  Set fact = CreateObject("CORBA.Factory")
2  Set gridDisp = fact.GetObject("grid:" + sIOR)

   width_val.Caption = gridDisp.Width
```

Example 1:

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

The preceding code can be explained as follows:

1. The call to `CreateObject` results in the creation of an instance of a CORBA object factory in the bridge. It is assigned a ProgID, `CORBA.Factory`.
2. After a `CORBA.Factory` object has been returned, the client can call `GetObject()` on the object factory, to request a particular object. The call to `GetObject()` achieves a connection between the client's `gridDisp` object reference (for the view) and the target `grid` object in the server.

   The call to `GetObject()` causes the following:

   i. The object factory creates an Automation view object that implements the `DIgrid` interface.

   ii. The view object is bound to an instance of the CORBA `grid` object named in the parameter for `GetObject()`.

   iii. The `grid` object is mapped onto a CORBA object reference. (This object reference is then bound to the view.)

   iv. A reference to the Automation view is returned to the client.

See "Obtaining a Reference to a CORBA Object" on page 86 and "DICORBAFactory" on page 228 for more details about `GetObject()`.

Step 3—Invoking Operations

After calling `GetObject()`, the client can implement the **Get** and **Set** buttons on the client GUI interface, by using the `gridDisp` object reference to invoke the OMG IDL operations on the `grid` object in the server. For example:

```
…
…gridDisp.set(…)
…
```

Step 4—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
```

Writing a PowerBuilder Client

Overview

This subsection describes the steps to use COMet to develop a simple PowerBuilder client of a CORBA server. The steps are:

Step	Action
1	Declare global data.
2	Connect to the Orbix `grid` serverm and obtain an object reference for the target CORBA `grid` object.
3	Invoke operations on the grid object.
4	Disconnect.

Note: There is no code generation genie available for PowerBuilder.

Any filenames mentioned in this subsection refer to files contained in the `*install-dir*\demos\comet\grid\pb_client` directory.

Step 1—Declaring Global Data

Start by declaring global variables for the bridge (`bridge`), the CORBA object factory (`fact`), and the Automation view object (`grid_client`).

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

Step 2—Connecting to the Orbix Grid Server

The following code is executed when you click **Connect** on the PowerBuilder client window shown in Figure 8 on page 42:

Example 2:

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

1  fact.ConnectToNewObject("CORBA.Factory")
```

Example 2:

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

grid_client = CREATE OleObject
2 grid_client = fact.GetObject("grid:" + sIOR)

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 can be explained as follows:

1. The call to `ConnectToNewObject` results in the creation of an instance of a CORBA object factory in the bridge. It is assigned a ProgID, `CORBA.Factory`.
2. After a `CORBA.Factory` object has been returned, the client can call `GetObject()` on the object factory, to request a particular object. The call to `GetObject()` achieves a connection between the client's `grid_client` object reference (for the view) and the target `grid` object in the server.

   The call to `GetObject()` causes the following:

   i. The object factory creates an Automation view object that implements the `DIgrid` interface.

   ii. The view object is bound to an instance of the CORBA `grid` object named in the parameter for `GetObject()`.

   iii. The `grid` object is mapped onto a CORBA object reference. (This object reference is then bound to the view.)

   iv. A reference to the Automation view is returned to the client.

See "Obtaining a Reference to a CORBA Object" on page 86 and "DICORBAFactory" on page 228 for more details about `GetObject()`.

Step 3—Invoking Operations

After calling `GetObject()`, the client can implement the **Get** and **Set** buttons on the client GUI interface, by using the `grid_client` object reference to invoke the OMG IDL operations on the `grid` object in the server. For example:

```
…
…grid_client.set(…)
…
```

Step 4—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
```

Running the Client

Overview

This subsection describes the steps to run the client application.

Steps

The steps to run the client are:

Step	Action
1	If you are using: • Visual Basic, run `vbgrid.exe`. This opens the client window shown in Figure 7 on page 41. • PowerBuilder, run `grid.exe`. This opens the client window shown in Figure 8 on page 42.
2	Specify the hostname in the appropriate field and click **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	Click **Set** to modify values in the grid, or **Get** to obtain values from the grid.
5	Click **Disconnect** when you are finished.

Using DCOM with COMet

Overview

This section describes how to use COMet to develop Automation clients that launch the COMet bridge out-of-process, and hence use DCOM as the wire protocol for communication.

In This Section

This section discusses the following topics:

Introduction

Overview

This subsection provides an introduction to the concept of launching the bridge out-of-process, and the mandates and recommendations associated with it. The following topics are discussed:

- "In-Process versus Out-of-Process" on page 55.
- "Automation Clients versus COM Clients" on page 55.
- "IDispatch Interfaces" on page 55.
- "Windows 2000" on page 56.

In-Process versus Out-of-Process

The examples provided in "Developing Automation Clients" on page 38 create an instance of the `CORBA.Factory` object in the Automation client's address space, which means the COMet bridge is launched in-process to the client. Launching the bridge in-process is the recommended deployment scenario with COMet, because it involves the use of IIOP as the wire protocol for communication between the client machine and the CORBA server.

Launching the bridge out-of-process involves the use of DCOM as the wire protocol for communication between the client and the COMet bridge. If the bridge is launched out-of-process on the same machine as the client, it is referred to as a local server. If the bridge is launched on a separate machine, it is referred to as a remote server. Launching the bridge out-of-process comes with certain mandates and recommendations, which are described next.

Automation Clients versus COM Clients

COMet only allows Automation clients to launch the bridge out-of-process. It does not support COM clients with the bridge loaded out-of-process. COM clients must launch the bridge in-process.

IDispatch Interfaces

If you want to launch the bridge out-of-process, your Automation clients must use `IDispatch` interfaces. The use of dual interfaces is not supported with DCOM.

Windows 2000

If you want to launch the bridge out-of-process, the bridge machine must be running on Windows. For reasons of scalability, it is recommended that the bridge machine is running on Windows 2000. Running the bridge on any other version of Windows limits the number of clients that it can handle.

Launching the COMet Bridge Out-of-Process

Overview

This subsection describes how to write a client that can launch the bridge out-of-process. The following topics are discussed:

- "Example" on page 57.
- "Explanation" on page 58.
- "Required Setting" on page 58.
- "The custsur.exe Executable" on page 58.
- "The CreateObject() Method" on page 59.

Example

Example 3 shows a sample Visual Basic client that can launch the bridge out-of-process.

Example 3: *Sample Visual Basic Client for Out-of-Process Launching*

```
  ' Visual Basic
  Private Sub ConnectBtn_Click()
  On Error GoTo errortrap
1 If inprocess.Value <> Checked Then
2     set objFactory = CreateObject("CORBA.Factory", HostName.Text)
  Else
      set objFactory = CreateObject("CORBA.Factory")
  End If
  inprocess.Enabled = False
3 Set srvObj = objFactory.GetObject("grid:" + sIOR)
  StartBtn.Enabled = True
  ConnectBtn.Enabled = False
  Exit Sub
  errortrap:
  MsgBox (Err.Description & ", in " & Err.Source)
  End Sub
```

Explanation

The client code shown in Example 3 can be explained as follows:

1. The client implements a check button (inprocess), to let the user decide at runtime whether to launch the bridge in-process or out-of-process. Because the decision is controlled by a simple If…Else statement, both configurations are equally easy to use from the client programmer's point of view.
2. The Visual Basic CreateObject() method allows you to specify a hostname as an optional, extra parameter. The hostname specified is the name of the machine on which you want to launch the bridge. The call to CreateObject() creates an instance of the CORBA.Factory object in the bridge.

   The Visual Basic CreateObject() method is similar to the COM CoCreateInstanceEx() method. Most Automation controllers allow you to specify an optional hostname to their equivalent of the Visual Basic CreateObject() method.
3. The client calls GetObject() on the object factory, to invoke on the target grid object. The call to GetObject() achieves a connection between the client's srvObj object reference (for the view) and the target grid object in the server.

Required Setting

If you want to launch the bridge out-of-process, the *install-dir*\bin directory must be set on the system path. This might already have been done automatically at installation time. If not, you must do it manually.

The custsur.exe Executable

When COMet is launched in-process to the client, the COMet DLLs are hosted by a default surrogate executable, called DLLHOST.exe. However, when COMet is launched out-of-process, the COMet DLLs are instead hosted by a surrogate executable, called custsur.exe, on the bridge host.

The custsur.exe executable is supplied with your COMet installation. It is indicated by the following Windows registry value that is set during installation (where *version* represents the Orbix version number):

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

The CreateObject() Method

The Visual Basic `CreateObject()` method is completely independent of COMet, and can therefore be used on dedicated DCOM client machines. This is of particular use when you are using COMet with Internet Explorer. See “Using COMet with Internet Explorer” on page 61 for more details.

DCOM Security

Overview

This subsection addresses the subject of DCOM security, which is important for launching the bridge out-of-process. The following topics are discussed:

- "Addressing Security Issues" on page 60.
- "For More Information" on page 60.

Addressing Security Issues

Using DCOM as the wire protocol for communication between the client machine and the bridge machine requires that DCOM security issues are addressed. Security can be dealt with either by using `DCOMCNFG.EXE`, or programmatically via API security functions, or using a combination of these two approaches.

For More Information

A full treatment of COM security is outside the scope of this guide. For more details see the COM security FAQ at:

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

Using COMet with Internet Explorer

Overview

This section describes how to use a tool such as VBScript to set up a web-based Automation client that runs in Internet Explorer and uses COMet to communicate with CORBA objects in a remote web server.

Note: Before reading this section, ensure that you have read "Using DCOM with COMet" on page 54.

In This Section

This section discusses the following topics:

Specifying the Bridge Location

Overview

This subsection describes how to specify the location of the bridge for use with an Internet Explorer client. The following topics are discussed:

- "Supplied DLL" on page 62.
- "Referencing the DLL in HTML" on page 62.
- "Attributes for the OBJECT Tag" on page 63.

Supplied DLL

Unlike the Visual Basic `CreateObject()` method, the `CreateObject()` method used in VBScript does not have the ability to pass an optional hostname parameter. COMet therefore supplies a file, called `IT_C2K_CCIExWrapper0_VC60.DLL`, which contains an ActiveX control used for wrapping the COM `CoCreateInstanceEx()` method. You can reference the `IT_C2K_CCIExWrapper0_VC60.DLL` file in HTML, by using the `OBJECT` tag.

Referencing the DLL in HTML

The following is an example of how to use the `OBJECT` tag in HTML, to reference the `IT_C2K_CCIExWrapper0_VC60.DLL` file:

```
<OBJECT ID="bridge" <
CLASSID="CLSID:3DA5B85F-F2FC-11D0-8D97-0060970557AC"
# change this to reflect the location of
# IT_C2K_CCIExWrapper0_VC60.DLL on your machine
CODEBASE="\\machine-name\install-dir\asp\x.x\bin\
    IT_C2K_CCIExWrapper0_VC60.DLL"
>
</OBJECT>
```

In the preceding example, *`install-dir`* represents the full path to your installation, and *`x.x`* represents the Orbix version number.

Attributes for the OBJECT Tag

The `OBJECT` tag that is used to reference the DLL contains attributes that can be explained as follows:

`ID` The value for this attribute specifies the object name. In the preceding example it is `bridge`.

`CLASSID` The value for this attribute specifies the object type (that is, the object implementation). The syntax for this attribute is `CLSID:`*`class-identifier`* for registered ActiveX controls.

`CODEBASE` The value for this attribute specifies the object location, by supplying a URL that identifies the codebase for the object. You might need to modify the *`machine-name`* in the HTML file before the demonstration can work.

The Supplied Demonstration

Overview

This subsection describes the sample Internet Explorer client demonstration supplied with your COMet installation. The following topics are discussed:

- "Downloading the HTML Demonstration" on page 64.
- "VBScript Example" on page 64.
- "VBScript Explanation" on page 65.
- "Location of the VBScript Example" on page 65.
- "Setting Internet Explorer Security" on page 65.
- "Specifying the Machine Name" on page 66.
- "Running the Demonstration" on page 66.

Downloading the HTML Demonstration

When the HTML file for the supplied demonstration is first downloaded to the client machine, the ActiveX control for wrapping `CoCreateInstanceEx()` is also retrieved and registers itself on your client machine (provided you agree, of course). This allows use of COMet from client machines, with no configuration effort required on the client's part.

The only requirement is that you must configure COMet on the server side with respect to type information, access permissions, and so on, and place the HTML file for the demonstration on the server where the bridge resides.

DCOM is used as the wire protocol for communication between the client and the bridge.

VBScript Example

The HTML file can contain VBScript or JavaScript for calling methods on the remote CORBA objects. For the purposes of this example, it contains VBScript. Example 4 shows the VBScript example. client connects to the `grid` object on the `"advice.iona.com"` machine and obtains the height and the width of the grid:

Example 4: *Sample VBScript Client*

```
<SCRIPT LANGUAGE="VBScript">
<!--

Dim Grid
Dim fact
```

Example 4: *Sample VBScript Client*

```
  Sub btnConnect_Onclick
  lblStatus.Value = "Connecting…"

  # DCOM on the wire…
  # the parameter should be the name of the
  # machine where the bridge is located
1 Set fact = bridge.IT_CreateRemoteFactory("advice.iona.com")

  # IIOP on the wire
  Set fact = CreateObject("CORBA.Factory")

2 Set Grid = fact.GetObject("grid:" + sIOR)
  lblStatus.Value = "Obtaining dimensions…"
  sleWidth.Value  = Grid.width
  sleHeight.Value = Grid.height
  lblStatus.Value = "Connected…"
  End Sub

  -->
  </SCRIPT>
```

VBScript Explanation

The code shown in Example 4 can be explained as follows:

1. The client creates an instance of the remote CORBA object factory on the `advice.iona.com` machine (that is, the host on which the bridge is to be launched).
2. The client calls `GetObject()` on the object factory, to invoke on the target `grid` object. The call to `GetObject()` achieves a connection between the client's `Grid` object reference (for the view) and the target `grid` object in the server.

Location of the VBScript Example

The full version of the preceding VBScript example is supplied in *`install-dir`*`\demos\comet\grid\ie_client`.

Setting Internet Explorer Security

To use the supplied VBScript example, you must set your Internet Explorer security settings to **medium** in your Windows **Control Panel**. A security setting of **medium** means that you are prompted whenever executable content is being downloaded. That is all you need to do. You do not need to have Orbix installed. You can now open the `install-dir\demos\comet\grid\ie_client\griddemo.htm` file.

Specifying the Machine Name

You must complete the following steps in the `griddemo.htm` file (where *x.x* represents the Orbix version number), to specify the name of the machine that is to be contacted (that is, the machine where the bridge is located) when the demonstration is downloaded to a client:

Step	Action
1	Edit the following line: `CODEBASE="\\machine-name\install-dir\asp\x.x\bin \IT_C2K_CCIExWrapper0_VC60.DLL"`

2	Edit either of the following lines:
	`Set fact = bridge.IT_CreateInstanceEx("{A8B553C5-3B72-11CF-BBFC-444553540000}", "machine-name")`
	or
	`Set fact = bridge.IT_CreateRemoteFactory("machine-name")`
	Note: In the preceding example, `IT_CreateInstanceEx` takes a stringified CLSID as the first parameter, which in this case is the CLSID for `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, the HTML file can be accessed from any Windows machine with Internet Explorer. Neither Orbix nor COMet are required on the client side for the demonstration to work.

Running the Demonstration

The first time you access the HTML page, a dialog box opens to tell you that unsigned executable content is being downloaded, which is acceptable in this case. You should be presented with a simple GUI, similar to the Visual Basic or PowerBuilder GUI screens in Figure 7 on page 41 and Figure 8 on page 42. The steps to use the demonstration are:

Step	Action
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

Overview

Some Automation controllers (for example, Visual Basic) provide clients the option of using either straight `IDispatch` interfaces or dual interfaces for invoking on a server. This section describes the use of dual interfaces. The following topics are discussed:

- "What is a Dual Interface?" on page 68.
- "Early Binding" on page 68.
- "Type Libraries" on page 69.
- "The ts2tlb Utility" on page 69.
- "Viewing the Type Library" on page 70.
- "Using the Type Library in a Client" on page 70.
- "Registering the Type Library" on page 70.

What is a Dual Interface?

An Automation dual interface is a COM vtable-based interface that derives from the `IDispatch` interface. The vtable, a standard feature of object-oriented programming, is a function table that contains entries corresponding to each operation defined in an interface. This means that its methods can be either late-bound, using `IDispatch::Invoke`, or early-bound through the vtable portion of the interface.

Note: If you want to use dual interfaces with COMet, you must load the bridge in-process to the client. COMet does not support the use of dual interfaces with the bridge loaded out-of-process.

Early Binding

The use of dual interfaces means that client invocations can be routed directly through the vtable. 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.

Type Libraries

The use of dual interfaces requires the use of a type library. To use dual interfaces in an Automation client that wants to communicate with a CORBA server, you must create a type library that is based on the OMG IDL type information implemented by the target CORBA server. This allows the Automation client to be presented with an Automation view of the target CORBA objects.

The ts2tlb Utility

COMet provides a type library generation tool, called ts2tlb, which produces type libraries, based on OMG IDL type information in the COMet type store. For example, 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
```

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 can only create a type library, however, if it finds all the OMG IDL types it looks for.

Note: You must ensure that your OMG IDL is registered with the Interface Repository before you add it to the type store and use ts2tlb to create type libraries from it. See "Development Support Tools" on page 171 for full details about ts2tlb and creating type libraries from OMG IDL.

Viewing the Type Library

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 by using the MD5 algorithm, which is described in the OMG *Interworking Architecture* specification at `ftp://ftp.omg.org/pub/docs/formal/01-12-55.pdf`.

Using the Type Library in a Client

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
```

Registering the Type Library

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. See "COMet Utility Arguments" on page 411 for more details about `tlibreg`.

Developing COM Clients

Overview

COMet provides support for COM customized interfaces. In other words, COMet not only supports standard Automation interfaces; it also supports COM interfaces, with all the extended types that they provide. 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. This section describes how to use COMet to develop COM clients in C++.

In This Section

This section discusses the following topics:

Generating Microsoft IDL from OMG IDL

Overview

The first step in implementing a COM client that can communicate with a CORBA server is to generate the Microsoft IDL definitions required by the COM client from existing OMG IDL for the CORBA objects. This allows the COM client to be presented with a COM view of the target CORBA objects.

This subsection describes how to generate Microsoft IDL from OMG IDL. The following topics are discussed:

- "The ts2idl Utility" on page 72.
- "OMG IDL grid Interface" on page 73.
- "Microsoft IDL Igrid Interface" on page 73.

The ts2idl Utility

COMet provides a COM IDL generation tool, called `ts2idl`, which produces Microsoft IDL, based on OMG IDL type information in the COMet type store. For example, the following `ts2idl` command creates a `grid.idl` Microsoft 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` argument with `ts2idl`, to completely resolve those types and to produce COM IDL for them also.

Note: You must ensure that your OMG IDL is registered with the Interface Repository before you add it to the type store and use `ts2idl` to create COM IDL from it. See "Development Support Tools" on page 171 for full details about `ts2idl` and creating COM IDL from OMG IDL.

OMG IDL grid Interface

The grid object in the CORBA server implements the following OMG IDL grid interface:

```
// OMG IDL
interface grid {
readonly attribute short height;
readonly attribute short width;
void set(in short n, in short m, in long value);
long get(in short n, in short m);
};
```

Microsoft IDL Igrid Interface

The corresponding COM interface for the preceding OMG IDL interface is called Igrid, and is defined as follows:

```
//Microsoft IDL
[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
```

Compiling Microsoft IDL

Overview

After generating the required Microsoft IDL definitions from OMG IDL, you must compile the Microsoft IDL. This subsection describes how to compile it and the resulting output. The following topics are discussed:

- "The midl.exe Compiler" on page 74.
- "Resulting Output" on page 74
- "Building the Proxy/Stub DLL" on page 74.

The midl.exe Compiler

Use the `midl.exe` compiler to compile the Microsoft IDL.

Resulting Output

The `midl.exe` compiler produces:

- The C++ interface definitions to be used within the COM client application.
- A proxy/stub DLL to marshal the customized Microsoft IDL interface.

This procedure is standard practice when writing COM applications.

Building the Proxy/Stub DLL

You can use `ts2idl` to produce a makefile that subsequently allows you to build and register the proxy/stub DLL. The steps are:

Step	Action
1	Use the `-p` argument with `ts2idl` to produce the makefile. For example, the following command produces a `grid.mk` file in addition to the `grid.idl` file already shown in "Generating Microsoft IDL from OMG IDL" on page 72: `ts2idl -p -f grid.idl grid` The generated makefile contains information on how to build and register the proxy/stub DLL.
2	Use the generated makefile to build the proxy/stub DLL as normal. **Note:** You need Visual C++ 6.0, to build the proxy/stub DLL.

Writing a COM C++ Client

Overview

This subsection describes the steps to use COMet to write a COM C++ client of a CORBA server. The steps are:

Step	Action
1	Make general declarations.
2	Connect to the CORBA factory.
3	Connect to the CORBA server.
4	Invoke operations on the `grid` object.

Note: The source for this demonstration is in *`install-dir`*`\demos\comet\grid\com_client`, where *`install-dir`* represents the Orbix installation directory.

Step 1—General Declarations

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

```
// COM C++
HRESULT         hr = NOERROR;
IUnknown       *pUnk = NULL;
ICORBAFactory  *pCORBAFact = NULL;
DWORD           ctx;
// our custom interface
Igrid          *pIBasic = NULL;
MULTI_QI        mqi;
```

Step 2—Connecting to the CORBA Object Factory

Create a remote instance of the CORBA object factory, which implements the `ICORBAFactory` interface, on the client machine. This involves calling the COM `CoCreateInstanceEx()` method as normal, to obtain a pointer to `ICORBAFactory`. The remote instance of the CORBA object factory is assigned the `IID_ICORBAFactory` IID:

```
// COM C++
// Call to CoInitialize(), some error handling,
// and so on, omitted for clarity

memset (&mqi, 0x00, sizeof (MULTI_QI));
mqi.pIID = &IID_ICORBAFactory;
ctx = CLSCTX_INPROC_SERVER;
hr = CoCreateInstanceEx(IID_ICORBAFactory, NULL, ctx, NULL, 1,
   &mqi);
CheckHRESULT("CoCreateInstanceEx()", hr, FALSE);
pCORBAFact = (ICORBAFactory*)mqi.pItf;
```

Step 3—Connecting to the CORBA Server

Call `GetObject()` on the CORBA object factory, to get a pointer to the `IUnknown` interface of the COM view of the target `grid` CORBA object.

```
// COM C++
sprintf(szObjectName,"grid:%s",sIOR);
hr = pCORBAFact->GetObject(szObjectName, &pUnk);
if(!CheckErrInfo(hr, pCORBAFact, IID_ICORBAFactory))
{
pCORBAFact->Release();
return;
}
pCORBAFact->Release();
```

In the preceding code, `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 that might be raised during the course of a call. See "Exception Handling" on page 113 for more details about exception handling.

See "Obtaining a Reference to a CORBA Object" on page 103 and "ICORBAFactory" on page 254 for more details about `GetObject()`.

Step 4—Invoking Operations on the Grid Object

Call `QueryInterface()` on the pointer to the `IUnknown` interface of the COM view object, to obtain a pointer to the customized `Igrid` interface. The client can then use the `pIF` object reference to invoke operations on the target `grid` object in the server:

```
// COM C++
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 COMet Type Store Cache

Overview

This section describes the concept of *priming* the type store cache. The following topics are discussed:

- "What is Priming?" on page 78.
- "Relevance of Priming" on page 78.
- "For More Information" on page 78.

What is Priming?

When you are ready to run your application for the first time, you have the option of improving the runtime performance by adding the OMG IDL type information required by the application to the COMet type store. This is also known as *priming* the type store cache. Priming the cache means that the type store already holds the required OMG IDL type information in memory before you run your application. Therefore, the application does not have to keep contacting the Interface Repository for each IDL type required.

Relevance of Priming

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.

For More Information

See "Development Support Tools" on page 171 for details about the workings of the COMet type store cache and how to prime it.

CHAPTER 5

Developing an Automation Client

This chapter expands on what you learned in “Getting Started” on page 35. It uses the example of a distributed telephone book application to show how to write Automation clients in PowerBuilder or Visual Basic that can communicate with an existing CORBA C++ server.

In This Chapter

This chapter discusses the following topics:

Note: This chapter assumes that you are familiar with the CORBA Interface Definition Language (OMG IDL). See “Introduction to OMG IDL” on page 269 for more details.

The Telephone Book Example

Overview

This section provides an introduction to the telephone book application developed in this chapter. The following topics are discussed:

- "Application Summary" on page 80.
- "Graphical Overview" on page 81.
- "OMG IDL PhoneBook Interface" on page 81.
- "Location of Source Files" on page 82.
- "Client GUI Layout" on page 82.

Note: You do not need to understand how the demonstration server is implemented, to follow the examples in this chapter.

Application Summary

In the supplied telephone book application, the Automation client makes requests on a PhoneBook object implemented in a CORBA C++ server. As explained in "How COMet Implements the Model" on page 20, the client actually makes its method calls on a view object in the COMet bridge. The principal task of the Automation client in this example is, therefore, 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. (See "Mapping CORBA to Automation" on page 313 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.

Graphical Overview

Figure 10 provides a graphical overview of the components of the telephone book application.

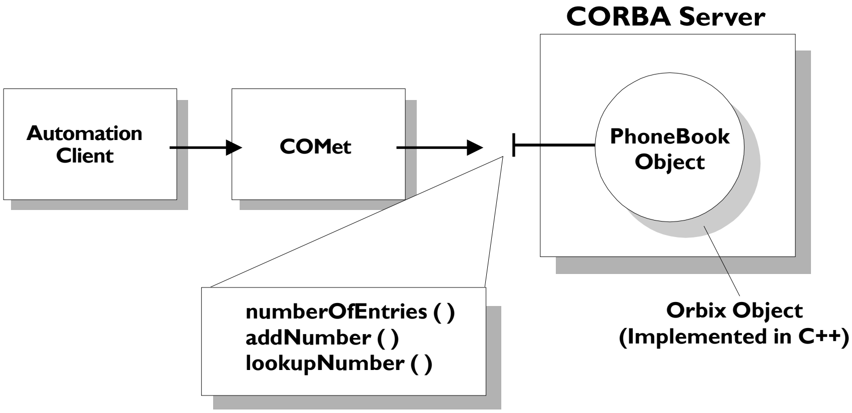


Figure 10: *Telephone Book Example with Automation Client*

OMG IDL PhoneBook Interface

The PhoneBook object in the CORBA server implements the following OMG IDL PhoneBook interface:

```
// OMG IDL
interface PhoneBook {
    readonly attribute long numberOfEntries;

    boolean addNumber(in string name, in long number);
    long lookupNumber(in string name);
};
```

Automation DIPhoneBook Interface

The corresponding Automation interface for the "OMG IDL PhoneBook Interface" on page 81 is called `DIPhoneBook`, and is defined as follows:

```
[odl,…]
interface DIPhoneBook : IDispatch {
[id(0x00000001)]
HRESULT addNumber(
     [in] BSTR name,
     [in] long number,
     [in, out, optional] VARIANT* excep_OBJ,
     [out, retval] VARIANT_BOOL* val);
[id(0x00000002)]
HRESULT lookupNumber(
     [in] BSTR name,
     [in, out, optional] VARIANT* excep_OBJ,
     [out, retval] long* val);
[id(0x00000003), propget]
HRESULT numberOfEntries([out, retval] long* val);
};
```

Location of Source Files

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\phonebook\vb_client`
PowerBuilder	*`install-dir`*`\demos\comet\phonebook\pb_client`
Internet Explorer	*`install-dir`*`\demos\comet\phonebook\ie_client`

The CORBA server application is supplied in the *`install-dir`*`\demos\comet\phonebook\cxx_server` directory.

Client GUI Layout

Figure 11 shows the layout of the client GUI interface that is developed in this chapter.

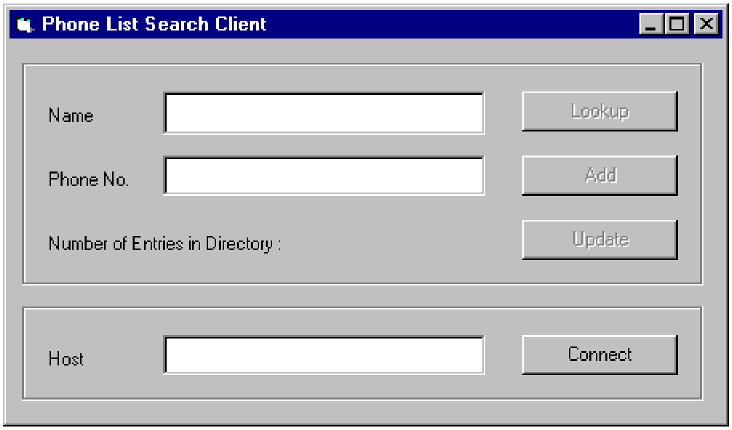


Figure 11: *Phone List Search Client GUI Interface*

Using Automation Dual Interfaces

Overview

This section describes the use of Automation dual interfaces. The following topics are discussed:

- "IDispatch versus Dual Interfaces" on page 84.
- "Creating Type Libraries" on page 84.

IDispatch versus Dual Interfaces

"Automation Dual Interface Support" on page 68 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.

Note: The use of dual interfaces is only supported when the bridge is loaded in-process to the client. If the bridge is loaded out-of-process, you must use `IDispatch`.

Creating Type Libraries

If you want to use dual interfaces, you must create a type library. To create an Automation client that uses dual interfaces and communicates with a CORBA server, 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 COMet `ts2tlb` utility. See "Creating a Type Library" on page 190 for more details.

Writing the Client

Overview

This section describes how to write a Visual Basic version of the client, without using the code generation genie. It also describes how to write a PowerBuilder version of the client.

Note: There is no code generation genie available for PowerBuilder. If you want to use the code generation genie for Visual Basic, see "Using the Visual Basic Genie" on page 43 for a detailed introduction, and "Generating Visual Basic Client Code" on page 199 for full details of how to use it.

In This Section

This section discusses the following topics:

Obtaining a Reference to a CORBA Object

Overview

This subsection provides Visual Basic and PowerBuilder examples of the client code that is used to obtain a reference to a CORBA object. See “The Visual Basic Client Code in Detail” on page 89 and “The PowerBuilder Client Code in Detail” on page 92 for the complete client code. The following topics are discussed:

- “Visual Basic Example” on page 86.
- “PowerBuilder Example” on page 86.
- “Explanation of Examples” on page 87.
- “Format of Parameter for GetObject()” on page 87.
- “Purpose of GetObject()” on page 88.
- “Explanation of GetObject()” on page 88.

Visual Basic Example

The following is a Visual Basic example of how to obtain a CORBA object reference:

Example 5:

```
   ' Visual Basic
   Dim ObjFactory As Object
   Dim phoneBookObj As Object
   …
1  Set ObjFactory = CreateObject("CORBA.Factory")
   …
2  Set phoneBookObj = ObjFactory.GetObject("PhoneBook:" + sIOR)
```

PowerBuilder Example

The following is a PowerBuilder example of how to obtain a CORBA object reference:

Example 6:

```
   // PowerBuilder
   OleObject ObjFactory
   OleObject phoneBookObj
   …
   ObjFactory = CREATE OleObject
1  ObjFactory.ConnectToNewObject("CORBA.Factory")
```

Example 6:

```
…
phoneBookObj = CREATE OleObject
phoneBookObj = ObjFactory.GetObject("PhoneBook:" + sIOR)
```
2

Explanation of Examples

The preceding examples can be explained as follows:

1. The client 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 calls `GetObject()` on the CORBA object factory. It passes the name of the `PhoneBook` object in the CORBA server in the parameter for `GetObject()`.

Format of Parameter for GetObject()

The parameter for `GetObject()` takes the following format:

```
"interface:TAG:Tag Data"
```

The *`TAG`* variable 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.staff.PD.Tom")
  ```

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

Purpose of 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 12 shows how the call to `GetObject()` achieves this.

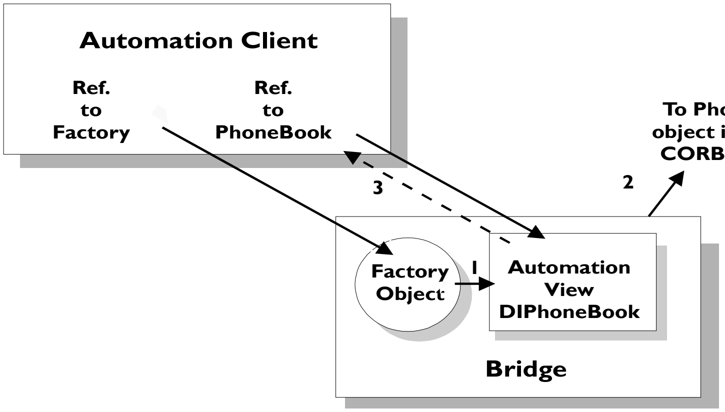


Figure 12: *Binding to the CORBA PhoneBook Object*

Explanation of GetObject()

In Figure 12, `GetObject()`:

1. Creates an Automation view object in the COMet bridge that implements the `DIPhoneBook` dual interface.
2. Binds the Automation view object to the CORBA `PhoneBook` implementation object named in the parameter for `GetObject()`.
3. Returns a reference to the Automation 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. This is further illustrated for Visual Basic in "Step 4—Invoking Operations on the PhoneBook Object" on page 90 and for PowerBuilder in "Step 4—Invoking Operations on the PhoneBook Object" on page 93.

The Visual Basic Client Code in Detail

Overview

This subsection describes the steps to write the complete Visual Basic client application. It shows how the Visual Basic code extracts shown in "Obtaining a Reference to a CORBA Object" on page 86 fit into the overall client program. The steps are:

Step	Action
1	Make general declarations.
2	Create the form.
3	Connect to the CORBA server.
4	Invoke operations on the PhoneBook object.
5	Unload the form.

Step 1—General Declarations

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

```
' Visual Basic
Dim ObjFactory As Object
Dim phoneBookObj As Object
```

Step 2—Connecting to the CORBA Object Factory

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

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

Step 3—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()`:

```
' Visual Basic
Private Sub ConnectBtn_Click()
Set phoneBookObj = ObjFactory.GetObject("PhoneBook:" + sIOR)
…
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 "Step 4—Invoking Operations on the PhoneBook Object".

Step 4—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:

```
' Visual Basic
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
```

Step 5—Unloading the Form

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

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

The PowerBuilder Client Code in Detail

Overview

This subsection describes the steps to write the complete PowerBuilder client application. It shows how the PowerBuilder code extracts shown in "Obtaining a Reference to a CORBA Object" on page 86 fit into the overall client program. The steps are:

Step	Action
1	Make general declarations.
2	Load the window.
3	Connect to the CORBA server.
4	Invoke operations on the PhoneBook object.
5	Unload the window.

Step 1—General Declarations

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

```
// PowerBuilder
OleObject ObjFactory
OleObject phoneBookObj
```

Step 2—Connecting to the CORBA Object Factory

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

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

Step 3—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()`:

```
// PowerBuilder
phoneBookObj = CREATE OleObject
phoneBookObj = ObjFactory.GetObject("PhoneBook:" + sIOR)
…
```

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 "Step 4—Invoking Operations on the PhoneBook Object".

Step 4—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:

```
// PowerBuilder
// 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)
```

Step 5—Unloading the Window

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

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

Building and Running the Client

Overview

This section describes how to build and run the client. The following topics are discussed:

- "Building the Client" on page 95.
- "Running the Client" on page 95.

Building the Client

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

Running the Client

The steps to run the client are:

Step	Action
1	Ensure that an activator daemon is running on the CORBA server's host. This allows the locator daemon to automatically activate the server. (See the *CORBA Administrator's Guide* for more details.)
2	Register the CORBA server with the Implementation Repository. (Usually, it is not necessary to register a server, if the server has been written and registered by someone else.) See the Orbix documentation set for more details.
3	Run the client. On the **Phone List Search Client** screen, shown in Figure 11 on page 83, type the server's hostname in the **Host** text box, and select **Connect**. You can now add and look up telephone book entries.

Note: 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.

CHAPTER 6

Developing a COM Client

This chapter expands on what you learned in "Getting Started" on page 35. 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.

In This Chapter

This chapter discusses the following topics:

Note: This chapter assumes that you are familiar with the CORBA Interface Definition Language (OMG IDL). See "Introduction to OMG IDL" on page 269 for more details.

The Telephone Book Example

Overview

This section provides an introduction to the telephone book application developed in this chapter. The following topics are discussed:

"Application Summary" on page 98.

"Graphical Overview" on page 99.

"OMG IDL PhoneBook Interface" on page 99.

"Microsoft IDL IPhoneBook Interface" on page 100.

"Location of Source Files" on page 100.

Note: You do not need to understand how the demonstration server is implemented, to follow the example in this chapter.

Application Summary

In the supplied telephone book application, the COM client makes requests on a PhoneBook object implemented in a CORBA C++ server. As explained in "How COMet Implements the Model" on page 20, the client actually makes its method calls on a view object in the COMet bridge. The principal task of the COM client in this example is, therefore, to obtain a reference to a COM PhoneBook view object in the bridge.

The PhoneBook view object exposes a COM IPhoneBook interface, generated from the OMG IDL PhoneBook interface. (See "Mapping CORBA to COM" on page 357 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.

Graphical Overview

Figure 13 provides a graphical overview of the components of the telephone book application.

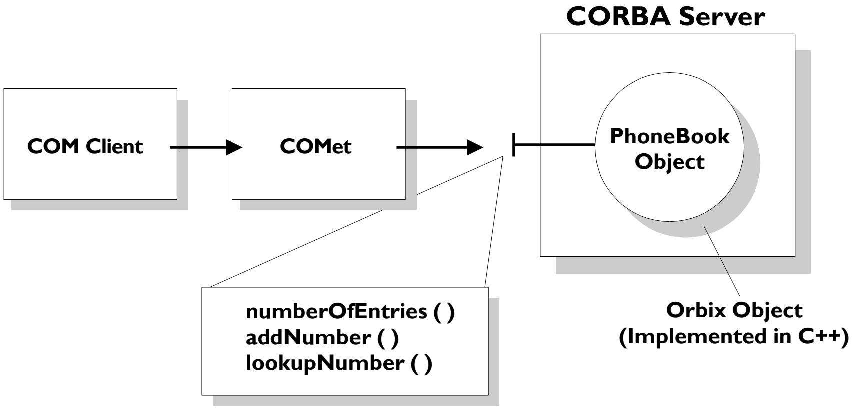


Figure 13: *Telephone Book Example with COM Client*

OMG IDL PhoneBook Interface

The PhoneBook object in the CORBA server implements the following OMG IDL PhoneBook interface:

```
// OMG IDL
interface PhoneBook {
    readonly attribute long numberOfEntries;

    boolean addNumber(in string name, in long number);
    long lookupNumber(in string name);
};
```

Microsoft IDL IPhoneBook Interface

The corresponding COM interface for the preceding OMG IDL interface is called `IPhoneBook`, and is defined as follows:

```
//COM IDL
[object,…]
interface IPhoneBook : IUnknown
{
    HRESULT addNumber([in, string] LPSTR name,
        [in] long number,
        [out] boolean *val);
    HRESULT lookupNumber([in, string] LPSTR name,
        [out] long *val);
    HRESULT _get_numberOfEntries([out] long *val);
};
```

Location of Source Files

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

The CORBA server application is supplied in the *`install-dir`*`\demos\comet\phonebook\cxx_server` directory.

Prerequisites

Overview

This section describes the prerequisities to writing a COM client with COMet. The following topics are discussed:

- "Generating Microsoft IDL from OMG IDL" on page 101.
- "Building a Proxy/Stub DLL" on page 101.

Generating Microsoft IDL from OMG IDL

As explained in "Generating Microsoft IDL from OMG IDL" on page 72, 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 COMet `ts2idl` utility. See "Development Support Tools" on page 171 for details of how to use it.

Building a Proxy/Stub DLL

If the COMet 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 COMet 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
```

The `-p` argument with `ts2idl` creates 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 supplied in the *`install-dir`*`\demos\comet\phonebook\com_client` directory.

Writing the Client

Overview

The section describes how to write the COM C++ client.

In This Section

This section discusses the following topics:

Obtaining a Reference to a CORBA Object

Overview

This subsection shows how the COM C++ client obtains a reference to a CORBA object. See the "The COM C++ Client Code in Detail" on page 107 for the complete client code. The following topics are discussed:

- "Example" on page 103.
- "Explanation" on page 104.
- "Format of Parameter for GetObject()" on page 104.
- "Purpose of GetObject()" on page 105.
- "Explanation of GetObject()" on page 105.
- "Using CoCreateInstance()" on page 106.

Example

The following is a COM C++ example of how to obtain a CORBA object reference:

Example 7:

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

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

   // Connecting to the CORBA Server
   // Read IOR from file
   // …
   sprintf(szObjectName,"PhoneBook:%s", szIOR);

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

Explanation

The preceding example can be explained as follows:

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()`.

Format of Parameter for GetObject()

The parameter for `GetObject()` takes the following format:

```
"interface:TAG:Tag Data"
```

The *TAG* variable 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.staff.PD.Tom")
  ```

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

Purpose of GetObject()

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 14 shows how the call to `GetObject()` achieves this.

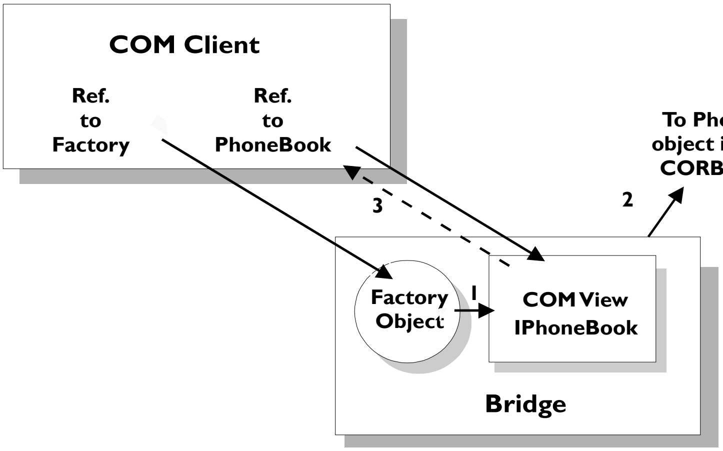


Figure 14: *Binding to the CORBA PhoneBook Object*

Explanation of GetObject()

In Figure 14, `GetObject()`:

1. Creates a COM view object in the COMet 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. This is further illustrated in "Step 5—Invoking Operations on the PhoneBook Object" on page 108.

Using CoCreateInstance()

The CORBA object factory allows you to obtain a reference to a CORBA object in a manner that is compliant with the OMG specification. However, as an alternative, COMet also allows a COM client to use the standard `CoCreateInstance()` COM API call, to connect directly to a CORBA server.

The COM C++ Client Code in Detail

Overview

This subsection describes the steps to write the complete COM C++ client application. It shows how the code extracts shown in "Obtaining a Reference to a CORBA Object" on page 103 fit into the overall client program. The steps are:

Step	Action
1	Make include statements.
2	Make general declarations.
3	Connect to the CORBA factory.
4	Connect to the CORBA server.
5	Invoke operations on the PhoneBook object.

Step 1—Include Statements

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

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

Step 2—General Declarations

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

```
// COM C++
IUnknown*pUnk = NULL;
IPhoneBook*pIPhoneBook = NULL;
ICORBAFactory*pCORBAFact = NULL;
char szObjectName[128];
```

Step 3—Connecting to the CORBA Object 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.

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

Step 4—Connecting to the CORBA Server

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

```
// COM C++
sprintf(szObjectName,"PhoneBook:%s", szIOR);

hr = pCORBAFact->GetObject(szObjectName,&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 “Step 5—Invoking Operations on the PhoneBook Object”.

Step 5—Invoking Operations on the PhoneBook Object

Invoke operations on the PhoneBook object in the CORBA server, which allow you to add a number to the telephone book and look up entries:

```
// COM C++
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
long phoneNumber=0;
pIF->lookupNumber("IONA Freephone USA", &phoneNumber);
cout << "The number for IONA Freephone USA is " << phoneNumber <<
   endl;
```

Building and Running the Client

Overview

This section describes how to build and run the client. The following topics are discussed:

- "Building the Client" on page 110.
- "Running the Client" on page 110.
- "Client Output" on page 111.

Building the Client

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

Running the Client

The steps to run the client are:

Step	Action
1	Ensure that an activator daemon is running on the CORBA server's host. This allows the locator daemon to automatically activate the server. (See the *CORBA Administrator's Guide* for more details.)
2	Register the CORBA server with the Implementation Repository. (Usually, it is not necessary to register a server, if the server has been written and registered by someone else.) See the Orbix documentation set for more details.
3	Run the client.

Client Output

The client produces output such as 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
```

CHAPTER 7

Exception Handling

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

In This Chapter

This chapter discusses the following topics:

Note: See the Orbix documentation set for details of system exceptions.

CORBA Exceptions

Overview

This section introduces the concept of CORBA exceptions. The following topics are discussed:

- "Standard System Exceptions" on page 114
- "Application-Specific User Exceptions" on page 114.
- "Exception Handling versus Exception Raising" on page 114.

Standard System Exceptions

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. COMet can raise system exceptions either during a remote invocation or through calls to COMet. These exceptions range from reporting network problems to failure to marshal operation parameters.

Application-Specific User Exceptions

CORBA also allows users to define application-specific exceptions that allow an application to define the set of exception conditions associated with it. These user exceptions are defined in the `raises` clause of an OMG IDL operation, and can be raised by a call to that OMG IDL operation. See the Orbix documentation set for more details.

Exception Handling versus Exception Raising

Applications do not (and should not) explicitly raise system exceptions. However, client applications should be able to handle both standard system exceptions and application-specific user exceptions. See "Exception Handling in Automation" on page 122 and "Exception Handling in COM" on page 131 for details of how clients can handle exceptions.

Example of a User Exception

Overview

This section provides an example of a typical user exception. The following topics are discussed:

- "OMG IDL Example" on page 115.
- "Explanation" on page 115.
- "Corresponding Automation Interface" on page 116.
- "Corresponding COM Interface" on page 117.

OMG IDL Example

The following is an example of an OMG IDL Bank interface, which contains a newAccount operation that raises a Reject exception:

Example 8:

```
   //OMG IDL
   interface Bank {

1      exception Reject {
2          string reason;
       };

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

An operation can raise more than one user exception. For example:

```
Account newAccount(in string owner) raises (Reject, BankClosed);
```

Explanation

The preceding example can be explained as follows:

1. The Bank interface defines a user exception called Reject.
2. The Reject exception contains one member, of the string type, which is used to specify the reason why the request for a new account was rejected.

3. The `newAccount()` operation can raise the `Reject` user exception (for example, if the bank cannot create an account, because the owner already has an account at the bank).

> **Note:** If COMet encounters some problem during the operation invocation, the `newAccount()` operation can then, of course, raise a system exception. However, system exceptions are not listed in a `raises` clause, and user code should never explicitly raise a system exception.

Corresponding Automation Interface

The Automation view of the preceding OMG IDL 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);
}
```

See “Mapping CORBA to Automation” on page 313 for details of how OMG IDL interfaces and exceptions map to Automation.

Corresponding COM Interface

The COM view of the preceding OMG IDL is as follows:

```
// COM 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);
};
```

See "Mapping CORBA to COM" on page 357 for details of how OMG IDL interfaces and exceptions map to COM.

Exception Properties

Overview

This section describes system and user exception properties.

In This Section

This section discusses the following topics:

General Exception Properties

Overview

This subsection describes the general exception properties that allow you to find information about a system or user exception that has occurred. The following topics are discussed:

- "(D)IForeignException Definition" on page 119.
- "Explanation" on page 119.

(D)IForeignException Definition

All exceptions expose the (D)IForeignException interface. It 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);
};
```

Explanation

The methods relating to (D)IForeignException 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 115).

Additional System Exception Properties

Overview

This subsection describes the additional system exception properties. The following topics are discussed:

- "(D)ICORBASystemException Definition" on page 120.
- "Explanation" on page 120.

System exceptions have additional properties, which are defined in the (D)ICORBASystemException interface.

(D)ICORBASystemException Definition

Additional system exceptions are defined in the (D)ICORBASystemException interface. It is defined as follows:

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

Explanation

The methods relating to (D)ICORBASystemException 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 whether or not it had completed.

`EX_minorCode()`

This returns a code describing the type of system exception that has occurred. See the *CORBA Programmer's Guide, C++* for details of minor exception codes and their associated textual descriptions.

Exception Handling in Automation

Overview

CORBA exceptions are mapped to Automation exceptions by the bridge. This means that exceptions raised by calls to CORBA objects can 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 115, 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 client.

In This Section

This section discusses the following topics:

Exception Handling in Visual Basic

Overview

This subsection describes how to use the On Error GoTo clause and standard Err object for exception handling, in a controller such as Visual Basic. The following topics are discussed:

- "Example" on page 123.
- "Triggering an Automation Exception" on page 123.
- "The Err Object" on page 123.

Example

In Visual Basic, exceptions can be trapped using the On Error GoTo clause, and handled using the standard Err object. (See your Visual Basic documentation for full details of the Err object.) The following Visual Basic code shows how a client can trap and handle an exception:

```
' 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
```

Triggering an Automation Exception

Even though the client cannot call the COM GetErrorInfo() function, to retrieve the error information, most controllers can trigger an Automation exception when the view calls the SetErrorInfo() function to populate the Err object with exception details. In the case of Visual Basic, for example, the currently active error trap is called and the Err object is used to contain the details of the exception that occurred.

The Err Object

The standard Err object contains properties that provide details of any exception that occurs. These properties can be described as follows:

`Err.Description` This 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 as follows:

CORBA User Exception :[Bank::Reject]

For a system exception, an example of the string in `Err.Description` is as follows:

```
CORBA System Exception :[CORBA::COMM_FAILURE]
   minor code [10087][NO]
```

`Err.Source` This indicates the operation that raised the exception (for example, `Bank.newAccount`).

Inline Exception Handling

Overview

This subsection describes exception handling in Automation controllers that do not support the concept of the standard Err object. The following topics are discussed:

- "How It Works" on page 125.
- "Example" on page 125.
- "IT_Ex Parameter" on page 126.
- "Disabling Standard Exception Handling" on page 126.
- "Uses of Inline Exception Handling" on page 126.

How It Works

Automation controllers that do not support the concept of the standard Err object can use inline exception handling as an alternative. Inline exception handling involves passing an additional parameter to each method, to obtain any error information that might occur. Any exception that does occur, in this case, is returned to the client via the DICORBASystemException interface, whose properties allow access to the error information.

Note: You must use inline exception handling if you want to access the members in a user exception. See "Mapping CORBA to Automation" on page 313 for details of how OMG IDL user exceptions map to Automation.

Example

As described in "Mapping for System Exceptions" on page 343, an OMG IDL operation maps to an Automation method that has an additional optional parameter. For example:

1. Consider the following OMG IDL:

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

2. This maps to the following COM IDL:

```
// COM IDL
interface DIAccount : IDispatch {
    …
    HRESULT makeDeposit([in] float amount,
        [out] float* balance,
        [optional, in, out] VARIANT* IT_Ex);
}
```

IT_Ex Parameter

A client can pass the IT_Ex parameter, shown in the preceding example, 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
…
```

When the IT_Ex parameter is subsequently passed in a method call, COMet does not translate any CORBA exceptions that might occur during the call into an Automation exception. Instead, an instance of DICORBASystemException is created and inserted into the IT_Ex parameter. This means that the IT_Ex parameter is populated with error information relating to any CORBA exception that occurs. This allows the client to retrieve the exception parameter in the context of the invoked method.

Disabling Standard Exception Handling

Passing the IT_Ex parameter means that standard Automation exception handling is disabled, so the view makes no calls to SetErrorInfo(). The corresponding operation returns HRESULT_FALSE, which prevents an active error trap from being called.

Uses of Inline Exception Handling

A user exception can define one or more members that translate to COM IDL methods. The client can pass the IT_Ex parameter in calls to these methods, so that if a user exception does occur, the IT_Ex parameter is populated with additional error information that the client in turn can extract.

Because the error-handling code must be written inline, the value of the exception can be examined inline. The ability to handle user exceptions inline is useful, because user exceptions can be thrown to indicate logical errors rather than unrecoverable errors.

Using Type Information

Overview

This subsection describes how you can use type information to check the type of exception that occured. The following topics are discussed:

- "Example for Type Library Usage" on page 128.
- "Explanation" on page 129.
- "Example for Non-Usage of Type Library" on page 129.

Example for Type Library Usage

Consider the following Visual Basic example, which assumes that a type library is being used:

```
' 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)

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

Explanation

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 COMet when using late binding (that is, when using `IDispatch` interfaces) in an Automation client.

Example for Non-Usage of Type Library

The following Visual Basic example assumes that a type library is not being used:

```
' 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)

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

    ' Which user exception?
    If ex.EX_Id = bank::reject
        MsgBox ex.EX_Id, "User Exception EX_Id :"
        MsgBox ex.INSTANCE_repositoryId, , "User
            Exception INSTANCE_repositoryId :"
        MsgBox ex.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
```

Exception Handling in COM

Overview

As explained in "Mapping for System Exceptions" on page 381, a CORBA exception maps to a COM IDL interface and an exception structure that appears as the last parameter of any mapped operation. This section describes two alternative ways of handling exceptions in COM. The one you use depends on how you build your COM client.

Note: See the Orbix documentation set for details of system exceptions.

In This Section

This section discusses the following topics:

Catching COM Exceptions

Overview

This subsection describes the standard method of CORBA exception handling in COM clients. The following topics are discussed:

- “How It Works” on page 132.
- “Example” on page 132.
- “Explanation” on page 133.
- “Memory Handling” on page 133.

How It Works

COMet maps CORBA exceptions to standard COM exceptions. There are two parts to the exception. The first part, HRESULT, gives the class of the exception. The second part is a human-readable form of the exception, which is exposed through the ISupportErrorInfo interface that is supported by all COM views of CORBA objects.

Example

Consider the following client 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);
```

Explanation

If the bridge makes a call to the server that subsequently raises a system or user exception, the COM view in the bridge calls the COM `SetErrorInfo()` function, to set the COM error object in the client thread. This allows the client code to subsequently call the `GetErrorInfo()` function, to retrieve the error object for reporting to the user.

The preceding code does the same as a COM client would do to report a COM exception, if a COM server were using the COM `SetErrorInfo()` method.

If no exception is raised, the COM view in the bridge calls `SetErrorInfo()` with a null value for the `ISupportErrInfo` pointer parameter. This assures the error object that the client thread is thoroughly destroyed.

The client can indicate that no exception information should be returned, by specifying null as the value for the operation’s exception parameter.

Memory Handling

If the client expects to receive exception information, it must pass the address of a pointer to the memory in which the exception information is to be placed. The client must subsequently release this memory when it is no longer required.

The COM view is responsible for the allocation of memory used to hold exception information being returned.

Using Direct-to-COM Support

Overview

This subsection describes an alternative to standard CORBA exception handling in COM clients. The following topics are discussed:

- “How It Works” on page 134.
- “Example” on page 134.
- “Explanation” on page 135.

How It Works

In some cases, the IDL for a CORBA operation can define that it raises only one user exception, COM_ERROR. This happens, for example, in the case of a CORBA implementation of an already existing COM interface. Specifying COM_ERROR in an OMG IDL raises clause indicates that the operation was originally defined as a COM operation.

Example

Consider the following client example:In this case, CORBA exceptions are mapped to the standard _com_error exception. For example:

Example 9: *Using Direct-to-COM Exception Handling (Sheet 1 of 2)*

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

   fact.CreateInstance("CORBA.Factory");
1  bank = fact->GetObject(szObjectName, 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;
   }
2  catch (_com_error &e)
   {
   print_error(e);
   }
```

Example 9: *Using Direct-to-COM Exception Handling (Sheet 2 of 2)*

```
catch (…)
{
cerr << “Caught unknown exception ” << endl;
}
```

Explanation

1. The `szObjectName` parameter to `GetObject()` takes the format `"bank:IOR:`*xxxxxxxx*`"` (where *xxxxxxxx* represents the IOR string).
2. CORBA exceptions are mapped to, and caught by, the standard `_com_error` exception.

CHAPTER 8

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 how to implement client callbacks.

In This Chapter

This chapter discusses the following topics:

Introduction to Callbacks

Overview

This chapter introduces the concept of client callbacks. The following topics are discussed:

- "What Is a Callback?" on page 138.
- "Typical Use" on page 138.

What Is a Callback?

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.

Typical Use

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. 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.

Note: The COMet bridge holds an Orbix proxy object, as well as a COM or Automation view object, for each implementation object to which it has a reference.

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.

Implementing Callbacks

Overview

This section describes how to implement callbacks.

In This Section

This section discusses the following topics:

Note: A demonstration that implements callback functionality is provided in `*install-dir*\demos\corba\COMet\callback`, where `*install-dir*` represents your Orbix installation directory.

Defining the OMG IDL Interfaces

Overview

This section describes the first step in implementing client callback functionality, which is to define the OMG IDL interfaces for the server objects and client objects. The following topics are discussed:

- "Client Interface Example" on page 140.
- "Client Interface Explanation" on page 140.
- "Server Interface Example" on page 140.
- "Server Interface Explanation" on page 140.

Client Interface Example

The client implements an IDL interface that the server uses to call back clients. A suitable IDL interface for the client might be defined as follows:

```
// OMG IDL
interface NotifyCallback{
    oneway void notifyClient();
}
```

Client Interface Explanation

In the preceding example, the `notifyClient()` operation is declared as oneway, because it is important that the server is not blocked when it calls back its clients.

Server Interface Example

The server implements an IDL interface that allows it to maintain a list of clients that should be notified of changes in its objects' data. A suitable IDL interface for the server might be defined as follows:

```
// OMG IDL
interface RegisterCallback{
    void registerClient(in NotifyCallback client);
    void unregisterClient(in NotifyCallback client);
}
```

Server Interface Explanation

The preceding example can be explained as follows:

- 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 remove the client from its list of interested clients.

Generating Stub Code for the Callback Objects

Overview

After you have defined the OMG IDL interfaces for the server and client, you can generate the stub code for the callback objects from the OMG IDL.

For More Information

See "Creating Stub Code for Client Callbacks" on page 194 for full details of how to do this.

Implementing the Client

Overview

To write a client, you must implement the `NotifyCallback` interface defined for the client objects. You can use the generated stub code for the callback objects as a starting point.

In This Section

This section discusses the following topics:

Note: 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.

Implementing the Client in Visual Basic

Overview

This subsection describes how to implement the client in Visual Basic. The following topics are discussed:

- "Code for Generated Class File" on page 144.
- "Code for Client Form" on page 144.
- "Explanation" on page 145.

Code for Generated Class File

The following is the code in the generated `NotifyCallback.cls` file:

```
' Visual Basic
Public Sub notifyClient(Optional ByRef IT_Ex As Variant)

… ' Your code goes here

End Sub
…
```

Code for Client Form

The following is the code in the `client.frm` file for the Visual Basic client's form:

Example 10:

```
  ' Visual Basic
1 Dim clientObj as New NotifyCallback

  Dim ObjFactory As Object
  Set ObjFactory = CreateObject("CORBA.Factory")
  …
  Dim serverObj as clientBridge.DIRegisterCallback
  Set serverObj =
2     ObjFactory.GetObject("RegisterCallback:"&IOR_file)
3 serverObj.registerClient clientObj
```

Explanation

The preceding client code can be explained as follows:

1. It creates an implementation object, `clientObj`, which is of the `NotifyCallback` type.
2. It binds to an object of the `RegisterCallback` type in the server. At this point, the client holds both of the following:
    - An implementation object for the `NotifyCallback` type.
    - A reference to an Automation view object, `serverObj`, for an object of the `RegisterCallback` type.
3. 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`.

Implementing the Client in PowerBuilder

Overview

This subsection describes how to implement the client in PowerBuilder. The following topics are discussed:

- "Example" on page 146.
- "Explanation" on page 146.

Example

The following is the code for the PowerBuilder client:

Example 11:

```
    //PowerBuilder
    integer success
    OleObject clientObj
    OleObject ObjFactory
1   success = clientObj.ConnectToNewObject
        ("PBcallback.NotifyCallback")

    ObjFactory = CREATE OleObject
    serverObj = CREATE OleObject

    if success != 2 then
2       serverObj = ObjFactory.GetObject(“CallBack:"&IOR_file)
3       serverObj.Register(clientObj)
```

Explanation

The preceding client code can be explained as follows:

1. It creates an implementation object, `clientObj`, which is of the `NotifyCallback` type.
2. It binds to an object of the `CallBack` type in the server. At this point, the client holds both of the following:
    - An implementation object for the `NotifyCallback` type.
    - A reference to an Automation view object, `serverObj`, for an object of the `CallBack` type.

3. 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 `Register()` operation on the `serverObj` view object, and passes it a reference to its implementation object, `clientObj`.

Implementing the Client in COM C++

Example

The following is the code for the COM C++ client:

Example 12:

```
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
char *sIOR;
// read IOR
char *szObjectName;
// allocate memory for string
sprintf(szObjectName,"Callback:%s", sIOR);
hr = pCORBAFact->GetObject(szObjectName,&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 until we explicitly quit for the none console application
```

Example 12:

```
StartCOMServerLOOP(10000);
poImpl->Release();
```

Implementing the Server

Overview

This section describes the steps to implement a server for the purpose of client callbacks. The steps are:

Step	Action
1	Implement the `RegisterCallback` interface.
2	Invoke the `notifyClient()` operation.

Note: See the *CORBA Programmer's Guide, C++* for more details of how to implement servers.

Step 1—Implementing the RegisterCallback Interface

You must provide an implementation class for the `RegisterCallback` interface. You can use the stub code generated for the callback objects as a starting point to do this.

The implementation of the `registerClient()` operation receives an object reference from the client. When this object reference enters the server address space, a CORBA view for the client's `NotifyCallback` object is created in the client's bridge.

The server uses the created view to call back to the client. The implementation of the `registerClient()` operation should store the reference to the view for this purpose.

Step 2—Invoking the notifyClient() Operation

After the COM or Automation view for the client's `NotifyCallback` object has been created in the server address space, the server can then invoke the `notifyClient()` operation on the view.

CHAPTER 9

Deploying a COMet Application

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

In This Chapter

This chapter discusses the following topics:

Deployment Models

Overview

"Usage Models and Bridge Locations" on page 27 outlines the various deployment scenarios that are supported with COMet. When it comes to Automation clients, COMet supports communication using either DCOM or IIOP. When it comes to COM clients, COMet only supports communication using IIOP. This means Automation clients enjoy a good deal of flexibility about where the bridge can be installed, whereas COM clients enjoy no such flexibility. This section provides some more details about the various possible deployment scenarios that COMet offers.

In This Section

This section discusses the following topics:

Bridge In-Process to Each Client

Overview

This subsection describes a scenario where the bridge is loaded in-process to each client. The following topics are discussed:

- "Details" on page 153.
- "Graphical Overview" on page 153.

Details

This has the COMet bridge loaded in-process to each COM or Automation client (that is, in each client's address space). In this case:

- The bridge on each client machine uses IIOP to communicate with the CORBA server.
- Each client machine can be running on Windows NT, Windows 98, or Windows 2000.
- Each client can be COM-based or Automation-based.
- The CORBA server machine can be running on any platform, such as Windows, UNIX, or OS/390.

For Automation clients, this is the recommended COMet deployment scenario. For COM clients, this is the only supported COMet deployment scenario.

Graphical Overview

Figure 15 provides a graphical overview of a scenario where the COMet bridge is loaded in-process to each COM or Automation client.

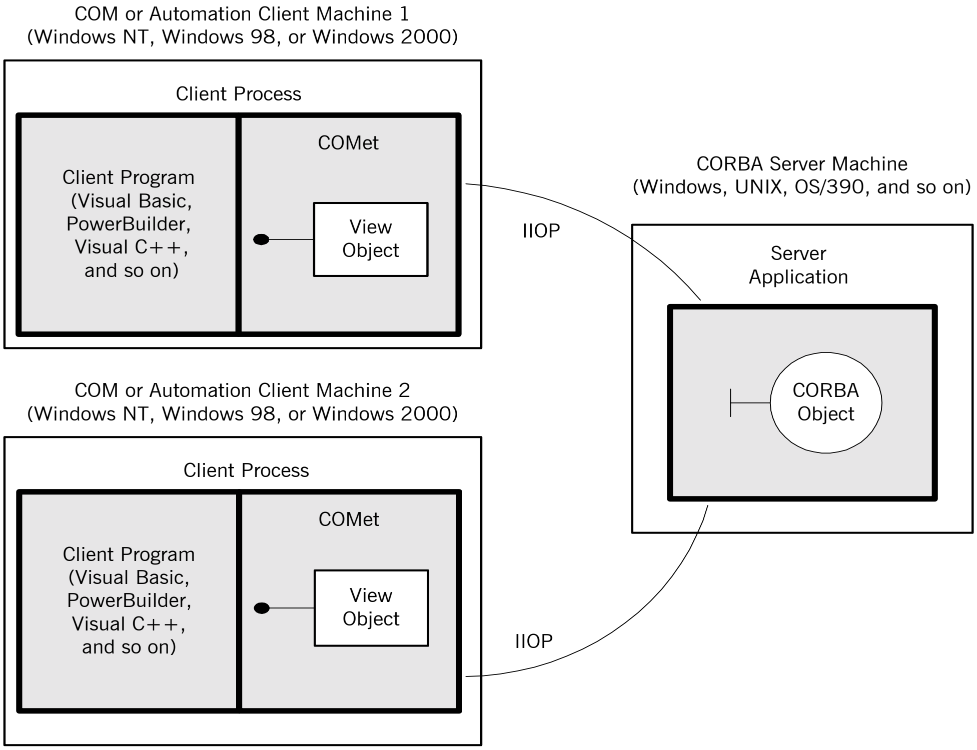


Figure 15: *Bridge In-Process to Each Client*

Bridge Out-of-Process on Each Client Machine

Overview

This subsection describes a scenario where the bridge is launched out-of-process on each client machine. The following topics are discussed:

- "Details" on page 155.
- "Graphical Overview" on page 155.

Details

This has the COMet bridge launched out-of-process on each client machine. In this case:

- The bridge is referred to as a local server.
- The bridge on each client machine uses IIOP to communicate with the CORBA server.
- Each client machine should preferably be running on Windows 2000, for reasons of scalability. Otherwise, it limits the number of clients that can be handled.
- Each client must be Automation-based and using `IDispatch` interfaces rather than dual interfaces.
- The CORBA server machine can be running on any platform, such as Windows, UNIX, or OS/390.

Graphical Overview

Figure 16 provides a graphical overview of a scenario where the COMet bridge is loaded out-of-process on each Automation client machine.

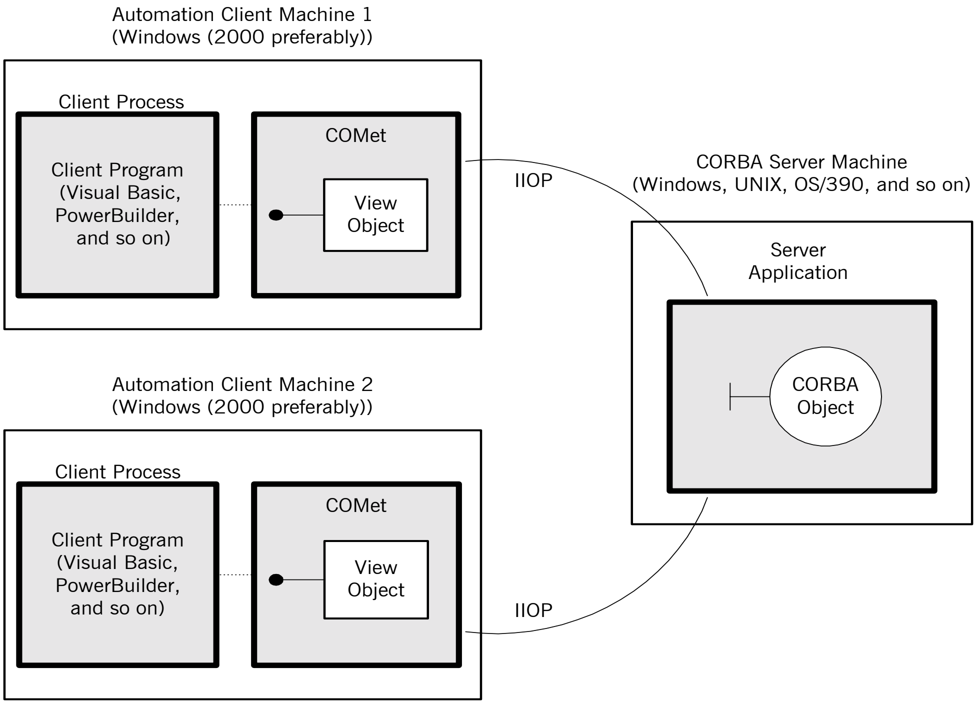


Figure 16: *Bridge Out-Of-Process On Each Client Machine*

Bridge on Intermediary Machine

Overview

This subsection describes a scenario where the bridge is launched on a single intermediary machine. The following topics are discussed:

- "Details" on page 157.
- "Creating a Remote Instance of the CORBA Object Factory" on page 157.
- "TYPEMAN_READONLY Configuration Setting" on page 158.
- "Graphical Overview" on page 158.

Details

This has the COMet bridge launched on a single intermediary machine. In this case:

- The bridge is referred to as a remote server.
- Each client machine can be running on Windows NT, Windows 98, or Windows 2000.
- Each client must be Automation-based and using `IDispatch` interfaces rather than dual interfaces.
- Each client uses DCOM to communicate with the bridge.
- The bridge machine must be running on Windows. It should preferably be running on Windows 2000, for reasons of scalability. Otherwise, it limits the number of clients that can be handled.
- The bridge uses IIOP to communicate with the CORBA server.
- The CORBA server machine can be running on any platform, such as Windows, UNIX, or OS/390.

Creating a Remote Instance of the CORBA Object Factory

For the purposes of this deployment scenario, 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 COM `CoCreateInstanceEx()` method. Most Automation controllers now allow you to supply a hostname as an optional extra parameter to their equivalent of the Visual Basic `CreateObject()` method, similar to the `CoCreateInstanceEx()` method.

TYPEMAN_READONLY Configuration Setting

When using multiple DCOM clients with a single bridge, as shown in Figure 17, the setting of the `COMet.Typeman.TYPEMAN_READONLY` configuration variable is particularly important. See "COMet Configuration" on page 399 for details.

Graphical Overview

Figure 17 provides a graphical overview of a scenario where the COMet bridge is installed on a single separate machine.

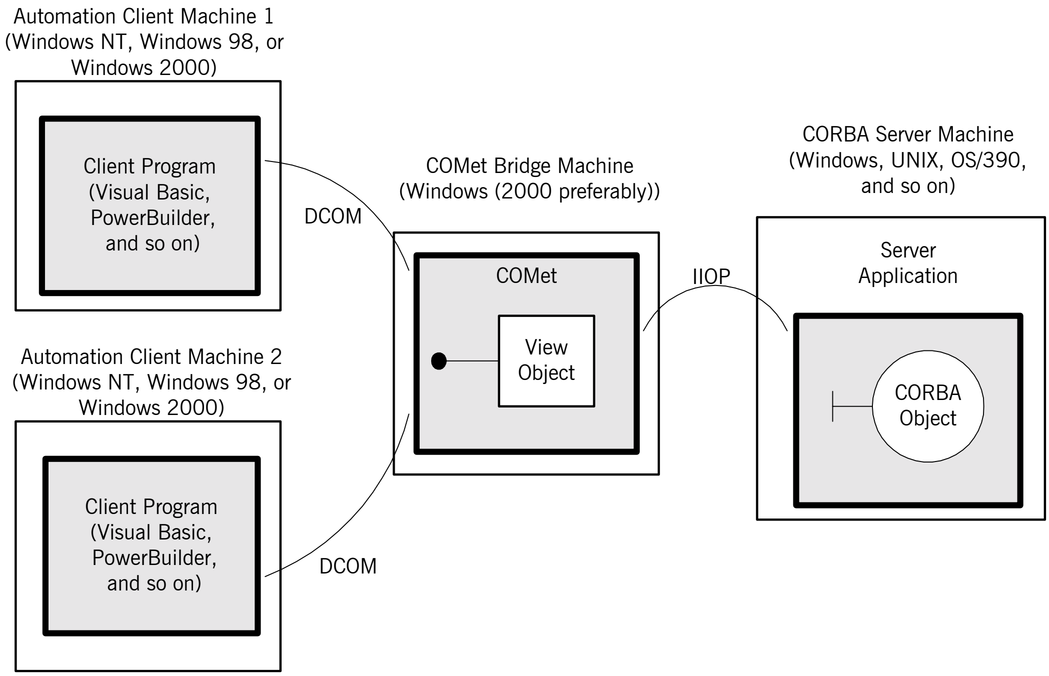


Figure 17: *Bridge on Intermediary Machine*

Bridge on Server Machine

Overview

This subsection describes a scenario where the bridge is launched on the CORBA server machine. The following topics are discussed:

- "Details" on page 159.
- "Creating a Remote Instance of the CORBA Object Factory" on page 159.
- "TYPEMAN_READONLY Configuration Setting" on page 159.
- "Graphical Overview" on page 160.

Details

This has the COMet bridge installed on the CORBA server machine. In this case:

- The bridge is referred to as a remote server.
- Each client machine can be running on Windows NT, Windows 98, or Windows 2000.
- Each client must be Automation-based and using `IDispatch` interfaces rather than dual interfaces.
- Each client uses DCOM to communicate with the CORBA server machine.
- The CORBA server machine must be running on Windows. It should preferably be running on Windows 2000, for reasons of scalability. Otherwise, it limits the number of clients that can be handled.

Creating a Remote Instance of the CORBA Object Factory

For the purposes of this deployment scenario, 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 COM `CoCreateInstanceEx()` method. Most Automation controllers now allow you to supply a hostname as an optional extra parameter to their equivalent of the Visual Basic `CreateObject()` method, similar to the `CoCreateInstanceEx()` method.

TYPEMAN_READONLY Configuration Setting

When using multiple DCOM clients with a single bridge, as shown in Figure 18 on page 160, the setting of the `COMet.Typeman.TYPEMAN_READONLY` configuration variable is particularly important. See "COMet Configuration" on page 399 for details.

Graphical Overview

Figure 18 provides a graphical overview of a scenario where the COMet bridge is installed on the CORBA server machine.

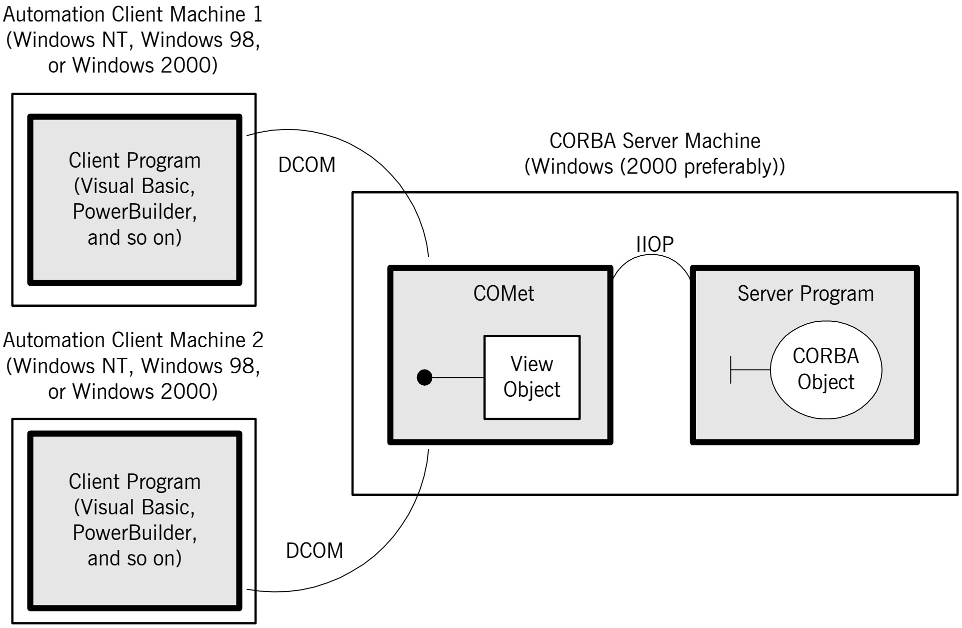


Figure 18: *Bridge on Server Machine*

Internet Deployment

Overview

This subsection discusses deploying a COMet application on the Internet. There are two deployment options to choose from. The following topics are discussed:

- "Dowloading the Bridge to the Client" on page 161.
- "Leaving the Bridge on the Internet Server" on page 161.

Dowloading the Bridge to the Client

You can choose to download the entire COMet 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.

Leaving the Bridge on the Internet Server

You can alternatively choose to download only the `IT_C2K_CCIExWrapper0_VC60.DLL` file and leave the bridge on the Internet server. In this case, your ActiveX control uses DCOM to communicate with your Internet server.

Deployment Steps

Overview

This section describes the steps you must follow to deploy a COMet application. The following topics are discussed:

- "Installing Your Application Runtime" on page 162.
- "Installing the Development Language Runtime" on page 162.
- "Installing the Orbix Deployment Environment" on page 162.
- "Configuring COMet" on page 162.
- "Installing and Registering Type Libraries" on page 163.

Installing Your Application Runtime

The components associated with your COMet 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).

See the documentation set for the specific development language you are using for details of the runtime requirements of that language.

Installing the Orbix Deployment Environment

Regardless of the model you adopt in deploying your COMet applications, the Orbix deployment environment requirements remain the same. See the Orbix 6.1 *Deployment Guide* for full details of Orbix deployment environment requirements.

Configuring COMet

You must set the COMet configuration variables required by your COMet application at the location where the COMet runtime is installed. You must modify the configuration entries in the configuration domain appropriately for your system.

When specifying a path name for a specific directory, you must provide the full path name and ensure it is valid. You must also ensure the activator and locator daemons have read/write permissions on the directories specified in these path names.

See “COMet Configuration” on page 399 for details of the COMet configuration variables. See the *CORBA Administrator’s Guide* for details of the core Orbix configuration variables.

Installing and Registering 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 use the supplied tlibreg utility to register a type library. See “Creating a Type Library” on page 190 and “Tlibreg Arguments” on page 418 for more details.

Minimizing the Client-Side Footprint

Overview

This section describes how to minimize the client-side footprint in your COMet deployment. The following topics are discussed:

- "Zero-Install Configuration" on page 164.
- "Internet-Based Deployment" on page 164.
- "Automation-Based Clients" on page 164.
- "COM-Based Clients" on page 165.

Zero-Install Configuration

In certain scenarios, COMet allows you to deploy your client application without requiring any COMet footprint on the client machine. This is normally referred to as a zero-install configuration. This means you can use a centralised installation of the COMet bridge for your clients that 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 COMet supports the DCOM wire protocol, your web-based clients can use DCOM to communicate with your installation of COMet, which then forwards the calls to the appropriate CORBA server. If your scripting language supports the creation of a remote DCOM object, no COMet runtime needs to be downloaded to that machine.

Automation-Based Clients

If you are developing client applications that use Automation late binding (that is, they use `IDispatch` interfaces), you can choose to use DCOM as the wire protocol. In this scenario, you do not need any COMet installation on your client machine, provided the Automation language supports connection to a remote DCOM object (which in this case is the COMet bridge).

If your client applications use early binding (that is, they use dual interfaces rather than straight `IDispatch` interfaces), the type library that you created via the `COMetCfg` tool or the `ts2tlb` command-line utility must be included with your client application. (This means that the type library file must be copied along with the client executable file to any machine on which you want to run the application.) 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. COMet can automatically generate the COM IDL definitions and makefile, which are needed to create this DLL, by using the `COMetCfg` tool or the `ts2idl` command line tool.

If your COM client application uses the standard COMet interfaces, such as `ICORBAFactory`, you must also include the COMet proxy/stub DLL. This is called `IT_C2K_PROXY_STUB5_VC60.DLL` and is located in the *`install-dir`*`\asp\`*`version`*`\bin` directory, where *`version`* represents the Orbix version number.

If your COM client uses pure DCOM calls, you must register forwarding entries in your client-side registry, to indicate the COMet CORBA location information for your CORBA server. You can use the `SrvAlias` utility to create the extra registry entries. For deployment purposes, you can use the `AliasSrv.exe` to restore these settings during installation. See the `demo\COM\coCreate` demonstration for an example. See “Replacing an Existing DCOM Server” on page 196 for more details about the `AliasSrv` and `SrvAlias` utilities.

Deploying Multiple Hosts

Overview

A typical scenario might involve multiple clients running simultaneously, with each client configured to connect to a different server on a different host. This section describes how this scenario can be easily achieved.

Graphical overview

Figure 19 provides a graphical overview of a deployment scenario involving different COMet clients, each of which contacts a different server host at application runtime.

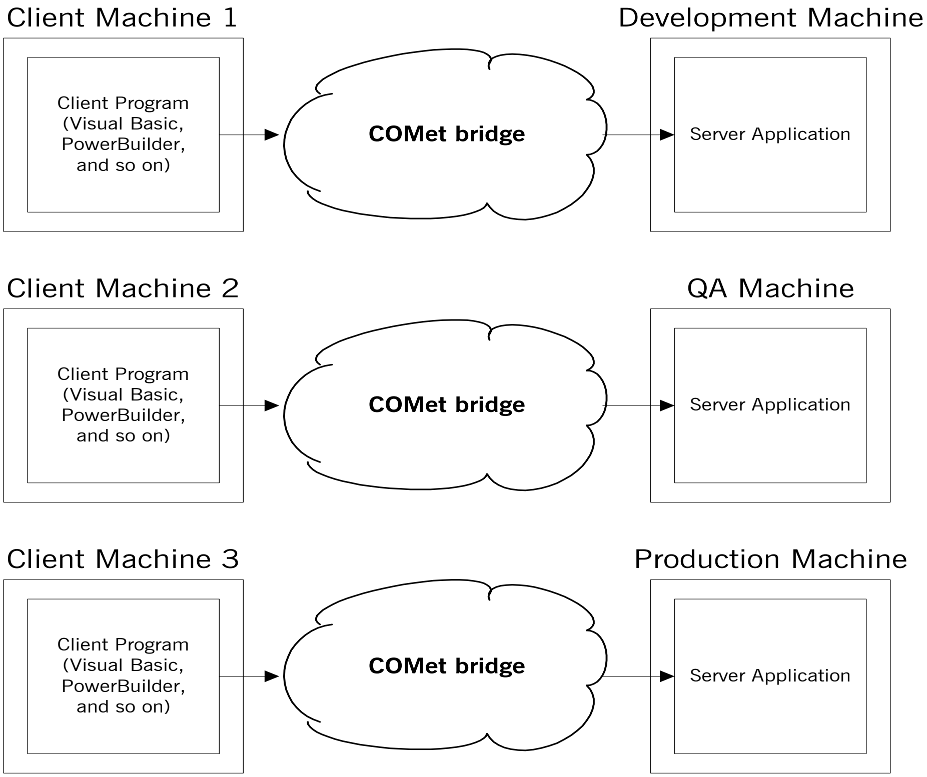


Figure 19: *Deploying Multiple Hosts*

> **Note:** In reality, the COMet bridge could be deployed in a number of different ways, as explained in "Deployment Models" on page 152. Even though it is possible to deploy just one COMet bridge to mediate between all clients and servers, this is not recommended because of the performance overheads it could incur at application runtime.

Steps

The steps to deploy this type of scenario are:

1. Ensure that your server-side configuration includes the Naming Service and IFR. See the Orbix *Deployment Guide* and Orbix *Administrator's Guide* for more details of how to set up configuration domains and configuration scopes. See the Orbix *Configuration Reference* for more details of how to configure Orbix services such as the Naming Service and IFR.
2. Ensure that your client program calls `GetObject()` to obtain the relevant object references via the Naming Service. For example:

   ```
   // Visual Basic
   …
   obj = fact.GetObject("interface_type:NAME_SERVICE:name")
   …
   ```

   See "Format of Parameter for GetObject()" on page 87 for more details of the format of the preceding example.
3. Ensure that your client-side configuration includes the `initial_references:NameService:reference` and `initial_references:InterfaceRepository:reference` configuration items. The values that can be specified for these items can take either of the following formats:
   - `"IOR:…"`
     In this case, the IOR string for the Naming Service or IFR can be obtained from the server-side configuration.
   - `"corbaloc:iiop:`*`host`*`:`*`port`*`:/NameService"` or `"corbaloc:iiop:`*`host`*`:`*`port`*`:/InterfaceRepository"`
     In this case, *`host`* and *`port`* specify where the locator daemon is running. This format is particularly useful in allowing you to quickly change the details of the host (for example, Development

machine, QA machine, Production machine) to which you want to point the client.

By encapsulating these variables in configuration scopes specific to each deployment scenario, as shown in the following example, it is possible at runtime to dynamically change the configuration. For example:

```
…
Development
{
host="123.45.67.89";
port="3075";
initial_references:NamingService:reference="corbaloc:iiop:"
   +host+":"+port+"/NameService"
initial_references:InterfaceRepository:reference="corbaloc:
   iiop:"+host+":"+port+"/InterfaceRepository";
};

QA
{
host="123.45.66.123";
port="3075";
initial_references:NamingService:reference="corbaloc:iiop:"
   +host+":"+port+"/NameService"
initial_references:InterfaceRepository:reference="corbaloc:
   iiop:"+host+":"+port+"/InterfaceRepository";
};

Production
{
host="123.45.70.14";
port="3075";
initial_references:NamingService:reference="corbaloc:iiop:"
   +host+":"+port+"/NameService"
initial_references:InterfaceRepository:reference="corbaloc:
   iiop:"+host+":"+port+"/InterfaceRepository";
};
…
```

Note: Any variable defined in the global configuration scope can also be included within scopes such as those in the preceding example. This allows you to fine-tune your configuration for specific clients.

4. To specify which ORB is to be used, ensure that the form load at the start of your client program calls `SetOrbName()`, passing the name of the relevant configuration scope (that is, `"Development"`, `"QA"`, or `"Production"`) as a parameter.

   An alternative to setting the ORB name programatically is to set the `IT_ORB_NAME` environment variable with the relevant ORB name. You can set this environment variable either globally through the Windows Control Panel or locally through a batch file.

CHAPTER 10

Development Support Tools

This chapter first describes the central role played by the COMet type store in terms of the development support tools supplied with COMet. It then describes the tools you can use to manage the type store cache and to generate Microsoft IDL and type library information from existing OMG IDL, which is necessary to allow COM or Automation clients to communicate with CORBA servers. It also describes how to generate stub code, if you want to avail of client callback functionality in your applications. Finally, it describes the tools you can use to generate Visual Basic code from OMG IDL, and to replace an existing COM or Automation server with a CORBA server.

In This Chapter

This chapter discusses the following topics:

Note: The `typeman`, `ts2idl`, and `ts2tlb` command-line utilities described in this chapter are located in *`install-dir`*`\bin`, where *`install-dir`* represents your Orbix installation directory.

The COMet Type Store

Overview

This section describes the COMet type store in terms of its role and how it works.

In This Section

This section discusses the following topics:

The Central Role of the Type Store

Overview

This subsection describes the role of the type store. The following topics are discussed:

- "Graphical Overview" on page 174.
- "Role" on page 175.

Graphical Overview

Figure 20 provides a graphical overview of the central role played by the type store in the use of the COMet development utilities.

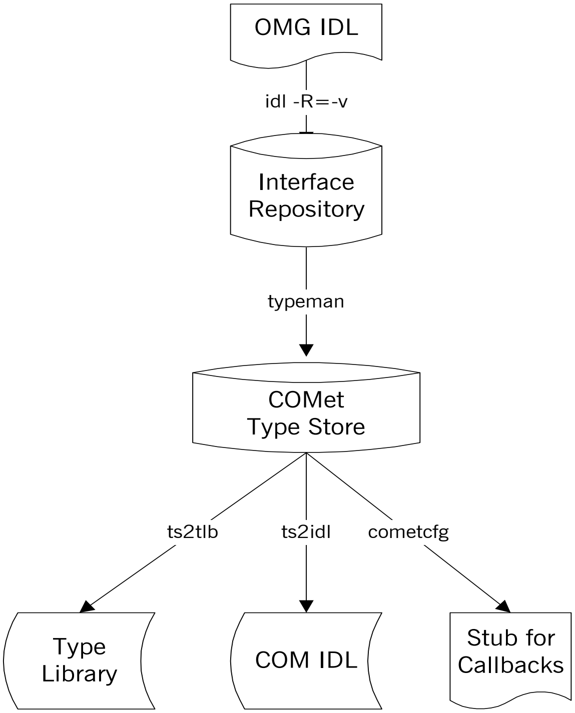


Figure 20: *COMet Type Store and the Development Utilities*

Role

As shown in Figure 20 on page 174, the type store plays a central role in the use of the COMet development utilities. The `typeman` utility manages the OMG IDL information in the type store cache. The `ts2tlb`, `ts2idl`, and `cometcfg` utilities use the OMG IDL type information in the cache to respectively generate the Microsoft IDL, type library information, and callback stub code used by your COM or Automation clients for communicating with CORBA servers.

The Caching Mechanism of the Type Store

Overview

This subsection describes how type information is stored in the type store. The following topics are discussed:

- "OMG IDL" on page 176.
- "Memory and Disk Cache" on page 176.
- "Type Information Management" on page 176.

OMG IDL

OMG IDL files define the IDL interfaces for CORBA objects. (See "Introduction to OMG IDL" on page 269 for more details.) As shown in Figure 20 on page 174, you can register OMG IDL in a CORBA Interface Repository, where it is stored in binary format. (See the Orbix documentation set for full details of how to register OMG IDL.)

COMet uses the OMG IDL type information available in the Interface Repository. The type information can consist of module names, interface names, or data types.

Memory and Disk Cache

A possible performance bottleneck might result at application runtime, if COMet needs to contact the Interface Repository for each OMG IDL definition. This is because every query might involve multiple remote invocations.

To avoid any bottlenecks, COMet uses a memory and disk cache of type information. The `typeman` utility converts OMG IDL type information into an ORB-neutral binary format, and caches it in memory. The use of a memory cache means that COMet has to query the Interface Repository only once for each OMG IDL definition.

Type Information Management

At application runtime, when COMet 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 that 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 a COMet 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 "COMet Configuration" on page 399.) 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 in the disk cache, `typeman` pulls it out of the Interface Repository. The related type information then becomes the most recent item in the queue in the type store memory cache.

The COMet Tools Window

Overview

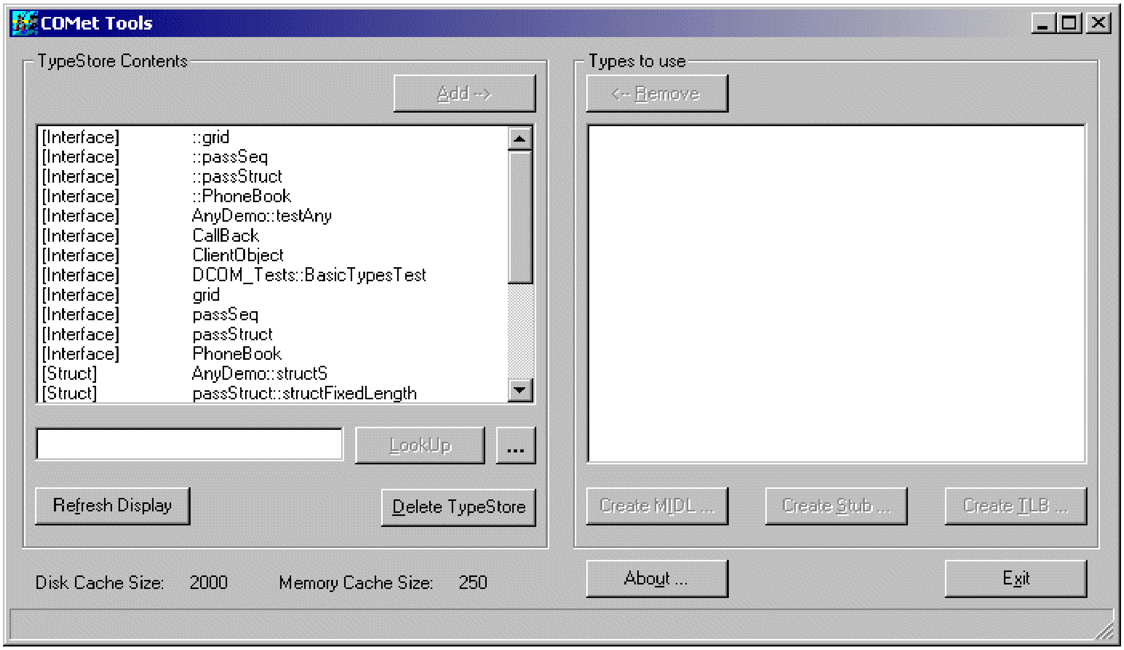
This section describes the **COMet Tools** window, which allows you to:

- Add new OMG IDL information to the type store.
- Delete the type store contents.
- Create Microsoft IDL from cached OMG IDL.
- Create Automation type libraries from cached OMG IDL.

Note: You can ignore this section if you intend using only the `typeman`, `ts2idl`, and `ts2tlb` utilities from the command line.

Window Layout

Figure 21 shows the layout of the **COMet Tools** window.



```
[Interface]      ::grid
[Interface]      ::passSeq
[Interface]      ::passStruct
[Interface]      ::PhoneBook
[Interface]      AnyDemo::testAny
[Interface]      CallBack
[Interface]      ClientObject
[Interface]      DCOM_Tests::BasicTypesTest
[Interface]      grid
[Interface]      passSeq
[Interface]      passStruct
[Interface]      PhoneBook
[Struct]         AnyDemo::structS
[Struct]         passStruct::structFixedLength
```

Disk Cache Size: 2000   Memory Cache Size: 250

Figure 21: *COMet Tools Window*

Opening the COMet Tools Window

To open the **COMet Tools** window, enter `cometcfg` on the command line, or select the **Configure COMet** icon in the **Orbix Configuration** panel on the **IONA Central** window. (You can open the **IONA Central** window by entering `itcentral` on the command line.) When you open the **COMet Tools** window, the **TypeStore Contents** panel automatically lists all the OMG IDL type information that is currently held in the type store cache.

Viewing Command-Line Changes

If you are using both the GUI and the `typeman` command-line utility to manage the type store, changes made via the `typeman` command-line utility do not appear automatically in the **TypeStore Contents** panel on the **COMet Tools** window, shown in Figure 21 on page 178. In this case, you must select **Refresh Display**, to allow the GUI tool to reflect any changes that were made via the command line.

Adding New Information to the Type Store

Overview

This section describes how to add new OMG IDL type information to the COMet type store, by using either the GUI tool or the `typeman` command-line utiilty.

"The Caching Mechanism of the Type Store" on page 176 describes 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 OMG IDL 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 a COMet application that is using previously unseen OMG IDL types. After COMet has obtained the type information from the Interface Repository, either through cache priming or during the first run of an application, all subsequent queries for that type information are satisfied by the cache.

In This Section

This section discusses the following topics:

Note: An OMG IDL interface must be registered in the Interface Repository before you can add it to the COMet type store. See the *CORBA Administrator's Guide* for more details about registering OMG IDL.

Using the GUI Tool

Overview

This subsection describes how to use the GUI tool to add OMG IDL type information to the type store.

Steps

The steps to add new information to the type store are:

Step	Action
1	Open the **COMet Tools** window shown in Figure 21 on page 178.
2	In the field beside the **LookUp** button, enter the name of an OMG IDL interface that you want to add. This enables the **LookUp** button.
3	Select the **LookUp** button. COMet searches both the Interface Repository and the type store cache 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.

Using the Command Line

Overview

This section describes how to use the typeman command-line utility to add OMG IDL type information to the type store. (See "COMet Utility Arguments" on page 411 for details of each of the arguments available with typeman.)

Example

The following command adds the grid interface to the type store:

```
typeman -e grid
```

Usage String

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.
```

Priming the Type Store with an Individual Entry

To prime the type store with the OMG IDL `mygrid` interface, enter:

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

In this case, the `-e` argument 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. This means you must precede the interface name with the module name (that is, `mymodule::` in the previous example).

Priming the Type Store with Multiple Entries

To prime the type store with multiple OMG IDL entries simultaneously, create a text file that lists any number of OMG IDL typenames. 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
Chat::ChatClient
Chat::ChatServer
```

As shown in the preceding example, OMG IDL typenames must be fully scoped (that is, precede the interface name with *`modulename`*`::`). 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.

Deleting the Type Store Contents

Overview

You can use either the GUI tool or the command-line utilities to delete the entire contents of the type store. 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 cache, select **Delete TypeStore** on the **COMet Tools** window shown in Figure 21 on page 178.

Using the Command Line

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 that the `typeman` data files are held in `c:\temp`. (The `COMet.TypeMan.TYPEMAN_CACHE_FILE` configuration variable determines where the data files are stored. See "COMet Configuration" on page 399 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 specify `typeman -wm` to ensure that this file is also deleted.

Dumping the Type Store Contents

Overview

The typeman utility is also a useful diagnostic utility, because it allows you to dump the contents of the type store cache.

Example

The following command prints the methods of the grid interface in both alphabetic and vtable order (the vtable order is determined by the OMG *Interworking Architecture* specifiction at ftp://ftp.omg.org/pub/docs/formal/01-12-55.pdf):

```
[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 operations for the get attribute. The absence of height set and width set implies that these are readonly attributes.

Creating a Microsoft IDL File

Overview

The normal procedure for writing a COM or Automation client to communicate with a CORBA server is to first obtain a Microsoft IDL definition of the target CORBA interface, which the COM or Automation client can understand. You can generate Microsoft IDL definitions from existing OMG IDL information in the type store. To minimize manual lookups, you should ensure that each IDL file contains a module.

Note: Creating Microsoft IDL in this way allows you to create a standard DCOM proxy/stub DLL that can be installed with a COM or Automation client. This means that you do not have to install any CORBA components on the client machine. In this case, the distribution model is exactly the same as for a standard DCOM application. This means that it includes a COM or Automation client and a proxy/stub DLL.

In This Section

This section discusses the following topics:

Using the GUI Tool

Overview

This subsection describes how to use the GUI tool to create a Microsoft IDL file from OMG IDL.

Steps

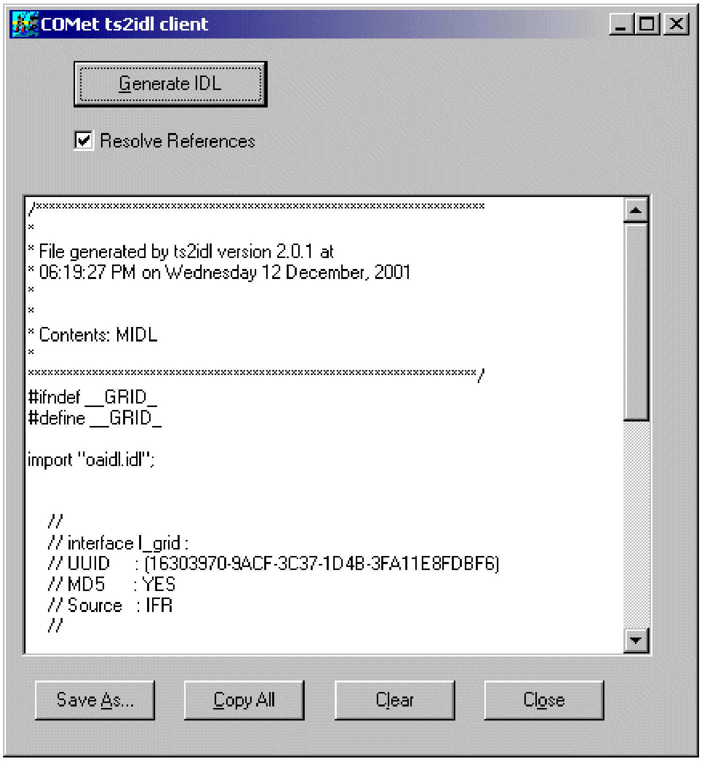
The steps to create a Microsoft IDL file are:

1. Open the **COMet Tools** window shown in Figure 21 on page 178.
2. From the **TypeStore Contents** panel, select the item of OMG IDL type information on which you want to base the Microsoft IDL file.
3. Select **Add**. 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 Microsoft IDL file.
4. Select **Create MIDL**. This opens the **COMet ts2idl** client window shown in Figure 22 on page 188.
5. If you want to:
   - Ensure that Microsoft 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 Microsoft IDL file to your development environment, select the **Copy All** button.
   - Refresh the window, select the **Clear** button.
   - Assign a Microsoft IDL filename, select the **Save As** button.
6. Select the **Generate IDL** button. This creates the Microsoft IDL file.

COMet ts2idl Client Window

Figure 22 shows the **COMet ts2idl client** window, which you can use to create a Microsoft IDL file from OMG IDL.



```
/**************************************************************************
*
* File generated by ts2idl version 2.0.1 at
* 06:19:27 PM on Wednesday 12 December, 2001
*
*
* Contents: MIDL
*
**************************************************************************/
#ifndef __GRID_
#define __GRID_

import "oaidl.idl";


   //
   // interface I_grid :
   // UUID     : {16303970-9ACF-3C37-1D4B-3FA11E8FDBF6}
   // MD5      : YES
   // Source   : IFR
   //
```

Figure 22: *Creating a Microsoft IDL File from OMG IDL*

Using the Command Line

Overview

This subsection describes how to use the ts2idl command-line utility to create a Microsoft IDL file from existing OMG IDL type information. (See "COMet Utility Arguments" on page 411 for details of each of the arguments available with ts2idl.)

Example

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

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

Usage String

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:
   -c : Don't connect to the IFR (e.g. if cache is fully primed).
   -r : Resolve referenced types.
   -m : Generate Microsoft 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.
```

Creating a Type Library

Overview

When using an Automation client, you have the option in some controllers (for example, Visual Basic) of using straight `IDispatch` interfaces or dual interfaces.

Using IDispatch Interface

If you want to develop an Automation client that is to only use the straight `IDispatch` interface, there is no need to create a type library from existing OMG IDL information in the type store. This is because COMet automatically copies the related type information into the type store when it uses `GetObject` to perform a lookup on the target CORBA object.

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")
```

Using Dual Interfaces

If you want to develop an Automation client that uses dual interfaces, instead of using the straight `IDispatch` interface, you must use either the GUI tool or the `ts2tlb` command-line utility to create a type library from existing OMG IDL information in the type store.

Note: If you intend to use dual interfaces, the bridge must be loaded in-process to the client (that is, in the client's address space). The use of dual interfaces is not supported with the bridge loaded out-of-process.

Using the GUI Tool

Overview

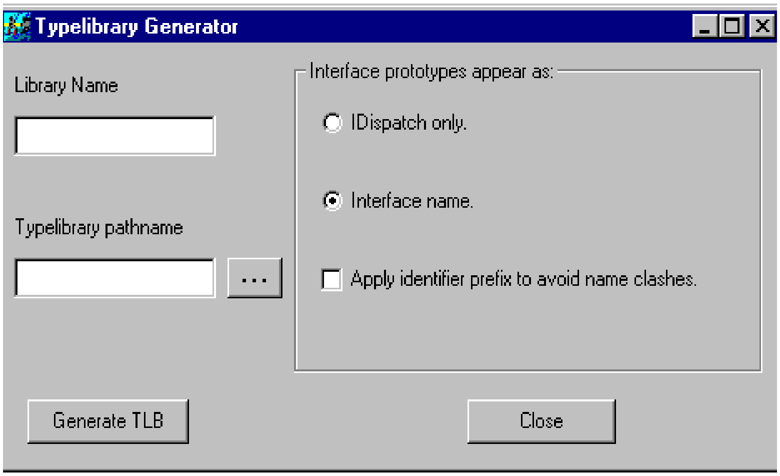
This subsection describes how to use the GUI tool to create a type library from OMG IDL.

Steps

The steps to create a type library are:

1. Open the **COMet Tools** window shown in Figure 21 on page 178.
2. From the **TypeStore Contents** panel, select an item of OMG IDL type information on which you want to base the type library.
3. Select **Add**. 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.
4. Select Create TLB. This opens the Typelibrary Generator window shown in Figure 23.



Interface prototypes appear as:

○ IDispatch only.

● Interface name.

□ Apply identifier prefix to avoid name clashes.

Figure 23: *Creating a Type Library from OMG IDL*

5. In the **Library Name** field, type the internal library name. This can be the same as the type library path name if you wish, but ensure that the library does not have the same name as any of the types that it contains.
6. In the **Typelibrary pathname** field, type the full path name for the type library.
7. If you want interface prototypes to:
   - Appear as `IDispatch`, select **IDispatch only**.
   - Use the specific interface name, select **Interface name**.
8. 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 Microsoft IDL keywords.
9. Click **Generate TLB**. This creates the type library.

Using the Command Line

Overview

This subsection describes how to use the ts2tlb command-line utility to create a type library from existing OMG IDL type information. (See "COMet Utility Arguments" on page 411 for details of each of the arguments available with ts2tlb.)

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
```

Usage String

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.
```

Creating Stub Code for Client Callbacks

Overview

When you want your application to have client callback functionality, you must provide an implementation for the callback objects. This section describes how to use the GUI tool to generate Visual Basic or PowerBuilder stub code for callbacks.

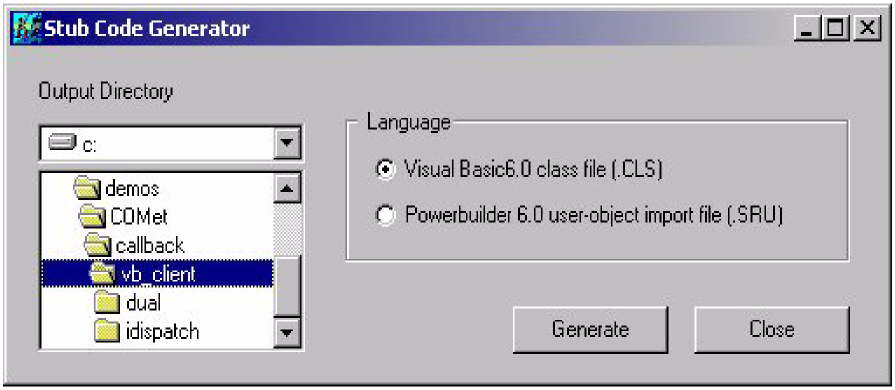
Note: There is no equivalent command-line utility available for creating stub code for callbacks.

Steps

The steps to create stub code for callbacks are:

1. Open the **COMet Tools** window shown in Figure 21 on page 178.
2. From the **TypeStore Contents** panel, select the item of OMG IDL type information on which you want to base the stub code.
3. 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 stub code.
4. Select the **Create Stub** button. This opens the **Stub Code Generator** window shown in Figure 24.



Stub Code Generator

Output Directory

c:

- demos
- COMet
- callback
- vb_client
- dual
- idispatch

Language

● Visual Basic6.0 class file (.CLS)

○ Powerbuilder 6.0 user-object import file (.SRU)

Figure 24: *Creating Stub Code for Callbacks*

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

Replacing an Existing DCOM Server

Overview

This section describes the concept of replacing an existing DCOM server with a CORBA server, and how to do it. The following topics are discussed:

- "Background" on page 196.
- "The srvAlias Utility" on page 196.
- "The Server Aliasing Registry Editor Window" on page 197.
- "Using the Window" on page 197.
- "The aliassrv Utility" on page 198.

Background

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.

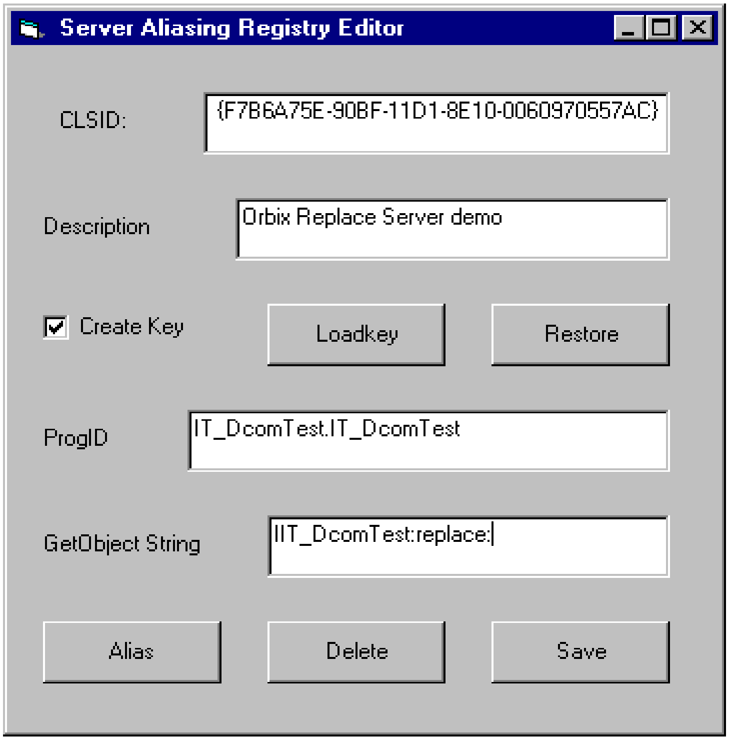
The solution is to allow such clients to continue to use their normal `CoCreateInstanceEx()` or `CreateObject()` calls. This means that you must retrofit the bridge to serve these clients' activation requests. In other words, you must alias the bridge to the legacy COM or Automation server. This ensures that when the client is subsequently run, the bridge is activated in response to the client's `CoCreateInstanceEx()` or `CreateObject()` calls.

The srvAlias Utility

COMet supplies a `srvAlias` utility, which you can enter at the command line, to open the **Server Aliasing Registry Editor** window shown in Figure 25 on page 197.

The Server Aliasing Registry Editor Window

Figure 25 shows the layout of the **Server Aliasing Registry Editor** window, which you can open by running srvAlias from the command line.



CLSID: {F7B6A75E-90BF-11D1-8E10-0060970557AC}

Description: Orbix Replace Server demo

☑ Create Key

ProgID: IT_DcomTest.IT_DcomTest

GetObject String: IIT_DcomTest:replace:

Figure 25: *Aliasing the Bridge*

Using the Window

The **Server Aliasing Registry Editor** window 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 should 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 aliassrv Utility

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 COMet:

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

The next time a DCOM client of the server is run, COMet is used instead.

Generating Visual Basic Client Code

Overview

This section describes how to use the Visual Basic genie, to generate Visual Basic client code from OMG IDL definitions.

In This Section

This section discusses the following topics:

Introduction

Overview

This subsection provides an introduction to the concept of using the genie to generate Visual Basic client code. The following topics are discussed:

- "Introduction to the Genie" on page 200.
- "Development Steps" on page 200.
- "Generated Files" on page 201.

Introduction to the Genie

COMet is shipped with a Visual Basic code generation genie that can automatically generate the bulk of the application code for a Visual Basic client, based on OMG IDL definitions. The use of the Visual Basic genie is not compulsory for creating Visual Basic clients with COMet. However, using the genie makes the development of Visual Basic clients much faster and easier.

The genie is designed to generate Visual Basic clients. These clients can communicate with C++ servers that have been generated via the C++ genie supplied with the CORBA Code Generation Toolkit. (See the *CORBA Code Generation Toolkit Guide* for more details about the C++ genie.)

Development Steps

The steps to create and build a distributed COMet client-server application via code generation are:

Step	Action
1	Generate the CORBA server code, by using the `idlgen cpp_poa_genie` supplied with the CORBA Code Generation Toolkit. See the *CORBA Code Generation Toolkit Guide* for more details.
2	Generate the Visual Basic client, by using the `idlgen vb_genie` supplied with COMet. The following subsections describe how to use either the command-line or GUI version of the genie to do this. See "Generated Files" on page 201 for a list of the files that the Visual Basic genie creates.

Step	Action
3	Register the OMG IDL file with the Orbix Interface Repository. This step is only required if using the command-line version of the genie.
4	Load the `client.vbp` file into the Visual Basic IDE, and build the client.

Generated Files

The files that the Visual Basic genie creates are:

`client.vbp`	This is the Visual Basic project file for the client.
`client.frm`	This is the main Visual Basic form for the client.
`FindIOR.frm`	This form contains the functions needed by the client to select a `.ref` file. The `.ref` file is written by the server and contains the server object's IOR.
`Call_Funcs.bas`	This contains the Visual Basic code for implementing the operations defined in the selected interface(s).
`Print_Funcs.bas`	This contains functions for printing the values of all the CORBA simple types supported by COMet. It also contains functions for printing any user-defined types declared in the IDL file.
`Random_Funcs.bas`	This contains functions for generating random values for all the CORBA simple types supported by COMet. It also contains functions for generating random values for any user-defined types declared in the IDL file.
`IT_Random.cls`	This class is a random number generator that is used in the generated `Random_Funcs.bas` file.

Using the GUI Tool

Overview

This subsection describes the steps to use the GUI tool to generate Visual Basic client code from existing OMG IDL are:

1 From the **Visual Basic** project dialog shown in Figure 26, select the **COMet Wizard** icon.

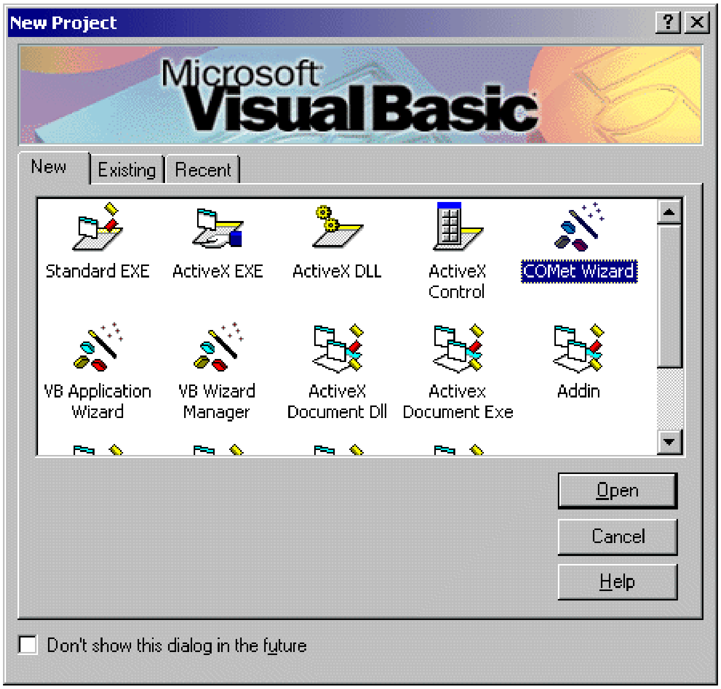
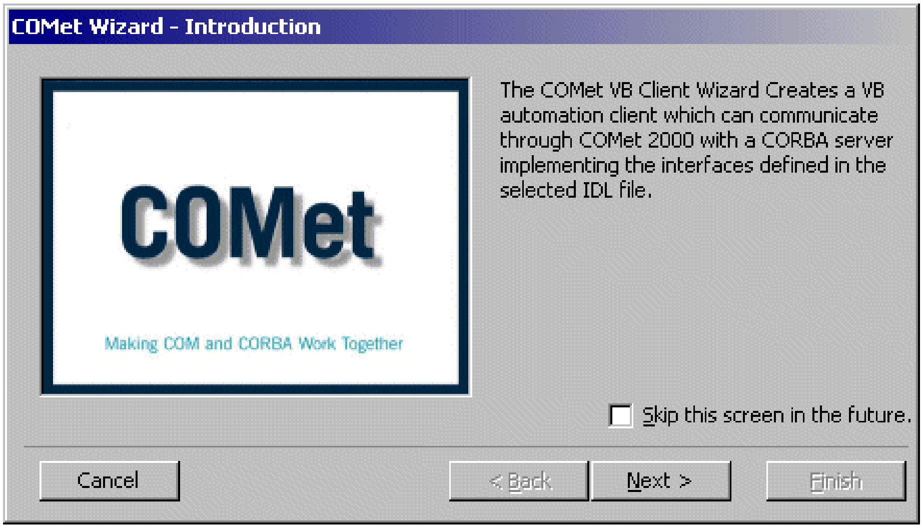


Figure 26: *Visual Basic Project Dialog Window*

This opens the **COMet Wizard** Introduction window shown in Figure 27 on page 203.

2 Select the **Next** button on the **COMet Wizard - Introduction** window shown in Figure 27.


COMet Wizard - Introduction
COMet
Making COM and CORBA Work Together

The COMet VB Client Wizard Creates a VB automation client which can communicate through COMet 2000 with a CORBA server implementing the interfaces defined in the selected IDL file.

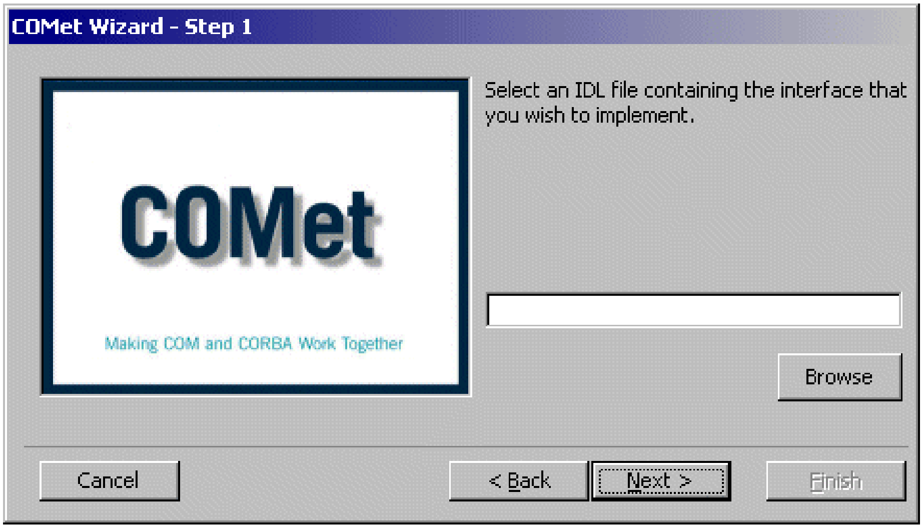
□ Skip this screen in the future.

Cancel
< Back
Next >
Finish

Figure 27: *COMet Wizard - Introduction Window*

This opens the **COMet Wizard - Step 1** window shown in Figure 28 on page 204.

3 Select the **Browse** button on the **COMet Wizard - Step 1** window in Figure 28.



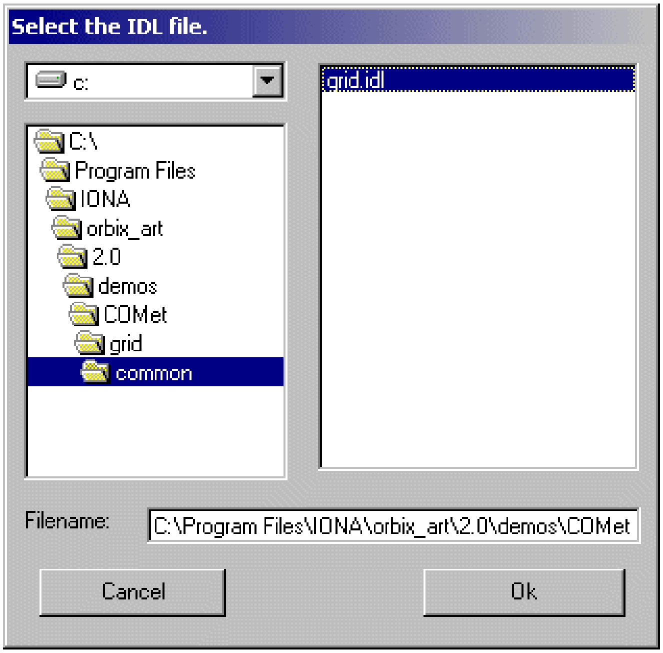
COMet Wizard - Step 1

Select an IDL file containing the interface that you wish to implement.

Figure 28: *COMet Wizard - Step 1 Window*

This opens the **Select the IDL file** window shown in Figure 29 on page 205.

4 From the **Select the IDL file** window in Figure 29, select the OMG IDL file on which you want to base the Visual Basic client.



Filename: C:\Program Files\IONA\orbix_art\2.0\demos\COMet

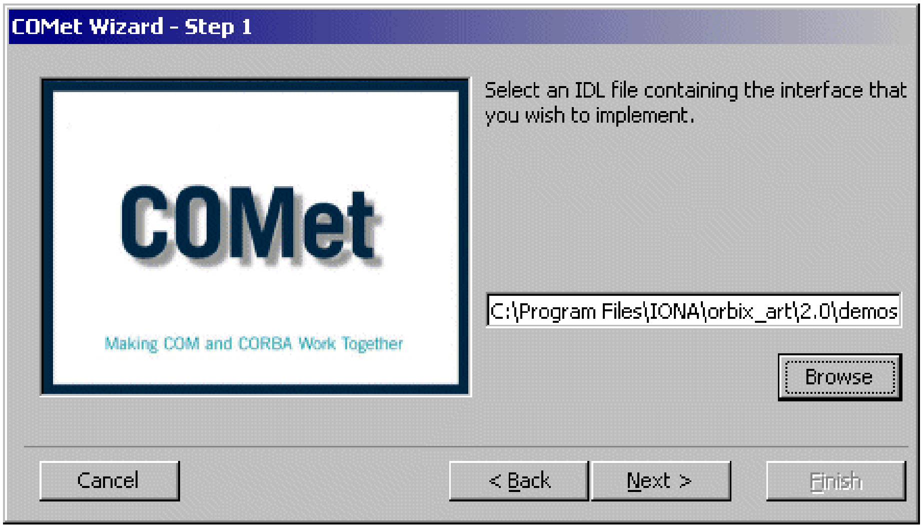
Figure 29: *Select the IDL File Window*

The **Filename** field displays the full path to the OMG IDL file that you select.

5 Select the **Ok** button on the **Select the IDL file** window.

This opens the **COMet Wizard - Step 1** window again, this time with the full path to the selected OMG IDL file displayed, as shown in Figure 30 on page 206.

6 Select the **Next** button on the **COMet Wizard - Step 1** window in Figure 30.



COMet Wizard - Step 1

Select an IDL file containing the interface that you wish to implement.

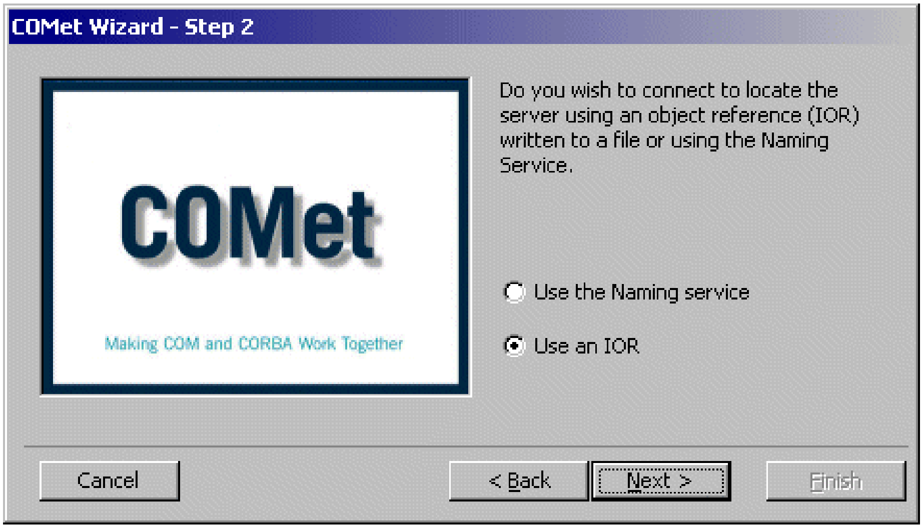
C:\Program Files\IONA\orbix_art\2.0\demos

Figure 30: *Step 1 Window Displaying Full Path to the Selected File*

This opens the **COMet Wizard - Step 2** window shown in Figure 31 on page 207.

7 Select the appropriate radio button on the **COMet Wizard - Step 2** window in Figure 31, depending on whether you want to connect to the server by using an IOR or the Naming Service.

Note: The option you choose must correspond with the option selected for the C++ server when it was created via the CORBA Code Generation Toolkit.



COMet Wizard - Step 2

Do you wish to connect to locate the server using an object reference (IOR) written to a file or using the Naming Service.

○ Use the Naming service

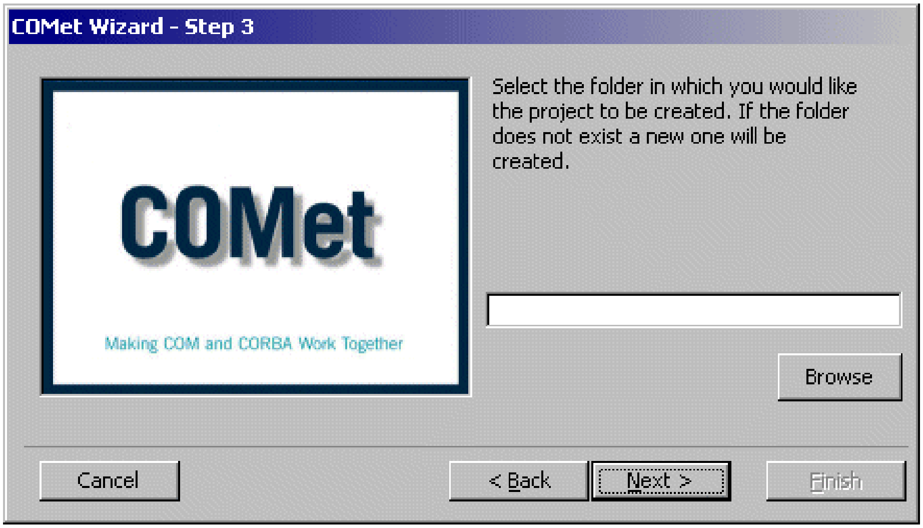
● Use an IOR

Figure 31: *COMet Wizard - Step 2 Window*

8 Select the **Next** button on the **COMet Wizard - Step 2** window.

This opens the **COMet Wizard - Step 3** window shown in Figure 32 on page 208.

9 Select the **Browse** button on the **COMet Wizard - Step 3** window in Figure 32.



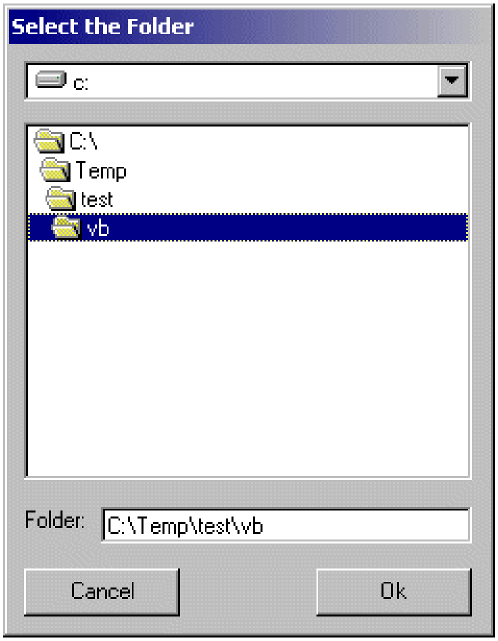
COMet Wizard - Step 3

Select the folder in which you would like the project to be created. If the folder does not exist a new one will be created.

Figure 32: *COMet Wizard - Step 3 Window*

This opens the **Select the Folder** window shown in Figure 33 on page 209.

10 From the **Select the Folder** window in Figure 33, select the path to the folder in which you want to store your Visual Basic client project.



Folder: C:\Temp\test\vb

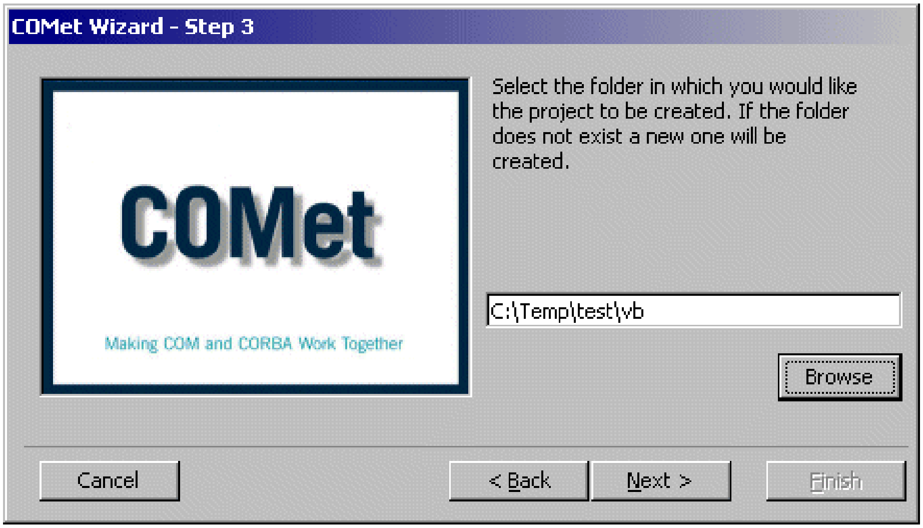
Figure 33: *Selecting a Folder*

The **Folder** field displays the full path to the folder that you select.

11 Select the **Ok** button on the **Select the Folder** window.

This opens the **COMet Wizard - Step 3** window again, this time with the full path to the selected folder displayed, as shown in Figure 34 on page 210.

12 Select the **Next** button on the **COMet Wizard - Step 3** window in Figure 34.



COMet Wizard - Step 3

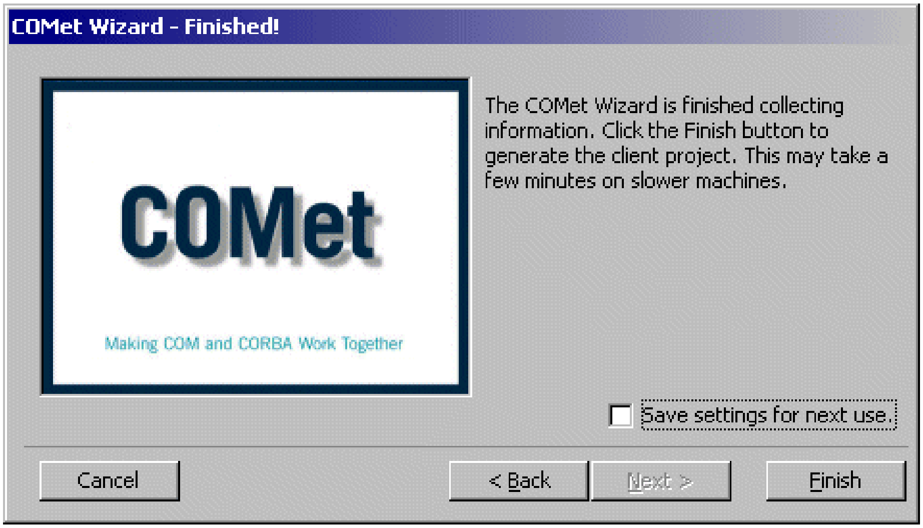
Select the folder in which you would like the project to be created. If the folder does not exist a new one will be created.

C:\Temp\test\vb

Figure 34: *Step 3 Window Displaying Full Path to the Selected Folder*

This opens the **COMet Wizard - Finished** window shown in Figure 35 on page 211.

13 Select the **Finish** button on the **COMet Wizard - Finished** window in Figure 35.



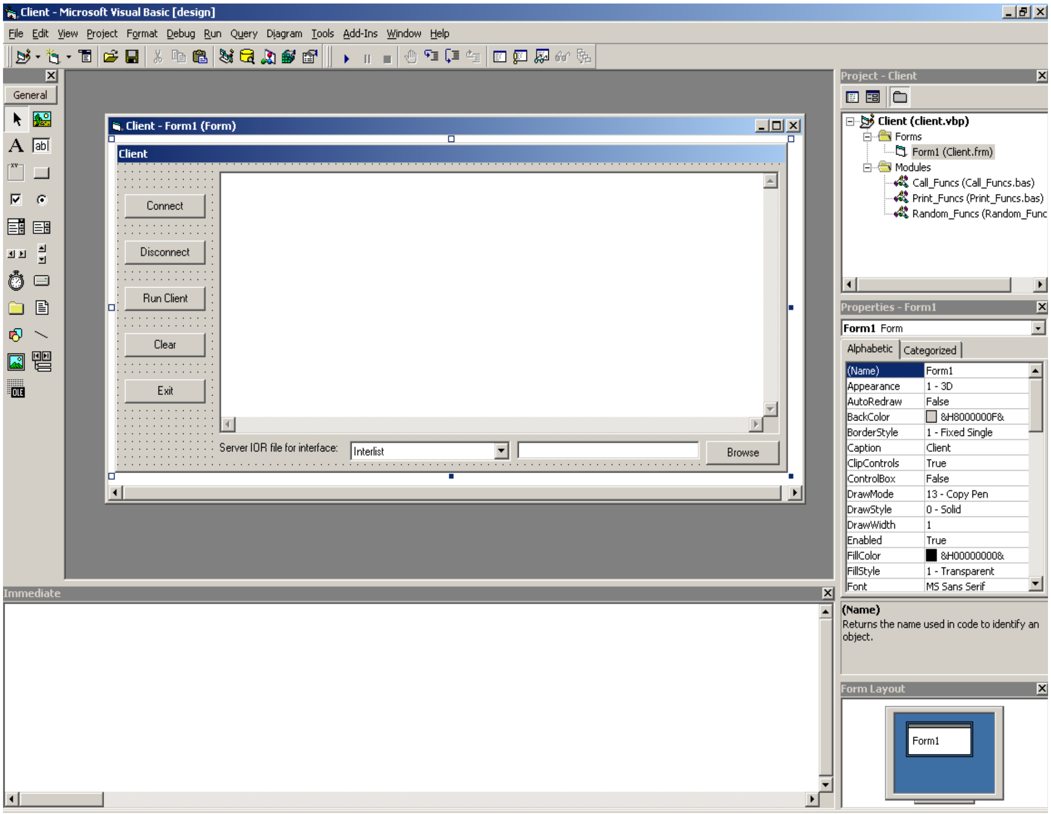
The COMet Wizard is finished collecting information. Click the Finish button to generate the client project. This may take a few minutes on slower machines.

□ Save settings for next use.

Figure 35: *COMet Wizard - Finished Window*

This automatically generates the Visual Basic client project for you. It also automatically registers the corresponding OMG IDL file in the Interface Repository.

When the genie has completed its processing, the generated client application appears, as shown in Figure 36 on page 212.



Client (client.vbp)
- Forms
  - Form1 (Client.frm)
- Modules
  - Call_Funcs (Call_Funcs.bas)
  - Print_Funcs (Print_Funcs.bas)
  - Random_Funcs (Random_Func

(Name)	Form1
Appearance	1 - 3D
AutoRedraw	False
BackColor	&H8000000F&
BorderStyle	1 - Fixed Single
Caption	Client
ClipControls	True
ControlBox	False
DrawMode	13 - Copy Pen
DrawStyle	0 - Solid
DrawWidth	1
Enabled	True
FillColor	&H00000000&
FillStyle	1 - Transparent
Font	MS Sans Serif

(Name)
Returns the name used in code to identify an object.

Figure 36: *Example of a Generated Client Application*

Using the Command Line

The `idlgen vb_genie` utility can create the bulk of a Visual Basic client application from existing OMG IDL definitions. The command-line syntax for the genie is as follows, where *`filename`* represents the name of the OMG IDL file:

```
idlgen vb_genie.tcl [options] filename.idl [interface wildcard]*
```

You can generate a Visual Basic client, based on any of the following:

- All interfaces in an OMG IDL file.

  For example, the following command creates a Visual Basic client, based on all the interfaces contained in the `grid.idl` file:

  ```
  idlgen vb_genie.tcl grid.idl *
  ```

- A specific interface in an OMG IDL file.

  For example, the following command creates a Visual Basic client, based on the `grid` interface in the `grid.idl` file:

  ```
  idlgen vb_genie.tcl modulename::grid grid.idl
  ```

  In this case, you must supply the fully scoped name of the interface (that is, the interface name preceded by module name and `::`).

- A range of selected interfaces in an OMG IDL file, by using wildcard characters.

  For example, the following command creates a Visual Basic client, based on all interfaces in `foo.idl` that are within the `Test` module, and which have names that begin with `Foo` or end with `Bar`:

  ```
  idlgen vb_genie.tcl Test::* foo.idl "Foo*" "*Bar"
  ```

Note: Remember that the command-line version of the genie does not automatically register OMG IDL in the Interface Repository. You must do this manually after the genie has created the Visual Basic client application. For example, to register the OMG IDL in a file called `grid.idl`, enter the command `idl -R=-v grid.idl`.

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

```
idlgen vb_genie -h
```

The usage string for the genie is:

```
usage: idlgen vb_genie.tcl [options] file.idl [interface
   wildcard]*
[options]

-I<directory>        Passed to preprocessor.
-D<name>[=value]     Passed to preprocessor.
-h                   Prints this help message.
-v                   Verbose mode.
-s                   Silent mode (opposite of -v option).
-dir <directory>     Put generated files in the specified
                     directory.
-include             Process interfaces in #include'd file too.
-(no)ns              Use the Naming Service (default no).
```

See “Idlgen vb_genie.tcl Arguments” on page 419 for details of each of the arguments available with the genie.

Part 3

Programmer's Reference

In This Part

This part contains the following chapters:

CHAPTER 11

COMet API Reference

This chapter describes the application programming interface (API) for COMet, which is defined in Microsoft IDL.

In This Chapter

This chapter discusses the following topics:

Common Interfaces

Overview

This section describes the interfaces that are common to both COM and Automation.

In This Section

This section discusses the following topics:

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

Mapping object references through views, and passing those object references back and forth through the bridge, could potentially lead to the creation of indefinitely long chains of views that delegate to other views, and so on indefinitely. The `IForeignObject` interface is provided as a deterrent to this potential problem, in that it provides a mechanism to extract a valid CORBA object reference from a view.

To effect this solution, each COM and Automation view object must map onto one and only one CORBA object reference, and it must also expose the `IForeignObject` interface. This in turn means that an unambiguous CORBA object reference can be obtained via `IForeignObject` from any COM or Automation view.

Note: The matching Automation interface for a constructed OMG IDL type (such as struct, union, or exception) exposes `DIForeignComplexType` instead of `IForeignObject`.

Methods

The methods for the `IForeignObject` interface are:

`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

The COM standard does not provide any mechanism for clients to deal with server objects that are inherently persistent (that is, server objects that store their own state instead of having their state stored through an external interface such as `IPersistStorage`). Databases are a typical example of inherently persistent server objects. COM does have the concept of monikers, which are the conceptual equivalent of CORBA persistent object references, but they are used in only a limited capacity in the COM world.

The `IMonikerProvider` interface allows clients to obtain an `IMoniker` interface pointer from COM and Automation views. The resulting moniker can be used as a persistent reference to the CORBA object that relates to the view from which the moniker was obtained.

Both COM and Automation views can support the `IMonikerProvider` interface. It allows clients to persistently save object references for later use, without needing to keep the view in memory.

Methods

The methods for the `IMonikerProvider` interface are:

`get_moniker()`	This returns a moniker that allows the CORBA object to be converted to persistent form for storage in a file, and so on. Once it is stored in persistent form, by means of this moniker, the CORBA object can be reconnected to again, by using the standard COM moniker semantics.

UUID

`{ecce76fe-39ce-11cf-8e92-080000970dac7}`

Notes

COM/CORBA-compliant.

Automation-Specific Interfaces

Overview

This section describes the interfaces that are specific to Automation.

In This Section

This section discusses the following topics:

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);

//  tk_array, tk_sequence
```

```
    HRESULT insert_safearray([in] VARIANT val, [in] BSTR
       typeName);
};
```

Description

The OMG IDL any type maps to the DICORBAAny Automation interface. You can use DICORBAAny to find details about the type of value stored by an any. The particular methods that you can call on DICORBAAny depend on the kind of value it contains. A BadKind exception is raised if a method is called on DICORBAAny that is not appropriate to the kind of value it contains.

You can use the kind() method to find the kind of value contained. The kind() 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.

Because DICORBAAny derives from the DIForeignComplexType interface, objects that implement it are effectively pseudo-objects.

If the any contains a CORBA sequence or array type, the VARIANT value property contains an Automation safearray or an OLE collection. If the any contains a complex CORBA type, such as a struct or union, the VARIANT value property contains an IDispatch pointer to the Automation interface to which that type is mapped.

Methods

The methods for the DICORBAAny interface are:

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 kind. 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 kind. If called on a DICORBAAny of a different kind, it raises a BadKind exception.

It 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.

It 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.

It 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, it returns the value of the bound; a return value of 0 indicates an unbounded string or sequence. For an array, it 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 a `DICORBAAny` of a different kind, it raises a `BadKind` exception. For a sequence or array, it returns the type of element contained in the sequence or array. For an alias, it returns the type aliased by the typedef definition.
`insert_safearray()`	This can be called on a `DICORBAAny` of the `tk_sequence` or `tk_array` kind. If called on a `DICORBAANY` of a different kind, it raises a `BadKind` exception. This is used to insert sequences or arrays into anys. The typename of the sequence or array must be supplied along with the array itself.

UUID

`{A8B553C4-3B72-11CF-BBFC-444553540000}`

Notes

Automation/CORBA-compliant.

DICORBAFactory

Synopsis

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

Description

The DICORBAFactory interface is used to make CORBA objects available to Automation clients, in a manner that is similar to the GetActiveObject method in Automation (already described in "COM and CORBA Principles" on page 3). It is a factory class that allows an Automation client to create new CORBA object instances and bind to existing CORBA objects. It is designed to be similar to the Visual Basic CreateObject and GetObject functions.

The Automation/CORBA-compliant ProgID for this class is CORBA.Factory. An instance of this class must be registered in the Windows system registry on the client machine.

In COMet, the name CORBA.Factory.Orbix is also registered as an alias for CORBA.Factory. This allows access to the Orbix instance in the event of a subsequent installation of an ORB other than Orbix.

Methods

The methods for the `DICORBAFactory` interface are:

`GetObject()`

This allows a client to specify the name of a target object to which it wants to connect. It creates an Automation view of the specified target object, binds this view to the target, and provides the client with a reference to the view, which the client can then use to makes its requests.

The `objectName` parameter specifies the target CORBA object to which the client wants to connect. In COMet, the format of this parameter is as follows:

```
"interface:TAG:Tag data"
```

The *`interface`* component represents the IDL interface that the target object supports. If the interface is scoped (for example, "*`module_name`*`::`*`interface_name`*"), the interface token is "*`module_name/interface_name`*".

The *`TAG`* component can be either of the following:

- `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.
      staff.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 that are instances of CORBA complex types, such as structs, unions, and exceptions.

`DICORBAFactoryEx` derives from `DICORBAFactory`. You can call `DICORBAFactoryEx` methods on an instance of `DICORBAFactory`.

Methods

The methods for `DICORBAFactoryEx` are:

`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 by calling `DIForeignComplexType::INSTANCE_repositoryID()`.

This method requires runtime access to the Interface Repository.

UUID

`{A8B553C5-3B72-11CF-BBFC-444553540000}`

Notes

Automation/CORBA-compliant. There is no corresponding `ICORBAFactoryEx` COM API, because CORBA structures map to native COM structures.

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 Automation views of CORBA objects expose the `DICORBAObject` interface. It provides a number of Automation/CORBA-compliant methods that all CORBA (and hence, Orbix) objects support.

An Automation client must call `DIORBObject::GetCORBAObject()`, to obtain an `IDispatch` pointer to the `DICORBAObject` interface.

Methods

The methods for the `DICORBAObject` interface are:

`GetInterface()`	This returns a reference to an object in the Interface Repository that provides type information about the target object. This method requires runtime access to the Interface Repository.
`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() 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 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 method is 0).

UUID {204F6244-3AEC-11CF-BBFC-444553540000}

Notes Automation/CORBA-compliant.

DICORBAStruct

Synopsis

```
[oleautomation,dual,uuid(…)]
interface DICORBAStruct : DIForeignComplexType {};
```

Description

The `DICORBAStruct` interface is used to show that an Automation interface has been translated from an OMG IDL struct definition. Any Automation interface that results from the translation of an OMG IDL struct supports `DICORBAStruct`.

`DICORBAStruct` derives from the `DIForeignComplexType` interface. It has no associated methods.

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

The `DICORBASystemException` interface is used to show that an Automation interface has been translated from a CORBA system exception. (CORBA system exceptions are not defined in OMG IDL.) Any Automation interface that results from the translation of a CORBA system exception supports `DICORBASystemException`, which in turn derives from `DIForeignException`.

Methods

The methods for the `DICORBASystemException` interface are:

`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` This indicates that the operation had completed before the exception was raised.
- `COMPLETION_NO = 1` This indicates that the operation had not completed before the exception was raised.
- `COMPLETION_MAYBE = 2` This indicates that 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

The DICORBATypeCode interface is used to show that an Automation interface has been translated from an OMG IDL typecode definition. Any Automation interface that results from the translation of an OMG IDL typecode supports DICORBATypeCode, which in turn derives from DIForeignComplexType.

Methods

The methods for the `DICORBATypeCode` interface are:

`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 `CORBA_TCKind` 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 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 Interface Repository ID that globally identifies the type.

  This method requires runtime access to the Interface Repository.

`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.

It 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 specified in 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 specified in 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, it returns the bound value. A return value of `0` indicates an unbounded string or sequence. For an array, it 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, it 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

The `DICORBAUnion` interface is used to show that an Automation interface has been translated from an OMG IDL union definition. Any Automation interface that results from the translation of an OMG IDL union supports `DICORBAUnion`, which in turn derives from `DIForeignComplexType`.

Methods

The methods for the `DICORBAUnion` interface are:

`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

The `DICORBAUserException` interface is used to show that an Automation interface has been translated from an OMG IDL user-defined exception. Any Automation interface that results from the translation of an OMG IDL user-defined exception supports `DICORBAUserException`, which in turn derives from `DIForeignException`. `DICORBAUserException` has no associated methods.

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

The `DIForeignComplexType` interface is used to show that an Automation interface has been translated from an OMG IDL complex type (for example, a struct, union, or exception). Any Automation interface that results from the translation of an OMG IDL complex type supports `DIForeignComplexType`.

The interfaces that derive from `DIForeignComplexType` are `DICORBAAny`, `DICORBAStruct`, `DICORBATypeCode`, `DICORBAUnion`, and `DIForeignException` (that is, the matching Automation interface for any CORBA constructed type).

Methods

The methods for the `DIForeignComplexType` interface are:

`INSTANCE_repositoryId()` This returns the repository ID of a complex type. The `DICORBAFactoryEx::CreateTypeById()` method can subsequently use the repository ID to create an instance of a complex type, based on the repository ID.

`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 to use `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

The DIForeignException interface is used to show that an Automation interface has been translated from either an OMG IDL user-defined exception or a CORBA system exception. Any Automation interface that results from the translation of either an OMG IDL user-defined or system exception supports DIForeignException.

The interfaces that derive from DIForeignException are DICORBASystemException and DICORBAUserException The DIForeignException interface in turn derives from DIForeignComplexType.

Methods

The methods for the DIForeignException interface are:

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

The `DIObject` interface is the object wrapper for the OMG IDL `Object` type. It has no associated methods.

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

The `DIObjectInfo` interface allows you to retrieve information about a complex data type (such as a union, structure, or exception) that is held as an `IDispatch` pointer. It derives from the `DICORBAFactoryEx` interface..

Note: The recommended way to obtain a repository ID is to call `DIObjectInfo::unique_id()`, followed by `DIObjectInfo::clone()`.

Methods

The methods for the `DIObjectInfo` interface are:

`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.

DIOrbixORBObject

Synopsis

```
[oleautomation,dual,uuid(…)]
interface DIOrbixORBObject : DIORBObject {
    HRESULT GetConfigValue([in] BSTR name, [out] BSTR *value,
        [in, out, optional] VARIANT *IT_Ex,
        [retval, out] VARIANT_BOOL * IT_retval);
    HRESULT StartUp([in, out, optional] VARIANT *IT_Ex,
        [retval, out] VARIANT_BOOL * IT_retval);
    HRESULT ShutDown([in, out, optional] VARIANT *IT_Ex,
        [retval, out] VARIANT_BOOL * IT_retval);
    HRESULT RunningInIDE([in, out, optional] VARIANT *IT_Ex,
        [retval, out] VARIANT_BOOL * IT_retval);
    HRESULT ReleaseCORBAView([in] IDispatch* poObj,
        [in] VARIANT_BOOL 1ToDestruction,
        [in, out, optional] VARIANT* IT_Ex,
        [retval, out] VARIANT_BOOL * IT_retval);
    HRESULT ProcessEvents([in, out, optional] VARIANT* IT_Ex,
        [retval, out] VARIANT_BOOL * IT_retval);
    HRESULT Narrow([in] IDispatch* poObj,
        [in] BSTR cNewIFaceName,
        [in, out, optional] VARIANT* IT_Ex,
        [out, retval] IDispatch** poDerivedObj);
    HRESULT SetOrbName([in] BSTR strOrbName,
        [in, out, optional] VARIANT* IT_Ex,
        [out, retval] VARIANT_BOOL* IT_retval);
};
```

Description

The `DIOrbixORBObject` interface provides Orbix-specific methods that allow you to control some aspects of the ORB (that is, Orbix) or to request it to perform actions. `DIOrbixORBObject` derives from `DIORBObject`. The `DIOrbixORBObject` methods augment the Automation/CORBA-compliant methods defined in `DIORBObject`.

The ORB has the `CORBA.ORB.2` ProgID, which is the Automation/CORBA-compliant name. In COMet, the `CORBA.ORB.Orbix` name is registered as an alias for `CORBA.ORB.2`. This allows access to the Orbix instance in the event of a subsequent installation of an ORB other than Orbix.

Methods

The methods for the `DIOrbixORBObject` interface are:

`GetConfigValue()` This obtains the value of the configuration entry specified in the `name` parameter.

See the Orbix documentation set for information on configuration values.

`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, for example, you are experiencing hang-on-exit problems or the `COMet.Config.COMET_SHUTDOWN_POLICY` configuration variable is set to `Disabled`. After this method is called, no more invocations can be made using CORBA.

`RunningInIDE()` This changes the internal shutdown policy, so COMet can run in the Visual Basic studio debugger. This call has no effect on the `COMet.Config.COMET_SHUTDOWN_POLICY` configuration variable.

`ReleaseCORBAView()` This is used by clients to free the CORBA view of a DCOM callback object when receipt of callbacks is no longer required.

`ProcessEvents()` 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.

If you want to use this method, set the `COMet.Config.SINGLE_THREADED_CALLBACK` configuration variable to `YES`.

`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 = obj.Fact.GetObject("A:" + ior)
Set cObj = orb.Narrow(aObj, "C")
If cObj Is Nothing Then
    MsgBox "Error: narrow failed"
End If
```

`SetOrbName()`

Every ORB is associated with a configuration domain that provides it with configuration information. A single configuration domain can hold configuration information for multiple ORBs, with each ORB using its ORB name as a "key" or configuration scope in which the particular configuration information relating to that ORB is located.

This method lets you programmatically specify, in the form load at the start of your applications, the ORB name that you want your COMet applications to use. This means that you can specify at runtime what configuration information is to be used by your COMet applications.

If you do not use this method to specify an ORB name, the configuration information relating to the default ORB name in the configuration repository is used instead.

Note: Only one COMet ORB object should be created in any COMet application. Therefore, `SetOrbName` should only be called once during each run of an application, and it should be the first call that is made.

UUID

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

Notes

Automation/CORBA-compliant.

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

All Automation views of CORBA objects expose the `DIORBObject` interface. It provides Automation/CORBA-compliant methods that allow Automation clients to request the ORB to perform various operations. You can call the `DICORBAFactory::GetObject()` method, to obtain a reference to `DIORBObject`.

The ORB has the `CORBA.ORB.2` ProgID. In COMet, the `CORBA.ORB.Orbix` name is registered as an alias for `CORBA.ORB.2`. This allows access to the Orbix instance in the event of a subsequent installation of an ORB other than Orbix.

Methods

The methods for the `DIORBObject` interface are:

`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 Interface Repository 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 Interface Repository 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, to gain access to operations on the CORBA object reference interface.

UUID `{204F6246-3AEC-11CF-BBFC-444553540000}`

Notes Automation/CORBA-compliant.

COM-Specific Interfaces

Overview

This section describes the interfaces that are specific to COM.

In This Section

This section discusses the following topics:

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 maps to the ICORBA_Any COM interface. You can use ICORBA_Any to get the type of an any, and to get or set its value.

Methods

The methods for the ICORBA_Any interface are:

`_get_value()` This returns the value of a CORBA any that can be contained by a VARIANT (that is, if the value of the any is a simple type or an interface pointer).

`_put_value()` This sets the value of a CORBA any that can be contained by a VARIANT (that is, if the value of the any is a simple type or an interface pointer).

`_get_CORBAAnyData()` This returns the value of a CORBA any that cannot be contained by a VARIANT (that is, if the value of the any is a complex type, such as a struct or union).

`_put_CORBAAnyData()` This sets the value of a CORBA any that cannot be contained by a VARIANT (that is, if the value of the any is a complex type, such as a struct or union).

`_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 GetObject ([in] LPSTR objectName, [out] IUnknown **
       val);
};
```

Description

The `ICORBAFactory` interface is used to make CORBA objects available to COM clients, in a manner that is similar to `GetObject` method in COM (already described in "COM and CORBA Principles" on page 3). It is a factory class that allows a COM client to create new CORBA object instances and bind to existing CORBA objects.

An instance of this class must be registered in the Windows system registry on the client machine, using the following settings:

```
{913D82C0-3B00-11cf-BBFC-444553540000}
DEFINE_GUID(IID_ICORBAFactory, 0x913d82c0, 0x3b00, 0x11cf, 0xbb,
   0xfc, 0x44, 0x45, 0x53, 0x54, 0x0, 0x0);
"CORBA.Factory.COM"
```

Your COM clients can obtain a pointer to `ICORBAFactory`, by making the COM `CoCreateInstanceEx()` call as normal. The IID that the client assigns to the factory (for example, `IID_ICORBAFactory`) is specified in the parameter to `CoCreateInstanceEx()`. The call to `CoCreateInstanceEx()` creates a remote instance of the CORBA object factory on the client machine.

Methods

The methods for the `ICORBAFactory` interface are:

`GetObject()`

This allows a client to specify the name of a target object to which it wants to connect. It creates a COM view of the specified target object, binds this view to the target, and sets up a pointer to the `IUnknown` interface of the view object. After calling `GetObject()`, the COM client can then call `QueryInterface()` on the pointer to `IUnknown`, to obtain a reference to the view, which the client can then use to makes its requests.

The `objectName` parameter specifies the target CORBA object to which the client wants to connect. In COMet, the format of this parameter is as follows:

```
"interface:TAG:Tag data"
```

The *`interface`* component represents the IDL interface that the target object supports. If the interface is scoped (for example, "*`Module`*`::`*`Interface`*"), the interface token is "*`Module/Interface`*".

`TAG` can be either of the following:

- `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.
     staff.PD.Tom")
  ```

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

All COM views of CORBA objects expose the ICORBAObject interface. It provides a number of COM/CORBA-compliant methods that all CORBA (and hence, Orbix) objects support.

ICORBAObject allows COM clients to have access to operations on the CORBA object references, which are defined on the CORBA::Object pseudo-interface. A COM client can call QueryInterface() to obtain a pointer to ICORBAObject.

Methods

The methods for the ICORBAObject interface are:

GetInterface()	This returns a reference to an object in the Interface Repository that provides type information about the target object. This method requires runtime access to the Interface Repository.
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 in the repositoryID parameter, or an instance of a derived type of the type specified 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 specified 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 value of the method 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

The `ICORBA_TypeCode` interface is used to show that a COM interface has been translated from an OMG IDL typecode definition. Any COM interface that results from the translation of an OMG IDL typecode supports `ICORBA_TypeCode`. It describes arbitrarily complex OMG IDL type structures at runtime.

Methods

The methods for the ICORBA_TypeCode interface are:

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 CORBA_TCKind 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 Interface Repository ID that globally identifies the type.

This method requires runtime access to the Interface Repository.

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` 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 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.

It 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` 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 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, it returns the bound value. A return value of 0 indicates an unbounded string or sequence.

For an array, it 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 ICORBA_TypeCode of a different kind, it raises a BadKind exception.

For a sequence or array, it returns the type of element contained in the sequence or array. For an alias, it returns the type aliased by the typedef definition.

UUID

{9556EA21-3889-11cf-9586AA0004004A09}

Notes

COM/CORBA-compliant.

ICORBA_TypeCodeExceptions

Synopsis

```
typedef struct tagTypeCodeBounds {long 1;} TypeCodeBounds;
typedef struct tagTypeCodeBadKind {long 1;} TypeCodeBadKind;

[object, uuid(…), pointer_default(unique)]
interface ICORBA_TypeCodeExceptions : IUnknown
{
    HRESULT _get_Bounds([out] TypeCodeBounds * pExceptionBody);
    HRESULT _get_BadKind([out] TypeCodeBadKind * pExceptionBody);
};
typedef struct tagCORBA_TypeCodeExceptions
{
    CORBA_ExceptionType type;
    LPSTR repositoryId;
    ICORBA_TypeCodeExceptions *pUserException;
} CORBA_TypeCodeExceptions;
```

Description

The ICORBA_TypeCodeExceptions interface allows for the raising of exceptions that can occur with ICORBA_TypeCode at runtime.

Methods

The methods for the ICORBA_TypeCodeExceptions interface are:

_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.

IOrbixORBObject

Synopsis

```
[object, uuid(…)]
interface IOrbixORBObject : IORBObject {
    HRESULT GetConfigValue([in] LPSTR name,
        [out] LPSTR *value,
        [out] BOOLEAN * IT_retval);
    HRESULT StartUp([out] BOOLEAN * IT_retval);
    HRESULT ShutDown([out] BOOLEAN * IT_retval);
    HRESULT ReleaseCORBAView([in IDispatch * poObj,
        [in] VARIANT_BOOL 1ToDestruction,
        [optional,in,out] VARIANT *IT_Ex,
        [retval,out] VARIANT_BOOL * IT_retval);
    HRESULT ProcessEvents(in, out, optional] VARIANT* IT_Ex,
        [retval, out] VARIANT_BOOL * IT_retval);
    HRESULT SetOrbName([in] LPSTR strOrbName,
        [out] BOOLEAN * IT_retval);
};
```

Description

The `IOrbixORBObject` interface provides Orbix-specific methods that allow you to control some aspects of the ORB (that is, Orbix) or to request it to perform actions. `IOrbixORBObject` derives from `IORBObject`. The `IOrbixORBObject` methods augment the COM/CORBA-compliant methods defined in the `IORBObject` interface.

The ORB has the `CORBA.ORB.2` ProgID, which is the COM/CORBA-compliant name. In COMet, the name `CORBA.ORB.Orbix` is registered as an alias for `CORBA.ORB.2`. This allows access to the Orbix instance in the event of a subsequent installation of an ORB other than Orbix.

Methods

The methods for the `IOrbixORBObject` interface are:

`GetConfigValue()` This obtains the value of the configuration entry specified in the `name` parameter.

See the Orbix documentation set for information on configuration values.

`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 `ICORBAFactory`).

`ShutDown()` This shuts down the bridge. Invoking this method might be necessary if, for example, you are experiencing hang-on-exit problems or the `COMet:Config:COMET_SHUTDOWN_POLICY` configuration variable is set to `Disabled`. After this method is called, no more invocations can be made using CORBA.

`ReleaseCORBAView()` This is used by clients to free the CORBA view of a DCOM callback object when receipt of callbacks is no longer required.

`ProcessEvents()` 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.

If you want to use this method, set the `COMet.Config.SINGLE_THREADED_CALLBACK` configuration variable to `YES`.

`SetOrbName()`

Every ORB is associated with a configuration domain that provides it with configuration information. A single configuration domain can hold configuration information for multiple ORBs, with each ORB using its ORB name as a "key" or configuration scope in which the particular configuration information relating to that ORB is located.

This method lets you programmatically specify the ORB name that you want your COMet applications to use. This means that you can specify at runtime what configuration information is to be used by your COMet applications.

If you do not use this method to specify an ORB name, the configuration information relating to the default ORB name in the configuration repository is used instead.

Note: Only one COMet ORB object should be created in any COMet application. Therefore, `SetOrbName` should only be called once during each run of an application, and it should be the first call that is made.

UUID

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

Notes

Automation/CORBA-compliant.

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

All COM views of CORBA objects expose the IORBObject interface. It provides COM/CORBA-compliant methods that allow COM clients to request the ORB to perform various operations. You can call the ICORBAFactory::GetObject() method, to obtain a reference to IORBObject.

The ORB has the CORBA.ORB.2 ProgID. In COMet, the CORBA.ORB.Orbix name is registered as an alias for CORBA.ORB.2. This allows access to the Orbix instance in the event of a subsequent installation of an ORB other than Orbix.

Methods

The methods for the IORBObject interface are:

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 Interface Repository 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 COM/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 Interface Repository or one of the CORBA services) can be used. The `name` parameter specifies the desired service. A list of supported services can be obtained via `DIORBObject::GetInitialReferences()`.

UUID `{204F6245-3AEC-11CF-BBFC-444553540000}`

Notes COM/CORBA-compliant.

CHAPTER 12

Introduction to OMG IDL

An object's interface describes that object to potential clients through its attributes and operations, and their signatures. This chapter describes the semantics and uses of the CORBA Interface Definition Language (OMG IDL), which is used to describe the interfaces to CORBA objects.

In This Chapter

This chapter discusses the following topics:

Note: COMet does not support all the OMG IDL types described in this chapter. See "Mapping CORBA to Automation" on page 313 and "Mapping CORBA to COM" on page 357 for details of the OMG IDL types that COMet supports.

IDL

Overview

An IDL-defined object can be implemented in any language that IDL maps to, including C++, Java, COBOL, and PL/I. By encapsulating object interfaces within a common language, IDL facilitates interaction between objects regardless of their actual implementation. Writing object interfaces in IDL is therefore central to achieving the CORBA goal of interoperability between different languages and platforms.

IDL Standard Mappings

CORBA defines standard mappings from IDL to several programming languages, including C++, Java, COBOL, and PL/I. Each IDL mapping specifies how an IDL interface corresponds to a language-specific implementation. The Orbix 2000 IDL compiler uses these mappings to convert IDL definitions to language-specific definitions that conform to the semantics of that language.

Overall Structure

You create an application's IDL definitions within one or more IDL modules. Each module provides a naming context for the IDL definitions within it. Modules and interfaces form naming scopes, so identifiers defined inside an interface need to be unique only within that interface.

IDL Definition Structure

In the following example, two interfaces, `Bank` and `Account`, are defined within the `BankDemo` module:

```
module BankDemo
{
interface Bank {
        //…
    };

    interface Account {
        //…
    };
};
```

Modules and Name Scoping

Resolving a Name

To resolve a name, the IDL compiler conducts a search among the following scopes, in the order outlined:

1. The current interface.
2. Base interfaces of the current interface (if any).
3. The scopes that enclose the current interface.

Referencing Interfaces

Interfaces can reference each other by name alone within the same module. If an interface is referenced from outside its module, its name must be fully scoped, with the following syntax:

```
module-name::interface-name
```

For example, the fully scoped names of the `Bank` and `Account` interfaces shown in “IDL Definition Structure” on page 270 are, respectively, `BankDemo::Bank` and `BankDemo::Account`.

Nesting Restrictions

A module cannot be nested inside a module of the same name. Likewise, you cannot directly nest an interface inside a module of the same name. To avoid name ambiguity, you can provide an intervening name scope as follows:

```
module A
{
    module B
    {
        interface A {
            //…
        };
    };
};
```

Interfaces

Overview

This section provides details about OMG IDL interfaces.

In This Section

The following topics are discussed in this section:

Introduction to Interfaces

Overview

This subsection provides an introductory overview of OMG IDL interfaces.

What Are Interfaces?

Interfaces are the fundamental abstraction mechanism of CORBA. An interface defines a type of object, including the operations that object supports in a distributed enterprise application.

Objects and Interfaces

Every CORBA object has exactly one interface. However, the same interface can be shared by many CORBA objects in a system. CORBA object references specify CORBA objects (that is, interface instances). Each reference denotes exactly one object, which provides the only means by which that object can be accessed for operation invocations.

Public Members

Because an interface does not expose an object's implementation, all members are public. A client can access variables in an object's implementation only through an interface's operations and attributes.

Operations and Attributes

An IDL interface generally defines an object's behavior through operations and attributes:

- Operations of an interface give clients access to an object's behavior. When a client invokes an operation on an object, it sends a message to that object. The ORB transparently dispatches the call to the object, whether it is in the same address space as the client, in another address space on the same machine, or in an address space on a remote machine.
- An IDL attribute is short-hand for a pair of operations that get and, optionally, set values in an object.

Account Interface IDL Sample

In the following example, the `Account` interface in the `BankDemo` module describes the objects that implement the bank accounts:

```
module BankDemo
{
     typedef float CashAmount; // Type for representing cash
     typedef string AccountId; // Type for representing account
                               // ids
     //…
     interface Account {
         readonly attribute AccountId  account_id;
         readonly attribute CashAmount balance;

         void
         withdraw(in CashAmount amount)
         raises (InsufficientFunds);

         void
         deposit(in CashAmount amount);
     };
};
```

Code Explanation

This interface has two readonly attributes, `AccountId` and `balance`, which are respectively defined as typedefs of the `string` and `float` types. The interface also defines two operations, `withdraw()` and `deposit()`, which a client can invoke on this object.

Interface Contents

IDL Interface Components

An IDL interface definition typically has the following components.

- Operation definitions.
- Attribute definitions
- Exception definitions.
- Type definitions.
- Constant definitions.

Of these, operations and attributes must be defined within the scope of an interface, all other components can be defined at a higher scope.

Operations

Overview

Operations of an interface give clients access to an object’s behavior. When a client invokes an operation on an object, it sends a message to that object. The ORB transparently dispatches the call to the object, whether it is in the same address space as the client, in another address space on the same machine, or in an address space on a remote machine.

Operation Components

IDL operations define the signature of an object’s function, which client invocations on that object must use. The signature of an IDL operation is generally composed of three components:

- Return value data type.
- Parameters and their direction.
- Exception clause.

An operation’s return value and parameters can use any data types that IDL supports.

Operations IDL Sample

In the following example, the `Account` interface defines two operations, `withdraw()` and `deposit()`, and an `InsufficientFunds` exception:

```
module BankDemo
{
    typedef float CashAmount;  // Type for representing cash
    //...
    interface Account {
        exception InsufficientFunds {};

        void
        withdraw(in CashAmount amount)
        raises (InsufficientFunds);

        void
        deposit(in CashAmount amount);
    };
};
```

Code Explanation

On each invocation, both operations expect the client to supply an argument for the `amount` parameter, and return `void`. Invocations on the `withdraw()` operation can also raise the `InsufficientFunds` exception, if necessary.

Parameter Direction

Each parameter specifies the direction in which its arguments are passed between client and object. Parameter-passing modes clarify operation definitions and allow the IDL compiler to accurately map operations to a target programming language. The COBOL runtime uses parameter-passing modes to determine in which direction or directions it must marshal a parameter.

Parameter-Passing Mode Qualifiers

There are three parameter-passing mode qualifiers:

`in` This means that the parameter is initalized only by the client and is passed to the object.

`out` This means that the parameter is initialized only by the object and returned to the client.

`inout` This means that the parameter is initialized by the client and passed to the server; the server can modify the value before returning it to the client.

In general, you should avoid using `inout` parameters. Because an `inout` parameter automatically overwrites its initial value with a new value, its usage assumes that the caller has no use for the parameter's original value. Thus, the caller must make a copy of the parameter in order to retain that value. By using the two parameters, `in` and `out`, the caller can decide for itself when to discard the parameter.

One-Way Operations

By default, IDL operations calls are *synchronous*—that is, a client invokes an operation on an object and blocks until the invoked operation returns. If an operation definition begins with the keyword `oneway`, a client that calls the operation remains unblocked while the object processes the call.

The COBOL runtime cannot guarantee the success of a one-way operation call. Because one-way operations do not support return data to the client, the client cannot ascertain the outcome of its invocation. The COBOL runtime indicates failure of a one-way operation only if the call fails before it exits the client's address space; in this case, the COBOL runtime raises a system exception.

A client can also issue non-blocking, or asynchronous, invocations. See the *CORBA Programmer's Guide, C++* for more details.

One-Way Operation Constraints

Three constraints apply to a one-way operation:

- The return value must be set to `void`.
- Directions of all parameters must be set to `in`.
- No `raises` clause is allowed.

One-Way Operation IDL Sample

In the following example, the `Account` interface defines a one-way operation that sends a notice to an `Account` object:

```
module BankDemo {
     //…
     interface Account {
         oneway void notice(in string text);
         //…
     };
};
```

Attributes

Attributes Overview

An interface's attributes correspond to the variables that an object implements. Attributes indicate which variables in an object are accessible to clients.

Qualified and Unqualified Attributes

Unqualified attributes map to a pair of `get` and `set` functions in the implementation language, which allow client applications to read and write attribute values. An attribute that is qualified with the `readonly` keyword maps only to a `get` function.

IDL Readonly Attributes Sample

For example the `Account` interface defines two readonly attributes, `AccountId` and `balance`. These attributes represent information about the account that only the object's implementation can set; clients are limited to readonly access:

```
module BankDemo
{
    typedef float CashAmount; // Type for representing cash
    typedef string AccountId; //Type for representing account
       ids
    //…
    interface Account {
        readonly attribute AccountId  account_id;
        readonly attribute CashAmount balance;

        void
        withdraw(in CashAmount amount)
        raises (InsufficientFunds);

        void
        deposit(in CashAmount amount);
    };
};
```

Code Explanation

The `Account` interface has two readonly attributes, `AccountId` and `balance`, which are respectively defined as typedefs of the `string` and `float` types. The interface also defines two operations, `withdraw()` and `deposit()`, which a client can invoke on this object.

Exceptions

IDL and Exceptions

IDL operations can raise one or more CORBA-defined system exceptions. You can also define your own exceptions and explicitly specify these in an IDL operation. An IDL exception is a data structure that can contain one or more member fields, formatted as follows:

```
exception exception-name {
    [member;]…
};
```

Exceptions that are defined at module scope are accessible to all operations within that module; exceptions that are defined at interface scope are accessible on to operations within that interface.

The raises Clause

After you define an exception, you can specify it through a `raises` clause in any operation that is defined within the same scope. A `raises` clause can contain multiple comma-delimited exceptions:

```
return-val operation-name( [params-list] )
    raises( exception-name[, exception-name] );
```

Example of IDL-Defined Exceptions

The `Account` interface defines the `InsufficientFunds` exception with a single member of the `string` data type. This exception is available to any operation within the interface. The following IDL defines the `withdraw()` operation to raise this exception when the withdrawal fails:

```
module BankDemo
{
    typedef float CashAmount;  // Type for representing cash
    //…
    interface Account {
        exception InsufficientFunds {};

        void
        withdraw(in CashAmount amount)
        raises (InsufficientFunds);
        //…
    };
};
```

Empty Interfaces

Defining Empty Interfaces

IDL allows you to define empty interfaces. This can be useful when you wish to model an abstract base interface that ties together a number of concrete derived interfaces.

IDL Empty Interface Sample

In the following example, the CORBA `PortableServer` module defines the abstract `Servant Manager` interface, which serves to join the interfaces for two servant manager types, `ServantActivator` and `ServantLocator`:

```
module PortableServer
{
    interface ServantManager {};

    interface ServantActivator : ServantManager {
        //…
    };

    interface ServantLocator : ServantManager {
        //…
    };
};
```

Inheritance of Interfaces

Inheritance Overview

An IDL interface can inherit from one or more interfaces. All elements of an inherited, or *base* interface, are available to the *derived* interface. An interface specifies the base interfaces from which it inherits, as follows:

```
interface new-interface : base-interface[, base-interface]…
{…};
```

Inheritance Interface IDL Sample

In the following example, the `CheckingAccount` and `SavingsAccount` interfaces inherit from the `Account` interface, and implicitly include all its elements:

```
module BankDemo{
     typedef float CashAmount;  // Type for representing cash
     interface Account {
         //…
     };

     interface CheckingAccount : Account {
         readonly attribute CashAmount overdraftLimit;
         boolean orderCheckBook ();
     };

     interface SavingsAccount : Account {
         float calculateInterest ();
     };
};
```

Code Sample Explanation

An object that implements the `CheckingAccount` interface can accept invocations on any of its own attributes and operations as well as invocations on any of the elements of the `Account` interface. However, the actual implementation of elements in a `CheckingAccount` object can differ from the implementation of corresponding elements in an `Account` object. IDL inheritance only ensures type-compatibility of operations and attributes between base and derived interfaces.

Multiple Inheritance

Multiple Inheritance IDL Sample

In the following IDL definition, the `BankDemo` module is expanded to include the `PremiumAccount` interface, which inherits from the `CheckingAccount` and `SavingsAccount` interfaces:

```
module BankDemo {
    interface Account {
        //…
    };

    interface CheckingAccount : Account {
        //…
    };

    interface SavingsAccount : Account {
        //…
    };

    interface PremiumAccount :
        CheckingAccount, SavingsAccount {
        //…
    };
};
```

Multiple Inheritance Constraints

Multiple inheritance can lead to name ambiguity among elements in the base interfaces. The following constraints apply:

- Names of operations and attributes must be unique across all base interfaces.
- If the base interfaces define constants, types, or exceptions of the same name, references to those elements must be fully scoped.

Inheritance Hierarchy Diagram

Figure 37 shows the inheritance hierarchy for the `Account` interface, which is defined in “Multiple Inheritance IDL Sample” on page 283.

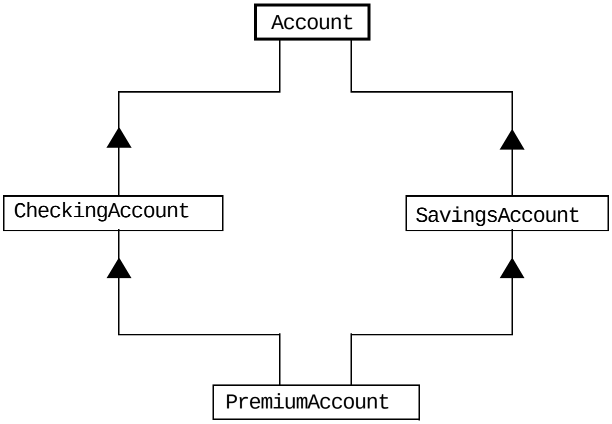


Figure 37: *Inheritance Hierarchy for PremiumAccount Interface*

Inheritance of the Object Interface

User-Defined Interfaces

All user-defined interfaces implicitly inherit the predefined `Object` interface. Thus, all `Object` operations can be invoked on any user-defined interface. You can also use `Object` as an attribute or parameter type, to indicate that any interface type is valid for the attribute or parameter.

Object Locator IDL Sample

For example, the following `getAnyObject()` operation serves as an all-purpose object locator:

```
interface ObjectLocator {
    void getAnyObject (out Object obj);
};
```

Note: It is illegal in IDL syntax to explicitly inherit the `Object` interface.

Inheritance Redefinition

Overview

A derived interface can modify the definitions of constants, types, and exceptions that it inherits from a base interface. All other components that are inherited from a base interface cannot be changed.

Inheritance Redefinition IDL Sample

In the following example, the `CheckingAccount` interface modifies the definition of the `InsufficientFunds` exception, which it inherits from the `Account` interface:

```
module BankDemo
{
    typedef float CashAmount;  // Type for representing cash
    //…
    interface Account {
        exception InsufficientFunds {};
        //…
    };
    interface CheckingAccount : Account {
        exception InsufficientFunds {
            CashAmount overdraftLimit;
        };
    };
    //…
};
```

Note: While a derived interface definition cannot override base operations or attributes, operation overloading is permitted in interface implementations for those languages, such as C++, that support it. However, COBOL does not support operation overloading.

Forward Declaration of IDL Interfaces

Overview

An IDL interface must be declared before another interface can reference it. If two interfaces reference each other, the module must contain a forward declaration for one of them; otherwise, the IDL compiler reports an error. A forward declaration only declares the interface's name; the interface's actual definition is deferred until later in the module.

Forward Declaration IDL Sample

In the following example, the `Bank` interface defines a `create_account()` and `find_account()` operation, both of which return references to `Account` objects. Because the `Bank` interface precedes the definition of the `Account` interface, `Account` is forward-declared:

```
module BankDemo
{
    typedef float CashAmount; // Type for representing cash
    typedef string AccountId; //Type for representing account ids

    // Forward declaration of Account
    interface Account;

    // Bank interface...used to create Accounts
    interface Bank {
        exception AccountAlreadyExists { AccountId account_id; };
        exception AccountNotFound      { AccountId account_id; };

        Account
        find_account(in AccountId account_id)
        raises(AccountNotFound);

        Account
        create_account(
            in AccountId account_id,
            in CashAmount initial_balance
        ) raises (AccountAlreadyExists);
    };

    // Account interface…used to deposit, withdraw, and query
    // available funds.
    interface Account { //…
    };
};
```

Local Interfaces

Overview

An interface declaration that contains the IDL `local` keyword defines a *local interface*. An interface declaration that omits this keyword can be referred to as an *unconstrained interface*, to distinguish it from local interfaces. An object that implements a local interface is a *local object*.

Characteristics

Local interfaces differ from unconstrained interfaces in the following ways:

- A local interface can inherit from any interface, whether local or unconstrained. Unconstrained interfaces cannot inherit from local interfaces.
- Any non-interface type that uses a local interface is regarded as a local type. For example, a struct that contains a local interface member is regarded as a local struct, and is subject to the same localization constraints as a local interface.
- Local types can be declared as parameters, attributes, return types, or exceptions only in a local interface, or as state members of a valuetype.
- Local types cannot be marshalled, and references to local objects cannot be converted to strings through `ORB::object_to_string()`. Any attempts to do so throw a `CORBA::MARSHAL` exception.
- Any operation that expects a reference to a remote object cannot be invoked on a local object. For example, you cannot invoke any DII operations or asynchronous methods on a local object; similarly, you cannot invoke pseudo-object operations such as `is_a()` or `validate_connection()`. Any attempts to do so throw a `CORBA::NO_IMPLEMENT` exception.
- The ORB does not mediate any invocations on a local object. Thus, local interface implementations are responsible for providing the parameter copy semantics that a client expects.
- Instances of local objects that the OMG defines, as supplied by ORB products, are exposed either directly or indirectly through `ORB::resolve_initial_references()`.

Implementation

Local interfaces are implemented by `CORBA::LocalObject` to provide implementations of `Object` pseudo-operations, and other ORB-specific support mechanisms that apply. Because object implementations are language-specific, the `LocalObject` type is only defined by each language mapping.

Local Object Pseudo-Operations

The `LocalObject` type implements the `Object` pseudo-operations shown in Table 3.

Table 3: *CORBA::LocalObject Pseudo-Operations and Return Values*

Operation	Always returns
`is_a()`	An exception of `NO_IMPLEMENT`.
`get_interface()`	An exception of `NO_IMPLEMENT`.
`get_domain_managers()`	An exception of `NO_IMPLEMENT`.
`get_policy()`	An exception of `NO_IMPLEMENT`.
`get_client_policy()`	An exception of `NO_IMPLEMENT`.
`set_policy_overrides()`	An exception of `NO_IMPLEMENT`.
`get_policy_overrides()`	An exception of `NO_IMPLEMENT`.
`validate_connection()`	An exception of `NO_IMPLEMENT`.
`non_existent()`	False.
`hash()`	A hash value that is consistent with the object's lifetime.
`is_equivalent()`	True, if the references refer to the same `LocalObject` implementation.

Valuetypes

Overview

Valuetypes enable programs to pass objects by value across a distributed system. This type is especially useful for encapsulating lightweight data such as linked lists, graphs, and dates.

Characteristics

Valuetypes can be seen as a cross between the following:

- Data types, such as `long` and `string`, which can be passed by value over the wire as arguments to remote invocations.
- Objects, which can only be passed by reference.

When a program supplies an object reference, the object remains in its original location; subsequent invocations on that object from other address spaces move across the network, rather than the object moving to the site of each request.

Valuetype Support

Like an interface, a valuetype supports both operations and inheritance from other valuetypes; it also can have data members. When a valuetype is passed as an argument to a remote operation, the receiving address space creates a copy of it. The copied valuetype exists independently of the original; operations that are invoked on one have no effect on the other.

Valuetype Invocations

Because a valuetype is always passed by value, its operations can only be invoked locally. Unlike invocations on objects, valuetype invocations are never passed over the wire to a remote valuetype.

Valuetype Implementations

Valuetype implementations necessarily vary, depending on the languages used on sending and receiving ends of the transmission, and their respective abilities to marshal and demarshal the valuetype’s operations. A receiving process that is written in C++ must provide a class that implements valuetype operations and a factory to create instances of that class. These classes must be either compiled into the application, or made available through a shared library. Conversely, Java applications can marshal enough information on the sender, so the receiver can download the bytecodes for the valuetype operation implementations.

Abstract Interfaces

Overview

An application can use abstract interfaces to determine at runtime whether an object is passed by reference or by value.

IDL Abstract Interface Sample

In the following example, the IDL definitions specify that the `Example::display()` operation accepts any derivation of the abstract interface, `Describable`:

```
abstract interface Describable {
    string get_description();
};

interface Example {
    void display(in Describable someObject);
};
```

Abstract Interface IDL Sample

Based on the preceding IDL, you can define two derivations of the `Describable` abstract interface—the `Currency` valuetype and the `Account` interface:

```
interface Account : Describable {
    // body of Account definition not shown
};

valuetype Currency supports Describable {
    // body of Currency definition not shown
};
```

Note: Because the parameter for `display()` is defined as a `Describable` type, invocations on this operation can supply either `Account` objects or `Currency` valuetypes.

IDL Data Types

In This Section

The following topics are discussed in this section:

Data Type Categories

In addition to IDL module, interface, valuetype, and exception types, IDL data types can be grouped into the following categories:

- Built-in types such as `short`, `long`, and `float`.
- Extended built-in types such as `long long` and `wstring`.
- Complex types such as `enum`, `struct`, and `string`.
- Pseudo objects.

Built-in Data Types

List of Types, Sizes, and Values

Table 4 shows a list of CORBA IDL built-in data types (where the ≤ symbol means ’less than or equal to’).

Table 4: *Built-in IDL Data Types, Sizes, and Values*

Data type	Size	Range of values
short	≤ 16 bits	$-2^{15}...2^{15}-1$
unsigned short	≤ 16 bits	$0...2^{16}-1$
long	≤ 32 bits	$–2^{31}...2^{31}-1$
unsigned long	≤ 32 bits	$0...2^{32}-1$
float	≤ 32 bits	IEEE single-precision floating point numbers
double	≤ 64 bits	IEEE double-precision floating point numbers
char	≤ 8 bits	ISO Latin-1
string	Variable length	ISO Latin-1, except NUL
string<bound>	Variable length	ISO Latin-1, except NUL
boolean	Unspecified	TRUE or FALSE
octet	≤ 8 bits	0x0 to 0xff
any	Variable length	Universal container type

Floating Point Types

The float and double types follow IEEE specifications for single-precision and double-precision floating point values, and on most platforms map to native IEEE floating point types.

Char Type

The `char` type can hold any value from the ISO Latin-1 character set. Code positions 0-127 are identical to ASCII. Code positions 128-255 are reserved for special characters in various European languages, such as accented vowels.

String Type

The `string` type can hold any character from the ISO Latin-1 character set, except `NUL`. IDL prohibits embedded `NUL` characters in strings. Unbounded string lengths are generally constrained only by memory limitations. A bounded string, such as `string<10>`, can hold only the number of characters specified by the bounds, excluding the terminating `NUL` character. Thus, a `string<6>` can contain the six-character string, `cheese`.

Bounded and Unbounded Strings

The declaration statement can optionally specify the string's maximum length, thereby determining whether the string is bounded or unbounded:

```
string[length] name
```

For example, the following code declares the `ShortString` type, which is a bounded string with a maximum length of 10 characters:

```
typedef string<10> ShortString;
attribute ShortString shortName; // max length is 10 chars
```

Octet Type

Octet types are guaranteed not to undergo any conversions in transit. This lets you safely transmit binary data between different address spaces. Avoid using the char type for binary data, because characters might be subject to translation during transmission. For example, if a client that uses ASCII sends a string to a server that uses EBCDIC, the sender and receiver are liable to have different binary values for the string's characters.

Any Type

The `any` type allows specification of values that express any IDL type, which is determined at runtime, thereby allowing a program to handle values whose types are not known at compile time. An `any` logically contains a `TypeCode` and a value that is described by the `TypeCode`. A client or server can construct an `any` to contain an arbitrary type of value and then pass this call in a call to the operation. A process receiving an any must determine what type of value it stores and then extract the value via the typecode. See the *CORBA Programmer's Guide, C++* for more details about the any type.

Extended Built-in Data Types

List of Types, Sizes, and Values

Table 5 shows a list of CORBA IDL extended built-in data types (where the ≤ symbol means ’less than or equal to’).

Table 5: *Extended built-in IDL Data Types, Sizes, and Values*

Data Type	Size	Range of Values
long long[a]	≤ 64 bits	$–2^{63}...2^{63}-1$
unsigned long long[a]	≤ 64 bits	$0...-2^{64}-1$
long double[b]	≤ 79 bits	IEEE double-extended floating point number, with an exponent of at least 15 bits in length and a signed fraction of at least 64 bits. The `long double` type is currently not supported on Windows NT.
wchar	Unspecified	Arbitrary codesets
wstring	Variable length	Arbitrary codesets
fixed[c]	Unspecified	≤ 31significant digits

a. Due to compiler restrictions, the COBOL range of values for the `long long` and `unsigned long long` types is the same range as for a `long` type (that is, $0...2^{31}-1$).

b. Due to compiler restrictions, the COBOL range of values for the `long double` type is the same range as for a double type (that is, ≤ 64 bits).

c. Due to compiler restrictions, the COBOL range of values for the fixed type is ≤ 18 significant digits.

Long Long Type

The 64-bit integer types, `long long` and `unsigned long long`, support numbers that are too large for 32-bit integers. Platform support varies. If you compile IDL that contains one of these types on a platform that does not support it, the compiler issues an error.

Long Double Type

Like 64-bit integer types, platform support varies for the `long double` type, so its use can yield IDL compiler errors.

Wchar Type

The `wchar` type encodes wide characters from any character set. The size of a `wchar` is platform-dependent. Because Orbix 2000 currently does not support character set negotiation, use this type only for applications that are distributed across the same platform.

Wstring Type

The `wstring` type is the wide-character equivalent of the `string` type. Like `string` types, `wstring` types can be unbounded or bounded. Wide strings can contain any character except `NUL`.

Fixed Type

IDL specifies that the `fixed` type provides fixed-point arithmetic values with up to 31 significant digits. However, due to restrictions in the COBOL compiler for OS/390, only up to 18 significant digits are supported.

You specify a `fixed` type with the following format:

```
typedef fixed<digit-size,scale> name
```

The format for the fixed type can be explained as follows:

- The *`digit-size`* represents the number's length in digits. The maximum value for *`digit-size`* is 31 and it must be greater than *`scale`*. A `fixed` type can hold any value up to the maximum value of a `double` type.
- If *`scale`* is a positive integer, it specifies where to place the decimal point relative to the rightmost digit. For example, the following code declares a fixed type, `CashAmount`, to have a digit size of 10 and a scale of 2:

  ```
  typedef fixed<10,2> CashAmount;
  ```

  Given this typedef, any variable of the `CashAmount` type can contain values of up to (+/-)99999999.99.

- If *scale* is a negative integer, the decimal point moves to the right by the number of digits specified for *scale*, thereby adding trailing zeros to the fixed data type's value. For example, the following code declares a fixed type, bigNum, to have a digit size of 3 and a scale of -4:

  ```
  typedef fixed <3,-4> bigNum;
  bigNum myBigNum;
  ```

  If myBigNum has a value of 123, its numeric value resolves to 1230000. Definitions of this sort allow you to efficiently store numbers with trailing zeros.

Constant Fixed Types

Constant fixed types can also be declared in IDL, where *digit-size* and *scale* are automatically calculated from the constant value. For example:

```
module Circle {
    const fixed pi = 3.142857;
};
```

This yields a fixed type with a digit size of 7, and a scale of 6.

Fixed Type and Decimal Fractions

Unlike IEEE floating-point values, the fixed type is not subject to representational errors. IEEE floating point values are liable to inaccurately represent decimal fractions unless the value is a fractional power of 2. For example, the decimal value, 0.1, cannot be represented exactly in IEEE format. Over a series of computations with floating-point values, the cumulative effect of this imprecision can eventually yield inaccurate results.

The fixed type is especially useful in calculations that cannot tolerate any imprecision, such as computations of monetary values.

Complex Data Types

IDL Complex Data Types

IDL provide the following complex data types:

- Enums.
- Structs.
- Multi-dimensional fixed-sized arrays.
- Sequences.

Enum Data Type

Overview

An enum (enumerated) type lets you assign identifiers to the members of a set of values.

Enum IDL Sample

For example, you can modify the BankDemo IDL with the balanceCurrency enum type:

```
module BankDemo {
    enum Currency {pound, dollar, yen, franc};

    interface Account {
        readonly attribute CashAmount balance;
        readonly attribute Currency balanceCurrency;
        //…
    };
};
```

In the preceding example, the balanceCurrency attribute in the Account interface can take any one of the values pound, dollar, yen, or franc.

Ordinal Values of Enum Type

The ordinal values of an enum type vary according to the language implementation. The CORBA specification only guarantees that the ordinal values of enumerated types monotonically increase from left to right. Thus, in the previous example, dollar is greater than pound, yen is greater than dollar, and so on. All enumerators are mapped to a 32-bit type.

Struct Data Type

Overview

A struct type lets you package a set of named members of various types.

Struct IDL Sample

In the following example, the `CustomerDetails` struct has several members. The `getCustomerDetails()` operation returns a struct of the `CustomerDetails` type, which contains customer data:

```
module BankDemo{
    struct CustomerDetails {
        string custID;
        string lname;
        string fname;
        short age;
        //…
    };

    interface Bank {
        CustomerDetails getCustomerDetails(in string custID);
        //…
    };
};
```

Note: A struct type must include at least one member. Because a struct provides a naming scope, member names must be unique only within the enclosing structure.

Union Data Type

Overview

A union type lets you define a structure that can contain only one of several alternative members at any given time. A union type saves space in memory, because the amount of storage required for a union is the amount necessary to store its largest member.

Union Declaration Syntax

You declare a union type with the following syntax:

```
union name switch (discriminator) {
     case label1 : element-spec;
     case label2 : element-spec;
     […]
     case labeln : element-spec;
     [default : element-spec;]
};
```

Discriminated Unions

All IDL unions are *discriminated*. A discriminated union associates a constant expression (`label1`…`label`*n*) with each member. The discriminator's value determines which of the members is active and stores the union's value.

IDL Union Date Sample

The following IDL defines a `Date` union type, which is discriminated by an enum value:

```
enum dateStorage
{ numeric, strMMDDYY, strDDMMYY };

struct DateStructure {
     short Day;
     short Month;
     short Year;
};
union Date switch (dateStorage) {
     case numeric: long digitalFormat;
     case strMMDDYY:
     case strDDMMYY: string stringFormat;
     default: DateStructure structFormat;
};
```

Sample Explanation

Given the preceding IDL:

- If the discriminator value for `Date` is numeric, the `digitalFormat` member is active.
- If the discriminator's value is `strMMDDYY` or `strDDMMYY`, the `stringFormat` member is active.
- If neither of the preceding two conditions apply, the default `structFormat` member is active.

Rules for Union Types

The following rules apply to union types:

- A union's discriminator can be `integer`, `char`, `boolean`, `enum`, or an alias of one of these types; all case label expressions must be compatible with the relevant type.
- Because a union provides a naming scope, member names must be unique only within the enclosing union.
- Each union contains a pair of values: the discriminator value and the active member.
- IDL unions allow multiple case labels for a single member. In the previous example, the `stringFormat` member is active when the discriminator is either `strMMDDYY` or `strDDMMYY`.
- IDL unions can optionally contain a `default` case label. The corresponding member is active if the discriminator value does not correspond to any other label.

Arrays

Overview

IDL supports multi-dimensional fixed-size arrays of any IDL data type, with the following syntax (where *dimension-spec* must be a non-zero positive constant integer expression):

```
[typedef] element-type array-name [dimension-spec]…
```

IDL does not allow open arrays. However, you can achieve equivalent functionality with sequence types.

Array IDL Sample

For example, the following defines a two-dimensional array of bank accounts within a portfolio:

```
typedef Account portfolio[MAX_ACCT_TYPES][MAX_ACCTS]
```

Note: For an array to be used as a parameter, an attribute, or a return value, the array must be named by a typedef declaration. You can omit a typedef declaration only for an array that is declared within a structure definition.

Array Indexes

Because of differences between implementation languages, IDL does not specify the origin at which arrays are indexed. For example, C and C++ array indexes always start at `0`, but COBOL, PL/I, and Pascal always start at `1`. Consequently, clients and servers cannot exchange array indexes unless they both agree on the origin of array indexes and make adjustments, as appropriate, for their respective implementation languages. Usually, it is easier to exchange the array element itself, instead of its index.

Sequence

Overview

IDL supports sequences of any IDL data type with the following syntax:

`[typedef] sequence <` *`element-type[, max-elements]`* `>` *`sequence-name`*

An IDL sequence is similar to a one-dimensional array of elements; however, its length varies according to its actual number of elements, so it uses memory more efficiently.

For a sequence to be used as a parameter, an attribute, or a return value, the sequence must be named by a typedef declaration. You can omit a typedef declaration only for a sequence that is declared within a structure definition.

A sequence's element type can be of any type, including another sequence type. This feature is often used to model trees.

Bounded and Unbounded Sequences

The maximum length of a sequence can be fixed (bounded) or unfixed (unbounded):

- Unbounded sequences can hold any number of elements, up to the memory limits of your platform.
- Bounded sequences can hold any number of elements, up to the limit specified by the bound.

Bounded and Unbounded IDL Definitions

The following code shows how to declare bounded and unbounded sequences as members of an IDL struct:

```
struct LimitedAccounts {
    string bankSortCode<10>;
    sequence<Account, 50> accounts; // max sequence length is 50
};

struct UnlimitedAccounts {
    string bankSortCode<10>;
    sequence<Account> accounts; // no max sequence length
};
```

Pseudo Object Types

Overview

CORBA defines a set of pseudo-object types that ORB implementations use when mapping IDL to a programming language. These object types have interfaces defined in IDL; however, these object types do not have to follow the normal IDL mapping rules for interfaces and they are not generally available in your IDL specifications.

Defining

You can use only the following pseudo-object types as attribute or operation parameter types in an IDL specification:

```
CORBA::NamedValue
CORBA::TypeCode
```

To use these types in an IDL specification, include the `orb.idl` file in the IDL file as follows:

```
#include <orb.idl>
//…
```

This statement instructs the IDL compiler to allow the `NamedValue` and `TypeCode` types.

Defining Data Types

Overview

With typedef, you can define more meaningful or simpler names for existing data types, regardless of whether those types are IDL-defined or user-defined.

The following code defines the typedef identifier, StandardAccount, so that it can act as an alias for the Account type in later IDL definitions:

```
module BankDemo {
    interface Account {
        //…
    };

    typedef Account StandardAccount;
};
```

In This Section

This section contains the following subsections:

Constants

Overview

IDL lets you define constants of all built-in types except the any type. To define a constant's value, you can use either another constant (or constant expression) or a literal. You can use a constant wherever a literal is permitted.

Integer Constants

IDL accepts integer literals in decimal, octal, or hexadecimal:

```
const short     I1 = -99;
const long      I2 = 0123;  // Octal 123, decimal 83
const long long I3 = 0x123; // Hexadecimal 123, decimal 291
const long long I4 = +0xaB; // Hexadecimal ab, decimal 171
```

Both unary plus and unary minus are legal.

Floating-Point Constants

Floating-point literals use the same syntax as C++:

```
const float       f1 = 3.1e-9; // Integer part, fraction part,
                               // exponent
const double      f2 = -3.14;  // Integer part and fraction part
const long double f3 = .1      // Fraction part only
const double      f4 = 1.      // Integer part only
const double      f5 = .1E12   // Fraction part and exponent
const double      f6 = 2E12    // Integer part and exponent
```

Character and String Constants

Character constants use the same escape sequences as C++:

```
const char C1 = 'c';        // the character c
const char C2 = '\007';     // ASCII BEL, octal escape
const char C3 = '\x41';     // ASCII A, hex escape
const char C4 = '\n';       // newline
const char C5 = '\t';       // tab
const char C6 = '\v';       // vertical tab
const char C7 = '\b';       // backspace
const char C8 = '\r';       // carriage return
const char C9 = '\f';       // form feed
const char C10 = '\a';      // alert
const char C11 = '\\';      // backslash
const char C12 = '\?';      // question mark
const char C13 = '\'';      // single quote
// String constants support the same escape sequences as C++
const string S1 = "Quote: \"";      // string with double quote
const string S2 = "hello world";    // simple string
const string S3 = "hello" " world"; // concatenate
const string S4 = "\xA" "B";        // two characters
                                    // ('\xA' and 'B'),
                               // not the single character '\xAB'
```

Wide Character and String Constants

Wide character and string constants use C++ syntax. Use universal character codes to represent arbitrary characters. For example:

```
const wchar     C = L'X';
const wstring   GREETING = L"Hello";
const wchar     OMEGA = L'\u03a9';
const wstring   OMEGA_STR = L"Omega: \u3A9";
```

IDL files always use the ISO Latin-1 code set; they cannot use Unicode or other extended character sets.

Boolean Constants

Boolean constants use the FALSE and TRUE keywords. Their use is unnecessary, inasmuch as they create unnecessary aliases:

```
// There is no need to define boolean constants:
const CONTRADICTION = FALSE;    // Pointless and confusing
const TAUTOLOGY = TRUE;         // Pointless and confusing
```

Octet Constants

Octet constants are positive integers in the range 0-255.

```
const octet O1 = 23;
const octet O2 = 0xf0;
```

Octet constants were added with CORBA 2.3; therefore, ORBs that are not compliant with this specification might not support them.

Fixed-Point Constants

For fixed-point constants, you do not explicitly specify the digits and scale. Instead, they are inferred from the initializer. The initializer must end in `d` or `D`. For example:

```
// Fixed point constants take digits and scale from the
// initializer:
const fixed val1 = 3D;          // fixed<1,0>
const fixed val2 = 03.14d;      // fixed<3,2>
const fixed val3 = -03000.00D;  // fixed<4,0>
const fixed val4 = 0.03D;       // fixed<3,2>
```

The type of a fixed-point constant is determined after removing leading and trailing zeros. The remaining digits are counted to determine the digits and scale. The decimal point is optional.

Currently, there is no way to control the scale of a constant if it ends in trailing zeros.

Enumeration Constants

Enumeration constants must be initialized with the scoped or unscoped name of an enumerator that is a member of the type of the enumeration. For example:

```
enum Size { small, medium, large }

const Size DFL_SIZE = medium;
const Size MAX_SIZE = ::large;
```

Enumeration constants were added with CORBA 2.3; therefore, ORBs that are not compliant with this specification might not support them.

Constant Expressions

Overview

IDL provides a number of arithmetic and bitwise operators. The arithmetic operators have the usual meaning and apply to integral, floating-point, and fixed-point types (except for %, which requires integral operands). However, these operators do not support mixed-mode arithmetic: you cannot, for example, add an integral value to a floating-point value.

Arithmetic Operators

The following code contains several examples of arithmetic operators:

```
// You can use arithmetic expressions to define constants.
const long MIN = -10;
const long MAX = 30;
const long DFLT = (MIN + MAX) / 2;

// Can't use 2 here
const double TWICE_PI = 3.1415926 * 2.0;

// 5% discount
const fixed DISCOUNT = 0.05D;
const fixed PRICE = 99.99D;

// Can't use 1 here
const fixed NET_PRICE = PRICE * (1.0D - DISCOUNT);
```

Evaluating Expressions for Arithmetic Operators

Expressions are evaluated using the type promotion rules of C++. The result is coerced back into the target type. The behavior for overflow is undefined, so do not rely on it. Fixed-point expressions are evaluated internally with 31 bits of precision, and results are truncated to 15 digits.

Bitwise Operators

Bitwise operators only apply to integral types. The right-hand operand must be in the range 0-63. The right-shift operator, >>, is guaranteed to insert zeros on the left, regardless of whether the left-hand operand is signed or unsigned.

```
// You can use bitwise operators to define constants.
const long ALL_ONES = -1;               // 0xffffffff
const long LHW_MASK = ALL_ONES << 16;   // 0xffff0000
const long RHW_MASK = ALL_ONES >> 16;   // 0x0000ffff
```

IDL guarantees two's complement binary representation of values.

Precedence

The precedence for operators follows the rules for C++. You can override the default precedence by adding parentheses.

CHAPTER 13

Mapping CORBA to Automation

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

In This Chapter

This chapter discusses the following topics:

Note: For the purposes of illustration, this chapter describes a textual mapping between OMG IDL and COM IDL. COMet 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 automatically performed by COMet at application runtime.

Mapping for Basic Types

Overview

OMG IDL basic types translate to compatible types in Automation.

Mapping Rules

Table 6 shows the mapping rules for each basic type.

Table 6: *CORBA-to-Automation Mapping Rules for Basic Types*

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

[a] UI1 is supported in Windows 32-bit programs.

Limitations

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.)

Bidirectional Translation

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 mapping rules shown in Table 6 on page 315, 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.

Runtime Errors

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.

Mapping for Strings

Overview

OMG IDL bounded and unbounded strings map to an Automation BSTR.

Note: A runtime error occurs when mapping a fixed-length OMG IDL string, if the BSTR exceeds the maximum length of the OMG IDL string.

Example

The example can be broken down as follows:

1. Consider the following OMG IDL:

```
// OMG IDL
// This definition might appear within a struct definition.
string name<20>;
string address;
```

2. The preceding OMG IDL maps to the following COM IDL:

```
// COM IDL
BSTR name;
BSTR address;
```

Mapping for Interfaces

Overview

This section describes how OMG IDL interfaces map to Automation.

In This Section

This section discusses the following topics:

Basic Interface Mapping

Overview

An OMG IDL interface maps to an Automation view interface.

Example

The example can be broken down as follows:

1. Consider the following OMG IDL interface, `Bank`:

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

2. The preceding OMG IDL maps to the following Automation view interface, `DIBank`:

```
// COM IDL
// Definitions that are not of interest here.

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

The DIBank Interface

As shown in Figure 38 on page 320, 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.

Standard Automation View Interfaces

The Automation view also supports the following interfaces, by default:

- `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.

Graphical Overview

Figure 38 provides a graphical overview of the interfaces that the Automation view object supports, based on the example of the OMG IDL Bank interface.

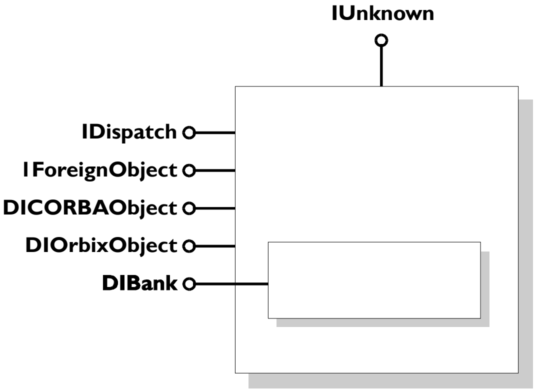


Figure 38: *Automation View of the Bank Interface*

Mapping for Attributes

Overview

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.

Example

The example can be broken down as follows:

1. Consider the following OMG IDL:

```
// 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);
};
```

2. The preceding OMG IDL maps to the following in Automation:

```
// COM IDL
[oleautomation, 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);
}
```

Note: The `get` method returns the attribute value contained in the `[retval,out]` parameter.

Visual Basic Example

The following is a Visual Basic example of how to set and get the balance of an account object, `accountObj`:

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

Dim myBalance as Single

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

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

PowerBuilder Example

The following is a PowerBuilder example of how to set and get the balance of an account object, `accountObj`:

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

integer myBalance

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

Mapping for Operations

Overview

An OMG IDL operation maps to an Automation method.

Example

The example can be broken down as follows:

1. Consider the following OMG IDL:

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

2. The preceding OMG IDL maps to the following in Automation:

```
// 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 );
}
```

Rules for Parameter Passing

The following mapping rules apply for 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.

Rules for Return Types

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 the 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. See "Mapping for System Exceptions" on page 343 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.

Visual Basic Example

The following is a Visual Basic example, based on the generated definitions in the preceding COM IDL example:

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

Mapping for Interface Inheritance

Overview

This section describes the CORBA-to-Automation mapping rules for both single and multiple interface inheritance.

In This Section

This section discusses the following topics:

Mapping for Single Inheritance

Overview

A hierarchy of singly-inherited OMG IDL interfaces maps to an identical hierarchy of Automation view interfaces.

Example

The example can be broken down as follows:

1. Consider the following OMG IDL 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();
     };
};
```

2. The preceding OMG IDL 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);
};
```

Mapping for Multiple Inheritance

Overview

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.

Interface Hierarchy Example

Figure 39 provides a graphical example of a CORBA interface hierarchy.

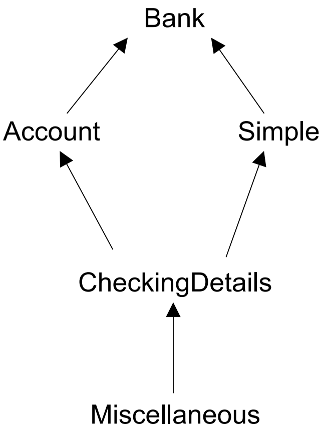


Figure 39: *Example of a CORBA Interface Hierarchy*

Interface Hierarchy Explanation

In Figure 39, the hierarchy can be read as follows:

- `Account` and `Simple` derive from `Bank`.
- `CheckingDetails` derives from `Account` and `Simple`.
- `Miscellaneous` derives from `CheckingDetails`.

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 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 the points of multiple inheritance. The operations of the Simple interface 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.

Code Example

The example can be broken down as follows:

1. Consider the following OMG IDL, which represents an interface hierarchy based on the example shown in Figure 39 on page 328:

```
// 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();
     };
};
```

2. The preceding OMG IDL 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);
}
```

Mapping for Complex Types

Overview

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.

In This Section

This section discusses the following topics:

Note: There is no standard CORBA-to-Automation mapping specified for OMG IDL context clauses.

Creating Constructed OMG IDL Types

Pseudo-Automation Interfaces

OMG IDL constructed types such as struct, union, and exception map to pseudo-Automation interfaces. The OMG *Interworking Architecture* specification at ftp://ftp.omg.org/pub/docs/formal/01-12-55.pdf chose this translation, because Automation does not allow Automation constructed types as valid parameter types.

Pseudo-Objects

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.

The CreateType() Method

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

Prototype for CreateType()

The prototype for CreateType() is:

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

Parameters for CreateType()

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 in 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.

See “Mapping for Structs” on page 333, “Mapping for Unions” on page 335, and “Mapping for System Exceptions” on page 343 for examples of how to use CreateType() to create structs, unions, and exceptions.

Mapping for Structs

Overview

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.

Example

The example can be broken down as follows:

1. Consider the following OMG IDL:

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

2. The preceding OMG IDL is mapped as if it were defined as follows:

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

Graphical Overview

Figure 40 provides a graphical overview of the interfaces that the Automation view object supports, based on the example of the OMG IDL AccountDetails struct.

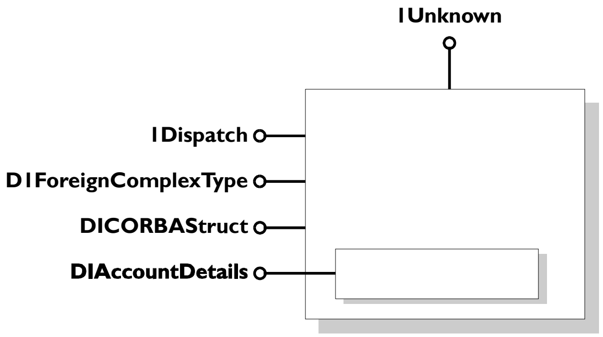


Figure 40: *Automation View of the OMG IDL AccoutDetails Struct*

Visual Basic Example

The following is a Visual Basic example, based on the preceding OMG IDL definition:

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

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

Mapping for Unions

Overview

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 translated from a union.

DICORBAUnion Interface

The following is a synopsis of the DICORBAUnion interface:

```
[oleautomation,dual,uuid(…)]
interface DICORBAUnion : DIForeignComplexType {
[id(400)] HRESULT Union_d ([retval,out] VARIANT * val);
};
```

DICORBAUnion has one method, Union_d, which returns the current value of the union’s discriminant.

DICORBAUnion2 Interface

The DICORBAUnion2 interface is defined to describe CORBA union types that support multiple case labels for each union branch. All mapped unions should support the DICORBAUnion2 interface.The DICORBAUnion2 provides two additional accessor methods, 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);
};
```

DICORBAUnion2 Methods

The methods provided by DICORBAUnion2 can be described as follows:

SetValue This can be used to set the discriminant and value simultaneously.

CurrentValue This uses the current discriminant value to initialize the VARIANT with the union element.

Example

The example can be broken down as follows:

1. Consider the following OMG IDL:

```
// OMG IDL
interface A {…};

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

2. The preceding OMG IDL maps to the following Automation pseudo-union::

```
// COM IDL
interface DIU : DICORBAUnion2{
     [propget] HRESULT get_UNION_d([retval,out] long * val);
     [propget] HRESULT 1([retval,out] long * 1);
     [propget] HRESULT 1([in] long 1);
     [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);
};
```

3. The following Visual Basic example is based on the preceding COM IDL:

```
' Visual Basic
Dim ObjFactory As CORBA_Orbix.DICORBAFactoryEx
Dim myUnion As DIU

…

Set myUnion = ObjFactory.CreateType(Nothing, "U")

myUnion.s = "This is a string"

Select Case(myUnion.UNION_d())
     Case 1: MsgBox ("Union (long):" & Str$(myUnion.l)
     Case 2: MsgBox ("Union (float):" & Str$(myUnion.f)
     Case Else : MsgBox ("Union contains object reference")
End Select
```

Explanation

The preceding COM IDL example in point 2 can be explained as follows:

- 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 type.
- 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 a case `FALSE` are both defined).

Graphical Overview

Figure 41 provides a graphical overview of the interfaces that the Automation view object supports, based on the example of the OMG IDL union, `U`.

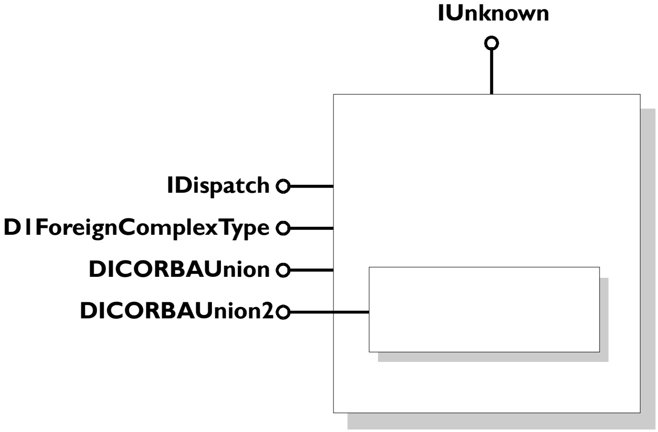


Figure 41: *Automation View of the OMG IDL Union, U*

Mapping for Sequences

Overview

An OMG IDL sequence maps to an Automation SafeArray.

Mapping to SafeArrays

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.

Example

The example can be broken down as follows:

1. Consider the following OMG IDL, which defines both a 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)
        …
    };
};
```

2. The preceding OMG IDL maps to the following in Automation:

```
// 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_reval);
    HRESULT sortAccounts ([in] IDispatch* toSort,
        [optional, out] VARIANT* IT_Ex,
        [retval, out] IDispatch** IT_retval);
};
```

3. The following Visual Basic example is based on the preceding COM IDL:

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

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

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

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

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

Mapping for Arrays

Overview

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

Mapping to SafeArrays

Multidimensional OMG IDL arrays map to `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.

Mapping to OLE Collections

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

Mapping for System Exceptions

Overview

The CORBA model uses exceptions to report error information. System exceptions can be raised by any operation. However, system exceptions are not defined at the OMG IDL level. A standard set of system exceptions is defined by CORBA, and Orbix provides a number of additional system exceptions. See the Orbix documentation set for details of the system exceptions available.

A CORBA system exception maps to the `DICORBASystemException` Automation interface, which is a pseudo-Automation interface (or pseudo-exception) that derives from `DIForeignException`. See “COMet API Reference” on page 217 for more details of these interfaces.

Example

Consider the following example of how a CORBA system exception is defined in Automation:

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

Explanation

The attributes shown in the preceding example for system exceptions can be described as follows:

`EX_minorCode` This defines the type of system exception raised.

`EX_completionStatus` This takes one of the following 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;
```

Mapping for User Exceptions

Overview

The CORBA model uses exceptions to report error information. 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 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.

Example

The example can be broken down as follows:

1. Consider the following OMG IDL:

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

2. The preceding OMG IDL maps to the following in Automation:

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

Graphical Overview for User Exceptions

Figure 42 provides a graphical overview of the interfaces that the Automation view object supports, based on the example of the OMG IDL `Bank::Reject` exception.

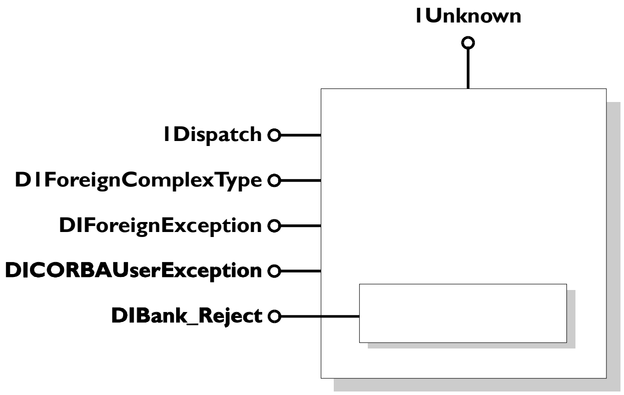


Figure 42: *Automation View of Bank_Reject*

Mapping for the Any Type

Overview

The OMG IDL any type translates to an OLE `VARIANT` type.

Containing a Simple Type

If the any contains a simple data type, it maps to a `VARIANT` type that contains a corresponding simple type. See Table 6 on page 315 for details of the mappings for basic types.

Containing a Complex Type

If the any contains a complex type, the `VARIANT` type contains an `IDispatch` view of the CORBA type.

Containing a Sequence or Array

If the any contains a CORBA sequence or `array` type, the `VARIANT` type contains an Automation SafeArray. See "Mapping for Sequences" on page 339 and "Mapping for Arrays" on page 342 for more details.

Mapping for Object References

Overview

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.

Example

The example can be broken down as follows:

1. Consider the following OMG IDL:

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

2. The preceding OMG IDL maps to the following in Automation:

```
// 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);
};
```

IForeignObject Interface

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

Visual Basic Example

The following Visual Basic example is based on the preceding mapping rules for object references:

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

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

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

' Use the returned object reference
accountObj.makeDeposit 231.98

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

Mapping for Modules

Overview

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.

Example

The example can be broken down as follows:

1. Consider the following OMG IDL:

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

2. The preceding OMG IDL maps to the following in Automation:

```
// COM IDL
[oleautomation, dual, uuid(…), helpstring("Finance_Bank")]
interface DIFinance_Bank : IDispatch {
    …
}
```

3. The preceding example can then be used as follows, for example, in Visual Basic:

```
' Visual Basic
Dim bankObj As DIFinance_Bank
```

Mapping for Constants

Overview

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. (See “Mapping for Scoped Names” on page 355 for more details.)

Example

The example can be broken down as follows:

1. Consider the following OMG IDL constant definition:

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

2. The preceding constant definition can be represented as follows in Visual Basic:

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

Alternatively, the preceding constant definition in point 1 can be represented as follows in PowerBuilder:

```
// PowerBuilder
CONSTANT long Max=1000
```

Mapping for Enums

Overview

A CORBA enum maps to an Automation enum.

Example

The example can be broken down as follows:

1. Consider the following OMG IDL:

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

2. The preceding OMG IDL maps to the following in Automation:

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

Runtime Errors

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

Enums within an Interface or Module

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. (See "Mapping for Scoped Names" on page 355 for more details.)

Enums at Global Scope

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

Mapping for Scoped Names

Overview

An OMG IDL scoped name maps to an Automation identifier where the scope operator, `::`, is replaced with an underscore.

Example

The example can be broken down as follows:

1. Consider the following OMG IDL:

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

2. The preceding OMG IDL yields the scoped name, `Finance::Bank::PersonnelRecord`.
3. The preceding scoped name maps to the Automation identifier, `Finance_Bank_PersonnelRecord`.

Mapping for Typedefs

Overview

The mapping of an OMG IDL typedef to Automation depends on the OMG IDL type for which the typedef is defined. A typedef definition is most often used for array and sequence definitions.

There is no mapping provided for typedefs for the basic OMG IDL types listed in Table 6 on page 315. Therefore, a Visual Basic programmer cannot make use of these typedef definitions for basic types.

Example

The example can be broken down as follows:

1. Consider the following OMG IDL:

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

2. The preceding OMG IDL can be used as follows in Visual Basic:

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

CHAPTER 14

Mapping CORBA to COM

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

In This Chapter

This chapter discusses the following topics:

Note: For the purposes of illustration, this chapter describes a textual mapping between OMG IDL and Microsoft IDL. COMet 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 COMet at runtime.

Basic Types

Overview

OMG IDL basic types translate to compatible types in COM.

Mapping Rules

Table 7 shows the mapping rules for each basic type.

Table 7: *CORBA-to-COM Mapping Rules for Basic Types*

OMG IDL	Description	Microsoft 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

Mapping for Strings

Overview

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

Example for Unbounded Strings

The example can be broken down as follows:

1. Consider the following OMG IDL definition for an unbounded string:

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

2. The preceding OMG IDL maps to the following Microsoft IDL:

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

Example for **Bounded Strings**

The example can be broken down as follows:

1. Consider the following OMG IDL definition for a bounded string:

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

2. The preceding OMG IDL maps to the following Microsoft IDL:

```
// Microsoft IDL
const long N = …;
typdef [string, unique] char (*BOUNDED_STRING) [N];
```

Mapping for Interfaces

Overview

This section describes how OMG IDL interfaces map to COM.

In This Section

This section discusses the following topics:

Mapping Interface Identifiers

Overview

An OMG IDL repository ID maps to a Microsoft IDL IID. All COM views that are mapped from a particular CORBA interface must share the same COM IID.

MD5 Algorithm

The mapping for interface identifiers is achieved by using a derivative of the RSA Data Security Inc. MD5 Message-Digest algorithm. The repository ID for the CORBA interface is fed into the algorithm to produce the IID, which is a 128-bit hash identifier. (A hash is a number generated by a formula from a text string.) The generated IID is then used for a COM view of a CORBA interface.

DCE UUID

One exception to the rule is if the repository ID is a DCE UUID, and the IID generated is for a COM interface (as opposed to an Automation or Automation dual interface). In this case, the DCE UUID (and not the generated IID) is used as the IID. This is to allow a scenario where CORBA server developers can implement existing COM interfaces.

Implicit Assumption

The mapping for interface identifiers implicitly assumes that repository IDs are identical across ORBs for the same interface, and unique across ORBs for different interfaces. This is necessary if IIOP is to function correctly across ORBs.

Mapping for Nested Types

Overview

OMG IDL and Microsoft IDL do not share the same rules for the scoping level of types declared within interfaces. OMG IDL considers a type to be scoped within its enclosing module or interface. Microsoft IDL considers all types to be declared at global scope. To avoid accidental name collisions, therefore, types declared within OMG IDL interfaces and modules must be fully qualified in Microsoft IDL.

Example

The example can be broken down as follows:

1. Consider the following OMG IDL:

```
// OMG IDL
module MyModule {
    interface MyInterface {
        enum type {TYPE1, TYPE2};
        struct MyStruct {
            string mystring;
            float myfloat;
            type mykind;
        };
        void myop (in MyStruct val);
    };
```

2. The preceding OMG IDL maps to the following Microsoft IDL:

```
// Microsoft IDL
[uuid(…), object]
interface IMyModule MyInterface : IUnknown {
   typedef [v1 enum] enum
      {MyModule MyInterface TYPE1,
      MyModule MyInterface TYPE2} MyModule MyInterface type;
   typedef struct {
       LPTSTR account;
       MyModule MyInterface type mykind;
   } MyModule MyInterface MyStruct;
   HRESULT myop (in MyModule MyInterface MyStruct *val);
};
```

Mapping for Attributes

Overview

An OMG IDL attribute maps to a Microsoft 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.

Example

The example can be broken down as follows:

1. Consider the following OMG IDL:

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

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

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

2. The Profile attribute in the preceding OMG IDL maps to the following Microsoft IDL:

   ```
   // Microsoft 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 attribute, Balance, in the preceding OMG IDL in point 1 maps to the following Microsoft IDL:

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

Note: The get method returns the attribute value contained in the [out] parameter.

Mapping for Operations

Overview

An OMG IDL operation maps to a Microsoft IDL method.

Example

The example can be broken down as follows:

1. Consider the following OMG IDL:

```
// 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);
};
```

2. The preceding OMG IDL maps to the following Microsoft IDL:

```
// Microsoft 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);
};
```

Rules for Parameter Passing and Return Types

The following mapping rules apply for parameter-passing modes and return types:

- An OMG IDL `in` parameter maps to a Microsoft IDL `[in]` parameter.
- An OMG IDL `out` parameter maps to a Microsoft IDL `[out]` parameter.
- An OMG IDL `inout` parameter maps to a Microsoft IDL `[in,out]` parameter.

- An OMG IDL return type maps to a Microsoft IDL [out] parameter as the last parameter in the signature.

Indirection Levels for Parameters

The following rules exist for operation parameters in terms of indirection levels:

- Integral types (for example, long, char, enum) are passed by value as in parameters, and are passed by reference as out parameters.
- Strings are passed as LPSTR as in parameters, and are passed as LPSTR* as out parameters.
- Complex types (for example, union, struct, exception) are always passed by reference.
- Optional parameters are passed using double indirection (for example, IntfException ** val).

Operations with Oneway Attribute

An OMG IDL operation that is defined with the oneway attribute maps to Microsoft IDL in the same way as an operation that has no output arguments.

Mapping for Interface Inheritance

Overview

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

Mapping Rules

The CORBA-to-COM mapping rules for an interface hierarchy are as follows:

- Each OMG IDL interface name is preceded by the letter `I` in the corresponding Microsoft IDL definition.
- If the interface is scoped by OMG IDL modules, using `::`, this is replaced by an underscore in Microsoft IDL (for example, `mymodule::myinterface` maps to `Imymodule_myinterface`).
- Each OMG IDL interface that does not have a parent maps to a Microsoft IDL interface derived from the `IUnknown` interface.
- Each OMG IDL interface that inherits from a single parent interface maps to a Microsoft IDL interface derived from the mapping for the parent interface.
- Each OMG IDL interface that inherits from multiple parent interfaces maps to a Microsoft IDL interface derived from the `IUnknown` interface. This Microsoft IDL interface then aggregates both base interfaces.
- For each CORBA interface, the mapping for operations precedes the mapping for attributes.
- Operations are sorted in ascending order, based on the ISO Latin-1 encoding values of the respective operation names.
- Attributes are sorted in ascending order, based on the ISO Latin-1 encoding values of the respective attribute names. For read-write attributes, the `get_`*`attribute_name`* method immediately precedes the `set_`*`attribute_name`* method.

Interface Hierarchy Example

Figure 43 shows an example of a CORBA interface hierarchy.

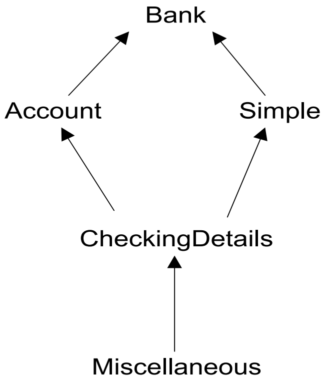


Figure 43: *Example of a CORBA Interface Hierarchy*

Interface Hierarchy Explanation

The hierarchy in Figure 43 can be explained as follows:

- `Account` and `Simple` derive from `Bank`.
- `CheckingDetails` derives from `Account` and `Simple`.
- `Miscellaneous` derives from `CheckingDetails`.

Code Example

The example can be broken down as follows:

1. Consider the following OMG IDL, which represents an interface hierarchy based on the example shown in Figure 43 on page 369:

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

2. The preceding OMG IDL maps to the following Microsoft IDL:

```
// Microsoft 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 IMiscellaneous: IUnknown
{
    HRESULT opMiscellaneous();
};
```

Mapping for Complex Types

Overview

OMG IDL includes a number of types that do not have counterparts in Microsoft IDL. This section describes the CORBA-to-COM mapping rules for these complex types.

In This Section

This section discusses the following topics:

Note: There is no standard CORBA-to-COM mapping specified for OMG IDL context clauses.

Creating Constructed OMG IDL Types

Overview

OMG IDL constructed types such as `struct`, `union`, `sequence`, and `exception` map to corresponding struct types in Microsoft IDL.

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

Mapping for Structs

Overview

An OMG IDL struct maps to a Microsoft IDL struct.

Example

The example can be broken down as follows:

1. Consider the following OMG IDL:

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

2. The preceding OMG IDL maps to the following Microsoft IDL:

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

Example for Self-Referential Types

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

1. Consider the following OMG IDL:

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

2. The preceding OMG IDL maps to the following Microsoft IDL:

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

Mapping for Unions

Overview

A discriminated union in OMG IDL maps to an encapsulated union in Microsoft IDL.

Example

The example can be broken down as follows:

1. Consider the following OMG IDL:

```
// 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]; };
```

2. The preceding OMG IDL maps to the following Microsoft IDL:

```
// Microsoft 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
```

Mapping for Sequences

Overview

OMG IDL sequences have no direct corresponding type in COM. An OMG IDL sequence can be bounded (that is, of fixed length) or unbounded (that is, of variable length). An OMG IDL sequence maps to a COM structure.

Example for Unbounded Sequences

The example can be broken down as follows:

1. Consider the following OMG IDL, which defines an unbounded sequence of some type, T:

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

2. The preceding OMG IDL maps to the following Microsoft IDL, which defines a COM structure containing a pointer to the first element, with a length and member indicating the total number of elements in the sequence:

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

Explanation for Unbounded Sequences

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

Example for Bounded Sequences

The example can be broken down as follows:

1. Consider the following OMG IDL, which defines a bounded sequence of some type, `T`, which can grow to be `N` size:

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

2. The preceding OMG IDL maps to the following Microsoft IDL, which defines a COM structure containing a fixed-size array of data elements:

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

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

Mapping for Arrays

Overview

OMG IDL arrays map to corresponding COM arrays. The array element types follow their standard mapping rules.

Example

The example can be broken down as follows:

1. Consider the following OMG IDL, which defines an array of some type, T:

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

2. The preceding OMG IDL maps to the following Microsoft IDL, which defines an array of type U:

```
// Microsoft IDL
const long N = …;
typedef … U;
typedef U ARRAY_OF_U[N];
```

Explanation

In the preceding example, the Microsoft IDL array of type U is the result of mapping the OMG IDL, T, into Microsoft IDL.

If the ellipsis (that is, …) shown in the preceding example represents octet in the OMG IDL, the ellipsis must be byte in the Microsoft IDL. This is why the types of the array elements have different names in the OMG IDL and Microsoft IDL defintions.

Mapping for System Exceptions

Overview

The CORBA model uses exceptions to report error information. System exceptions can be raised by any operation, regardless of the interface on which the operation was invoked. A standard set of system exceptions is defined by CORBA, and Orbix provides a number of additional system exceptions. See the Orbix documentation set for details about the system exceptions available.

Rules

There are two aspects to the mapping of CORBA system exceptions to COM:

- Exceptions must be returned to COM clients via the COM `HRESULT` return type. Therefore, the CORBA exception is mapped to one of the standard COM `HRESULT` values. When a CORBA system exception is raised, the COM view in the bridge returns the `HRESULT` to the client.
- Additional information pertaining to the system exception (for example, its minor code and repository ID) cannot be mapped to the `HRESULT` value. Instead, additional information can be returned to the client via a standard COM error object. Writing information to an error object is, however, optional.

Error Object

Because it is not possible to map information such as a CORBA system exception's minor code and repository ID to the `HRESULT` value, you can choose to have this additional exception information written to a COM error object, and returned to the client that way.

If you use an error object, the COM view must support the `ISupportErrorInfo` interface. If a COM client call results in a system exception, the COM view must call the COM `SetErrorInfo()` function, to set the error object to the client's calling thread. This allows the client to retrieve the error object, to report the error to the user. Even if no system exception occurs, the COM view must still call `SetErrorInfo()`, this time with a null value for the `IErrorInfo` pointer parameter, to ensure that the error object on that thread is destroyed.

Error Object Properties

The properties of the error object are set as shown in Table 8.

Table 8: *Using Error Object for CORBA System Exceptions*

Property	Description
`bstrSource`	This takes the following format: *`interfacename.operationname`* The interface and operation name pertain to the CORBA interface that the view represents.

`bstrDescription`	This takes the following format:
	`CORBA System Exception: [`*`repository ID`*`]`
	`minor code[`*`minor code`*`][`*`completion status`*`]`
	The *`repository ID`* and *`minor code`* are those of the system exception. The *`completion status`* can be `YES`, `NO`, or `MAYBE`, depending on the value of the system exception's CORBA completion status.

`bstrHelpFile`	This is unspecified.
`dwHelpContext`	This is unspecified.
`GUID`	This is the IDD of the COM view interface.

Example

The example can be broken down as follows:

1. Consider the following COM C++ code for a COM view that supports error objects:

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

2. The following COM C++ client code shows how a client can access the error object:

```
// 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(…);
     }
}
```

Mapping for User Exceptions

Overview

The CORBA model uses exceptions to report error information. 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.

An OMG IDL user-defined exception maps to a Microsoft IDL interface and an exception structure that describes the body of information to be returned for the exception to the client.

For the purpose of allowing access to user exception information, a Microsoft IDL interface is defined for each OMG IDL interface that can raise a user exception. The name of the Microsoft IDL interface is based on the fully scoped name of the OMG IDL interface on which the exception is raised.

An exception structure is defined for each user exception. The exception structure is specified as an output parameter, and it appears as the last parameter of any COM operation signature that has been mapped from any OMG IDL operation with a `raises` clause. For example, if an operation in `MyModule::MyInterface` raises a user exception, an exception structure named `MyModule_MyInterfaceExceptions` is created and mapped as an output parameter to Microsoft IDL. This extra parameter is passed by indirect reference, to allow it to be treated as optional by the target server side.

Exception Structure

Although a COM view can call `SetErrorInfo()` to indicate a CORBA user exception has occurred (as in the case of a CORBA system exception), there is no mechanism in COM to allow for accessing the additional data members defined on a user exception object. The additional error information is therefore mapped to an exception structure instead.

The exception structure contains:

- Members indicating the exception type.
- The repository ID for the exception definition in the CORBA Interface Repository.
- A pointer to the exception data.

Mapped Operations

Each exception that can be raised by an operation is mapped to an operation on the Exception interface. The mapped operation name is constructed by prefixing the exception name with get_. Each mapped operation takes one output parameter, of the struct type, which is used to return the exception information. Each mapped operation is defined to return a HRESULT value, for which the exact value depends on the type of exception raised and whether a structure has been specified by the client.

HRESULT for Successful Operations

If the call to a particular operation is successful and does not raise a user exception, a HRESULT value of S_OK is returned, to indicate that the operation has been successful.

Example

The example can be broken down as follows:

1. Consider the following OMG IDL:

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

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

2. The preceding OMG IDL maps to the following Microsoft IDL:

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

interface IBANK_AccountUserExceptions: IUnknown
{
    HRESULT get_InsufficientFunds([out] BANK_InsufficientFunds
        *exceptionBody);
    HRESULT get_InvalidAmount([out] BANK_InvalidAmount
        *exceptionBody);
    HRESULT get_NotAuthorised([out] BANK_Account_NotAuthorised
        *exceptionBody);
};
typedef struct
{
    ExceptionType type;
    LPSTR repositoryId;
    IBANK_AccountUserExceptions * piUserException;
} BANK_AccountExceptions
```

Mapping for the Any Type

Overview

The OMG IDL any type does not map directly to COM.

Example

The following is the Microsoft IDL interface definition to which the OMG IDL any type is mapped:

```
// Microsoft 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);
}
```

Mapping for Object References

Overview

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 Microsoft IDL.

Example

The example can be broken down as follows:

1. Consider the following OMG IDL:

```
// OMG IDL
interface Account {
     …
};
interface Bank {
     Account newAccount(in string name);
     deleteAccount(in Account a);
};
```

2. The preceding OMG IDL maps to the following Microsoft IDL:

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

3. The following COM C++ code is based on the preceding Microsoft IDL definition:

```
// COM C++
// 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();
```

Mapping for Modules

Overview

An OMG IDL definition contained within the scope of an OMG IDL module maps to its corresponding Microsoft IDL definition, by prefixing the name of the Microsoft IDL type definition with the name of the module.

Example

The example can be broken down as follows:

1. Consider the following OMG IDL:

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

2. The preceding OMG IDL maps to the following Microsoft IDL:

```
// Microsoft IDL
[object, uuid(…), helpstring("Finance_Bank")]
interface IFinance_Bank : IUnknown {
     …
}
```

Mapping for Constants

Overview

An OMG IDL const type maps to a Microsoft IDL const type.

Example

The example can be broken down as follows:

1. Consider the following OMG IDL:

```
// 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 = “…”;
```

2. The preceding OMG IDL maps to the following Microsoft IDL:

```
// Microsoft 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 = “…”;
```

Scoping of Constant Declarations

CORBA observes scoping of constant declarations, but COM ignores such scoping and always treats a constant declaration as though it were globally defined. To avoid potential name clashes, mapped constants in Microsoft IDL are prefixed with the enclosing type in which they are declared. For example, consider the following OMG IDL:

```
// OMG IDL
module PhoneCompany {
     interface CustomerServices {
         const float CallRate = 11.7;
     };
};
```

The preceding OMG IDL maps to the following Microsoft IDL:

```
// Microsoft IDL
const float PhoneCompany_CustomerServices_CallRate = 11.7;
```

Mapping for Enums

Overview

A CORBA enum maps to a COM enum.

Example

The example can be broken down as follows:

1. Consider the following OMG IDL:

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

2. The preceding OMG IDL maps to the following Microsoft IDL:

```
// Microsoft 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;
};
```

Values and Ordering

CORBA has enums that are not explicitly tagged with values. On the other hand, Microsoft 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.

Scoping

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 Microsoft 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`.

Transmitting as 32-Bit

The Microsoft 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.

Truncation of Identifiers

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

Mapping for Scoped Names

Overview

An OMG IDL scoped name must be fully qualified in Microsoft IDL, to prevent accidental name collisions.

Example

The example can be broken down as follows:

1. Consider the following OMG IDL:

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

2. The preceding OMG IDL maps to the following Microsoft IDL:

```
Microsoft 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);
};
```

Mapping for Typedefs

Overview

A CORBA typedef maps to a Microsoft IDL typedef. A typedef definition is most often used for array and sequence definitions.

Example

The example can be broken down as follows:

1. Consider the following OMG IDL:

```
// OMG IDL
interface Account {…};

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

2. The preceding OMG IDL maps to the following Microsoft IDL:

```
// Microsoft IDL
[object, UUID(…)]
interface IAccount : IUnknown {…};
Typedef struct {
…
} AccountList;
```

CHAPTER 15

COMet Configuration

This chapter describes the configuration variables that are specific to COMet, and their associated values.

In This Chapter

This chapter discusses the following topics:

Overview

Configuration Domains

Configuration variables are stored in a configuration domain. A configuration domain can be based on one of two distinct configuration models, depending on whether your deployment needs are small scale or large scale. For small-scale deployment, you can implement a configuration domain as an ASCII text file that is stored locally on each machine and edited directly. For large-scale deployment, Orbix provides a distributed configuration repository server that enables centralized configuration for all applications spread across a network.

The COMet: Scope

Configuration variables specific to COMet are grouped within various namespaces within a `COMet:` scope, as follows:

- `COMet:Config:…`
- `COMet:Mapping:…`
- `COMet:Debug:…`
- COMet:Typeman:…
- COMet:Services:…

See the *CORBA Administrator’s Guide* for details of CORBA configuration variables.

COMet:Config Namespace

Overview

This section describes the configuration variables within the `COMet:Config:` namespace.

`COMET_SHUTDOWN_POLICY` The default setting for this variable is:

```
COMet:Config:COMET_SHUTDOWN_POLICY="implicit"
```

The valid settings for this variable are:

`"implicit"`	This means that COMet shuts down the first time `DllCanUnloadNow` is about to return yes.
`"explicit"`	This means that you must make a call to `ORB::ShutDown()` to force COMet to shut down.
`"Disabled"`	This means that COMet 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 arises, however, if the DLL is re-used, because Orbix has already been shut down.
`"atExit"`	This means that the COMet bridge only shuts down at process-exit time. This is the recommended setting when running in the Visual Basic development environment.

`SINGLE_THREADED_CALLBACK` The default setting for this variable is:

```
COMet:Config:SINGLE_THREADED_CALLBACK="NO"
```

The valid settings for this variable are:

`"NO"`	This means that COMet dispatches callbacks as they arrive.
`"YES"`	This means that you can implement your own event loop for processing callbacks.

USE_INTERFACE_IN_IOR The default setting for this variable is:

```
COMet:Config:USE_INTERFACE_IN_IOR="YES"
```

The valid settings for this variable are:

"YES"	This means that COMet uses the type ID that is embedded in the IOR as the interface name when narrowing to derived interfaces. This can help to improve performance at application runtime.
"NO"	This means that COMet must make remote calls to get_interface() and possibly repeated calls on the IFR when narrowing to derived interfaces. This might have an adverse affect on performance at application runtime.

COMet:Mapping Namespace

Overview

This section describes the configuration variables within the COMet:Mapping: namespace.

SAFEARRAYS_CONTAIN_VARIANT The default setting for this variable is:

```
COMet:Mapping:SAFEARRAYS_CONTAIN_VARIANTS="yes"
```

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 they are of the VARIANT type. When constructing the out parameter, COMet 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.

The valid settings for this variable are:

"yes"	This means that COMet should treat, for example, a sequence of long types as mapping to a SafeArray of VARIANT types, where each VARIANT contains a long.
"no"	This means that COMet should treat, for example, a sequence of long types as mapping to a SafeArray of long types.

KEYWORDS An example setting for this variable is:

```
COMet:Mapping:KEYWORDS="grid, DialogBox, bar, Foobar, height"
```

This variable allows you to specify a list of words that are to be prefixed with IT_, to avoid clashes when using ts2idl to generate Microsoft IDL from existing OMG IDL type information in the type store.

COMet:Debug Namespace

Overview

This section describes the configuration variable within the `COMet:Debug:` namespace.

`MessageLevel` An example setting for this variable is:

```
COMet:Debug:MessageLevel="255, c:\temp\comet.log"
```

This variable 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.

COMet:TypeMan Namespace

Overview

This section describes the configuration variables within the `COMet:TypeMan:` namespace.

`TYPEMAN_CACHE_FILE` The default setting for this variable is:

```
COMet:TypeMan:TYPEMAN_CACHE_FILE="install-dir\var\it_domainname\
    dbs\comet"
```

COMet uses a memory and disk cache for efficient access to type information. This entry specifies the name and location of the file used. It is automatically set by the configuration script. In the preceding example, *`install-dir`* represents the Orbix installation directory, and *`domainname`* represents your domain name.

`TYPEMAN_DISK_CACHE_SIZE` The default setting for this variable is:

```
COMet:TypeMan:TYPEMAN_DISK_CACHE_SIZE="2000"
```

This variable is used in conjunction with `TYPEMAN_MEM_CACHE_SIZE`. It specifies the maximum number of entries allowed in the disk cache. When this value is exceeded, entries can be flushed from the cache. The nature of the applications using the bridge affects the value that should be assigned to this variable. However, as a general rule, the disk cache size should be about eight to ten times greater than the the memory cache. (See "TYPEMAN_MEM_CACHE_SIZE" on page 406 for more details about setting the maximum number of entries for the memory cache.)

A cache "entry" in this case corresponds to a user-defined type. For example, a union defined in OMG IDL results in one entry in the cache. An interface containing the definition of a structure results in two entries.

A good rule of thumb is that 1000 cache entries (given a representative cross section of user-defined types) corresponds to approximately 2 megabytes of disk space. Therefore, the default disk cache size of 2000 allows for a maximum disk cache file size of approximately 4 megabytes.

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 is more than three times the size of the original type library, because the information is stored in a format that optimizes speed.

TYPEMAN_MEM_CACHE_SIZE The default setting for this variable is:

```
COMet:TypeMan:TYPEMAN_MEM_CACHE_SIZE="250"
```

This variable is used in conjunction with TYPEMAN_DISK_CACHE_SIZE. It specifies the maximum number of entries allowed in the memory cache. When this value is exceeded, entries can be flushed from the cache. The nature of the applications using the bridge affects the value that should be assigned to this variable. 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. See "TYPEMAN_DISK_CACHE_SIZE" on page 405 for more details.

TYPEMAN_IFR_IOR_FILENAME The default setting for this variable is:

```
COMet:TypeMan:TYPEMAN_IFR_IOR_FILENAME=" "
```

When the dynamic marshalling engine in COMet encounters a type for which it cannot find corresponding type information in the type store, it must then retrieve the type information from the Interface Repository. The order in which COMet attempts to connect to the Interface Repository is as follows:

- If a name is specified in the COMet:TypeMan:TYPEMAN_IFR_NS_NAME variable, COMet looks up that name in the Naming Service to connect to the Interface Repository.
- If a name is not specified in COMet:TypeMan:TYPEMAN_IFR_NS_NAME, COMet checks to see if an IOR is specified in the initial_references:InterfaceRepository:reference variable. If so, it uses the Interface Repository associated with that IOR.

- If an IOR is not specified in initial_references:InterfaceRepository:reference, COMet checks to see if a filename is specified in the TYPEMAN_IFR_IOR_FILENAME variable.

Consequently, you must set the TYPEMAN_IFR_IOR_FILENAME variable if you do not set COMet:TypeMan:TYPEMAN:IFR_NS_NAME or initial_references:InterfaceRepository:reference. In this case, the value required is the full pathname to the file that contains the IOR for the Interface Repository you want to use.

TYPEMAN_IFR_NS_NAME The default setting for this variable is:

```
COMet:TypeMan:TYPEMAN_IFR_NS_NAME=" "
```

This variable is needed if you are using the Naming Service to resolve the Interface Repository. It specifies the name of the Interface Repository in the Naming Service. You should register an IOR for the Interface Repository in the Naming Service under a compound name. This variable should contain that compound name. As explained in "TYPEMAN_IFR_IOR_FILENAME" on page 406, this is the first configuration variable that COMet always checks if it needs to contact the Interface Repository for type information that it cannot find in the type store.

TYPEMAN_READONLY The default setting for this variable is:

```
COMet:TypeMan:TYPEMAN_READONLY="no"
```

The valid settings for this variable are:

"no"	This means that clients have write access to the type store.
"yes"	This means that clients have readonly access to the type store.

This variable specifies whether clients have write access or readonly access to the type store. If you have a scenario involving multiple Automation clients sharing a single out-of-process bridge, it means that all your clients are using one central type store. If clients are granted write access to the type store, the type store is blocked whenever it is in use by a particular client, and all other clients must wait until that client is finished using it. This can have a negative impact on both performance and scalability. It is therefore recommended that you set this configuration variable to "yes", to only allow clients readonly access to the type store.

`TYPEMAN_LOGGING` The default setting for this variable is:

```
COMet:TypeMan:TYPEMAN_LOGGING="none"
```

The valid settings for this variable are:

`"none"` This means that no logging information is output for the COMet type store manager (`typeman`).

`"stdout"` This means that logging information is used only with `typeman.exe`.

`"DBMon"` This means that logging information is output to `DBMon.exe`.

`"file"` This means that logging information is output to the file specified by the `COMet:Typeman:TYPEMAN_LOG_FILE` variable.

`TYPEMAN_LOG_FILE` An example setting for this variable is:

```
COMet:TypeMan:TYPEMAN_LOG_FILE="c:\temp\typeman.log"
```

If the value of the `TYPEMAN_LOGGING` variable is set to `"file"`, this variable specifies the full path to that output file for `typeman` logging information.

COMet:Services Namespace

Overview

This section describes the configuration variable within the COMet:Services: namespace.

NameService The default setting for this variable is:

```
COMet:Services:NameService=" "
```

By default, COMet uses the Naming Service that is specified in the Orbix initial_references:NameService: configuration scope. If (and only if) the value specified for that configuration variable is blank, or it relates to an invalid IOR, COMet then uses the Naming Service that is specified by the COMet:Services:NameService configuration variable. The value specified is the full pathname to the file that contains the IOR for the Naming Service you want to use.

CHAPTER 16

COMet Utility Arguments

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

In This Chapter

This chapter discusses the following topics:

Typeman Arguments

Overview

This section describes the arguments available with the typeman utility, which manages the COMet type store.

Summary of Arguments

The arguments 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. COMet generates UUIDs for OMG IDL types, using the MD5 algorithm, as specified by the OMG.)

-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. See "Adding New Information to the Type Store" on page 180 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 that 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 simultaneously with any number and combination of type names, type library pathnames, and COM UUIDs.

Ts2idl Arguments

Overview

This section describes the arguments available with the ts2idl utility, which allows you to create COM IDL definitions, based on existing OMG IDL type information in the type store.

Summary of Arguments

The arguments available with ts2idl are:

-c This instructs ts2idl not to query the Interface Repository for the specified OMG IDL interface. In this case, ts2idl searches only 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
```

-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.

-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 interfaces that employ user-defined types. This 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 Arguments

Overview

This section describes the arguments available with the `ts2tlb` utility, which allows you to create a type library, based on existing OMG IDL type information in the type store.

Summary of Arguments

The arguments 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.

Aliassrv Arguments

Overview

This section describes the arguments available with the `aliassrv` utility, which is used in association with the `srvAlias` GUI tool, to allow you to replace a legacy DCOM server with a CORBA server. See “Replacing an Existing DCOM Server” on page 196 for more details.

Summary of Arguments

The arguments available with `aliassrv` are:

`-c` This indicates the CLSID of the legacy DCOM server that is being replaced. You must qualify this argument 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` argument, which in turn must be qualified with the CLSID.

`-r` This aliases the specified CLSID to COMet, so that the next time you run a DCOM client of the legacy server whose CLSID is specified, COMet 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 `aliassrv`. You can also use `-?` for this.

Custsur Arguments

Overview

This section describes the arguments available with the `custsur` utility, which is a generic surrogate program that hosts the COMet DLLs when the bridge is loaded out-of-process. You can use `custsur` to generate IORs for non-Orbix clients.

Summary of Arguments

The arguments 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 timeout value, in milliseconds, for the server being implemented by `custsur`.
`-v`	This outputs the usage string for `custsur`. You can also use `-?` for this.

Tlibreg Arguments

Overview

This section describes the arguments available with the `tlibreg` utility, which allows you to register and unregister a type library that you have generated from OMG IDL via `ts2tlb`. The `tlibreg` utility registers the type library with the Windows registry.

Summary of Arguments

The arguments 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.

Idlgen vb_genie.tcl Arguments

Overview

The Visual Basic code generation genie allows for quick, easy, and automatic development of Visual Basic clients from existing OMG IDL definitions. It can be run from the command line, using the following command format:

```
idlgen vb_genie.tcl [options] filename.idl [interface wildcard]*
```

In the preceding format, *filename* represents the name of the OMG IDL file from which the Visual Basic code is generated.

Summary of Arguments

The arguments available with `idlgen vb_genie.tcl` are:

`-I` Before `idlgen` parses an IDL file, it sends the IDL file through an IDL preprocessor. The `-I` argument is one of two arguments that allow you to pass information to the IDL preprocessor. Specifically, `-I` lets you specify the include path for the preprocessor. For example:

```
idlgen vb_genie.tcl -I/inc -I…/std/inc bank.idl
```

`-D` The `-D` argument also allows you to pass information to the IDL preprocessor. Specifically, `-D` lets you define additional preprocessor symbols. For example:

```
idlgen vb_genie.tcl -I/inc -DDEBUG
```

`-h` This outputs the usage string for `idlgen vb_genie.tcl`.

`-v` This indicates that the genie is to run in verbose mode (that is, diagnostic messages are written to standard output when the genie is generating an output file).

`-s` This indicates that the genie is to run in silent mode (that is, diagnostic messages are not written to standard output when the genie is generating an output file).

`-dir` This specifies the directory path to which the generated file is to be output. This option must be qualified by a full directory path. If `-dir` is not specified, all output files are written to the current directory.

`-include` By default, the genie generates client code for the specified IDL files only. This argument allows you to specify that the genie must also generate code for all `#include` files specifed in the IDL. For example:

```
idlgen vb_genie.tcl -all -include grid.idl
```

The preceding example specifies that the genie is to generate Visual Basic client code from `grid.idl` and any IDL files that are included in it.

`-nons` This indicates that stringified object references are to be written to an IOR file, instead of using the Naming Service. This is the default setting. The IOR filename consists of the interface name and `.ref` suffix. This argument is mutually exclusive with the `-ns` argument.

Specify this argument only if it was also specified when generating the CORBA server with the CORBA Code Generation Toolkit.

`-ns` This indicates that the Naming Service is to be used to publish object references, instead of writing them to an IOR file by default. This argument is mutually exclusive with the `-nons` argument.

Specify this argument only if it was also specified when generating the CORBA server with the CORBA Code Generation Toolkit.

Index