The Common Object Request Broker: Architecture and Specification

Revision 2.0, July 1995
Updated July 1996

Table of Contents

Preface

0.1 About This Document

Under the terms of the collaboration between OMG and X/Open Co Ltd., this document is a candidate for endorsement by X/Open, initially as a Preliminary Specification and later as a full CAE Specification. The collaboration between OMG and X/Open Co Ltd. ensures joint review and cohesive support for emerging object-based specifications.

X/Open Preliminary Specifications undergo close scrutiny through a review process at X/Open before publication and are inherently stable specifications. Upgrade to full CAE Specification, after a reasonable interval, takes place following further review by X/Open. This further review considers the implementation experience of members and the full implications of conformance and branding.

0.1.1 Object Management Group

The Object Management Group, Inc. (OMG) is an international organization supported by over 500 members, including information system vendors, software developers and users. Founded in 1989, the OMG promotes the theory and practice of object-oriented technology in software development. The organization's charter includes the establishment of industry guidelines and object management specifications to provide a common framework for application development. Primary goals are the reusability, portability, and interoperability of object-based software in distributed, heterogeneous environments. Conformance to these specifications will make it possible to develop a heterogeneous applications environment across all major hardware platforms and operating systems.

OMG's objectives are to foster the growth of object technology and influence its direction by establishing the Object Management Architecture (OMA). The OMA provides the conceptual infrastructure upon which all OMG specifications are based.

0.1.2 X/Open

X/Open is an independent, worldwide, open systems organization supported by most of the world's largest information system suppliers, user organizations and software companies. Its mission is to bring to users greater value from computing, through the practical implementation of open systems. X/Open’s strategy for achieving its mission is to combine existing and emerging standards into a comprehensive, integrated systems environment called the Common Applications Environment (CAE).

The components of the CAE are defined in X/Open CAE specifications. These contain, among other things, an evolving portfolio of practical application programming interfaces (APIs), which significantly enhance portability of application programs at the source code level. The APIs also enhance the interoperability of applications by providing definitions of, and references to, protocols and protocol profiles.

The X/Open specifications are also supported by an extensive set of conformance tests and by the X/Open trademark (XPG brand), which is licensed by X/Open and is carried only on products that comply with the CAE specifications.

0.2 Intended Audience

The architecture and specifications described in this manual are aimed at software designers and developers who want to produce applications that comply with OMG standards for the Object Request Broker (ORB). The benefit of compliance is, in general, to be able to produce interoperable applications that are based on distributed, interoperating objects. As defined by the Object Management Group (OMG) in the *Object Management Architecture Guide,* the ORB provides the mechanisms by which objects transparently make requests and receive responses. Hence, the ORB provides interoperability between applications on different machines in heterogeneous distributed environments and seamlessly interconnects multiple object systems.

0.3 Context of CORBA

The key to understanding the structure of the CORBA architecture is the Reference Model, which consists of the following components:

- **Object Request Broker**, which enables objects to transparently make and receive requests and responses in a distributed environment. It is the foundation for building applications from distributed objects and for interoperability between applications in hetero- and homogeneous environments. The architecture and specifications of the Object Request Broker are described in this manual.
- **Object Services**, a collection of services (interfaces and objects) that support basic functions for using and implementing objects. Services are necessary to construct any distributed application and are always independent of application domains. For example, the Life Cycle Service defines conventions for creating, deleting, copying, and moving objects; it does not dictate how the objects are implemented in an application. Specifications for Object Services are contained in *CORBAservices: Common Object Services Specification.*

- **Common Facilities**, a collection of services that many applications may share, but which are not as fundamental as the Object Services. For instance, a system management or electronic mail facility could be classified as a common facility. Information about Common Facilities will be contained in *CORBAfacilities: Common Facilities Architecture*.
- **Application Objects,** which are products of a single vendor on in-house development group which controls their interfaces. Application Objects correspond to the traditional notion of applications, so they are not standardized by OMG. Instead, Application Objects constitute the uppermost layer of the Reference Model.

The Object Request Broker, then, is the core of the Reference Model. It is like a telephone exchange, providing the basic mechanism for making and receiving calls. Combined with the Object Services, it ensures meaningful communication between CORBA-compliant applications.

(For more information about the OMG Reference Model and the OMG Object Model, refer to the *Object Management Architecture Guide).*

0.4 Associated Documents

The CORBA documentation set includes the following books:

- *Object Management Architecture Guide* defines the OMG's technical objectives and terminology and describes the conceptual models upon which OMG standards are based. It also provides information about the policies and procedures of OMG, such as how standards are proposed, evaluated, and accepted.
- *CORBA: Common Object Request Broker Architecture and Specification* contains the architecture and specifications for the Object Request Broker.
- *CORBAservices: Common Object Services Specification* contains specifications for the Object Services.
- *CORBAfacilities: Common Facilities Architecture* contains the architecture for Common Facilities.

OMG collects information for each book in the documentation set by issuing Requests for Information, Requests for Proposals, and Requests for Comment and, with its membership, evaluating the responses. Specifications are adopted as standards only when representatives of the OMG membership accept them as such by vote.

To obtain books in the documentation set, or other OMG publications, refer to the enclosed subscription card or contact the Object Management Group, Inc. at:

OMG Headquarters
492 Old Connecticut Path
Framingham, MA 01701
USA
Tel: +1-508-820 4300
Fax: +1-508-820 4303
pubs@omg.org
http://www.omg.org/

0.5 Structure of This Manual

This manual is divided into the categories of Core, Interoperability, Interworking, and individual Language Mappings. These divisions reflect the compliance points of CORBA, as explained in Section 0.6, “Definition of CORBA Compliance,” on page 6. In addition to this preface, *CORBA: Common Object Request Broker Architecture and Specification* contains the following chapters*:*

Core

The Object Model describes the computation model that underlies the CORBA architecture.

Architecture describes the overall structure of the ORB architecture and includes information about CORBA interfaces and implementations.

OMG IDL Syntax and Semantics describes OMG interface definition language (OMG IDL), which is the language used to describe the interfaces that client objects call and object implementations provide.

The Dynamic Invocation Interface describes the DII, the client’s side of the interface that allows dynamic creation and invocation of request to objects.

The Dynamic Skeleton Interface describes the DSI, the server’s-side interface that can deliver requests from an ORB to an object implementation that does not have compile-time knowledge of the type of the object it is implementing. DSI is the server’s analogue of the client’s Dynamic Invocation Interface (DII).

Interface Repository describes the component of the ORB that manages and provides access to a collection of object definitions.

ORB Interface describes the interface to the ORB functions that do not depend on object adapters: these operations are the same for all ORBs and object implementations.

Basic Object Adapter describes the primary interface than an implementation uses to access ORB functions.

An **appendix** that contains standard OMG IDL types.

Interoperability

Interoperability Overview explains the interoperability architecture and introduces the subjects pertaining to interoperability: inter-ORB bridges; general and Internet inter-ORB protocols (GIOP and IIOP); and environment-specific, inter-ORB protocols (ESIOPs).

Interoperability Architecture introduces the framework of ORB interoperability, including information about domains; approaches to inter-ORB bridges; what it means to be compliant with ORB interoperability; and ORB Services and Requests.

Inter-ORB Bridges explains how to build bridges for an implementation of interoperating ORBs.

Inter-ORB Protocols describes the general inter-ORB protocol (GIOP) and includes information about the GIOP's goals, syntax, format, transport, and object location. This chapter also includes information about the Internet inter-ORB protocol (IIOP).

Environment-Specific Inter-ORB Protocol (ESIOP) describes a protocol for the OSF DCE environment. The protocol is called the DCE Environment Inter-ORB Protocol (DCE ESIOP).

An **appendix** containing OMG IDL tags that can identify an Object Service, a component, or a profile.

Interworking

Interworking Architecture describes the architecture for communication between two object management systems: Microsoft's COM (including OLE) and the OMG's CORBA.

Mapping: OLE Automation and CORBA describes the two-way mapping between OLE Automation (in ODL) and CORBA (in OMG IDL).

Mapping: COM and CORBA describes the data type and interface mapping between COM and CORBA. The mappings are described in the context of both Win16 and Win32 COM.

An **appendix** describing solutions that vendors might implement to support existing and older OLE Automation controllers.

An **appendix** that provides an example of how the Naming Service could be mapped to an OLE Automation interface according to the Interworking specification.

C Language Mapping

Mapping of OMG IDL to C maps OMG IDL to the C programming language.

C++ Language Mapping

C++ Mapping Overview introduces the mapping of OMG IDL to the C++ programming language.

Mapping of OMG IDL to C++ maps the constructs of OMG IDL to the C++ programming language.

Mapping of Pseudo Objects to C++ maps OMG IDL pseudo objects to the C++ programming language.

Server-Side Mapping explains the portability constraints for an object implementation written in C++.

The C++ language mapping also includes several appendices. One contains C++ definitions for CORBA, another contains alternate C++ mappings, and another contains C++ keywords.

Smalltalk Language Mapping

Smalltalk Mapping Overview introduces the mapping of OMG IDL to the Smalltalk programming language.

Mapping of OMG IDL to Smalltalk maps the constructs of OMG IDL to the Smalltalk programming language.

Mapping of Pseudo Objects to Smalltalk maps OMG IDL pseudo-objects to Smalltalk.

0.6 Definition of CORBA Compliance

As described in the *OMA Guide*, the OMG's Core Object Model consists of a core and components. Likewise, the body of *CORBA* specifications is divided into core and component-like specifications. The structure of this manual reflects that division.

The *CORBA* specifications are categorized as follows:

CORBA Core, as specified in Chapters 1-8

CORBA Interoperability, as specified in Chapters 9-13

CORBA Interworking, as specified in Chapters 13A, 13B, and 13C

Mapping of OMG IDL to the C programming language, as specified in Chapter 14

Mapping of OMG IDL to the C++ programming language, as specified in Chapters 15-18

Mapping of OMG IDL to the Smalltalk programming language, as specified in Chapters 19-21

(Additional OMG IDL mappings will be available with future updates of *CORBA*.)

The minimum required for a CORBA-compliant system is adherence to the specifications in CORBA Core and one mapping. Each additional language mapping is a separate, optional compliance point. Optional means users aren't required to implement these points if they are unnecessary at their site, but if implemented, they must adhere to the *CORBA* specifications to be called CORBA-compliant. For instance, if a vendor supports C++, their ORB must comply with the OMG IDL to C++ binding specified in this manual.

Interoperability and Interworking are separate compliance points. For detailed information about Interworking compliance, refer to Section 13.10.1, "Products Subject to Compliance," on page 13A-34.

0.7 Typographical Conventions

The type styles shown below are used in this document to distinguish programming statements from ordinary English. However, these conventions are not used in tables or section headings, where no distinction is necessary, nor are the type styles used in text where their density would be distracting.

Helvetica bold OMG Interface Definition Language (OMG IDL) language and syntax elements.

Times bold Pseudo-OMG IDL (PIDL) language elements.

`Courier bold` Programming language elements or any interface definition language **other than** OMG IDL.

Code examples written in PIDL and programming languages are further identified by a comment; unidentified examples are written in OMG IDL.

0.8 Acknowledgements

The following companies submitted parts of the specifications that were approved by the Object Management Group to become *CORBA:*

- BNR Europe Ltd.
- Expersoft Corporation
- FUJITSU LIMITED
- Genesis Development Corporation
- IBM Corporation
- ICL plc
- IONA Technologies Ltd.
- Digital Equipment Corporation
- Hewlett-Packard Company
- HyperDesk Corporation
- NCR Corporation
- Novell USG
- Object Design, Inc.
- Siemens Nixdorf Informationssysteme AG
- Sun Microsystems Inc.
- SunSoft, Inc.
- Sybase, Inc.
- Visual Edge Software, Ltd.

In addition to the preceding contributors, the OMG would like to acknowledge Mark Linton at Silicon Graphics and Doug Lea at the State University of New York at Oswego for their work on the C++ mapping.

The Object Model 1

This chapter describes the concrete object model that underlies the CORBA architecture. The model is derived from the abstract Core Object Model defined by the Object Management Group in the *Object Management Architecture Guide*. (Information about the *OMA Guide* and other books in the CORBA documentation set is provided in this document's preface.)

1.1 Overview

The object model provides an organized presentation of object concepts and terminology. It defines a partial model for computation that embodies the key characteristics of objects as realized by the submitted technologies. The OMG object model is *abstract* in that it is not directly realized by any particular technology. The model described here is a *concrete* object model. A concrete object model may differ from the abstract object model in several ways:

- It may *elaborate* the abstract object model by making it more specific, for example, by defining the form of request parameters or the language used to specify types
- It may *populate* the model by introducing specific instances of entities defined by the model, for example, specific objects, specific operations, or specific types
- It may *restrict* the model by eliminating entities or placing additional restrictions on their use

An object system is a collection of objects that isolates the requestors of services (clients) from the providers of services by a well-defined encapsulating interface. In particular, clients are isolated from the implementations of services as data representations and executable code.

The object model first describes concepts that are meaningful to clients, including such concepts as object creation and identity, requests and operations, types and signatures. It then describes concepts related to object implementations, including such concepts as methods, execution engines, and activation.

The object model is most specific and prescriptive in defining concepts meaningful to clients. The discussion of object implementation is more suggestive, with the intent of allowing maximal freedom for different object technologies to provide different ways of implementing objects.

There are some other characteristics of object systems that are outside the scope of the object model. Some of these concepts are aspects of application architecture, some are associated with specific domains to which object technology is applied. Such concepts are more properly dealt with in an architectural reference model. Examples of excluded concepts are compound objects, links, copying of objects, change management, and transactions. Also outside the scope of the object model is the model of control and execution.

This object model is an example of a classical object model, where a client sends a message to an object. Conceptually, the object interprets the message to decide what service to perform. In the classical model, a message identifies an object and zero or more actual parameters. As in most classical object models, a distinguished first parameter is required, which identifies the operation to be performed; the interpretation of the message by the object involves selecting a method based on the specified operation. Operationally, of course, method selection could be performed either by the object or the ORB.

1.2 Object Semantics

An object system provides services to clients. A *client* of a service is any entity capable of requesting the service.

This section defines the concepts associated with object semantics, that is, the concepts relevant to clients.

1.2.1 Objects

An object system includes entities known as objects. An *object* is an identifiable, encapsulated entity that provides one or more services that can be requested by a client.

1.2.2 Requests

Clients request services by issuing requests. A *request* is an event, i.e. something that occurs at a particular time. The information associated with a request consists of an operation, a target object, zero or more (actual) parameters, and an optional request context.

A *request form* is a description or pattern that can be evaluated or performed multiple times to cause the issuing of requests. As described in the OMG IDL Syntax and Semantics chapter, request forms are defined by particular language bindings. An alternative request form consists of calls to the dynamic invocation interface to create an invocation structure, add arguments to the invocation structure, and to issue the invocation (refer to the C Language Mapping chapter and the Dynamic Invocation Interface chapter for descriptions of these request forms).

A *value* is anything that may be a legitimate (actual) parameter in a request. A value may identify an object, for the purpose of performing the request. A value that identifies an object is called an *object name*. More particularly, a value is an instance of an OMG IDL data type.

An *object reference* is an object name that reliably denotes a particular object. Specifically, an object reference will identify the same object each time the reference is used in a request (subject to certain pragmatic limits of space and time). An object may be denoted by multiple, distinct object references.

A request may have parameters that are used to pass data to the target object; it may also have a request context which provides additional information about the request.

A request causes a service to be performed on behalf of the client. One outcome of performing a service is returning to the client the results, if any, defined for the request.

If an abnormal condition occurs during the performance of a request, an exception is returned. The exception may carry additional return parameters particular to that exception.

The request parameters are identified by position. A parameter may be an input parameter, an output parameter, or an input-output parameter. A request may also return a single *result value*, as well as any output parameters.

The following semantics hold for all requests:

- Any aliasing of parameter values is neither guaranteed removed nor guaranteed to be preserved
- The order in which aliased output parameters are written is not guaranteed
- Any output parameters are undefined if an exception is returned
- The values that can be returned in an input-output parameter may be constrained by the value that was input

Descriptions of the values and exceptions that are permitted, see Types on page 1-4 and Exceptions on page 1-6.

1.2.3 Object Creation and Destruction

Objects can be created and destroyed. From a client’s point of view, there is no special mechanism for creating or destroying an object. Objects are created and destroyed as an outcome of issuing requests. The outcome of object creation is revealed to the client in the form of an object reference that denotes the new object.

1.2.4 Types

A *type* is an identifiable entity with an associated predicate (a single-argument mathematical function with a boolean result) defined over values. A value *satisfies* a type if the predicate is true for that value. A value that satisfies a type is called a *member of the type*.

Types are used in signatures to restrict a possible parameter or to characterize a possible result.

The *extension of a type* is the set of values that satisfy the type at any particular time.

An *object type* is a type whose members are objects (literally, values that identify objects). In other words, an object type is satisfied only by (values that identify) objects.

Constraints on the data types in this model are shown in this section.

Basic types:

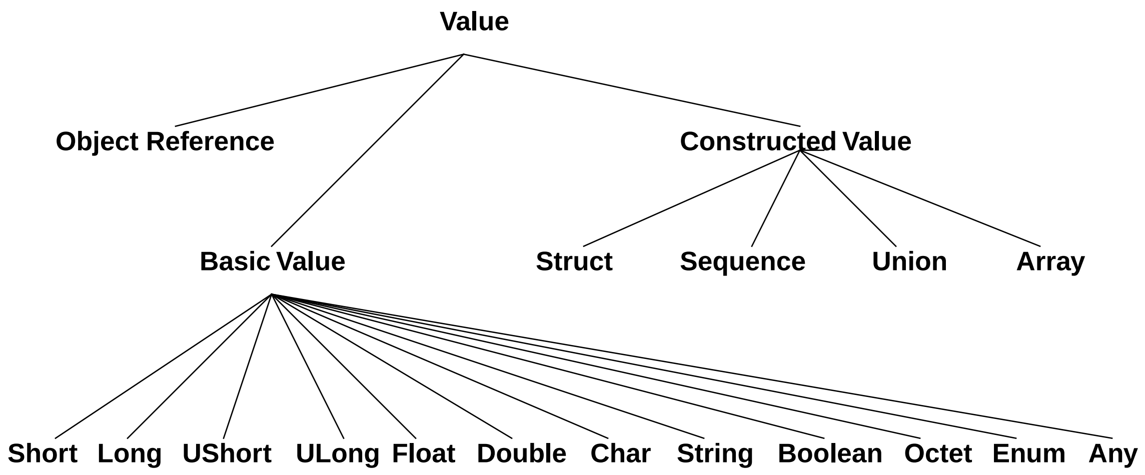
- 16-bit and 32-bit signed and unsigned 2's complement integers
- 32-bit and 64-bit IEEE floating point numbers
- Characters, as defined in ISO Latin-1 (8859.1)
- A boolean type taking the values TRUE and FALSE
- An 8-bit opaque detectable, guaranteed to *not* undergo any conversion during transfer between systems
- Enumerated types consisting of ordered sequences of identifiers
- A string type which consists of a variable-length array of characters; the length of the string is available at run-time
- A type "any" which can represent any possible basic or constructed type

Constructed types:

- A record type (called struct), consisting of an ordered set of (name,value) pairs
- A discriminated union type, consisting of a discriminator followed by an instance of a type appropriate to the discriminator value
- A sequence type which consists of a variable-length array of a single type; the length of the sequence is available at run-time
- An array type which consists of a fixed-length array of a single type
- An interface type, which specifies the set of operations which an instance of that type must support

Values in a request are restricted to values that satisfy these type constraints. The legal values are shown in FIG. 1 on page 1-5. No particular representation for values is defined.

FIG. 1 Legal Values



1.2.5 Interfaces

An *interface* is a description of a set of possible operations that a client may request of an object. An object *satisfies* an interface if it can be specified as the target object in each potential request described by the interface.

An *interface type* is a type that is satisfied by any object (literally, any value that identifies an object) that satisfies a particular interface.

Interfaces are specified in OMG IDL. Interface inheritance provides the composition mechanism for permitting an object to support multiple interfaces. The *principal interface* is simply the most-specific interface that the object supports, and consists of all operations in the transitive closure of the interface inheritance graph.

1.2.6 Operations

An *operation* is an identifiable entity that denotes a service that can be requested.

An operation is identified by an *operation identifier.* An operation is not a value.

An operation has a signature that describes the legitimate values of request parameters and returned results. In particular, a *signature* consists of:

- A specification of the parameters required in requests for that operation
- A specification of the result of the operation
- A specification of the exceptions that may be raised by a request for the operation and the types of the parameters accompanying them
- A specification of additional contextual information that may affect the request
- An indication of the execution semantics the client should expect from a request for the operation

Operations are (potentially) generic, meaning that a single operation can be uniformly requested on objects with different implementations, possibly resulting in observably different behavior. Genericity is achieved in this model via interface inheritance in IDL and the total decoupling of implementation from interface specification.

The general form for an operation signature is:

[oneway] <op_type_spec> <identifier> (param1, ..., paramL)
[raises(except1,...,exceptN)] [context(name1, ..., nameM)]

where:

- The optional **oneway** keyword indicates that best-effort semantics are expected of requests for this operation; the default semantics are exactly-once if the operation successfully returns results or at-most-once if an exception is returned
- The **<op_type_spec>** is the type of the return result
- The **<identifier>** provides a name for the operation in the interface
- The operation parameters needed for the operation; they are flagged with the modifiers **in**, **out**, or **inout** to indicate the direction in which the information flows (with respect to the object performing the request)
- The optional **raises** expression indicates which user-defined exceptions can be signaled to terminate a request for this operation; if such an expression is not provided, no user-defined exceptions will be signaled
- The optional **context** expression indicates which request context information will be available to the object implementation; no other contextual information is required to be transported with the request

Parameters

A parameter is characterized by its mode and its type. The *mode* indicates whether the value should be passed from client to server (**in**), from server to client (**out**), or both (**inout**). The parameter's type constrains the possible value which may be passed in the directions dictated by the mode.

Return Result

The return result is a distinguished **out** parameter.

Exceptions

An exception is an indication that an operation request was not performed successfully. An exception may be accompanied by additional, exception-specific information.

The additional, exception-specific information is a specialized form of record. As a record, it may consist of any of the types described in Section 1.2.4.

All signatures implicitly include the standard exceptions described in Section 3.15, "Standard Exceptions," on page 3-33.

Contexts

A request context provides additional, operation-specific information that may affect the performance of a request.

Execution Semantics

Two styles of execution semantics are defined by the object model:

- At-most-once: if an operation request returns successfully, it was performed exactly once; if it returns an exception indication, it was performed at-most-once.
- Best-effort: a best-effort operation is a request-only operation, i.e. it cannot return any results and the requester never synchronizes with the completion, if any, of the request.

The execution semantics to be expected is associated with an operation. This prevents a client and object implementation from assuming different execution semantics.

Note that a client is able to invoke an at-most-once operation in a synchronous or deferred-synchronous manner.

1.2.7 Attributes

An interface may have attributes. An attribute is logically equivalent to declaring a pair of accessor functions: one to retrieve the value of the attribute and one to set the value of the attribute.

An attribute may be read-only, in which case only the retrieval accessor function is defined.

1.3 Object Implementation

This section defines the concepts associated with object implementation, i.e. the concepts relevant to realizing the behavior of objects in a computational system.

The implementation of an object system carries out the computational activities needed to effect the behavior of requested services. These activities may include computing the result of the request and updating the system state. In the process, additional requests may be issued.

The implementation model consists of two parts: the execution model and the construction model. The execution model describes how services are performed. The construction model describes how services are defined.

1.3.1 The Execution Model: Performing Services

A requested service is performed in a computational system by executing code that operates upon some data. The data represents a component of the state of the computational system. The code performs the requested service, which may change the state of the system.

Code that is executed to perform a service is called a *method*. A method is an immutable description of a computation that can be interpreted by an execution engine. A method has an immutable attribute called a *method format* that defines the set of execution engines that can interpret the method. An *execution engine* is an abstract machine (not a program) that can interpret methods of certain formats, causing the described computations to be performed. An execution engine defines a dynamic context for the execution of a method. The execution of a method is called a *method activation*.

When a client issues a request, a method of the target object is called. The input parameters passed by the requestor are passed to the method and the output parameters and return value (or exception and its parameters) are passed back to the requestor.

Performing a requested service causes a method to execute that may operate upon an object's persistent state. If the persistent form of the method or state is not accessible to the execution engine, it may be necessary to first copy the method or state into an execution context. This process is called *activation*; the reverse process is called *deactivation*.

1.3.2 The Construction Model

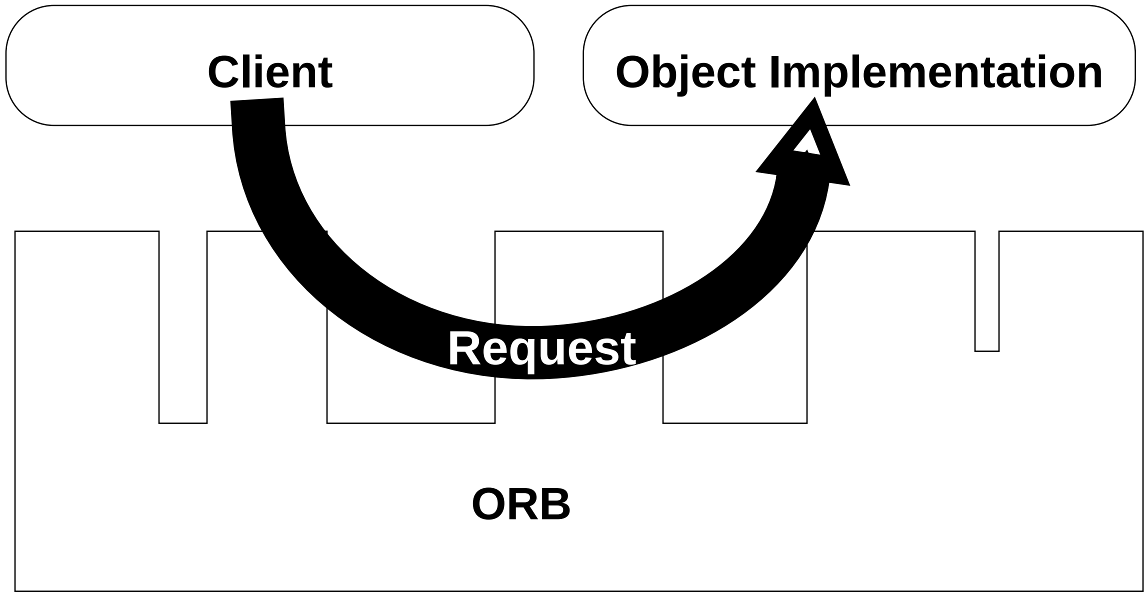
A computational object system must provide mechanisms for realizing behavior of requests. These mechanisms include definitions of object state, definitions of methods, and definitions of how the object infrastructure is to select the methods to execute and to select the relevant portions of object state to be made accessible to the methods. Mechanisms must also be provided to describe the concrete actions associated with object creation, such as association of the new object with appropriate methods.

An *object implementation*—or *implementation*, for short—is a definition that provides the information needed to create an object and to allow the object to participate in providing an appropriate set of services. An implementation typically includes, among other things, definitions of the methods that operate upon the state of an object. It also typically includes information about the intended type of the object.

CORBA Overview 2

The Common Object Request Broker Architecture (CORBA) is structured to allow integration of a wide variety of object systems. The motivation for some of the features may not be apparent at first, but as we discuss the range of implementations, policies, optimizations, and usages we expect to encompass, the value of the flexibility becomes more clear.

FIG. 2 A Request Being Sent Through the Object Request Broker



2.1 Structure of an Object Request Broker

FIG. 2 on page 2-1 shows a request being sent by a client to an object implementation.The Client is the entity that wishes to perform an operation on the object and the Object Implementation is the code and data that actually implements the object. The ORB is responsible for all of the mechanisms required to find the object implementation for the request, to prepare the object implementation to receive the request, and to communicate the data making up the ‘request. The interface the client sees is completely independent of where the object is located, what programming language it is implemented in, or any other aspect which is not reflected in the object’s interface.

FIG. 3 The Structure of Object Request Broker Interfaces

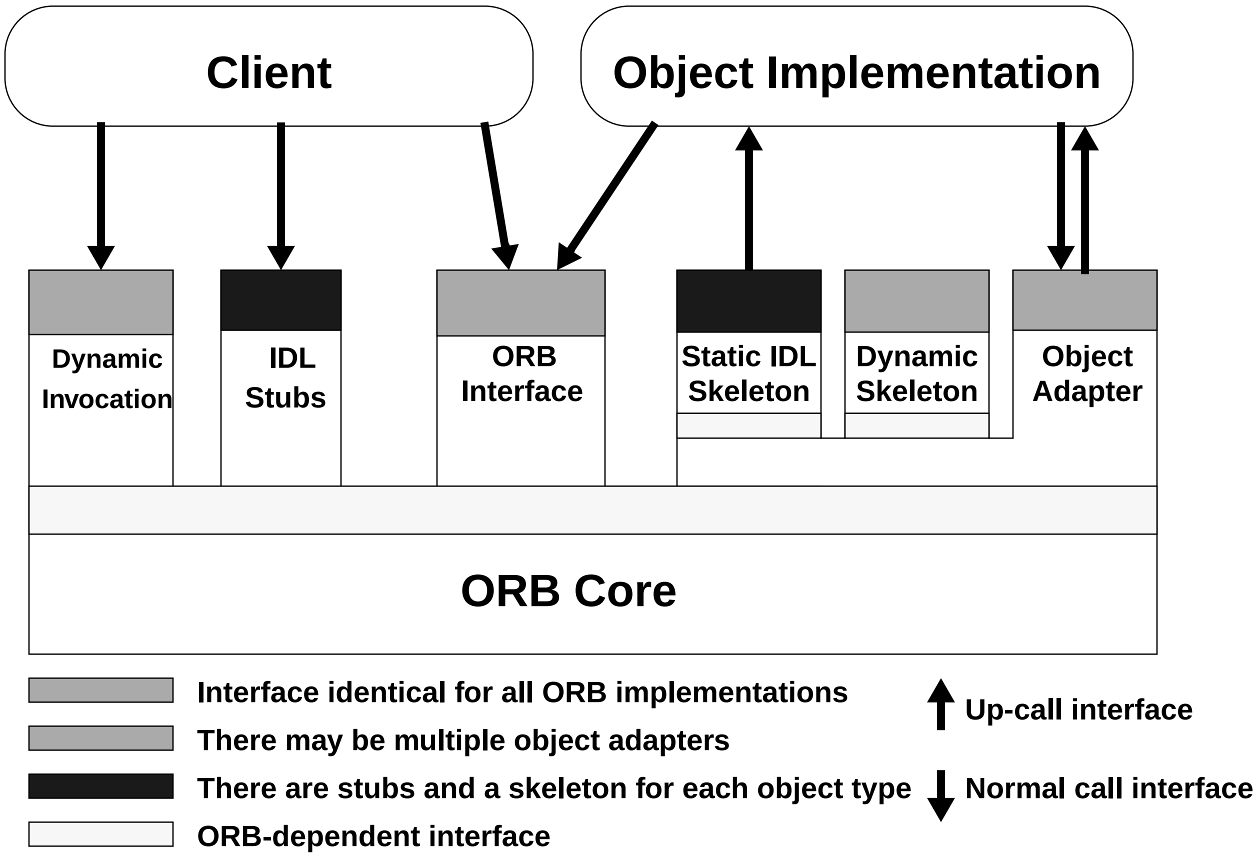


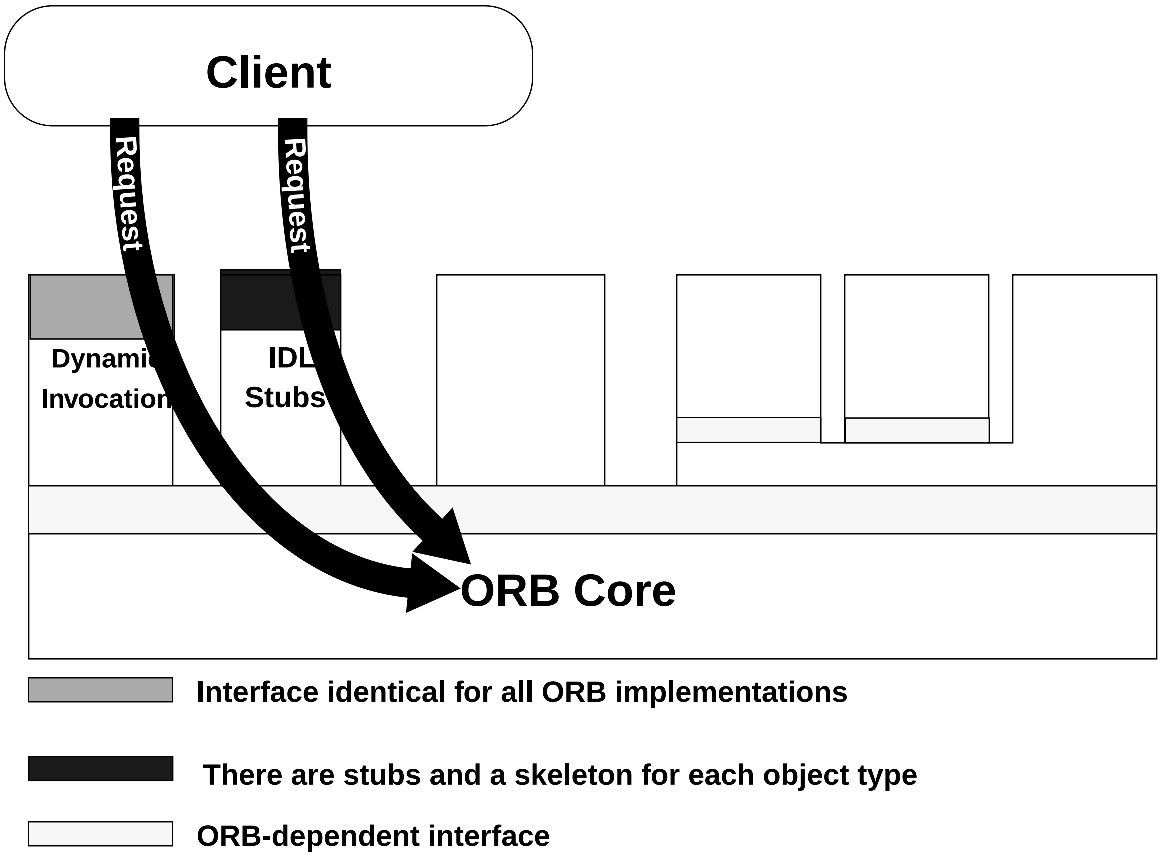
FIG. 3 on page 2-2 shows the structure of an individual Object Request Broker (ORB). The interfaces to the ORB are shown by striped boxes, and the arrows indicate whether the ORB is called or performs an up-call across the interface.

To make a request, the Client can use the Dynamic Invocation interface (the same interface independent of the target object’s interface) or an OMG IDL stub (the specific stub depending on the interface of the target object). The Client can also directly interact with the ORB for some functions.

The Object Implementation receives a request as an up-call either through the OMG IDL generated skeleton or through a dynamic skeleton. The Object Implementation may call the Object Adapter and the ORB while processing a request or at other times.

Definitions of the interfaces to objects can be defined in two ways. Interfaces can be defined statically in an interface definition language, called the OMG Interface Definition Language (OMG IDL). This language defines the types of objects according to the operations that may be performed on them and the parameters to those operations. Alternatively, or in addition, interfaces can be added to an Interface Repository service; this service represents the components of an interface as objects, permitting run-time access to these components. In any ORB implementation, the Interface Definition Language (which may be extended beyond its definition in this document) and the Interface Repository have equivalent expressive power.

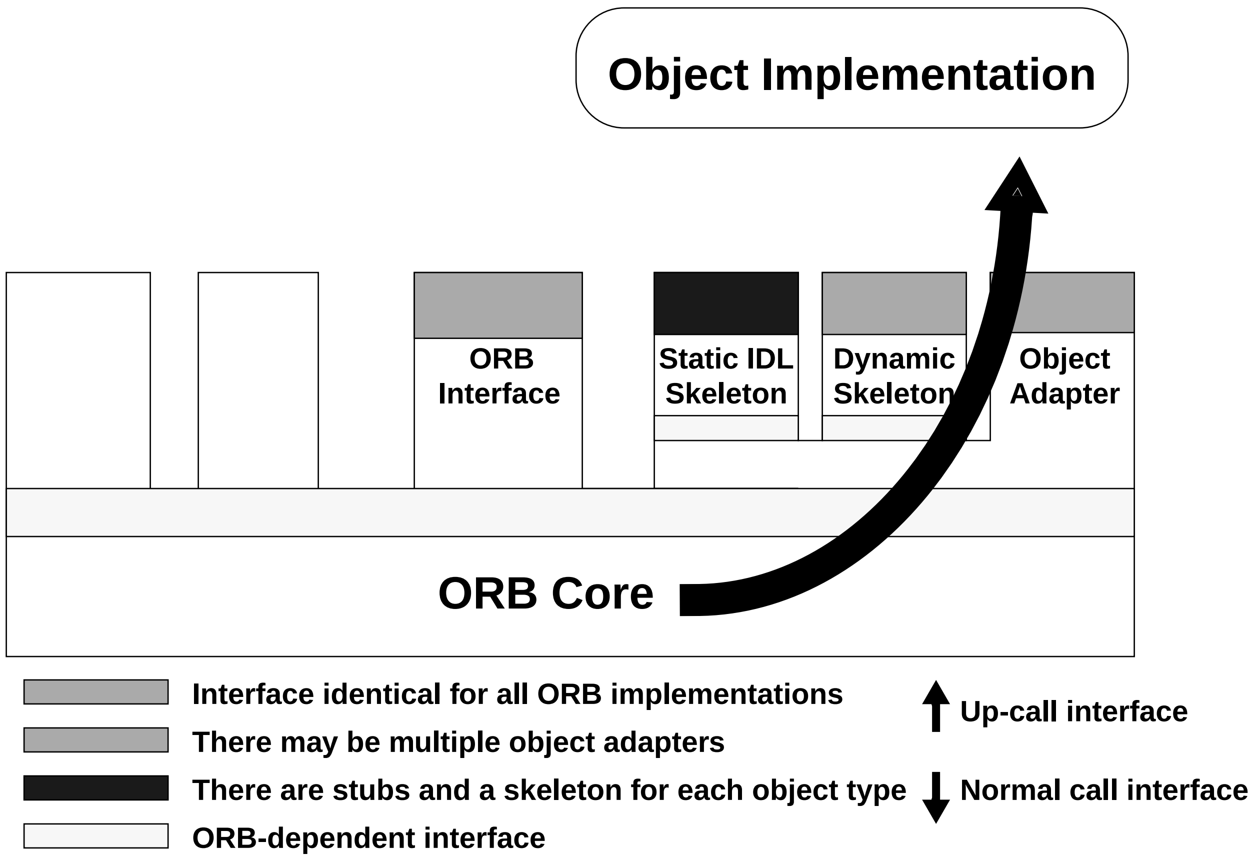
FIG. 4 A Client using the Stub or Dynamic Invocation Interface


Client
Request
Request
Dynamic Invocation
IDL Stubs
ORB Core
Interface identical for all ORB implementations
There are stubs and a skeleton for each object type
ORB-dependent interface

The client performs a request by having access to an Object Reference for an object and knowing the type of the object and the desired operation to be performed. The client initiates the request by calling stub routines that are specific to the object or by constructing the request dynamically (see FIG. 4 on page 2-3).

The dynamic and stub interface for invoking a request satisfy the same request semantics, and the receiver of the message cannot tell how the request was invoked.

FIG. 5 An Object Implementation Receiving a Request



The ORB locates the appropriate implementation code, transmits parameters and transfers control to the Object Implementation through an IDL skeleton or a dynamic skeleton (see FIG. 5 on page 2-4). Skeletons are specific to the interface and the object adapter. In performing the request, the object implementation may obtain some services from the ORB through the Object Adapter. When the request is complete, control and output values are returned to the client.

The Object Implementation may choose which Object Adapter to use. This decision is based on what kind of services the Object Implementation requires.

FIG. 6 Interface and Implementation Repositories

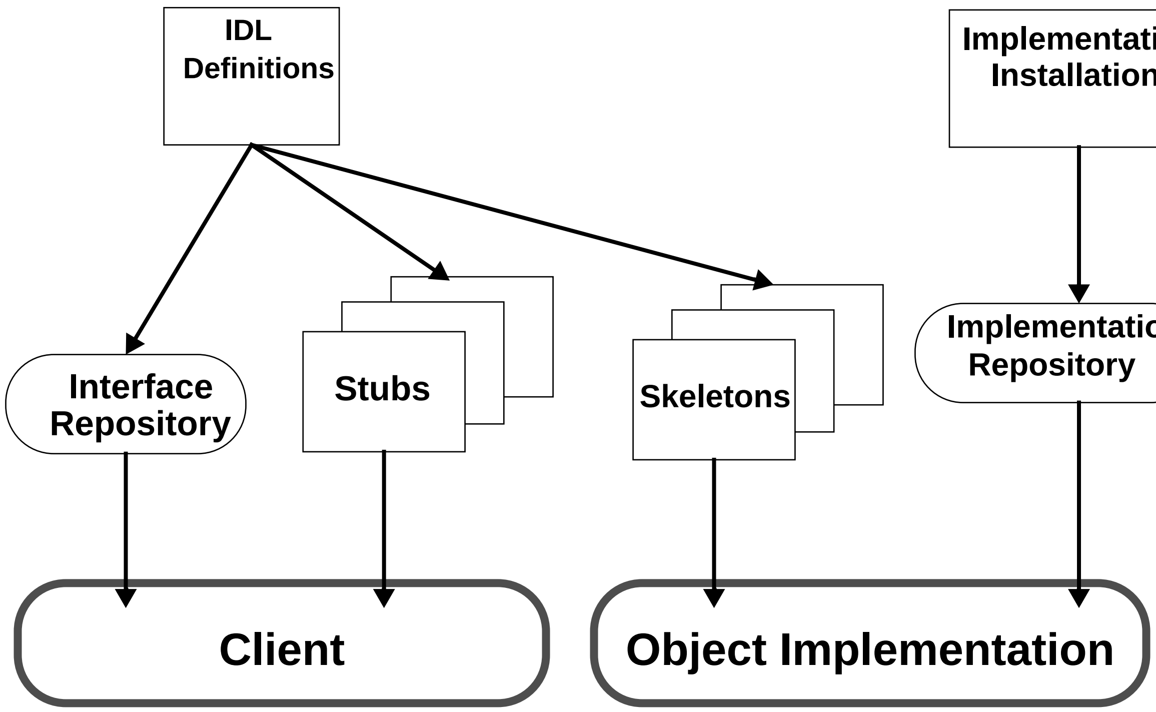


FIG. 6 on page 2-5 shows how interface and implementation information is made available to clients and object implementations. The interface is defined in OMG IDL and/or in the Interface Repository; the definition is used to generate the client Stubs and the object implementation Skeletons.

The object implementation information is provided at installation time and is stored in the Implementation Repository for use during request delivery.

2.1.1 Object Request Broker

In the architecture, the ORB is not required to be implemented as a single component, but rather it is defined by its interfaces. Any ORB implementation that provides the appropriate interface is acceptable. The interface is organized into three categories:

1. **Operations that are the same for all ORB implementations**
2. **Operations that are specific to particular types of objects**
3. **Operations that are specific to particular styles of object implementations**

Different ORBs may make quite different implementation choices, and, together with the IDL compilers, repositories, and various Object Adapters, provide a set of services to clients and implementations of objects that have different properties and qualities.

There may be multiple ORB implementations (also described as multiple ORBs) which have different representations for object references and different means of performing invocations. It may be possible for a client to simultaneously have access to two object references managed by different ORB implementations. When two ORBs are intended to work together, those ORBs must be able to distinguish their object references. It is not the responsibility of the client to do so.

The ORB Core is that part of the ORB that provides the basic representation of objects and communication of requests. CORBA is designed to support different object mechanisms, and it does so by structuring the ORB with components above the ORB Core, which provide interfaces that can mask the differences between ORB Cores.

2.1.2 Clients

A client of an object has access to an object reference for the object, and invokes operations on the object. A client knows only the logical structure of the object according to its interface and experiences the behavior of the object through invocations. Although we will generally consider a client to be a program or process initiating requests on an object, it is important to recognize that something is a client relative to a particular object. For example, the implementation of one object may be a client of other objects.

Clients generally see objects and ORB interfaces through the perspective of a language mapping, bringing the ORB right up to the programmer's level. Clients are maximally portable and should be able to work without source changes on any ORB that supports the desired language mapping with any object instance that implements the desired interface. Clients have no knowledge of the implementation of the object, which object adapter is used by the implementation, or which ORB is used to access it. Object Implementations

An object implementation provides the semantics of the object, usually by defining data for the object instance and code for the object's methods. Often the implementation will use other objects or additional software to implement the behavior of the object. In some cases, the primary function of the object is to have side-effects on other things that are not objects.

A variety of object implementations can be supported, including separate servers, libraries, a program per method, an encapsulated application, an object-oriented database, etc. Through the use of additional object adapters, it is possible to support virtually any style of object implementation.

Generally, object implementations do not depend on the ORB or how the client invokes the object. Object implementations may select interfaces to ORB-dependent services by the choice of Object Adapter.

2.1.3 Object References

An Object Reference is the information needed to specify an object within an ORB. Both clients and object implementations have an opaque notion of object references according to the language mapping, and thus are insulated from the actual representation of them. Two ORB implementations may differ in their choice of Object Reference representations.

The representation of an object reference handed to a client is only valid for the lifetime of that client.

All ORBs must provide the same language mapping to an object reference (usually referred to as an Object) for a particular programming language. This permits a program written in a particular language to access object references independent of the particular ORB. The language mapping may also provide additional ways to access object references in a typed way for the convenience of the programmer.

There is a distinguished object reference, guaranteed to be different from all object references, that denotes no object.

2.1.4 OMG Interface Definition Language

The OMG Interface Definition Language (OMG IDL) defines the types of objects by specifying their interfaces. An interface consists of a set of named operations and the parameters to those operations. Note that although IDL provides the conceptual framework for describing the objects manipulated by the ORB, it is not necessary for there to be IDL source code available for the ORB to work. As long as the equivalent information is available in the form of stub routines or a run-time interface repository, a particular ORB may be able to function correctly.

IDL is the means by which a particular object implementation tells its potential clients what operations are available and how they should be invoked. From the IDL definitions, it is possible to map CORBA objects into particular programming languages or object systems.

2.1.5 Mapping of OMG IDL to Programming Languages

Different object-oriented or non-object-oriented programming languages may prefer to access CORBA objects in different ways. For object-oriented languages, it may be desirable to see CORBA objects as programming language objects. Even for non-object-oriented languages, it is a good idea to hide the exact ORB representation of the object reference, method names, etc. A particular mapping of OMG IDL to a programming language should be the same for all ORB implementations. Language mapping includes definition of the language-specific data types and procedure interfaces to access objects through the ORB. It includes the structure of the client stub interface (not required for object-oriented languages), the dynamic invocation interface, the implementation skeleton, the object adapters, and the direct ORB interface.

A language mapping also defines the interaction between object invocations and the threads of control in the client or implementation. The most common mappings provide synchronous calls, in that the routine returns when the object operation completes. Additional mappings may be provided to allow a call to be initiated and control returned to the program. In such cases, additional language-specific routines must be provided to synchronize the program's threads of control with the object invocation.

2.1.6 Client Stubs

For the mapping of a non–object–oriented language, there will be a programming interface to the stubs for each interface type. Generally, the stubs will present access to the OMG IDL-defined operations on an object in a way that is easy for programmers to predict once they are familiar with OMG IDL and the language mapping for the particular programming language. The stubs make calls on the rest of the ORB using interfaces that are private to, and presumably optimized for, the particular ORB Core. If more than one ORB is available, there may be different stubs corresponding to the different ORBs. In this case, it is necessary for the ORB and language mapping to cooperate to associate the correct stubs with the particular object reference.

Object-oriented programming languages, such as C++ and Small-time, do not require stub interfaces.

2.1.7 Dynamic Invocation Interface

An interface is also available that allows the dynamic construction of object invocations, that is, rather than calling a stub routine that is specific to a particular operation on a particular object, a client may specify the object to be invoked, the operation to be performed, and the set of parameters for the operation through a call or sequence of calls. The client code must supply information about the operation to be performed and the types of the parameters being passed (perhaps obtaining it from an Interface Repository or other runtime source). The nature of the dynamic invocation interface may vary substantially from one programming language mapping to another.

2.1.8 Implementation Skeleton

For a particular language mapping, and possibly depending on the object adapter, there will be an interface to the methods that implement each type of object. The interface will generally be an up-call interface, in that the object implementation writes routines that conform to the interface and the ORB calls them through the skeleton.

The existence of a skeleton does not imply the existence of a corresponding client stub (clients can also make requests via the dynamic invocation interface).

It is possible to write an object adapter that does not use skeletons to invoke implementation methods. For example, it may be possible to create implementations dynamically for languages such as Smalltalk.

2.1.9 Dynamic Skeleton Interface

An interface is available which allows dynamic handling of object invocations. That is, rather than being accessed through a skeleton that is specific to a particular operation, an object's implementation is reached through an interface that provides access to the operation name and parameters in a manner analogous to the client side's Dynamic Invocation Interface. Purely static knowledge of those parameters may be used, or dynamic knowledge (perhaps determined through an Interface Repository) may be also used, to determine the parameters.

The implementation code must provide descriptions of all the operation parameters to the ORB, and the ORB provides the values of any input parameters for use in performing the operation. The implementation code provides the values of any output parameters, or an exception, to the ORB after performing the operation. The nature of the dynamic skeleton interface may vary substantially from one programming language mapping or object adapter to another, but will typically be an up-call interface.

Dynamic skeletons may be invoked both through client stubs and through the dynamic invocation interface; either style of client request construction interface provides identical results.

2.1.10 Object Adapters

An object adapter is the primary way that an object implementation accesses services provided by the ORB. There are expected to be a few object adapters that will be widely available, with interfaces that are appropriate for specific kinds of objects. Services provided by the ORB through an Object Adapter often include: generation and interpretation of object references, method invocation, security of interactions, object and implementation activation and deactivation, mapping object references to implementations, and registration of implementations.

The wide range of object granularities, lifetimes, policies, implementation styles, and other properties make it difficult for the ORB Core to provide a single interface that is convenient and efficient for all objects. Thus, through Object Adapters, it is possible for the ORB to target particular groups of object implementations that have similar requirements with interfaces tailored to them.

2.1.11 ORB Interface

The ORB Interface is the interface that goes directly to the ORB which is the same for all ORBs and does not depend on the object's interface or object adapter. Because most of the functionality of the ORB is provided through the object adapter, stubs, skeleton, or dynamic invocation, there are only a few operations that are common across all objects. These operations are useful to both clients and implementations of objects.

2.1.12 Interface Repository

The Interface Repository is a service that provides persistent objects that represent the IDL information in a form available at run-time. The Interface Repository information may be used by the ORB to perform requests. Moreover, using the information in the Interface Repository, it is possible for a program to encounter an object whose interface was not known when the program was compiled, yet, be able to determine what operations are valid on the object and make an invocation on it.

In addition to its role in the functioning of the ORB, the Interface Repository is a common place to store additional information associated with interfaces to ORB objects. For example, debugging information, libraries of stubs or skeletons, routines that can format or browse particular kinds of objects, etc., might be associated with the Interface Repository.

2.1.13 Implementation Repository

The Implementation Repository contains information that allows the ORB to locate and activate implementations of objects. Although most of the information in the Implementation Repository is specific to an ORB or operating environment, the Implementation Repository is the conventional place for recording such information. Ordinarily, installation of implementations and control of policies related to the activation and execution of object implementations is done through operations on the Implementation Repository.

In addition to its role in the functioning of the ORB, the Implementation Repository is a common place to store additional information associated with implementations of ORB objects. For example, debugging information, administrative control, resource allocation, security, etc., might be associated with the Implementation Repository.

2.2 Example ORBs

There are a wide variety of ORB implementations possible within the Common ORB Architecture. This section will illustrate some of the different options. Note that a particular ORB might support multiple options and protocols for communication.

2.2.1 Client- and Implementation-resident ORB

If there is a suitable communication mechanism present, an ORB can be implemented in routines resident in the clients and implementations. The stubs in the client either use a location-transparent IPC mechanism or directly access a location service to establish communication with the implementations. Code linked with the implementation is responsible for setting up appropriate databases for use by clients.

2.2.2 Server-based ORB

To centralize the management of the ORB, all clients and implementations can communicate with one or more servers whose job it is to route requests from clients to implementations. The ORB could be a normal program as far as the underlying operating system is concerned, and normal IPC could be used to communicate with the ORB.

2.2.3 System-based ORB

To enhance security, robustness, and performance, the ORB could be provided as a basic service of the underlying operating system. Object references could be made unforgeable, reducing the expense of authentication on each request. Because the operating system could know the location and structure of clients and implementations, it would be possible for a variety of optimizations to be implemented, for example, avoiding marshalling when both are on the same machine.

2.2.4 Library-based ORB

For objects that are light-weight and whose implementations can be shared, the implementation might actually be in a library. In this case, the stubs could be the actual methods.

This assumes that it is possible for a client program to get access to the data for the objects and that the implementation trusts the client not to damage the data.

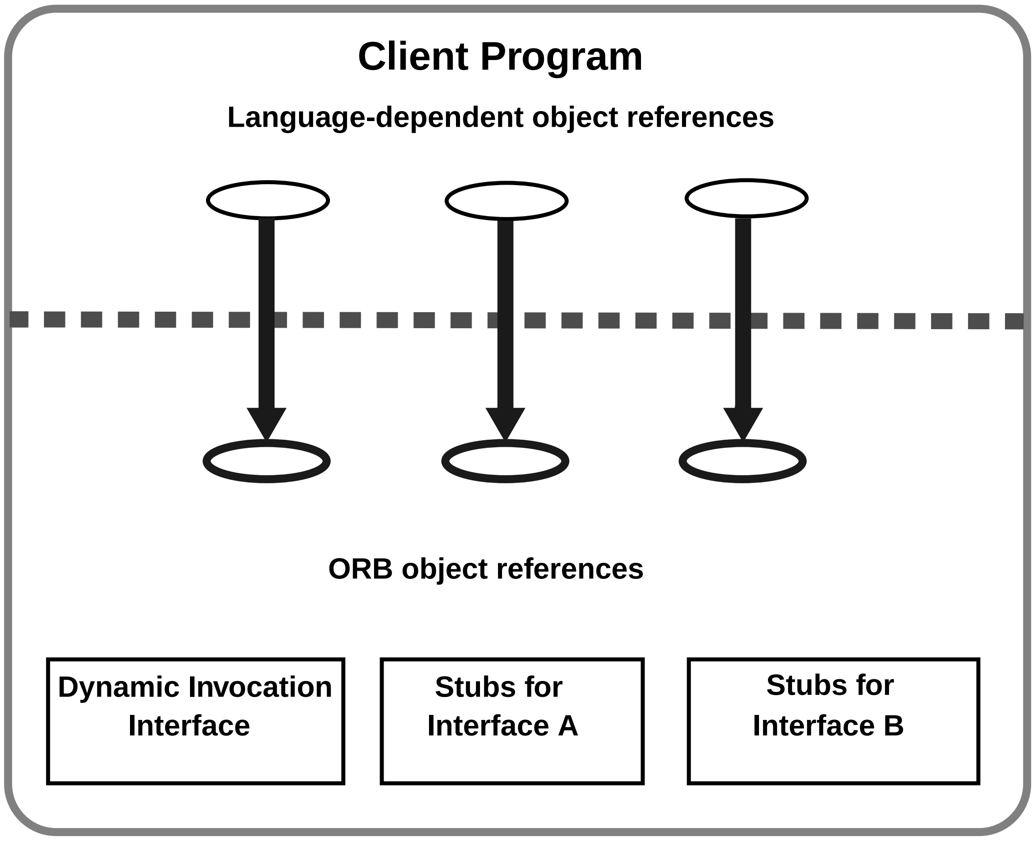
2.3 Structure of a Client

A client of an object has an object reference that refers to that object. An object reference is a token that may be invoked or passed as a parameter to an invocation on a different object. Invocation of an object involves specifying the object to be invoked, the operation to be performed, and parameters to be given to the operation or returned from it.

The ORB manages the control transfer and data transfer to the object implementation and back to the client. In the event that the ORB cannot complete the invocation, an exception response is provided. Ordinarily, a client calls a routine in its program that performs the invocation and returns when the operation is complete.

Clients access object-type-specific stubs as library routines in their program (see FIG. 7 on page 2-12). The client program thus sees routines callable in the normal way in its programming language. All implementations will provide a language-specific data type to use to refer to objects, often an opaque pointer. The client then passes that object reference to the stub routines to initiate an invocation. The stubs have access to the object reference representation and interact with the ORB to perform the invocation. (See Chapter 14 for additional, general information on language mapping of object references.)

FIG. 7 The Structure of a Typical Client



An alternative set of library code is available to perform invocations on objects, for example when the object was not defined at compile time. In that case, the client program provides additional information to name the type of the object and the method being invoked, and performs a sequence of calls to specify the parameters and initiate the invocation.

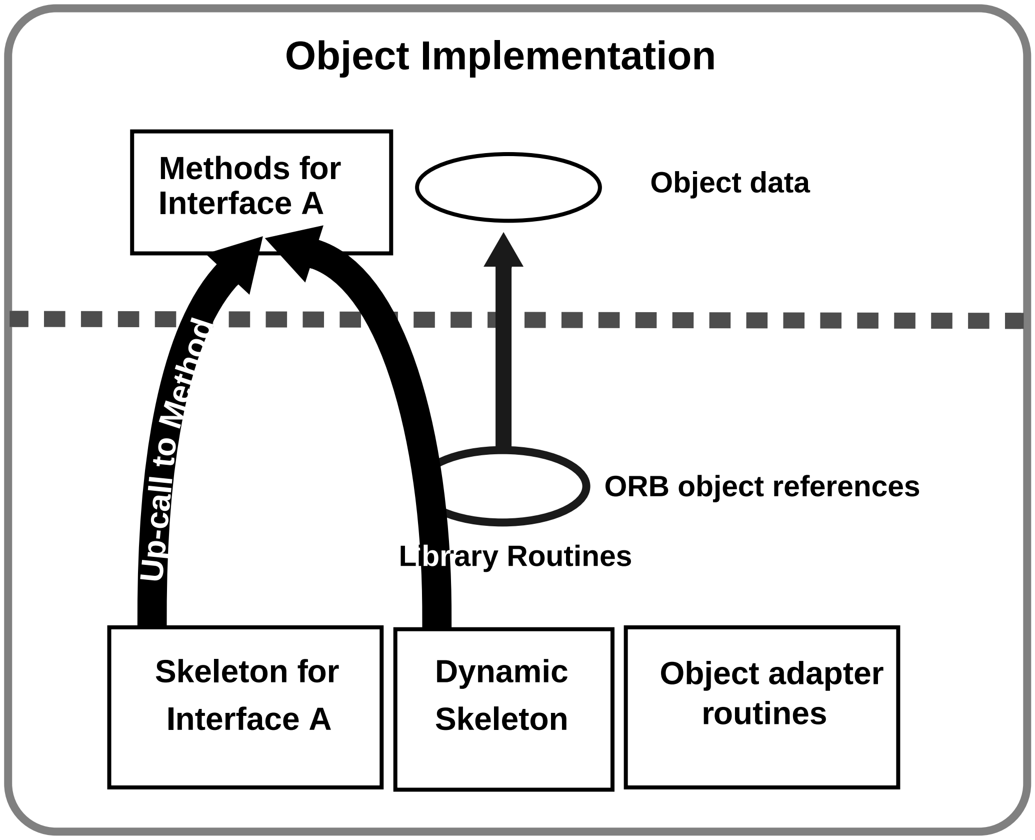
Clients most commonly obtain object references by receiving them as output parameters from invocations on other objects for which they have references. When a client is also an implementation, it receives object references as input parameters on invocations to objects it implements. An object reference can also be converted to a string that can be stored in files or preserved or communicated by different means and subsequently turned back into an object reference by the ORB that produced the string.

2.4 *Structure of an Object Implementation*

An object implementation provides the actual state and behavior of an object. The object implementation can be structured in a variety of ways. Besides defining the methods for the operations themselves, an implementation will usually define procedures for activating and deactivating objects and will use other objects or non-object facilities to make the object state persistent, to control access to the object, as well as to implement the methods.

The object implementation (see FIG. 8 on page 2-13) interacts with the ORB in a variety of ways to establish its identity, to create new objects, and to obtain ORB-dependent services. It primarily does this via access to an Object Adapter, which provides an interface to ORB services that is convenient for a particular style of object implementation.

FIG. 8 The Structure of a Typical Object Implementation



Because of the range of possible object implementations, it is difficult to be definitive about how in general an object implementation is structured. See the Basic Object Adapter chapter for the structure of object implementations that use the Basic Object Adapter.

When an invocation occurs, the ORB Core, object adapter, and skeleton arrange that a call is made to the appropriate method of the implementation. A parameter to that method specifies the object being invoked, which the method can use to locate the data for the object. Additional parameters are supplied according to the skeleton definition. When the method is complete, it returns, causing output parameters or exception results to be transmitted back to the client.

When a new object is created, the ORB may be notified so that the it knows where to find the implementation for that object. Usually, the implementation also registers itself as implementing objects of a particular interface, and specifies how to start up the implementation if it is not already running.

Most object implementations provide their behavior using facilities in addition to the ORB and object adapter. For example, although the Basic Object Adapter provides some persistent data associated with an object, that relatively small amount of data is typically used as an identifier for the actual object data stored in a storage service of the object implementation's choosing. With this structure, it is not only possible for different object implementations to use the same storage service, it is also possible for objects to choose the service that is most appropriate for them.

2.5 Structure of an Object Adapter

An object adapter (see FIG. 9 on page 2-15) is the primary means for an object implementation to access ORB services such as object reference generation. An object adapter exports a public interface to the object implementation, and a private interface to the skeleton. It is built on a private ORB-dependent interface.

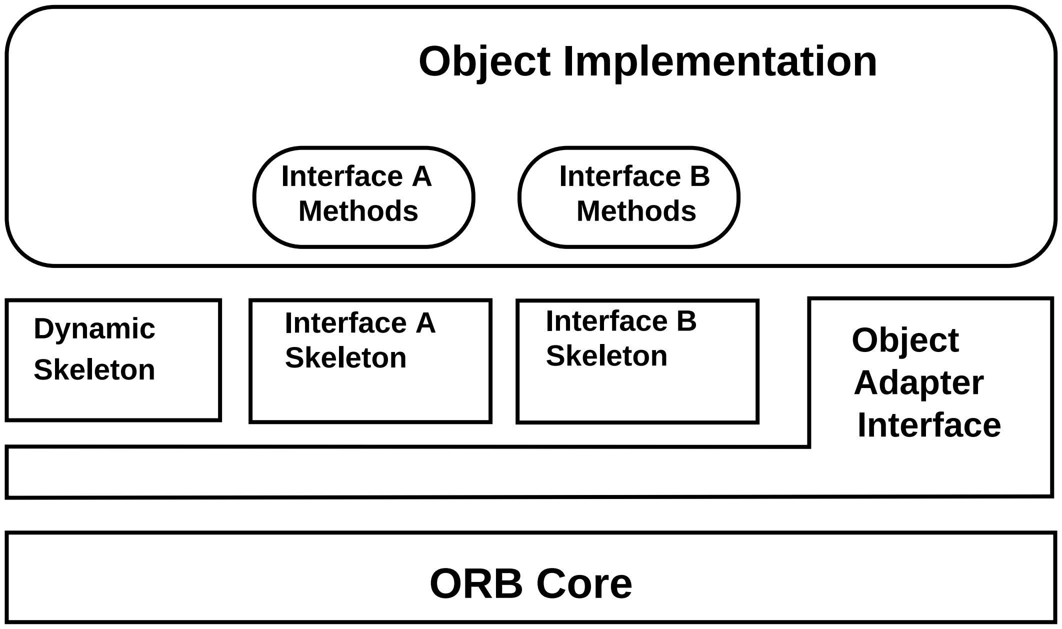
Object adapters are responsible for the following functions:

- Generation and interpretation of object references
- Method invocation
- Security of interactions
- Object and implementation activation and deactivation
- Mapping object references to the corresponding object implementations
- registration of implementations

These functions are performed using the ORB Core and any additional components necessary. Often, an object adapter will maintain its own state to accomplish its tasks. It may be possible for a particular object adapter to delegate one or more of its responsibilities to the Core upon which it is constructed.

As shown in FIG. 9 on page 2-15, the Object Adapter is implicitly involved in invocation of the methods, although the direct interface is through the skeletons. For example, the Object Adapter may be involved in activating the implementation or authenticating the request.

FIG. 9 The Structure of a Typical Object Adapter



The Object Adapter defines most of the services from the ORB that the Object Implementation can depend on. Different ORBs will provide different levels of service and different operating environments may provide some properties implicitly and require others to be added by the Object Adapter. For example, it is common for Object Implementations to want to store certain values in the object reference for easy identification of the object on an invocation. If the Object Adapter allows the implementation to specify such values when a new object is created, it may be able to store them in the object reference for those ORBs that permit it. If the ORB Core does not provide this feature, the Object Adapter would record the value in its own storage and provide it to the implementation on an invocation. With Object Adapters, it is possible for an Object Implementation to have access to a service whether or not it is implemented in the ORB Core—if the ORB Core provides it, the adapter simply provides an interface to it; if not, the adapter must implement it on top of the ORB Core. Every instance of a particular adapter must provide the same interface and service for all the ORBs it is implemented on.

It is also not necessary for all Object Adapters to provide the same interface or functionality. Some Object Implementations have special requirements, for example, an object-oriented database system may wish to implicitly register its many thousands of objects without doing individual calls to the Object Adapter. In such a case, it would be impractical and unnecessary for the object adapter to maintain any per-object state. By using an object adapter interface that is tuned towards such object implementations, it is possible to take advantage of particular ORB Core details to provide the most effective access to the ORB.

2.6 Example Object Adapters

There are a variety of possible object adapters. However, since the object adapter interface is something that object implementations depend on, it is desirable that there be as few as practical. Most object adapters are designed to cover a range of object implementations, so only when an implementation requires radically different services or interfaces should a new object adapter be considered. In this section, we describe three object adapters that might be useful.

2.6.1 Basic Object Adapter

This specification defines an object adapter that can be used for most ORB objects with conventional implementations. (See the Basic Object Adapter chapter for more information.) For this object adapter, implementations are generally separate programs. It allows there to be a program started per method, a separate program for each object, or a shared program for all instances of the object type. It provides a small amount of persistent storage for each object, which can be used as a name or identifier for other storage, for access control lists, or other object properties. If the implementation is not active when an invocation is performed, the BOA will start one.

2.6.2 Library Object Adapter

This object adapter is primarily used for objects that have library implementations. It accesses persistent storage in files, and does not support activation or authentication, since the objects are assumed to be in the clients program.

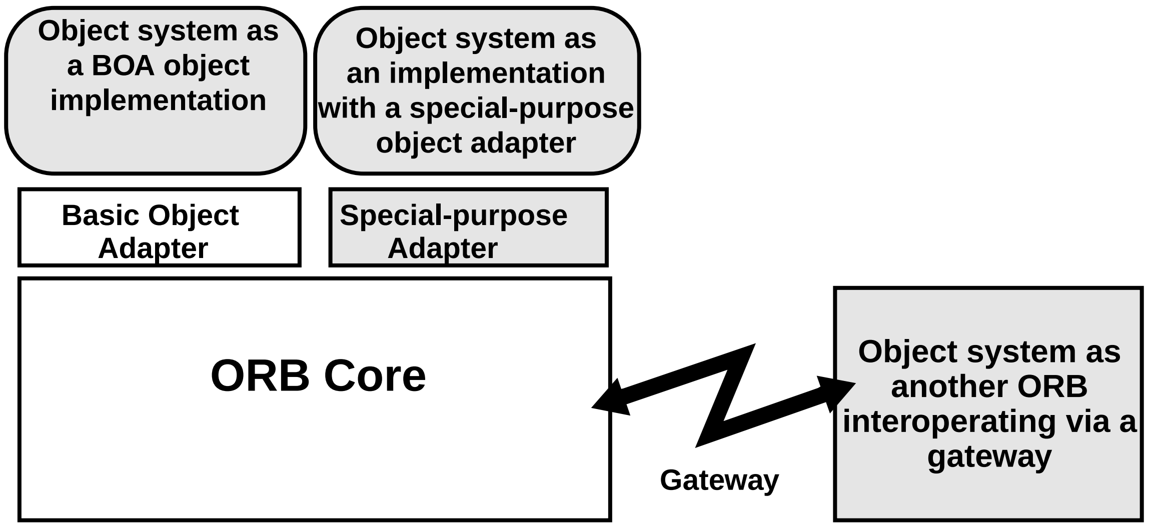
2.6.3 Object-Oriented Database Adapter

This adapter uses a connection to an object-oriented database to provide access to the objects stored in it. Since the OODB provides the methods and persistent storage, objects may be registered implicitly and no state is required in the object adapter.

2.7 The Integration of Foreign Object Systems

The Common ORB Architecture is designed to allow interoperation with a wide range of object systems (see FIG. 10 on page 2-17). Because there are many existing object systems, a common desire will be to allow the objects in those systems to be accessible via the ORB. For those object systems that are ORBs themselves, they may be connected to other ORBs through the mechanisms described in chapters 9, 10, 11, 12, and 13 in this manual.

FIG. 10 Different Ways to Integrate Foreign Object Systems


Object system as a BOA object implementation
Object system as an implementation with a special-purpose object adapter
Basic Object Adapter
Special-purpose Adapter
ORB Core
Gateway
Object system as another ORB interoperating via a gateway

For object systems that simply want to map their objects into ORB objects and receive invocations through the ORB, one approach is to have those object systems appear to be implementations of the corresponding ORB objects. The object system would register its objects with the ORB and handle incoming requests, and could act like a client and perform outgoing requests.

In some cases, it will be impractical for another object system to act like a BOA object implementation. An object adapter could be designed for objects that are created in conjunction with the ORB and that are primarily invoked through the ORB. Another object system may wish to create objects without consulting the ORB, and might expect most invocations to occur within itself rather than through the ORB. In such a case, a more appropriate object adapter might allow objects to be implicitly registered when they are passed through the ORB.

OMG IDL Syntax and Semantics 3

The OMG Interface Definition Language is the language used to describe the interfaces that client objects call and object implementations provide. An interface definition written in OMG IDL completely defines the interface and fully specifies each operation's parameters. An OMG IDL interface provides the information needed to develop clients that use the interface's operations. Clients are not written in OMG IDL, which is purely a descriptive language, but in languages for which mappings from OMG IDL concepts have been defined. The mapping of an OMG IDL concept to a client language construct will depend on the facilities available in the client language. For example, an OMG IDL exception might be mapped to a structure in a language that has no notion of exception, or to an exception in a language that does. The binding of OMG IDL concepts to the C, C++, and Smalltalk languages are described in this manual. Bindings from OMG IDL to additional programming languages will be added to future versions of *COBRA*.

OMG IDL obeys the same lexical rules as C++[1], although new keywords are introduced to support distribution concepts. It also provides full support for standard C++ preprocessing features. The OMG IDL specification is expected to track relevant changes to C++ introduced by the ANSI standardization effort.

3.1 About This Chapter

The description of OMG IDL's lexical conventions is presented in "Lexical Conventions" on page 3-2. A description of OMG IDL preprocessing is presented in "Preprocessing" on page 3-8. The scope rules for identifiers in an OMG IDL specification are described in "CORBA Module" on page 3-31.

1.Ellis, Margaret A. and Bjarne Stroustrup, *The Annotated C++ Reference Manual*, Addison-Wesley Publishing Company, Reading, Massachusetts, 1990, ISBN 0-201-51459-1

The OMG IDL grammar is a subset of the proposed ANSI C++ standard, with additional constructs to support the operation invocation mechanism. OMG IDL is a declarative language. It supports C++ syntax for constant, type, and operation declarations; it does not include any algorithmic structures or variables. The grammar is presented in "OMG IDL Grammar" on page 3-9.

OMG IDL-specific pragmas (those not defined for C++) may appear anywhere in a specification; the textual location of these pragmas may be semantically constrained by a particular implementation.

A source file containing interface specifications written in OMG IDL must have an ".idl" extension. The file orb.idl, which contains OMG IDL type definitions and is available on every ORB implementation, is described in Appendix A.

This chapter describes OMG IDL semantics and gives the syntax for OMG IDL grammatical constructs. The description of OMG IDL grammar uses a syntax notation that is similar to Extended Backus-Naur format (EBNF); Figure 1 on page 3-2 lists the symbols used in this format and their meaning.

TABLE 1. IDL EBNF Format

Symbol	Meaning
::=	Is defined to be
|	Alternatively
<text>	Nonterminal
"text"	Literal
*	The preceding syntactic unit can be repeated zero or more times
+	The preceding syntactic unit can be repeated one or more times
{}	The enclosed syntactic units are grouped as a single syntactic unit
[]	The enclosed syntactic unit is optional—may occur zero or one time

3.2 *Lexical Conventions*

This section[2] presents the lexical conventions of OMG IDL. It defines tokens in an OMG IDL specification and describes comments, identifiers, keywords, and literals—integer, character, and floating point constants and string literals.

An OMG IDL specification logically consists of one or more files. A file is conceptually translated in several phases.

The first phase is preprocessing, which performs file inclusion and macro substitution. Preprocessing is controlled by directives introduced by lines having # as the first character other than white space. The result of preprocessing is a sequence of tokens. Such a sequence of tokens, that is, a file after preprocessing, is called a translation unit.

2.This section is an adaptation of *The Annotated C++ Reference Manual*, Chapter 2; it differs in the list of legal keywords and punctuation.

OMG IDL uses the ISO Latin-1 (8859.1) character set. This character set is divided into alphabetic characters (letters), digits, graphic characters, the space (blank) character and formatting characters. Figure 2 on page 3-3 shows the OMG IDL alphabetic characters; upper- and lower-case equivalencies are paired.

TABLE 2. The 114 Alphabetic Characters (Letters)

Char.	Description	Char.	Description
Aa	Upper/Lower-case A	Àà	Upper/Lower-case A with grave accent
Bb	Upper/Lower-case B	Áá	Upper/Lower-case A with acute accent
Cc	Upper/Lower-case C	Ââ	Upper/Lower-case A with circumflex accent
Dd	Upper/Lower-case D	Ãã	Upper/Lower-case A with tilde
Ee	Upper/Lower-case E	Ää	Upper/Lower-case A with diaeresis
Ff	Upper/Lower-case F	Åå	Upper/Lower-case A with ring above
Gg	Upper/Lower-case G	Ææ	Upper/Lower-case dipthong A with E
Hh	Upper/Lower-case H	Çç	Upper/Lower-case C with cedilla
Ii	Upper/Lower-case I	Èè	Upper/Lower-case E with grave accent
Jj	Upper/Lower-case J	Éé	Upper/Lower-case E with acute accent
Kk	Upper/Lower-case K	Êê	Upper/Lower-case E with circumflex accent
Ll	Upper/Lower-case L	Ëë	Upper/Lower-case E with diaeresis
Mm	Upper/Lower-case M	Ìì	Upper/Lower-case I with grave accent
Nn	Upper/Lower-case N	Íí	Upper/Lower-case I with acute accent
Oo	Upper/Lower-case O	Îî	Upper/Lower-case I with circumflex accent
Pp	Upper/Lower-case P	Ïï	Upper/Lower-case I with diaeresis
Qq	Upper/Lower-case Q		Upper/Lower-case Icelandic eth
Rr	Upper/Lower-case R	Ññ	Upper/Lower-case N with tilde
Ss	Upper/Lower-case S	Òò	Upper/Lower-case O with grave accent
Tt	Upper/Lower-case T	Óó	Upper/Lower-case O with acute accent
Uu	Upper/Lower-case U	Ôô	Upper/Lower-case O with circumflex accent
Vv	Upper/Lower-case V	Õõ	Upper/Lower-case O with tilde
Ww	Upper/Lower-case W	Öö	Upper/Lower-case O with diaeresis
Xx	Upper/Lower-case X	Øø	Upper/Lower-case O with oblique stroke
Yy	Upper/Lower-case Y	Ùù	Upper/Lower-case U with grave accent
Zz	Upper/Lower-case Z	Úú	Upper/Lower-case U with acute accent
		Ûû	Upper/Lower-case U with circumflex accent
		Üü	Upper/Lower-case U with diaeresis
			Upper/Lower-case Y with acute accent
			Upper/Lower-case Icelandic thorn
		ß	Lower-case German sharp S
		ÿ	Lower-case Y with diaeresis

Figure 3 on page 3-4 lists the decimal digit characters.

TABLE 3. Decimal Digits

0 1 2 3 4 5 6 7 8 9

Figure 4 on page 3-4 shows the graphic characters.

TABLE 4. The 65 Graphic Characters

Char.	Description	Char.	Description
!	exclamation point	¡	inverted exclamation mark
"	double quote	¢	cent sign
#	number sign	£	pound sign
$	dollar sign	¤	currency sign
%	percent sign	¥	yen sign
&	ampersand		broken bar
’	apostrophe	§	section/paragraph sign
(	left parenthesis	¨	diaeresis
)	right parenthesis	©	copyright sign
*	asterisk	ª	feminine ordinal indicator
+	plus sign	«	left angle quotation mark
,	comma	¬	not sign
-	hyphen, minus sign		soft hyphen
.	period, full stop	®	registered trade mark sign
/	solidus	¯	macron
:	colon	°	ring above, degree sign
;	semicolon	±	plus-minus sign
<	less-than sign	^{2}	superscript two
=	equals sign	^{3}	superscript three
>	greater-than sign	´	acute
?	question mark	µ	micro
@	commercial at	¶	pilcrow
[	left square bracket	•	middle dot
\	reverse solidus	¸	cedilla
]	right square bracket	^{1}	superscript one
^	circumflex	º	masculine ordinal indicator
_	low line, underscore	»	right angle quotation mark
‘	grave		vulgar fraction 1/4
{	left curly bracket		vulgar fraction 1/2
|	vertical line		vulgar fraction 3/4
}	right curly bracket	¿	inverted question mark
~	tilde	×	multiplication sign
		÷	division sign

The formatting characters are shown in Figure 5 on page 3-5.

TABLE 5. The Formatting Characters

Description	Abbreviation	ISO 646 Octal Value
alert	BEL	007
backspace	BS	010
horizontal tab	HT	011
newline	NL, LF	012
vertical tab	VT	013
form feed	FF	014
carriage return	CR	015

3.2.1 Tokens

There are five kinds of tokens: identifiers, keywords, literals, operators, and other separators. Blanks, horizontal and vertical tabs, newlines, formfeeds, and comments (collective, “white space”), as described below, are ignored except as they serve to separate tokens. Some white space is required to separate otherwise adjacent identifiers, keywords, and constants.

If the input stream has been parsed into tokens up to a given character, the next token is taken to be the longest string of characters that could possibly constitute a token.

3.2.2 Comments

The characters /* start a comment, which terminates with the characters */. These comments do not nest. The characters // start a comment, which terminates at the end of the line on which they occur. The comment characters //, /*, and */ have no special meaning within a // comment and are treated just like other characters. Similarly, the comment characters // and /* have no special meaning within a /* comment. Comments may contain alphabetic, digit, graphic, space, horizontal tab, vertical tab, form feed and newline characters.

3.2.3 Identifiers

An identifier is an arbitrarily long sequence of alphabetic, digit, and underscore (“_”) characters. The first character must be an alphabetic character. All characters are significant.

Identifiers that differ only in case collide and yield a compilation error. An identifier for a definition must be spelled consistently (with respect to case) throughout a specification.

When comparing two identifiers to see if they collide:

- Upper- and lower-case letters are treated as the same letter. Figure 2 on page 3-3 defines the equivalence mapping of upper- and lower-case letters.

- The comparison does *not* take into account equivalences between digraphs and pairs of letters (e.g., “æ” and “ae” are not considered equivalent) or equivalences between accented and non-accented letters (e.g., “Á” and “A” are not considered equivalent).
- All characters are significant.

There is only one namespace for OMG IDL identifiers. Using the same identifier for a constant and an interface, for example, produces a compilation error.

3.2.4 Keywords

The identifiers listed in Figure 6 on page 3-6 are reserved for use as keywords, and may not be used otherwise.

TABLE 6. **Keywords**

any	default	inout	out	switch
attribute	double	interface	raises	TRUE
boolean	enum	long	readonly	typedef
case	exception	module	sequence	unsigned
char	FALSE	Object	short	union
const	float	octet	string	void
context	in	oneway	struct	

Keywords obey the rules for identifiers (see Section 3.2.3) and must be written exactly as shown in the above list. For example, “**boolean**” is correct; “**Boolean**” produces a compilation error.

OMG IDL specifications use the characters shown in Figure 7 on page 3-6 as punctuation.

TABLE 7. **Punctuation Characters**

;	{	}	:	,	=	+	-	(	)	<	>	[	]
'	"	\	|	^	&	*	/	%	~				

In addition, the tokens listed in Figure 8 on page 3-6 are used by the preprocessor.

TABLE 8. **Preprocessor Tokens**

#	##	!	||	&&

3.2.5 Literals

This section describes the following literals:

- Integer
- Character
- Floating-point
- String

Integer Literals

An integer literal consisting of a sequence of digits is taken to be decimal (base ten) unless it begins with 0 (digit zero). A sequence of digits starting with 0 is taken to be an octal integer (base eight). The digits 8 and 9 are not octal digits. A sequence of digits preceded by 0x or 0X is taken to be a hexadecimal integer (base sixteen). The hexadecimal digits include a or A through f or F with decimal values ten through fifteen, respectively. For example, the number twelve can be written 12, 014, or 0XC.

Character Literals

A character literal is one or more characters enclosed in single quotes, as in'x'. Character literals have type **char**.

A character is an 8-bit quantity with a numerical value between 0 and 255 (decimal). The value of a space, alphabetic, digit or graphic character literal is the numerical value of the character as defined in the ISO Latin-1 (8859.1) character set standard (See Figure 2 on page 3-3, Figure 3 on page 3-4, and Figure 4 on page 3-4). The value of a null is 0. The value of a formatting character literal is the numerical value of the character as defined in the ISO 646 standard (See Figure 5 on page 3-5). The meaning of all other characters is implementation-dependent.

Nongraphic characters must be represented using escape sequences as defined below in Figure 9 on page 3-7. Note that escape sequences must be used to represent single quote and backslash characters in character literals.

TABLE 9. Escape Sequences

Description	Escape Sequence
newline	\n
horizontal tab	\t
vertical tab	\v
backspace	\b
carriage return	\r
form feed	\f
alert	\a
backslash	\\
question mark	\?
single quote	\'
double quote	\"
octal number	\ooo
hexadecimal number	\xhh

If the character following a backslash is not one of those specified, the behavior is undefined. An escape sequence specifies a single character.

The escape \ooo consists of the backslash followed by one, two, or three octal digits that are taken to specify the value of the desired character. The escape \xhh consists of the backslash followed by x followed by one or two hexadecimal digits that are taken to specify the value of the desired character. A sequence of octal or hexadecimal digits is terminated by the first character that is not an octal digit or a hexadecimal digit, respectively. The value of a character constant is implementation dependent if it exceeds that of the largest char.

Floating-point Literals

A floating-point literal consists of an integer part, a decimal point, a fraction part, an e or E, and an optionally signed integer exponent. The integer and fraction parts both consist of a sequence of decimal (base ten) digits. Either the integer part or the fraction part (but not both) may be missing; either the decimal point or the letter e (or E) and the exponent (but not both) may be missing.

String Literals

A string literal is a sequence of characters (as defined in “Character Literals” on page 3-7) surrounded by double quotes, as in "...".

Adjacent string literals are concatenated. Characters in concatenated strings are kept distinct. For example,

```
"\xA" "B"
```

contains the two characters '\xA' and 'B' after concatenation (and not the single hexadecimal character '\xAB').

The size of a string literal is the number of character literals enclosed by the quotes, after concatenation. The size of the literal is associated with the literal. Within a string, the double quote character **"** must be preceded by a \.

A string literal may not contain the character '\0'.

3.3 Preprocessing

OMG IDL preprocessing, which is based on ANSI C++ preprocessing, provides macro substitution, conditional compilation, and source file inclusion. In addition, directives are provided to control line numbering in diagnostics and for symbolic debugging, to generate a diagnostic message with a given token sequence, and to perform implementation-dependent actions (the **#pragma** directive). Certain predefined names are available. These facilities are conceptually handled by a preprocessor, which may or may not actually be implemented as a separate process.

Lines beginning with # (also called “directives”) communicate with this preprocessor. White space may appear before the #. These lines have syntax independent of the rest of OMG IDL; they may appear anywhere and have effects that last (independent of the OMG IDL scoping rules) until the end of the translation unit. The textual location of OMG IDL-specific pragmas may be semantically constrained.

A preprocessing directive (or any line) may be continued on the next line in a source file by placing a backslash character (“\”), immediately before the newline at the end of the line to be continued. The preprocessor effects the continuation by deleting the backslash and the newline before the input sequence is divided into tokens. A backslash character may not be the last character in a source file.

A preprocessing token is an OMG IDL token (Section 3.2.1), a file name as in a **#include** directive, or any single character, other than white space, that does not match another preprocessing token.

The primary use of the preprocessing facilities is to include definitions from other OMG IDL specifications. Text in files included with a **#include** directive is treated as if it appeared in the including file. A complete description of the preprocessing facilities may be found in *The Annotated C++ Reference Manual*, Chapter 16. The #pragma directive that is used to include RepositoryIds is described in Section 6.6, “RepositoryIds,” on page 6-30.

3.4 OMG IDL Grammar

```
(1)   <specification>     ::=  <definition>+
(2)   <definition>        ::=  <type_dcl> “;”
                          |    <const_dcl> “;”
                          |    <except_dcl> “;”
                          |    <interface> “;”
                          |    <module> “;”
(3)   <module>            ::=  “module” <identifier> “{“ <definition>+ “}”
(4)   <interface>         ::=  <interface_dcl>
                          |    <forward_dcl>
(5)   <interface_dcl>     ::=  <interface_header> “{” <interface_body> “}”
(6)   <forward_dcl>       ::=  “interface” <identifier>
(7)   <interface_header>::= “interface” <identifier> [ <inheritance_spec> ]
(8)   <interface_body>    ::=  <export>*
(9)   <export>            ::=  <type_dcl> “;”
                          |    <const_dcl> “;”
                          |    <except_dcl> “;”
                          |    <attr_dcl> “;”
                          |    <op_dcl> “;”
(10)  <inheritance_spec>::= “:” <scoped_name> { “,” <scoped_name> }*
```

```
(11)   <scoped_name>      ::= <identifier>
                           |  "::" <identifier>
                           |  <scoped_name> "::" <identifier>

(12)   <const_dcl>        ::= "const" <const_type> <identifier> "=" <const_exp>

(13)   <const_type>       ::= <integer_type>
                           |  <char_type>
                           |  <boolean_type>
                           |  <floating_pt_type>
                           |  <string_type>
                           |  <scoped_name>

(14)   <const_exp>        ::= <or_expr>

(15)   <or_expr>          ::= <xor_expr>
                           |  <or_expr> "|" <xor_expr>

(16)   <xor_expr>         ::= <and_expr>
                           |  <xor_expr> "^" <and_expr>

(17)   <and_expr>         ::= <shift_expr>
                           |  <and_expr> "&" <shift_expr>

(18)   <shift_expr>       ::= <add_expr>
                           |  <shift_expr> ">>" <add_expr>
                           |  <shift_expr> "<<" <add_expr>

(19)   <add_expr>         ::= <mult_expr>
                           |  <add_expr> "+" <mult_expr>
                           |  <add_expr> "-" <mult_expr>

(20)   <mult_expr>        ::= <unary_expr>
                           |  <mult_expr> "*" <unary_expr>
                           |  <mult_expr> "/" <unary_expr>
                           |  <mult_expr> "%" <unary_expr>

(21)   <unary_expr>       ::= <unary_operator> <primary_expr>
                           |  <primary_expr>

(22)   <unary_operator> ::= "-"
                           |  "+"
                           |  "~"

(23)   <primary_expr>     ::= <scoped_name>
                           |  <literal>
                           |  "(" <const_exp> ")"

(24)   <literal>          ::= <integer_literal>
                           |  <string_literal>
                           |  <character_literal>
                           |  <floating_pt_literal>
                           |  <boolean_literal>

(25)   <boolean_literal>  ::= "TRUE"
                           |  "FALSE"

(26)   <positive_int_const>::=<const_exp>
```

```
(27)   <type_dcl>          ::=  "typedef" <type_declarator>
                            |   <struct_type>
                            |   <union_type>
                            |   <enum_type>

(28)   <type_declarator>  ::=  <type_spec> <declarators>

(29)   <type_spec>         ::=  <simple_type_spec>
                            |   <constr_type_spec>

(30)   <simple_type_spec>::=<base_type_spec>
                            |   <template_type_spec>
                            |   <scoped_name>

(31)   <base_type_spec>::=  <floating_pt_type>
                            |   <integer_type>
                            |   <char_type>
                            |   <boolean_type>
                            |   <octet_type>
                            |   <any_type>

(32)   <template_type_spec>::=<sequence_type>
                            |   <string_type>

(33)   <constr_type_spec>::=<struct_type>
                            |   <union_type>
                            |   <enum_type>

(34)   <declarators>       ::=  <declarator> { "," <declarator> }*

(35)   <declarator>        ::=  <simple_declarator>
                            |   <complex_declarator>

(36)   <simple_declarator>::=<identifier>

(37)   <complex_declarator>::=<array_declarator>

(38)   <floating_pt_type>::=  "float"
                            |   "double"

(39)   <integer_type>      ::=  <signed_int>
                            |   <unsigned_int>

(40)   <signed_int>        ::=  <signed_long_int>
                            |   <signed_short_int>

(41)   <signed_long_int> ::=  "long"

(42)   <signed_short_int>::= "short"

(43)   <unsigned_int>      ::=  <unsigned_long_int>
                            |   <unsigned_short_int>

(44)   <unsigned_long_int>::="unsigned" "long"

(45)   <unsigned_short_int>::="unsigned" "short"

(46)   <char_type>         ::=  "char"

(47)   <boolean_type>      ::=  "boolean"
```

(48)	<octet_type>	::=	“octet”
(49)	<any_type>	::=	“any”
(50)	<struct_type>	::=	“struct” <identifier> “{” <member_list> “}”
(51)	<member_list>	::=	<member>$^+$
(52)	<member>	::=	<type_spec> <declarators> “;”
(53)	<union_type>	::=	“union” <identifier> “switch” “(” <switch_type_spec> “)” “{” <switch_body> “}”
(54)	<switch_type_spec>	::=	<integer_type>
		|	<char_type>
		|	<boolean_type>
		|	<enum_type>
		|	<scoped_name>
(55)	<switch_body>	::=	<case>$^+$
(56)	<case>	::=	<case_label>$^+$ <element_spec> “;”
(57)	<case_label>	::=	“case” <const_exp> “:”
		|	“default” “:”
(58)	<element_spec>	::=	<type_spec> <declarator>
(59)	<enum_type>	::=	“enum” <identifier> “{” <enumerator> { “,” <enumerator> }* “}”
(60)	<enumerator>	::=	<identifier>
(61)	<sequence_type>	::=	“sequence” “<” <simple_type_spec> “,”<positive_int_const>“>”
		|	“sequence” “<” <simple_type_spec> “>”
(62)	<string_type>	::=	“string” “<” <positive_int_const> “>”
		|	“string”
(63)	<array_declarator>	::=	<identifier> <fixed_array_size>$^+$
(64)	<fixed_array_size>	::=	“[” <positive_int_const> “]”
(65)	<attr_dcl>	::=	[“readonly”] “attribute” <param_type_spec> <simple_declarator> { “,” <simple_declarator> }*
(66)	<except_dcl>	::=	“exception” <identifier> “{“ <member>* “}”
(67)	<op_dcl> <parameter_dcls>	::=	[<op_attribute>] <op_type_spec> <identifier> [<raises_expr>] [<context_expr>]
(68)	<op_attribute>	::=	“oneway”
(69)	<op_type_spec>	::=	<param_type_spec>
		|	“void”
(70)	<parameter_dcls>	::=	“(” <param_dcl> { “,” <param_dcl> }* “)”
		|	“(” “)”
(71)	<param_dcl>	::=	<param_attribute> <param_type_spec> <simple_declarator>

```
(72)    <param_attribute> ::=  “in”
                           |   “out”
                           |   “inout”

(73)    <raises_expr>     ::=  “raises” “(” <scoped_name> { “,” <scoped_name> }* “)”

(74)    <context_expr>    ::=  “context” “(” <string_literal> { “,” <string_literal> }* “)”

(75)    <param_type_spec>::=<base_type_spec>
                           |   <string_type>
                           |   <scoped_name>
```

3.5 OMG IDL Specification

An OMG IDL specification consists of one or more type definitions, constant definitions, exception definitions, or module definitions. The syntax is:

```
<specification>::=  <definition>+

<definition>    ::=  <type_dcl> “;”
                 |   <const_dcl> “;”
                 |   <except_dcl> “;”
                 |   <interface> “;”
                 |   <module> “;”
```

See “Constant Declaration” on page 3-17, “Type Declaration” on page 3-19, and “Exception Declaration” on page 3-26, respectively, for specifications of **<const_dcl>**, **<type_dcl>**, and **<except_dcl>**.

3.5.1 Module Declaration

A module definition satisfies the following syntax:

```
<module>        ::=  “module” <identifier> “{“ <definition>+ “}”
```

The module construct is used to scope OMG IDL identifiers; see “CORBA Module” on page 3-31 for details.

3.5.2 Interface Declaration

An interface definition satisfies the following syntax:

```
<interface>     ::=  <interface_dcl>
                 |   <forward_dcl>

<interface_dcl>::=  <interface_header> “{” <interface_body> “}”

<forward_dcl>  ::=  “interface” <identifier>

<interface_header>::=“interface” <identifier> [ <inheritance_spec> ]
```

```
<interface_body>::=<export>*

<export>       ::=  <type_dcl> ";"
                |   <const_dcl> ";"
                |   <except_dcl> ";"
                |   <attr_dcl> ";"
                |   <op_dcl> ";"
```

Interface Header

The interface header consists of two elements:

- The interface name. The name must be preceded by the keyword **interface**, and consists of an identifier that names the interface.
- An optional inheritance specification. The inheritance specification is described in the next section.

The **<identifier>** that names an interface defines a legal type name. Such a type name may be used anywhere an **<identifier>** is legal in the grammar, subject to semantic constraints as described in the following sections. Since one can only hold references to an object, the meaning of a parameter or structure member which is an interface type is as a *reference* to an object supporting that interface. Each language binding describes how the programmer must represent such interface references.

Inheritance Specification

The syntax for inheritance is as follows:

```
<inheritance_spec>::= ":" <scoped_name> {"," <scoped_name>}*

<scoped_name>::= <identifier>
               | "::" <identifier>
               | <scoped_name> "::" <identifier>
```

Each **<scoped_name>** in an **<inheritance_spec>** must denote a previously defined interface. See “Inheritance” on page 3-15 for the description of inheritance.

Interface Body

The interface body contains the following kinds of declarations:

- Constant declarations, which specify the constants that the interface exports; constant declaration syntax is described in “Constant Declaration” on page 3-17.
- Type declarations, which specify the type definitions that the interface exports; type declaration syntax is described in “Type Declaration” on page 3-19.
- Exception declarations, which specify the exception structures that the interface exports; exception declaration syntax is described in “Exception Declaration” on page 3-26.

- Attribute declarations, which specify the associated attributes exported by the interface; attribute declaration syntax is described in “Attribute Declaration” on page 3-30.
- Operation declarations, which specify the operations that the interface exports and the format of each, including operation name, the type of data returned, the types of all parameters of an operation, legal exceptions which may be returned as a result of an invocation, and contextual information which may affect method dispatch; operation declaration syntax is described in “Operation Declaration” on page 3-27.

Empty interfaces are permitted (that is, those containing no declarations).

Some implementations may require interface-specific pragmas to precede the interface body.

Forward Declaration

A forward declaration declares the name of an interface without defining it. This permits the definition of interfaces that refer to each other. The syntax consists simply of the keyword **interface** followed by an **<identifier>** that names the interface. The actual definition must follow later in the specification.

Multiple forward declarations of the same interface name are legal.

3.6 Inheritance

An interface can be derived from another interface, which is then called a *base* interface of the derived interface. A derived interface, like all interfaces, may declare new elements (constants, types, attributes, exceptions, and operations). In addition, unless redefined in the derived interface, the elements of a base interface can be referred to as if they were elements of the derived interface. The name resolution operator (“::”) may be used to refer to a base element explicitly; this permits reference to a name that has been redefined in the derived interface.

A derived interface may redefine any of the type, constant, and exception names which have been inherited; the scope rules for such names are described in “CORBA Module” on page 3-31.

An interface is called a direct base if it is mentioned in the **<inheritance_spec>** and an indirect base if it is not a direct base but is a base interface of one of the interfaces mentioned in the **<inheritance_spec>**.

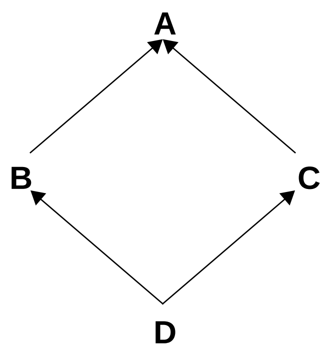
An interface may be derived from any number of base interfaces. Such use of more than one direct base interface is often called multiple inheritance. The order of derivation is not significant.

An interface may not be specified as a direct base interface of a derived interface more than once; it may be an indirect base interface more than once. Consider the following example:

```
interface A { ... }
interface B: A { ... }
interface C: A { ... }
interface D: B, C { ... }
```

The relationships between these interfaces is shown in Figure 11 on page 3-16. This "diamond" shape is legal.

FIGURE 11. **Legal Multiple Inheritance Example**



Reference to base interface elements must be unambiguous. Reference to a base interface element is ambiguous if the expression used refers to a constant, type, or exception in more than one base interface. (It is currently illegal to inherit from two interfaces with the same operation or attribute name, or to redefine an operation or attribute name in the derived interface.) Ambiguities can be resolved by qualifying a name with its interface name (that is, using a **<scoped_name>**).

References to constants, types, and exceptions are bound to an interface when it is defined i.e., replaced with the equivalent global **<scoped_name>**s. This guarantees that the syntax and semantics of an interface are not changed when the interface is a base interface for a derived interface. Consider the following example:

```
const long L = 3;

interface A {
    void f (in float s[L]);             // s has 3 floats
};

interface B {
    const long L = 4;
};

interface C: B, A {}// what is f()'s signature?
```

The early binding of constants, types, and exceptions at interface definition guarantees that the signature of operation **f** in interface **C** is

```
void f(in float s[3]);
```

which is identical to that in interface **A**. This rule also prevents redefinition of a constant, type, or exception in the derived interface from affecting the operations and attributes inherited from a base interface.

Interface inheritance causes all identifiers in the closure of the inheritance tree to be imported into the current naming scope. A type name, constant name, enumeration value name, or exception name from an enclosing scope can be redefined in the current scope. An attempt to use an ambiguous name without qualification is a compilation error.

Operation names are used at run-time by both the stub and dynamic interfaces. As a result, all operations that might apply to a particular object must have unique names. This requirement prohibits redefining an operation name in a derived interface, as well as inheriting two operations with the same name.

Note – It is anticipated that future revisions of the language may relax this rule in some way, perhaps allowing overloading or providing some means to distinguish operations with the same name.

3.7 Constant Declaration

This section describes the syntax for constant declarations.

3.7.1 Syntax

The syntax for a constant declaration is:

```
<const_dcl>    ::=  "const" <const_type> <identifier> "=" <const_exp>

<const_type>   ::=  <integer_type>
                |   <char_type>
                |   <boolean_type>
                |   <floating_pt_type>
                |   <string_type>
                |   <scoped_name>

<const_exp>    ::=  <or_expr>

<or_expr>      ::=  <xor_expr>
                |   <or_expr> "|" <xor_expr>

<xor_expr>     ::=  <and_expr>
                |   <xor_expr> "^" <and_expr>

<and_expr>     ::=  <shift_expr>
                |   <and_expr> "&" <shift_expr>
```

```
<shift_expr>    ::= <add_expr>
                 |  <shift_expr> ">>" <add_expr>
                 |  <shift_expr> "<<" <add_expr>

<add_expr>      ::= <mult_expr>
                 |  <add_expr> "+" <mult_expr>
                 |  <add_expr> "-" <mult_expr>

<mult_expr>     ::= <unary_expr>
                 |  <mult_expr> "*" <unary_expr>
                 |  <mult_expr> "/" <unary_expr>
                 |  <mult_expr> "%" <unary_expr>

<unary_expr>    ::= <unary_operator> <primary_expr>
                 |  <primary_expr>

<unary_operator>::="-"
                 |  "+"
                 |  "~"

<primary_expr>::= <scoped_name>
                 |  <literal>
                 |  "(" <const_exp> ")"

<literal>       ::= <integer_literal>
                 |  <string_literal>
                 |  <character_literal>
                 |  <floating_pt_literal>
                 |  <boolean_literal>

<boolean_literal>::="TRUE"
                 |  "FALSE"

<positive_int_const>::=<const_exp>
```

3.7.2 Semantics

The **<scoped_name>** in the **<const_type>** production must be a previously defined name of an **<integer_type>**, **<char_type>**, **<boolean_type>**, **<floating_pt_type>**, or **<string_type>** constant.

No infix operator can combine an integer and a float. Infix operators are not applicable to types other than integer and float.

An integer constant expression is evaluated as unsigned long unless it contains a negated integer literal or the name of an integer constant with a negative value. In the latter case, the constant expression is evaluated as signed long. The computed value is coerced back to the target type in constant initializers. It is an error if the computed value exceeds the precision of the target type. It is an error if any intermediate value exceeds the range of the evaluated-as type (long or unsigned long).

All floating-point literals are double, all floating-point constants are coerced to double, and all floating-point expressions are computed as doubles. The computed double value is coerced back to the target type in constant initializers. It is an error if this coercion fails or if any intermediate values (when evaluating the expression) exceed the range of double.

Unary (+ –) and binary (* / + –) operators are applicable in floating-point expressions. Unary (+ – ~) and binary (* / % + – << >> & | ^) operators are applicable in integer expressions.

The "~" unary operator indicates that the bit-complement of the expression to which it is applied should be generated. For the purposes of such expressions, the values are 2's complement numbers. As such, the complement can be generated as follows:

long	–(value+1)
unsigned long	(2**32 – 1) – value

The "%" binary operator yields the remainder from the division of the first expression by the second. If the second operand of "%" is 0, the result is undefined; otherwise

(a/b)*b + a%b

is equal to a. If both operands are nonnegative, then the remainder is nonnegative; if not, the sign of the remainder is implementation dependent.

The "<<"binary operator indicates that the value of the left operand should be shifted left the number of bits specified by the right operand, with 0 fill for the vacated bits. The right operand must be in the range 0 <= right operand < 32.

The ">>" binary operator indicates that the value of the left operand should be shifted right the number of bits specified by the right operand, with 0 fill for the vacated bits. The right operand must be in the range 0 <= right operand < 32.

The "&" binary operator indicates that the logical, bitwise AND of the left and right operands should be generated.

The "|" binary operator indicates that the logical, bitwise OR of the left and right operands should be generated.

The "^" binary operator indicates that the logical, bitwise EXCLUSIVE-OR of the left and right operands should be generated.

<positive_int_const> must evaluate to a positive integer constant.

3.8 Type Declaration

OMG IDL provides constructs for naming data types; that is, it provides C language-like declarations that associate an identifier with a type. OMG IDL uses the **typedef** keyword to associate a name with a data type; a name is also associated with a data type via the **struct**, **union**, and **enum** declarations; the syntax is:

```
<type_dcl>      ::=  "typedef" <type_declarator>
                |    <struct_type>
                |    <union_type>
                |    <enum_type>

<type_declarator>::=<type_spec> <declarators>
```

For type declarations, OMG IDL defines a set of type specifiers to represent typed values. The syntax is as follows:

```
<type_spec>     ::=  <simple_type_spec>
                |    <constr_type_spec>

<simple_type_spec>::=<base_type_spec>
                |    <template_type_spec>
                |    <scoped_name>

<base_type_spec>::=<floating_pt_type>
                |    <integer_type>
                |    <char_type>
                |    <boolean_type>
                |    <octet_type>
                |    <any_type>

<template_type_spec>::=<sequence_type>
                |    <string_type>

<constr_type_spec>::=<struct_type>
                |    <union_type>
                |    <enum_type>

<declarators>   ::=  <declarator> { "," <declarator> }*

<declarator>    ::=  <simple_declarator>
                |    <complex_declarator>

<simple_declarator>::=<identifier>

<complex_declarator>::=<array_declarator>
```

The **<scoped_name>** in **<simple_type_spec>** must be a previously defined type.

As seen above, OMG IDL type specifiers consist of scalar data types and type constructors. OMG IDL type specifiers can be used in operation declarations to assign data types to operation parameters. The next sections describe basic and constructed type specifiers.

3.8.1 Basic Types

The syntax for the supported basic types is as follows:

```
<floating_pt_type>::="float"
                |    "double"
```

```
<integer_type>::=  <signed_int>
                |  <unsigned_int>
<signed_int>   ::= <signed_long_int>
                |  <signed_short_int>
<signed_long_int>::="long"
<signed_short_int>::="short"
<unsigned_int>::=  <unsigned_long_int>
                |  <unsigned_short_int>
<unsigned_long_int>::="unsigned" "long"
<unsigned_short_int>::="unsigned" "short"
<char_type>    ::= "char"
<boolean_type>::= "boolean"
<octet_type>   ::= "octet"
<any_type>     ::= "any"
```

Each OMG IDL data type is mapped to a native data type via the appropriate language mapping. Conversion errors between OMG IDL data types and the native types to which they are mapped can occur during the performance of an operation invocation. The invocation mechanism (client stub, dynamic invocation engine, and skeletons) may signal an exception condition to the client if an attempt is made to convert an illegal value. The standard exceptions which are to be signalled in such situations are defined in “Standard Exceptions” on page 3-33.

Integer Types

OMG IDL supports **long** and **short** signed and **unsigned** integer data types. **long** represents the range $-2^{31} .. 2^{31} - 1$ while **unsigned long** represents the range $0 .. 2^{32} - 1$. **short** represents the range $-2^{15} .. 2^{15} - 1$, while **unsigned short** represents the range $0 .. 2^{16} - 1$.

Floating-Point Types

OMG IDL floating-point types are **float** and **double**. The **float** type represents IEEE single-precision floating point numbers; the **double** type represents IEEE double-precision floating point numbers.The IEEE floating point standard specification (*IEEE Standard for Binary Floating-Point Arithmetic*, ANSI/IEEE Std 754-1985) should be consulted for more information on the precision afforded by these types.

Implementations that do not fully support the value set of the IEEE 754 floating-point standard must completely specify their deviance from the standard.

Char Type

OMG IDL defines a **char** data type consisting of 8-bit quantities.

The ISO Latin-1 (8859.1) character set standard defines the meaning and representation of all possible graphic characters (i.e., the space, alphabetic, digit and graphic characters defined in Figure 2 on page 3-3, Figure 3 on page 3-4, and Figure 4 on page 3-4). The meaning and representation of the null and formatting characters (see Figure 5 on page 3-5) is the numerical value of the character as defined in the ASCII (ISO 646) standard. The meaning of all other characters is implementation-dependent.

During transmission, characters may be converted to other appropriate forms as required by a particular language binding. Such conversions may change the representation of a character but maintain the character's meaning. For example, a character may be converted to and from the appropriate representation in international character sets.

Boolean Type

The **boolean** data type is used to denote a data item that can only take one of the values TRUE and FALSE.

Octet Type

The **octet** type is an 8-bit quantity that is guaranteed not to undergo any conversion when transmitted by the communication system.

Any Type

The **any** type permits the specification of values that can express any OMG IDL type.

3.8.2 Constructed Types

The constructed types are:

```
<constr_type_spec>::=<struct_type>
                |    <union_type>
                |    <enum_type>
```

Although it is syntactically possible to generate recursive type specifications in OMG IDL, such recursion is semantically constrained. The only permissible form of recursive type specification is through the use of the **sequence** template type. For example, the following is legal:

```
struct foo {
    long value;
    sequence<foo> chain;
}
```

See “Sequences” on page 3-25 for details of the **sequence** template type.

Structures

The structure syntax is:

```
<struct_type>  ::=  “struct” <identifier> “{” <member_list> “}”
<member_list>::=  <member>+
<member>       ::=  <type_spec> <declarators> “;”
```

The **<identifier>** in **<struct_type>** defines a new legal type. Structure types may also be named using a **typedef** declaration.

Name scoping rules require that the member declarators in a particular structure be unique. The value of a **struct** is the value of all of its members.

Discriminated Unions

The discriminated **union** syntax is:

```
<union_type>  ::=  “union” <identifier> “switch” “(” <switch_type_spec> “)”
                   “{” <switch_body> “}”
<switch_type_spec>::=<integer_type>
                |  <char_type>
                |  <boolean_type>
                |  <enum_type>
                |  <scoped_name>
<switch_body>::=  <case>+
<case>          ::=  <case_label>+ <element_spec> “;”
<case_label>  ::=  “case” <const_exp> “:”
                |  “default” “:”
<element_spec>::=<type_spec> <declarator>
```

OMG IDL unions are a cross between the C **union** and **switch** statements. IDL unions must be discriminated; that is, the union header must specify a typed tag field that determines which union member to use for the current instance of a call. The **<identifier>** following the **union** keyword defines a new legal type. Union types may also be named using a **typedef** declaration. The **<const_exp>** in a **<case_label>** must be consistent with the **<switch_type_spec>**. A **default** case can appear at most once. The **<scoped_name>** in the **<switch_type_spec>** production must be a previously defined **integer**, **char**, **boolean** or **enum** type.

Case labels must match or be automatically castable to the defined type of the discriminator. The complete set of matching rules are shown in Figure 10 on page 3-24.

TABLE 10. Case Label Matching

Discriminator Type	Matched By
long	any integer value in the value range of long
short	any integer value in the value range of short
unsigned long	any integer value in the value range of unsigned long
unsigned short	any integer value in the value range of unsigned short
char	char
boolean	TRUE or FALSE
enum	any enumerator for the discriminator enum type

Name scoping rules require that the element declarators in a particular union be unique. If the **<switch_type_spec>** is an **<enum_type>**, the identifier for the enumeration is in the scope of the union; as a result, it must be distinct from the element declarators.

It is not required that all possible values of the union discriminator be listed in the **<switch_body>**. The value of a union is the value of the discriminator together with one of the following:

- If the discriminator value was explicitly listed in a **case** statement, the value of the element associated with that **case** statement;
- If a default **case** label was specified, the value of the element associated with the default **case** label;
- No additional value.

Access to the discriminator and the related element is language-mapping dependent.

Enumerations

Enumerated types consist of ordered lists of identifiers. The syntax is:

<enum_type> ::= "enum" <identifier> "{" <enumerator> { "," <enumerator> }* "}"

<enumerator> ::= <identifier>

A maximum of 2^{32} identifiers may be specified in an enumeration; as such, the enumerated names must be mapped to a native data type capable of representing a maximally-sized enumeration. The order in which the identifiers are named in the specification of an enumeration defines the relative order of the identifiers. Any language mapping which permits two enumerators to be compared or defines successor/predecessor functions on enumerators must conform to this ordering relation. The **<identifier>** following the **enum** keyword defines a new legal type. Enumerated types may also be named using a **typedef** declaration.

3.8.3 Template Types

The template types are:

```
<template_type_spec>:   ::=<sequence_type>
                |    <string_type>
```

Sequences

OMG IDL defines the sequence type **sequence**. A sequence is a one-dimensional array with two characteristics: a maximum size (which is fixed at compile time) and a length (which is determined at run time).

The syntax is:

```
<sequence_type>   ::=“sequence” “<” <simple_type_spec> “,”
<positive_int_const> “>”
                |    “sequence” “<” <simple_type_spec> “>”
```

The second parameter in a sequence declaration indicates the maximum size of the sequence. If a positive integer constant is specified for the maximum size, the sequence is termed a bounded sequence. Prior to passing a bounded sequence as a function argument (or as a field in a structure or union), the length of the sequence must be set in a language-mapping dependent manner. After receiving a sequence result from an operation invocation, the length of the returned sequence will have been set; this value may be obtained in a language-mapping dependent manner.

If no maximum size is specified, size of the sequence is unspecified (unbounded). Prior to passing such a sequence as a function argument (or as a field in a structure or union), the length of the sequence, the maximum size of the sequence, and the address of a buffer to hold the sequence must be set in a language-mapping dependent manner. After receiving such a sequence result from an operation invocation, the length of the returned sequence will have been set; this value may be obtained in a language-mapping dependent manner.

A sequence type may be used as the type parameter for another sequence type. For example, the following:

```
typedef sequence< sequence<long> > Fred;
```

declares Fred to be of type “unbounded sequence of unbounded sequence of long”. Note that for nested sequence declarations, white space must be used to separate the two “>” tokens ending the declaration so they are not parsed as a single “>>” token.

Strings

OMG IDL defines the string type **string** consisting of all possible 8-bit quantities except null. A string is similar to a sequence of char. As with sequences of any type, prior to passing a string as a function argument (or as a field in a structure or union), the length of the string must be set in a language-mapping dependent manner. The syntax is:

<string_type>: :="string" "<" <positive_int_const> ">"
| "string"

The argument to the string declaration is the maximum size of the string. If a positive integer maximum size is specified, the string is termed a bounded string; if no maximum size is specified, the string is termed an unbounded string.

Strings are singled out as a separate type because many languages have special built-in functions or standard library functions for string manipulation. A separate string type may permit substantial optimization in the handling of strings compared to what can be done with sequences of general types.

3.8.4 Complex Declarator

Arrays

OMG IDL defines multidimensional, fixed-size arrays. An array includes explicit sizes for each dimension.

The syntax for arrays is:

<array_declarator> ::=<identifier> <fixed_array_size>$^{+}$

<fixed_array_size> ::="[" <positive_int_const> "]"

The array size (in each dimension) is fixed at compile time. When an array is passed as a parameter in an operation invocation, all elements of the array are transmitted.

The implementation of array indices is language mapping specific; passing an array index as a parameter may yield incorrect results.

3.9 Exception Declaration

Exception declarations permit the declaration of struct-like data structures which may be returned to indicate that an exceptional condition has occurred during the performance of a request. The syntax is as follows:

<except_dcl> : :="exception" <identifier> "{" <member>* "}"

Each exception is characterized by its OMG IDL identifier, an exception type identifier, and the type of the associated return value (as specified by the **<member>**s in its declaration. If an exception is returned as the outcome to a request, then the value of the exception identifier is accessible to the programmer for determining which particular exception was raised.

If an exception is declared with members, a programmer will be able to access the values of those members when an exception is raised. If no members are specified, no additional information is accessible when an exception is raised.

A set of standard exceptions is defined corresponding to standard run-time errors which may occur during the execution of a request. These standard exceptions are documented in "Standard Exceptions" on page 3-33.

3.10 Operation Declaration

Operation declarations in OMG IDL are similar to C function declarations. The syntax is:

```
<op_dcl>          ::=  [ <op_attribute> ] <op_type_spec> <identifier>
<parameter_dcls>
                       [ <raises_expr> ] [ <context_expr> ]

<op_type_spec>::=<param_type_spec>
                  |    "void"
```

An operation declaration consists of:

- An optional operation attribute that specifies which invocation semantics the communication system should provide when the operation is invoked. Operation attributes are described in "Operation Attribute" on page 3-28.
- The type of the operation's return result; the type may be any type which can be defined in OMG IDL. Operations that do not return a result must specify the **void** type.
- An identifier that names the operation in the scope of the interface in which it is defined.
- A parameter list that specifies zero or more parameter declarations for the operation. Parameter declaration is described in "Parameter Declarations" on page 3-28.
- An optional raises expression which indicates which exceptions may be raised as a result of an invocation of this operation. Raises expressions are described in Section "Raises Expressions" on page 3-29.
- An optional context expression which indicates which elements of the request context may be consulted by the method that implements the operation. Context expressions are described in "Context Expressions" on page 3-29.

Some implementations and/or language mappings may require operation-specific pragmas to immediately precede the affected operation declaration.

3.10.1 Operation Attribute

The operation attribute specifies which invocation semantics the communication service must provide for invocations of a particular operation. An operation attribute is optional. The syntax for its specification is as follows:

```
<op_attribute> ::=  "oneway"
```

When a client invokes an operation with the **oneway** attribute, the invocation semantics are best-effort, which does not guarantee delivery of the call; best-effort implies that the operation will be invoked at most once. An operation with the **oneway** attribute must not contain any output parameters and must specify a **void** return type. An operation defined with the **oneway** attribute may not include a raises expression; invocation of such an operation, however, may raise a standard exception.

If an **<op_attribute>** is not specified, the invocation semantics is at-most-once if an exception is raised; the semantics are exactly-once if the operation invocation returns successfully.

3.10.2 Parameter Declarations

Parameter declarations in OMG IDL operation declarations have the following syntax:

```
<parameter_dcls>::="(" <param_dcl> { "," <param_dcl> }* ")"
                  |    "(" ")"
<param_dcl>   ::=  <param_attribute> <param_type_spec> <simple_declarator>
<param_attribute>::="in"
                  |    "out"
                  |    "inout"
<param_type_spec>::=<base_type_spec>
                  |    <string_type>
                  |    <scoped_name>
```

A parameter declaration must have a directional attribute that informs the communication service in both the client and the server of the direction in which the parameter is to be passed. The directional attributes are:

- **in** - the parameter is passed from client to server.
- **out** - the parameter is passed from server to client.
- **inout** - the parameter is passed in both directions.

It is expected that an implementation will *not* attempt to modify an **in** parameter. The ability to even attempt to do so is language-mapping specific; the effect of such an action is undefined.

If an exception is raised as a result of an invocation, the values of the return result and any **out** and **inout** parameters are undefined.

When an unbounded **string** or **sequence** is passed as an **inout** parameter, the returned value cannot be longer than the input value.

3.10.3 Raises Expressions

A **raises** expression specifies which exceptions may be raised as a result of an invocation of the operation. The syntax for its specification is as follows:

<raises_expr> ::= "raises" "(" <scoped_name> { "," <scoped_name> }* ")"

The **<scoped_name>**'s in the **raises** expression must be previously defined exceptions.

In addition to any operation-specific exceptions specified in the **raises** expression, there are a standard set of exceptions that may be signalled by the ORB. These standard exceptions are described in "Standard Exceptions" on page 3-33. However, standard exceptions may *not* be listed in a **raises** expression.

The absence of a **raises** expression on an operation implies that there are no operation-specific exceptions. Invocations of such an operation are still liable to receive one of the standard exceptions.

3.10.4 Context Expressions

A **context** expression specifies which elements of the client's context may affect the performance of a request by the object. The syntax for its specification is as follows:

<context_expr>::= "context" "(" <string_literal> { "," <string_literal> }* ")"

The run-time system guarantees to make the value (if any) associated with each **<string_literal>** in the client's context available to the object implementation when the request is delivered. The ORB and/or object is free to use information in this *request context* during request resolution and performance.

The absence of a context expression indicates that there is no request context associated with requests for this operation.

Each **string_literal** is an arbitrarily long sequence of alphabetic, digit, period ("."), underscore ("_"), and asterisk ("*") characters. The first character of the string must be an alphabetic character. An asterisk may only be used as the last character of the string. Some implementations may use the period character to partition the name space.

The mechanism by which a client associates values with the context identifiers is described in the Dynamic Invocation Interface chapter.

3.11 Attribute Declaration

An interface can have attributes as well as operations; as such, attributes are defined as part of an interface. An attribute definition is logically equivalent to declaring a pair of accessor functions; one to retrieve the value of the attribute and one to set the value of the attribute.

The syntax for **attribute** declaration is:

```
<attr_dcl>     ::=[ “readonly” ] “attribute” <param_type_spec> <simple_declarator>
                    { “,” <simple_declarator> }*
```

The optional **readonly** keyword indicates that there is only a single accessor function—the retrieve value function. Consider the following example:

```
interface foo {
enum material_t {rubber, glass};
struct position_t {
float x, y;
};

attribute float radius;
attribute material_t material;
readonly attribute position_t position;

• • •
};
```

The attribute declarations are equivalent to the following pseudo-specification fragment:

```
• • •
float _get_radius ();
void _set_radius (in float r);
material_t _get_material ();
void _set_material (in material_t m);
position_t _get_position ();
• • •
```

The actual accessor function names are language-mapping specific. The C, C++, and Smalltalk mappings are described in separate chapters. The attribute name is subject to OMG IDL’s name scoping rules; the accessor function names are guaranteed *not* to collide with any legal operation names specifiable in OMG IDL.

Attribute operations return errors by means of standard exceptions.

Attributes are inherited. An attribute name *cannot* be redefined to be a different type. See “CORBA Module” on page 3-31 for more information on redefinition constraints and the handling of ambiguity.

3.12 CORBA Module

In order to prevent names defined in the *CORBA* specification from clashing with names in programming languages and other software systems, all names defined in *CORBA* are treated as if they were defined within a module named CORBA. In an OMG IDL specification, however, OMG IDL keywords such as Object must not be preceded by a "CORBA::" prefix. Other interface names such as TypeCode are not OMG IDL keywords, so they must be referred to by their fully scoped names (e.g., CORBA::TypeCode) within an OMG IDL specification.

3.13 Names and Scoping

An entire OMG IDL file forms a naming scope. In addition, the following kinds of definitions form nested scopes:

- module
- interface
- structure
- union
- operation
- exception

Identifiers for the following kinds of definitions are scoped:

- types
- constants
- enumeration values
- exceptions
- interfaces
- attributes
- operations

An identifier can only be defined once in a scope. However, identifiers can be redefined in nested scopes. An identifier declaring a module is considered to be defined by its first occurrence in a scope. Subsequent occurrences of a module declaration within the same scope reopen the module allowing additional definitions to be added to it.

Due to possible restrictions imposed by future language bindings, OMG IDL identifiers are case insensitive; that is, two identifiers that differ only in the case of their characters are considered redefinitions of one another. However, all references to a definition must use the same case as the defining occurrence. (This allows natural mappings to case-sensitive languages.)

Type names defined in a scope are available for immediate use within that scope. In particular, see "Constructed Types" on page 3-22 on cycles in type definitions.

A name can be used in an unqualified form within a particular scope; it will be resolved by successively searching farther out in enclosing scopes. Once an unqualified name is used in a scope, it cannot be redefined—i.e. if one has used a name defined in an enclosing scope in the current scope, one cannot then redefine a version of the name in the current scope. Such redefinitions yield a compilation error.

A qualified name (one of the form <scoped-name>::<identifier>) is resolved by first resolving the qualifier <scoped-name> to a scope S, and then locating the definition of <identifier> within S. The identifier must be directly defined in S or (if S is an interface) inherited into S. The <identifier> is not searched for in enclosing scopes.

When a qualified name begins with "::", the resolution process starts with the file scope and locates subsequent identifiers in the qualified name by the rule described in the previous paragraph.

Every OMG IDL definition in a file has a global name within that file. The global name for a definition is constructed as follows.

Prior to starting to scan a file containing an OMG IDL specification, the name of the current root is initially empty ("") and the name of the current scope is initially empty (""). Whenever a **module** keyword is encountered, the string "::" and the associated identifier are appended to the name of the current root; upon detection of the termination of the **module**, the trailing "::" and identifier are deleted from the name of the current root. Whenever an **interface**, **struct**, **union**, or **exception** keyword is encountered, the string "::" and the associated identifier are appended to the name of the current scope; upon detection of the termination of the **interface**, **struct**, **union**, or **exception**, the trailing "::" and identifier are deleted from the name of the current scope. Additionally, a new, unnamed, scope is entered when the parameters of an operation declaration are processed; this allows the parameter names to duplicate other identifiers; when parameter processing has completed, the unnamed scope is exited.

The global name of an OMG IDL definition is the concatenation of the current root, the current scope, a "::", and the <identifier>, which is the local name for that definition.

Note that the global name in an OMG IDL files corresponds to an absolute **ScopedName** in the Interface Repository. (See "Supporting Type Definitions" on page 6-8.)

Inheritance produces shadow copies of the inherited identifiers; that is, it introduces names into the derived interface, but these names are considered to be semantically the same as the original definition. Two shadow copies of the same original (as results from the diamond shape in Figure 11 on page 3-16) introduce a single name into the derived interface and don't conflict with each other.

Inheritance introduces multiple global OMG IDL names for the inherited identifiers. Consider the following example:

```
interface A {
exception E {
long L;
};
void f() raises(E);
};

interface B: A {
void g() raises(E);
};
```

In this example, the exception is known by the global names **::A::E** and **::B::E**.

Ambiguity can arise in specifications due to the nested naming scopes. For example:

```
interface A {
typedef string<128> string_t;
};

interface B {
typedef string<256> string_t;
};

interface C: A, B {
attribute string_t Title;/* AMBIGUOUS!!! */
};
```

The attribute declaration in C is ambiguous, since the compiler does not know which **string_t** is desired. Ambiguous declarations yield compilation errors.

3.14 Differences from C++

The OMG IDL grammar, while attempting to conform to the C++ syntax, is somewhat more restrictive. The current restrictions are as follows:

- A function return type is mandatory.
- A name must be supplied with each formal parameter to an operation declaration.
- A parameter list consisting of the single token **void** is *not* permitted as a synonym for an empty parameter list.
- Tags are required for structures, discriminated unions, and enumerations.
- Integer types cannot be defined as simply int or unsigned; they must be declared explicitly as **short** or **long**.
- **char** cannot be qualified by **signed** or **unsigned** keywords.

3.15 Standard Exceptions

This section presents the standard exceptions defined for the ORB. These exception identifiers may be returned as a result of any operation invocation, regardless of the interface specification. Standard exceptions may not be listed in **raises** expressions.

In order to bound the complexity in handling the standard exceptions, the set of standard exceptions should be kept to a tractable size. This constraint forces the definition of equivalence classes of exceptions rather than enumerating many similar exceptions. For example, an operation invocation can fail at many different points due to the inability to allocate dynamic memory. Rather than enumerate several different exceptions corresponding to the different ways that memory allocation failure causes the exception (during marshalling, unmarshalling, in the client, in the object implementation, allocating network packets, ...), a single exception corresponding to dynamic memory allocation failure is defined. Each standard exception includes a minor code to designate the subcategory of the exception; the assignment of values to the minor codes is left to each ORB implementation.

Each standard exception also includes a **completion_status** code which takes one of the values {COMPLETED_YES, COMPLETED_NO, COMPLETED_MAYBE}. These have the following meanings:

COMPLETED_YES The object implementation has completed processing prior to the exception being raised.

COMPLETED_NO The object implementation was never initiated prior to the exception being raised.

COMPLETED_MAYBE The status of implementation completion is indeterminate.

3.15.1 Standard Exceptions Definitions

The standard exceptions are defined below.

```
#define ex_body {unsigned long minor; completion_status completed;}

enum completion_status {COMPLETED_YES, COMPLETED_NO,
COMPLETED_MAYBE};
enum exception_type {NO_EXCEPTION, USER_EXCEPTION,
SYSTEM_EXCEPTION};

exception UNKNOWN           ex_body;   // the unknown exception
exception BAD_PARAM         ex_body;   // an invalid parameter was
                                       // passed
exception NO_MEMORY         ex_body;   // dynamic memory allocation
                                       // failure
exception IMP_LIMIT         ex_body;   // violated implementation limit
exception COMM_FAILURE      ex_body;   // communication failure
exception INV_OBJREF        ex_body;   // invalid object reference
exception NO_PERMISSION     ex_body;   // no permission for attempted op.
exception INTERNAL          ex_body;   // ORB internal error
exception MARSHAL           ex_body;   // error marshalling param/result
exception INITIALIZE        ex_body;   // ORB initialization failure
exception NO_IMPLEMENT      ex_body;   // operation implementation
                                       // unavailable
exception BAD_TYPECODE      ex_body;   // bad typecode
exception BAD_OPERATION     ex_body;   // invalid operation
exception NO_RESOURCES      ex_body;   // insufficient resources for req.
exception NO_RESPONSE       ex_body;   // response to req. not yet
                                       // available
exception PERSIST_STORE     ex_body;   // persistent storage failure
exception BAD_INV_ORDER     ex_body;   // routine invocations out of order
exception TRANSIENT         ex_body;   // transient failure - reissue
                                       // request
exception FREE_MEM          ex_body;   // cannot free memory
exception INV_IDENT         ex_body;   // invalid identifier syntax
exception INV_FLAG          ex_body;   // invalid flag was specified
exception INTF_REPOS        ex_body;   // error accessing interface
                                       // repository
exception BAD_CONTEXT       ex_body;   // error processing context object
exception OBJ_ADAPTER       ex_body;   // failure detected by object
                                       // adapter
exception DATA_CONVERSION      ex_body;   // data conversion error
exception OBJECT_NOT_EXIST     ex_body;   // non-existent object, delete
                                          // reference
```

3.15.2 Object Non-Existence

This standard system exception is raised whenever an invocation on a deleted object was performed. It is an authoritative “hard” fault report. Anyone receiving it is allowed (even expected) to delete all copies of this object reference and to perform other appropriate “final recovery” style procedures.

Bridges forward this exception to clients, also destroying any records they may hold (for example, proxy objects used in reference translation). The clients could in turn purge any of their own data structures.

Dynamic Invocation Interface 4

4.1 Overview

The ORB *Dynamic Invocation* interface allows dynamic creation and invocation of requests to objects. A client using this interface to send a request to an object obtains the same semantics as a client using the operation stub generated from the type specification.

A request consists of an object reference, an operation, and a list of parameters. The ORB applies the implementation-hiding (encapsulation) principle to requests.

In the *Dynamic Invocation* interface, parameters in a request are supplied as elements of a list. Each element is an instance of a **NamedValue** (see "Common Data Structures" on page 4-1). Each parameter is passed in its native data form.

Parameters supplied to a request may be subject to run-time type checking upon request invocation. Parameters must be supplied in the same order as the parameters defined for the operation in the Interface Repository.

All types defined in this chapter are part of the CORBA module. When referenced in OMG IDL, the type names must be prefixed by "CORBA::".

4.1.1 Common Data Structures

The type **NamedValue** is a well-known data type in OMG IDL. It can be used either as a parameter type directly or as a mechanism for describing arguments to a request. The type NVList is a pseudo-object useful for constructing parameter lists. The types are described in OMG IDL and C, respectively, as:

```
typedef unsigned long Flags;

struct NamedValue {
      Identifier        name;       // argument name
      any               argument;   // argument
      long              len;        // length/count of argument value
      Flags             arg_modes;// argument mode flags
};
CORBA_NamedValue * CORBA_NVList;                          /* C */
```

NamedValue and **Flags** are defined in the CORBA module.

The **NamedValue** and **NVList** structures are used in the request operations to describe arguments and return values. They are also used in the context object routines to pass lists of property names and values. Despite the above declaration for **NVList**, the **NVList** structure is partially opaque and may only be created by using the ORB **create_list** operation.

A named value includes an argument name, argument value (as an **any**), length of the argument, and a set of argument mode flags. When named value structures are used to describe arguments to a request, the names are the argument identifiers specified in the OMG IDL definition for a specific operation.

As described in Section 14.7, “Mapping for Basic Data Types,” on page 14-8, an **any** consists of a **TypeCode** and a pointer to the data value. The TypeCode is a well-known opaque type that can encode a description of any type specifiable in OMG IDL. A full description of TypeCodes is Section 14.7, “Mapping for Basic Data Types,” on page 14-8.

For most datatypes, **len** is the actual number of bytes that the value occupies. For object references, **len** is 1. TABLE 11. on page 4-2 shows the length of data values for the C language binding. The behavior of a NamedValue is undefined if the **len** value is inconsistent with the TypeCode.

TABLE 11. **C Type Lengths**

Data type: X	Length (X)
short	sizeof (CORBA_short)
unsigned short	sizeof (CORBA_unsigned_short)
long	sizeof (CORBA_long)
unsigned long	sizeof (CORBA_unsigned_long)
float	sizeof (CORBA_float)
double	sizeof (CORBA_double)
char	sizeof (CORBA_char)
boolean	sizeof (char)
octet	sizeof (CORBA_octet)
string	strlen (string) /* does NOT include ‘\0’ byte! */

TABLE 11. C Type Lengths *(Continued)*

Data type: X	Length (X)
enum E {};	sizeof (CORBA_enum)
union U { };	sizeof (U)
struct S { };	sizeof (S)
Object	1
array N of type T1	Length (T1) * N
sequence V of type T2	Length (T2) * V /* V is the actual, dynamic, number of elements */

The **arg_modes** field is defined as a bitmask (long) and may contain the following flag values:

CORBA::ARG_IN the associated value is an input only argument

CORBA::ARG_OUT the associated value is an output only argument

CORBA::ARG_INOUT the associated value is an in/out argument

These flag values identify the parameter passing mode for arguments. Additional flag values have specific meanings for request and list routines, and are documented with their associated routines.

All other bits are reserved. The high-order 16 bits are reserved for implementation-specific flags.

4.1.2 Memory Usage

The values for output argument data types that are unbounded strings or unbounded sequences are returned as pointers to dynamically allocated memory are shown in Table 21. In order to facilitate the freeing of all “out-arg memory”, the request routines provide a mechanism for grouping, or keeping track of, this memory. If so specified, out-arg memory is associated with the argument list passed to the create request routine. When the list is deleted the associated out-arg memory will automatically be freed.

If the programmer chooses not to associate out-arg memory with an argument list, the programmer is responsible for freeing each out parameter using **`CORBA_free()`**, which is discussed in Section 14.17, “Argument Passing Considerations,” on page 14-16.

4.1.3 Return Status and Exceptions

In the *Dynamic Invocation* interface, many routines return a **Status** result, which is intended as a status code. **Status** is defined in the CORBA modules as:

```
typedef unsigned long Status;
```

Conforming CORBA implementations are *not* required to return this status code; instead, the definition

typedef void Status;

is a conforming implementation (in which case no status code result is returned, except in the usual **inout Environment** argument). Implementations are required to specify which **Status** behavior is supported.

4.2 Request Operations

The request operations are defined in terms of the Request pseudo-object. The Request routines use the **NVList** definition defined in the preceding section.

```
module CORBA {

    interface Request {                                             // PIDL

        Status add_arg (
            in Identifier       name,           // argument name
            in TypeCode         arg_type,       // argument datatype
            in void             * value,        // argument value to be added
            in long             len,            // length/count of argument value
            in Flags            arg_flags       // argument flags
        );
        Status invoke (
            in Flags            invoke_flags    // invocation flags
        );
        Status delete ();
        Status send (
            in Flags            invoke_flags// invocation flags
        );
        Status get_response (
            in Flags            response_flags  // response flags
        );
    };
};
```

4.2.1 create_request

Because it creates a pseudo-object, this operation is defined in the Object interface (see “Object Reference Operations” on page 7-2 for the complete interface definition). The **create_request** operation is performed on the Object which is to be invoked.

```
Status create_request (                                              // PIDL
    in Context          ctx,            // context object for operation
    in Identifier       operation,      // intended operation on object
    in NVList           arg_list,       // args to operation
    inout NamedValue result,            // operation result
    out Request         request,        // newly created request
    in Flags            req_flags       // request flags
);
```

This operation creates an ORB request. The actual invocation occurs by calling **invoke** or by using the **send** / **get_response** calls.

The operation name specified on **create_request** is the same operation identifier that is specified in the OMG IDL definition for this operation. In the case of attributes, it is the name as constructed following the rules specified in the ServerRequest interface as described in the DSI in Section 5.2

The **arg_list**, if specified, contains a list of arguments (input, output, and/or input/output) which become associated with the request. If **arg_list** is omitted (specified as NULL), the arguments (if any) must be specified using the **add_arg** call below.

Arguments may be associated with a request by passing in an argument list or by using repetitive calls to **add_arg**. One mechanism or the other may be used for supplying arguments to a given request; a mixture of the two approaches is not supported.

If specified, the **arg_list** becomes associated with the request; until the **invoke** call has completed (or the request has been deleted), the ORB assumes that **arg_list** (and any values it points to) remains unchanged.

When specifying an argument list, the **value** and **len** for each argument must be specified. An argument's datatype, name, and usage flags (i.e., in, out, inout) may also be specified; if so indicated, arguments are validated for datatype, order, name, and usage correctness against the set of arguments expected for the indicated operation.

An implementation of the request services may relax the order constraint (and allow arguments to be specified out of order) by doing ordering based upon argument name.

The context properties associated with the operation are passed to the object implementation. The object implementation may not modify the context information passed to it.

The operation result is placed in the **result** argument after the invocation completes.

The **req_flags** argument is defined as a bitmask (**long**) that may contain the following flag values:

CORBA::OUT_LIST_MEMORYIndicates that any out-arg memory is associated with the argument list (**`NVList`**).

Setting the OUT_LIST_MEMORY flag controls the memory allocation mechanism for out-arg memory (output arguments, for which memory is dynamically allocated). If OUT_LIST_MEMORY is specified, an argument list must also have been specified on the **create_request** call. When output arguments of this type are allocated, they are associated with the list structure. When the list structure is freed (see below), any associated out-arg memory is also freed.

If OUT_LIST_MEMORY is *not* specified, then each piece of out-arg memory remains available until the programmer explicitly frees it with procedures provided by the language mappings (See Section 14.17, “Argument Passing Considerations,” on page 14-16; Section 17.6, “NVList,” on page 17-5; and Section 20.19, “Argument Passing Considerations,” on page 20-11.)

4.2.2 add_arg

```
Status add_arg (                                                    // PIDL
    in Identifier       name,       // argument name
    in TypeCode         arg_type,   // argument datatype
    in void             * value,    // argument value to be added
    in long             len,        // length/count of argument value
    in Flags            arg_flags   // argument flags
);
```

add_arg incrementally adds arguments to the request.

For each argument, minimally its **value** and **len** must be specified. An argument’s datatype, name, and usage flags (i.e in, out, inout) may also be specified. If so indicated, arguments are validated for datatype, order, name, and usage correctness against the set of arguments expected for the indicated operation.

An implementation of the request services may relax the order constraint (and allow arguments to be specified out of order) by doing ordering based upon argument name.

The arguments added to the request become associated with the request and are assumed to be unchanged until the invoke has completed (or the request has been deleted).

Arguments may be associated with a request by specifying them on the **create_request** call or by adding them via calls to **add_arg**. Using both methods for specifying arguments, for the same request, is not currently supported.

In addition to the argument modes defined in Section 4.1.1, **arg_flags** may also take the flag value:IN_COPY_VALUE. The argument passing flags defined in Section 4.1.1 may be used here to indicate the intended parameter passing mode of an argument.

If the IN_COPY_VALUE flag is set, a copy of the argument value is made and used instead. This flag is ignored for inout and out arguments.

4.2.3 invoke

```
Status invoke (                                         // PIDL
    in Flags         invoke_flags     // invocation flags
              );
```

This operation calls the ORB, which performs method resolution and invokes an appropriate method. If the method returns successfully, its result is placed in the **result** argument specified on cr**eate_request**.

4.2.4 delete

```
Status delete ( );                                      // PIDL
```

This operation deletes the request. Any memory associated with the request (i.e. by using the **IN_COPY_VALUE** flag) is also freed.

4.3 Deferred Synchronous Operations

4.3.1 send

```
Status send (                                   // PIDL
    in Flags         invoke_flags         // invocation flags
              );
```

send initiates an operation according to the information in the Request. Unlike **invoke**, **send** returns control to the caller without waiting for the operation to finish. To determine when the operation is done, the caller must use the **get_response** or **get_next_response** operations described below. The out parameters and return value must not be used until the operation is done.

Although it is possible for some standard exceptions to be raised by the **send** operation, there is no guarantee that all possible errors will be detected. For example, if the object reference is not valid, **send** might detect it and raise an exception, or might return before the object reference is validated, in which case the exception will be raised when **get_response** is called.

If the operation is defined to be **oneway** or if INV_NO_RESPONSE is specified, then **get_response** does not need to be called. In such cases, some errors might go unreported, since if they are not detected before **send** returns there is no way to inform the caller of the error.

The following invocation flags are currently defined for **send**:

CORBA::INV_NO_RESPONSE Indicates that the invoker does not intend to wait for a response, nor does it expect any of the output arguments (in/out and out) to be updated. This option may be specified even if the operation has not been defined to be **oneway**.

4.3.2 send_multiple_requests

```
/* C */

CORBA_Status CORBA_send_multiple_requests (
      CORBA_Request     reqs[],        /* array of Requests */
      CORBA_Environment *env,
      CORBA_long        count,         /* number of Requests */
      CORBA_Flags       invoke_flags
   );

// C++

class ORB
{
   public:
    typedef sequence<Request_ptr> RequestSeq;
...
    Status send_multiple_requests_oneway(const RequestSeq &);
    Status send_multiple_requests_deferred(const RequestSeq &);
};
```

The Smalltalk mapping of send multiple requests is as follows:

```
sendMultipleRequests: aCollection
sendMultipleRequestOneway: aCollection
```

send_multiple_requests initiates more than one request in parallel. Like **send**, **send_multiple_requests** returns to the caller without waiting for the operations to finish. To determine when each operation is done, the caller must use the **get_response** or **get_next_response** operations described below.

The degree of parallelism in the initiation and execution of the requests is system dependent. There are no guarantees about the order in which the requests are initiated. If INV_TERM_ON_ERR is specified, and the ORB detects an error initiating one of the requests, it will not initiate any further requests from this list. If INV_NO_RESPONSE is specified, it applies to all of the requests in the list.

The following invocation flags are currently defined for **send_multiple_requests**:

CORBA::INV_NO_RESPONSE indicates that at the invoker does not intend to wait for a response, nor does it expect any of the output arguments (inout and out) to be updated. This option may be specified even if the operation has not been defined to be **oneway**.

CORBA::INV_TERM_ON_ERR means that if one of the requests causes an error, the remaining requests are not sent.

4.3.3 get_response

```
Status get_response (                              // PIDL
    in Flags            response_flags          // response flags
```

get_response determines whether a request has completed. If **get_response** indicates that the operation is done, the out parameters and return values defined in the Request are valid, and they may be treated as if the Request **invoke** operation had been used to perform the request.

If the RESP_NO_WAIT flag is set, **get_response** returns immediately even if the request is still in progress. Otherwise, **get_response** waits until the request is done before returning.

The following response flags are defined for **get_response**:

CORBA::RESP_NO_WAIT indicates that the caller does not want to wait for a response.

4.3.4 get_next_response

```
/* C */

CORBA_Status CORBA_get_next_response (
   CORBA_Environment*env,
   CORBA_Flags    response_flags,
   CORBA_Request  *req
);



// C++
class ORB
{
   public:
      Boolean poll_next_response();
      Status get_next_response(RequestSeq*&);
};
```

The Smalltalk mapping of get_next_response is as follows:

```
pollNextResponse
getNextResponse
```

`get_next_response` returns the next request that completes. Despite the name, there is no guaranteed ordering among the completed requests, so the order in which they are returned from successive `get_next_response` calls is not necessarily related to the order in which they finished.

If the RESP_NO_WAIT flag is set, and there are no completed requests pending, then `get_next_response` returns immediately. Otherwise, `get_next_response` waits until some request finishes.

The following response flags are defined for `get_next_response`:

CORBA::RESP_NO_WAIT Indicates that the caller does not want to wait for a response.

4.4 List Operations

The list operations use the named-value structure defined above.

The list operations that create **NVList** objects are defined in the ORB interface described in Chapter 7, but are described in this section. The **NVList** interface is shown below.

```
interface NVList {                                                    // PIDL
    Status add_item (
        in Identifier   item_name,        // name of item
        in TypeCode     item_type,        // item datatype
        in void         *value,           // item value
        in long         value_len,        // length of item value
        in Flags        item_flags        // item flags
    );
    Status free ( );
    Status free_memory ( );
    Status get_count (
        out long        count             // number of entries in the list
    );
};
```

Interface NVList is defined in the CORBA module.

4.4.1 create_list

This operation, which creates a pseudo-object, is defined in the ORB interface and excerpted below.

```
Status create_list (                                                  //PIDL
    in long         count,      // number of items to allocate for list
    out NVList      new_list    // newly created list
        );
```

This operation allocates a list of the specified size, and clears it for initial use. List items may be added to the list using the **add_item** routine. Alternatively, they may be added by indexing directly into the list structure. A mixture of the two approaches for initializing a list, however, is not supported.

An **NVList** is a partially opaque structure. It may only be allocated via a call to **create_list**.

4.4.2 add_item

```
Status add_item (                                           // PIDL
    in Identifier    item_name,        // name of item
    in TypeCode      item_type,        // item datatype
    in void          *value,           // item value
    in long          value_len,        // length of item value
    in Flags         item_flags        // item flags
        );
```

This operation adds a new item to the indicated list. The item is added after the previously added item.

In addition to the argument modes defined in Section 4.1.1, **item_flags** may also take the following flag values: IN_COPY_VALUE, DEPENDENT_LIST. The argument passing flags defined in Section 4.1.1 may be used here to indicate the intended parameter passing mode of an argument.

If the IN_COPY_VALUE flag is set, a copy of the argument value is made and used instead.

If a list structure is added as an item (e.g. a “sublist”) the DEPENDENT_LIST flag may be specified to indicate that the sublist should be freed when the parent list is freed.

4.4.3 free

```
Status free ( );                                            // PIDL
```

This operation frees the list structure and any associated memory (an implicit call to the list **free_memory** operation is done).

4.4.4 free_memory

```
Status free_memory ( );                                     // PIDL
```

This operation frees any dynamically allocated out-arg memory associated with the list. The list structure itself is not freed.

4.4.5 *get_count*

```
Status get_count (                                                   // PIDL
    out long                  count          // number of entries in the list
            );
```

This operation returns the total number of items allocated for this list.

4.4.6 *create_operation_list*

This operation, which creates a pseudo-object, is defined in the ORB interface.

```
Status create_operation_list (                                       // PIDL
    in OperationDef         oper,              // operation
    out NVList              new_list           // argument definitions
            );
```

This operation returns an `NVList` initialized with the argument descriptions for a given operation. The information is returned in a form that may be used in *Dynamic Invocation* requests. The arguments are returned in the same order as they were defined for the operation.

The list **free** operation is used to free the returned information.

4.5 *Context Objects*

A context object contains a list of properties, each consisting of a name and a string value associated with that name. By convention, context properties represent information about the client, environment, or circumstances of a request that are inconvenient to pass as parameters.

Context properties can represent a portion of a client's or application's environment that is meant be propagated to (and made implicitly part of) a server's environment (for example, a window identifier, or user preference information). Once a server has been invoked (i.e., after the properties are propagated), the server may query its context object for these properties.

In addition, the context associated with a particular operation is passed as a distinguished parameter, allowing particular ORBs to take advantage of context properties, for example, using the values of certain properties to influence method binding behavior, server location, or activation policy.

An operation definition may contain a clause specifying those context properties that may be of interest to a particular operation. These context properties comprise the minimum set of properties that will be propagated to the server's environment (although a specified property may have no value associated with it). The ORB may choose to pass more properties than those specified in the operation declaration.

When a context clause is present on an operation declaration, an additional argument is added to the stub and skeleton interfaces. When an operation invocation occurs via either the stub or *Dynamic Invocation* interface, the ORB causes the properties which were named in the operation definition in IDL and which are present in the client's context object, to be provided in the context object parameter to the invoked method.

Context property names (which are strings) typically have the form of an OMG IDL identifier, or a series of OMG IDL identifiers separated by periods. A context property name pattern is either a property name, or a property name followed by a single "*". Property name patterns are used in the **context** clause of an operation definition, and in the **get_values** operation (described below).

A property name pattern without a trailing "*" is said to match only itself. A property name pattern of the form "<name>*" matches any property name that starts with <name> and continues with zero or more additional characters.

Context objects may be created and deleted, and individual context properties may be set and retrieved. There will often be context objects associated with particular processes, users, or other things depending on the operating system, and there may be conventions for having them supplied to calls by default.

It may be possible to keep context information in persistent implementations of context objects, while other implementations may be transient. The creation and modification of persistent context objects, however, is not addressed in this specification.

Context objects may be "chained" together to achieve a particular defaulting behavior.

Properties defined in a particular context object effectively override those properties in the next higher level. This searching behavior may be restricted by specifying the appropriate scope and the "restrict scope" option on the Context **get_values** call.

Context objects may be named for purposes of specifying a starting search scope.

4.6 Context Object Operations

When performing operations on a context object, properties are represented as named value lists. Each property value corresponds to a named value item in the list.

A property name is represented by a string of characters (see "Identifiers" on page 3-5 for the valid set of characters that are allowed). Property names are stored preserving their case, however names cannot differ simply by their case.

The Context interface is shown below.

```
module CORBA {

    interface Context {                                                                    // PIDL
        Status set_one_value (
            in Identifier          prop_name,          // property name to add
            in string              value               // property value to add
        );
        Status set_values (
            in NVList               values             // property values to be changed
        );
        Status get_values (
            in Identifier          start_scope,        // search scope
           in Flags               op_flags,           // operation flags
           in Identifier          prop_name,          // name of property(s) to retrieve
           out NVList             values              // requested property(s)
        );
        Status delete_values (
            in Identifier          prop_name           // name of property(s) to delete
        );
        Status create_child (
            in Identifier          ctx_name,           // name of context object
           out Context            child_ctx           // newly created context object
        );
        Status delete (
            in Flags               del_flags           // flags controlling deletion
        );
    };
};
```

4.6.1 *get_default_context*

This operation, which creates a Context pseudo-object, is defined in the ORB interface (see Section 7.1 for the complete ORB definition).

```
Status get_default_context (                                                               // PIDL
    out Context                ctx                     // context object
        );
```

This operation returns a reference to the default process context object. The default context object may be chained into other context objects. For example, an ORB implementation may chain the default context object into its User, Group, and System context objects.

4.6.2 set_one_value

```
Status set_one_value (                                              // PIDL
    in Identifier          prop_name,       // property name to add
    in string              value            // property value to add
        );
```

This operation sets a single context object property.

Currently, only string values are supported by the context object.

4.6.3 set_values

```
Status set_values (                                                 // PIDL
    in NVList          values             // property values to be changed
        );
```

This operation sets one or more property values in the context object. In the NVList, the flags field must be set to zero, and the TypeCode field associated with an attribute value must be TC_string.

Currently, only string values are supported by the context object.

4.6.4 get_values

```
Status get_values (                                                 // PIDL
    in Identifier        start_scope,        // search scope
    in Flags             op_flags,           // operation flags
    in Identifier        prop_name,          // name of property(s) to retrieve
    out NVList           values              // requested property(s)
        );
```

This operation retrieves the specified context property value(s). If **prop_name** has a trailing wildcard character (“*”), then all matching properties and their values are returned. The values returned may be freed by a call to the list **free** operation.

If no properties are found an error is returned, and no property list is returned.

Scope indicates the context object level at which to initiate the search for the specified properties (e.g. “_USER”, “_SYSTEM”). If the property is not found at the indicated level, the search continues up the context object tree until a match is found or all context objects in the chain have been exhausted.

Valid scope names are implementation-specific.

If scope name is omitted, the search begins with the specified context object. If the specified scope name is not found, an exception is returned.

The following operation flags may be specified:

CORBA::CTX_RESTRICT_SCOPE Searching is limited to the specified search scope or context object.

4.6.5 delete_values

```
Status delete_values (                                              // PIDL
    in Identifier       prop_name       // name of property(s) to delete
        );
```

This operation deletes the specified property value(s) values from the context object. If **prop_name** has a trailing wildcard character ("*"), then all property names that match will be deleted.

Search scope is always limited to the specified context object.

If no matching property is found, an exception is returned.

4.6.6 create_child

```
Status create_child (                                               // PIDL
    in Identifier       ctx_name,     // name of context object
    out Context         child_ctx     // newly created context object
);
```

This operation creates a child context object.

The returned context object is chained into its parent context. That is, searches on the child context object will look in the parent context (and so on, up the context tree), if necessary, for matching property names.

Context object names follow the rules for OMG IDL identifiers (see "Identifiers" on page 3-5).

4.6.7 delete

```
Status delete (                                                     // PIDL
    in Flags            del_flags             // flags controlling deletion
);
```

This operation deletes the indicated context object.

The following option flags may be specified:

CORBA::CTX_DELETE_DESCENDENTSDeletes the indicated context object and all of its descendent context objects, as well.

An exception is returned if there are one or more child context objects and the CTX_DELETE_DESCENDENTS flag was not set.

4.7 Native Data Manipulation

A future version of this specification will define routines to facilitate the conversion of data between the list layout found in **NVList** structures and the compiler native layout.

Dynamic Skeleton Interface 5

The Dynamic Skeleton interface (DSI) is a way to deliver requests from an ORB to an object implementation that does not have compile-time knowledge of the type of the object it is implementing. This contrasts with the type-specific, OMG IDL-based skeletons, but serves the same architectural role.

DSI is the server side's analogue to the client side's Dynamic Invocation Interface (DII). Just as the implementation of an object cannot distinguish whether its client is using type-specific stubs or the DII, the client who invokes an object cannot determine whether the implementation is using a type-specific skeleton or the DSI to connect the implementation to the ORB.

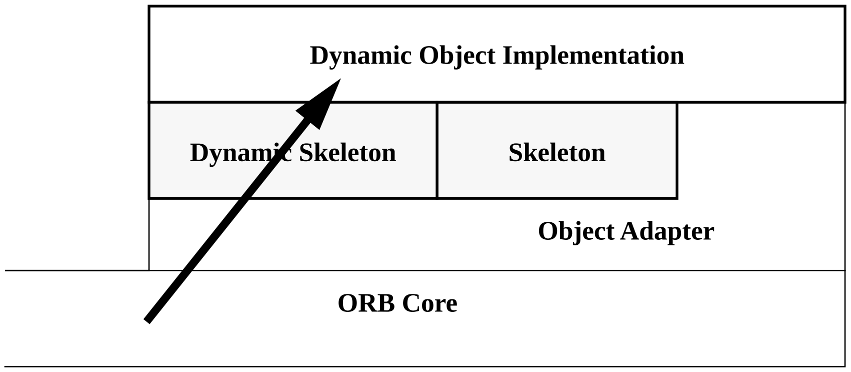


Figure 5-1 Requests are delivered through skeletons, including dynamic ones

DSI, like DII, has many applications beyond interoperability solutions. Uses include interactive software development tools based on interpreters, debuggers and monitors that want to dynamically interpose on objects, and support for dynamically-typed languages such as LISP.

5.1 Overview

The basic idea of the DSI is to implement all requests on a particular object by having the ORB invoke the same upcall routine, a Dynamic Implementation Routine (DIR). Since in any language binding all DIRs have the same signature, a single DIR could be used as the implementation for many objects, with different interfaces.

The DIR is passed all the explicit operation parameters, and an indication of the object that was invoked and the operation that was requested. The information is encoded in the request parameters. The DIR can use the invoked object, its object adapter, and the Interface Repository to learn more about the particular object and invocation. It can access and operate on individual parameters. It can make the same use of an object adapter as other object implementations.

The Dynamic Skeleton interface could be supported by any object adapter. Like type-specific skeletons, the DSI might have object adapter-specific details. This chapter describes a DSI interface for the Basic Object Adapter (BOA) and shows how it is mapped to C and C++.

5.2 Explicit Request State: ServerRequest Pseudo-Object

The **ServerRequest** pseudo-object captures the explicit state of a request for the DSI, analogous to the Request pseudo-object in the DII. The following shows how it provides access to the information:

```
module CORBA {
pseudo interface ServerRequest
{
    Identifier          op_name ();
    Context             ctx ();
    void                params (inout NVList params);
    Any                 result ();
};
}
```

The target object of the invocation is provided by the language binding for the DIR. In the context of a bridge, it will typically be a proxy for an object in some other ORB.

The **op_name** operation returns the name of the operation being invoked; according to OMG IDL's rules, these names must be unique among all operations supported by this object's "most-derived" interface. Note that the operation names for getting and setting attributes are **_get_<attribute_name>** and **_set_<attribute_name>**, respectively.

When the operation is not an attribute access, **ctx** will return the context information defined in OMG IDL for operation (if any). Otherwise, this context is empty.

Operation parameters will be retrieved with **params.** They appear in the NVList in the order in which they appear in the OMG IDL specification (left to right). This holds the “in”, “out” and “inout” values.

The **result** operation is used to find where to store any return value for the call. Reporting of exceptions (which preclude use of result and out/inout values in **params**) is a function of the language mapping.

See each language binding for a description of the memory management aspects of these parameters.

5.3 Dynamic Skeleton Interface: Language Mapping

Because DSI is defined in terms of a pseudo-object, special attention must be paid to it in the language mapping. This section provides general information about mapping the Dynamic Skeleton Interface to programming languages.

Section 14.24, “Mapping of the Dynamic Skeleton Interface to C,” on page 14-25 and Section 16.17, “Mapping of Dynamic Skeleton Interface to C++,” on page 16-43 provide mappings of the Dynamic Skeleton Interface (supporting the BOA) to the C language and C++ languages.

5.3.1 ServerRequest’s Handling of Operation Parameters

There is no requirement that a **ServerRequest** pseudo-object be usable as a general argument in OMG IDL operations, or listed in “orb.idl”.

The client side memory management rules normally applied to pseudo-objects do not strictly apply to a ServerRequest’s handling of operation parameters. Instead, the memory associated with parameters follows the memory management rules applied to data passed from skeletons into statically typed implementation routines, and vice versa.

In some language mappings, exceptions need special treatment. This is because the normal mapping for exceptions may require static knowledge of exception types. An example is the use of C++ exceptions, which require special run time typing information that can only be generated by a C++ compiler. Accordingly, the DSI and DII need an exception-reporting method that requires minimal compile-time support: the DIR needs to be able to provide the TypeCode for an exception as it reports the exception.

Finally, note that these APIs have been specified to support a performance model whereby the ORB doesn’t implicitly consult an interface repository (i.e. perform any remote object invocations, potentially slowing down a bridge) in order to handle an invocation. All the typing information is provided *to* the **ServerRequest** pseudo-object by an application. The ORB is allowed to verify that such information is correct, but such checking is not required.

5.3.2 Registering Dynamic Implementation Routines

Although it is not portably specified by previous CORBA specifications, any ORB and its BOA implementation must have some way of connecting type-specific skeletons to the methods that implement the operations. The Dynamic Skeleton interface uses the same mechanism.

A typical ORB/BOA implementation defines an operation, perhaps used when the object is activated, which specifies the methods to be used for a particular implementation class, for example, in C:

```
BOA_setimpl (BOA, ImplementationDef, MethodList,
skeleton);
```

The MethodList would be the DIR; the skeleton could be a Dynamic Skeleton, which would construct a ServerRequest object and invoke the DIR with it.

Whatever mechanism, whether at link time, run time, and so forth, is used to bind ordinary implementations to type-specific skeletons would also be used to bind dynamic implementations to dynamic skeletons. Such bindings could be maintained on a per-object, per-interface, per-class, or other basis.

The Interface Repository 6

The Interface Repository is the component of the ORB that provides persistent storage of interface definitions—it manages and provides access to a collection of object definitions specified in OMG IDL.

6.1 Overview

An ORB provides distributed access to a collection of objects using the objects' publicly defined interfaces specified in OMG IDL. The Interface Repository provides for the storage, distribution, and management of a collection of related objects' interface definitions.

For an ORB to correctly process requests, it must have access to the definitions of the objects it is handling. Object definitions can be made available to an ORB in one of two forms:

1. By incorporating the information procedurally into stub routines (e.g., as code that maps C language subroutines into communication protocols).
2. As objects accessed through the dynamically accessible Interface Repository (i.e., as "interface objects" accessed through OMG IDL-specified interfaces).

In particular, the ORB can use object definitions maintained in the Interface Repository to interpret and handle the values provided in a request:

- To provide type-checking of request signatures (whether the request was issued through the DII or through a stub).
- To assist in checking the correctness of interface inheritance graphs.
- To assist in providing interoperability between different ORB implementations.

As the interface to the object definitions maintained in an Interface Repository is public, the information maintained in the Repository can also be used by clients and services. For example, the Repository can be used:

- To manage the installation and distribution of interface definitions.

- To provide components of a CASE environment (for example, an interface browser).
- To provide interface information to language bindings (such as a compiler).
- To provide components of end-user environments (for example, a menu bar constructor).

The complete OMG IDL specification for the Interface Repository is in Section 6.8, “OMG IDL for Interface Repository,” on page 6-41. Fragments of the specification are used throughout this chapter as necessary.

6.2 Scope of an Interface Repository

Interface definitions are maintained in the Interface Repository as a set of objects that are accessible through a set of OMG IDL-specified interface definitions. An interface definition contains a description of the operations it supports, including the types of the parameters, exceptions it may raise, and context information it may use.

In addition, the interface repository stores constant values, which might be used in other interface definitions or might simply be defined for programmer convenience. And it stores typecodes, which are values that describe a type in structural terms.

The Interface Repository uses modules as a way to group interfaces and to navigate through those groups by name. Modules can contain constants, typedefs, exceptions, interface definitions, and other modules. Modules may, for example, correspond to the organization of OMG IDL definitions. They may also be used to represent organizations defined for administration or other purposes.

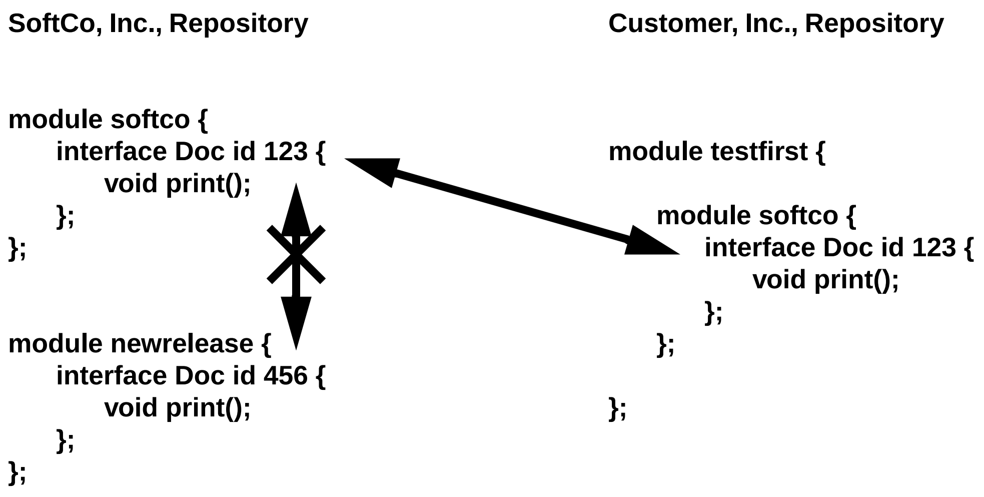
The Interface Repository is a set of objects that represent the information in it. There are operations that operate on this apparent object structure. It is an implementation’s choice whether these objects exist persistently or are created when referenced in an operation on the repository. There are also operations that extract information in an efficient form, obtaining a block of information that describes a whole interface or a whole operation.

An ORB may have access to multiple Interface Repositories. This may occur because two ORBs have different requirements for the implementation of the Interface Repository, because an object implementation (such as an OODB) prefers to provide its own type information, or because it is desired to have different additional information stored in different repositories. The use of typecodes and repository identifiers is intended to allow different repositories to keep their information consistent.

As shown in FIGURE 12. on page 6-3, the same interface **Doc** is installed in two different repositories, one at SoftCo, Inc., which sells Doc objects, and one at Customer, Inc., which buys Doc objects from SoftCo. SoftCo sets the repository id for the Doc interface when it defines it. Customer might first install the interface in its repository in a module where it could be tested before exposing it for general use. Because it has the same repository id, even though the Doc interface is stored in a different repository and is nested in a different module, it is known to be the same.

Meanwhile at SoftCo, someone working on a new Doc interface has given it a new repository id 456, which allows the ORBs to distinguish it from the current product Doc interface.

FIGURE 12. **Using Repository IDs to establish correspondence between repositories**



SoftCo, Inc., Repository

```
module softco {
    interface Doc id 123 {
        void print();
    };
};

module newrelease {
    interface Doc id 456 {
        void print();
    };
};
```

Customer, Inc., Repository

```
module testfirst {

    module softco {
        interface Doc id 123 {
            void print();
        };
    };

};
```

Not all interfaces will be visible in all repositories. For example, Customer employees cannot see the new release of the Doc interface. However, widely used interfaces will generally be visible in most repositories.

This Interface Repository specification defines operations for retrieving information from the repository as well as creating definitions within it. There may be additional ways to insert information into the repository (for example, compiling OMG IDL definitions, copying objects from one repository to another, etc.).

A critical use of the interface repository information is for connecting ORBs together. When an object is passed in a request from one ORB to another, it may be necessary to create a new object to represent the passed object in the receiving ORB. This may require locating the interface information in an interface repository in the receiving ORB. By getting the repository id from a repository in the sending ORB, it is possible to look up the interface in a repository in the receiving ORB. To succeed, the interface for that object must be installed in both repositories with the same repository id.

6.3 Implementation Dependencies

An implementation of an Interface Repository requires some form of persistent object store. Normally the kind of persistent object store used determines how interface definitions are distributed and/or replicated throughout a network domain. For example, if an Interface Repository is implemented using a filing system to provide object storage, there may be only a single copy of a set of interfaces maintained on a single machine. Alternatively, if an OODB is used to provide object storage, multiple copies of interface definitions may be maintained each of which is distributed across several machines to provide both high-availability and load-balancing.

The kind of object store used may determine the scope of interface definitions provided by an implementation of the Interface Repository. For example, it may determine whether each user has a local copy of a set of interfaces or if there is one copy per community of users. The object store may also determine whether or not all clients of an interface set see exactly the same set at any given point in time or whether latency in distributing copies of the set gives different users different views of the set at any point in time.

An implementation of the Interface Repository is also dependent on the security mechanism in use. The security mechanism (usually operating in conjunction with the object store) determines the nature and granularity of access controls available to constrain access to objects in the repository.

6.3.1 Managing Interface Repositories

Interface Repositories contain the information necessary to allow programs to determine and manipulate the type information at run-time. Programs may attempt to access the interface repository at any time by using the get_interface operation on the object reference. Once information has been installed in the repository, programs, stubs, and objects may depend on it. Updates to the repository must be done with care to avoid disrupting the environment. A variety of techniques are available to help do so.

A coherent repository is one whose contents can be expressed as a valid collection of OMG IDL definitions. For example, all inherited interfaces exist, there are no duplicate operation names or other name collisions, all parameters have known types, and so forth. As information is added to the repository, it is possible that it may pass through incoherent states. Media failures or communication errors might also cause it to appear incoherent. In general, such problems cannot be completely eliminated.

Replication is one technique to increase the availability and performance of a shared database. It is likely that the same interface information will be stored in multiple repositories in a computing environment. Using repository IDs, the repositories can establish the identity of the interfaces and other information across the repositories.

Multiple repositories might also be used to insulate production environments from development activity. Developers might be permitted to make arbitrary updates to their repositories, but administrators may control updates to widely used repositories. Some repository implementations might permit sharing of information, for example, several developers' repositories may refer to parts of a shared repository. Other repository implementations might instead copy the common information. In any case, the result should be a repository facility that creates the impression of a single, coherent repository.

The interface repository itself cannot make all repositories have coherent information, and it may be possible to enter information that does not make sense. The repository will report errors that it detects, e.g., defining two attributes with the same name, but might not report all errors, for example, adding an attribute to a base interface may or may not detect a name conflict with a derived interface. Despite these limitations, the expectation is that a combination of conventions, administrative controls, and tools that add information to the repository will work to create a coherent view of the repository information.

Transactions and concurrency control mechanisms defined by the Object Services may be used by some repositories when updating the repository. Those services are designed so that they can be used without changing the operations that update the repository. For example, a repository that supports the Transaction Service would inherit the Repository interface, which contains the update operations, as well as the Transaction interface, which contains the transaction management operations. (For more information about Object Services, including the Transaction and Concurrency Control Services, refer to *CORBAservices: Common Object Service Specifications*.)

Often, rather than change the information, new versions will be created, allowing the old version to continue to be valid. The new versions will have distinct repository IDs and be completely different types as far as the repository and the ORBs are concerned. The IR provides storage for version identifiers for named types, but does not specify any additional versioning mechanism or semantics.

6.4 Basics of the Interface Repository Interface

This section introduces some basic ideas that are important to understanding the Interface Repository. Topics addressed in this section are:

- Names and IDs
- Types and TypeCodes
- Interface Objects

6.4.1 Names and Identifiers

Simple names are not necessarily unique within an Interface Repository; they are always relative to an explicit or implicit module. In this context, interface definitions are considered explicit modules.

Scoped names uniquely identify modules, interfaces, constant, typedefs, exceptions, attributes, and operations in an Interface Repository.

Repository identifiers globally identify modules, interfaces, constants, typedefs, exceptions, attributes, and operations. They can be used to synchronize definitions across multiple ORBs and Repositories.

6.4.2 Types and TypeCodes

The Interface Repository stores information about types that are not interfaces in a data value called a TypeCode. From the TypeCode alone it is possible to determine the complete structure of a type. See “TypeCodes” on page 6-33 for more information on the internal structure of TypeCodes.

6.4.3 Interface Objects

Each interface managed in an Interface Repository is maintained as a collection of interface objects:

1. Repository: the top-level module for the repository name space; it contains constants, typedefs, exceptions, interface definitions, and modules.
2. ModuleDef: a logical grouping of interfaces; it contains constants, typedefs, exceptions, interface definitions, and other modules.
3. InterfaceDef: an interface definition; it contains lists of constants, types, exceptions, operations, and attributes.
4. AttributeDef: the definition of an attribute of the interface.
5. OperationDef: the definition of an operation on the interface; it contains lists of parameters and exceptions raised by this operation.
6. TypedefDef: base interface for definitions of named types that are not interfaces.
7. ConstantDef: the definition of a named constant.
8. ExceptionDef: the definition of an exception that can be raised by an operation.

The interface specifications for each interface object lists the attributes maintained by that object (see “Interface Repository Interfaces” on page 6-7). Many of these attributes correspond directly to OMG IDL statements. An implementation can choose to maintain additional attributes to facilitate managing the Repository or to record additional (proprietary) information about an interface. Implementations that extend the IR interfaces should do so by deriving new interfaces, not by modify the standard interfaces.

The *CORBA* specification defines a minimal set of operations for interface objects. Additional operations that an implementation of the Interface Repository may provide could include operations that provide for the versioning of interfaces and for the reverse compilation of specifications (i.e., the generation of a file containing an object’s OMG IDL specification).

6.4.4 Structure and Navigation of Interface Objects

The definitions in the Interface Repository are structured as a set of objects. The objects are structured the same way definitions are structured—some objects (definitions) “contain” other objects.

The containment relationships for the objects in the Interface Repository are shown in FIGURE 13. on page 6-7.

FIGURE 13. Interface Repository Object Containment

Repository	Each interface repository is represented by a global root repository object.
ConstantDef TypedefDef ExceptionDef InterfaceDef ModuleDef	The repository object represents the constants, typedefs, exceptions, interfaces and modules that are defined outside the scope of a module.
ConstantDef TypedefDef ExceptionDef ModuleDef InterfaceDef	The module object represents the constants, typedefs, exceptions, interfaces, and other modules defined within the scope of the module.
ConstantDef TypedefDef ExceptionDef AttributeDef OperationDef	An interface object represents constants, typedefs, exceptions, attributes, and operations defined within or inherited by the interface. Operation objects reference exception objects.

There are three ways to locate an interface in the Interface Repository:

1. By obtaining an **InterfaceDef** object directly from the ORB.
2. By navigating through the module name space using a sequence of names.
3. By locating the **InterfaceDef** object that corresponds to a particular repository identifier.

Obtaining an **InterfaceDef** object directly is useful when an object is encountered whose type was not known at compile time. By using the **get_interface()** operation on the object reference, it is possible to retrieve the Interface Repository information about the object. That information could then be used to perform operations on the object.

Navigating the module name space is useful when information about a particular named interface is desired. Starting at the root module of the repository, it is possible to obtain entries by name.

Locating the **InterfaceDef** object by ID is useful when looking for an entry in one repository that corresponds to another. A repository identifier must be globally unique. By using the same identifier in two repositories, it is possible to obtain the interface identifier for an interface in one repository, and then obtain information about that interface from another repository that may be closer or contain additional information about the interface.

6.5 Interface Repository Interfaces

Several abstract interfaces are used as base interfaces for other objects in the IR.

A common set of operations is used to locate objects within the Interface Repository. These operations are defined in the abstract interfaces **IRObject**, **Container**, and **Contained** described below. All IR objects inherit from the **IRObject** interface, which provides an operation for identifying the actual type of the object. Objects that are containers inherit navigation operations from the **Container** interface. Objects that are contained by other objects inherit navigation operations from the **Contained** interface.

The **IDLType** interface is inherited by all IR objects that represent IDL types, including interfaces, typedefs, and anonymous types. The **TypedefDef** interface is inherited by all named non-interface types.

The **IRObject**, **Contained**, **Container**, **IDLType**, and **TypedefDef** interfaces are not instantiable.

6.5.1 Supporting Type Definitions

Several types are used throughout the IR interface definitions.

```
module CORBA {
    typedef string              Identifier;
    typedef string              ScopedName;
    typedef string              RepositoryId;

    enum DefinitionKind {
        dk_none, dk_all,
        dk_Attribute, dk_Constant, dk_Exception, dk_Interface,
        dk_Module, dk_Operation, dk_Typedef,
        dk_Alias, dk_Struct, dk_Union, dk_Enum,
        dk_Primitive, dk_String, dk_Sequence, dk_Array,
        dk_Repository
    };
};
```

Identifiers are the simple names that identify modules, interfaces, constants, typedefs, exceptions, attributes, and operations. They correspond exactly to OMG IDL identifiers. An **Identifier** is not necessarily unique within an entire Interface Repository; it is unique only within a particular **Repository, ModuleDef**, **InterfaceDef**, or **Operation-Def**.

A **ScopedName** is a name made up of one or more **Identifier**s separated by the characters "::". They correspond to OMG IDL scoped names.

An *absolute* **ScopedName** is one that begins with "::" and unambiguously identifies a definition in a **Repository**. An *absolute* **ScopedName** in a **Repository** corresponds to a *global name* in an OMG IDL file (see Section 3.12). A *relative* **Scoped-Name** does not begin with "::" and must be resolved relative to some context.

A **RepositoryId** is an identifier used to uniquely and globally identify a module, interface, constant, typedef, exception, attribute or operation. As **RepositoryId**s are defined as strings, they can be manipulated (e.g., copied and compared) using a language binding's string manipulation routines.

A **DefinitionKind** identifies the type of an IR object.

6.5.2 IRObject

The **IRObject** interface represents the most generic interface from which all other Interface Repository interfaces are derived, even the Repository itself.

```
module CORBA {
    interface IRObject {
        // read interface
        readonly    attribute DefinitionKind    def_kind;

        // write interface
        void destroy ();
    };
};
```

Read Interface

The **def_kind** attribute identifies the type of the definition.

Write Interface

The **destroy** operation causes the object to cease to exist. If the object is a **Container**, **destroy** is applied to all its contents. If the object contains an **IDLType** attribute for an anonymous type, that **IDLType** is destroyed. If the object is currently contained in some other object, it is removed. Invoking **destroy** on a **Repository** or on a **Primitive-Def** is an error. Implementations may very in their handling of references to an object the is being destroyed, but the Repository should not be left in an incoherent state.

6.5.3 Contained

The **Contained** interface is inherited by all Interface Repository interfaces that are contained by other IR objects. All objects within the Interface Repository, except the root object (**Repository**) and definitions of anonymous (**ArrayDef**, **StringDef**, and **SequenceDef**), and primitive types are contained by other objects.

```
module CORBA {
    typedef string VersionSpec;

    interface Contained : IRObject {
        // read/write interface

                    attribute RepositoryId          id;
                    attribute Identifier            name;
                    attribute VersionSpec           version;

        // read interface

        readonly    attribute Container             defined_in;
        readonly    attribute ScopedName            absolute_name;
        readonly    attribute Repository            containing_repository;

        struct Description {
            DefinitionKind      kind;
            any                 value;
        };

        Description describe ();

        // write interface

        void move (
            in Container        new_container,
            in Identifier       new_name,
            in VersionSpec      new_version
        );
    };
};
```

Read Interface

An object that is contained by another object has an **id** attribute that identifies it globally, and a **name** attribute that identifies it uniquely within the enclosing **Container** object. It also has a **version** attribute that distinguishes it from other versioned objects with the same **name**. IRs are not required to support simultaneous containment of multiple versions of the same named object. Supporting multiple versions most likely requires mechanism and policy not specified in this document.

Contained objects also have a **defined_in** attribute that identifies the **Container** within which they are defined. Objects can be contained either because they are defined within the containing object (for example, an interface is defined within a module) or because they are inherited by the containing object (for example, an operation may be contained by an interface because the interface inherits the operation from another interface). If an object is contained through inheritance, the **defined_in** attribute identifies the **InterfaceDef** from which the object is inherited.

The **absolute_name** attribute is an absolute **ScopedName** that identifies a **Contained** object uniquely within its enclosing **Repository**. If this object's **defined_in** attribute references a **Repository**, the **absolute_name** is formed by concatenating the string "::" and this object's **name** attribute. Otherwise, the **absolute_name** is formed by concatenating the **absolute_name** attribute of the object referenced by this object's **defined_in** attribute, the string "::", and this object's **name** attribute.

The **containing_repository** attribute identifies the **Repository** that is eventually reached by recursively following the object's **defined_in** attribute.

The **describe** operation returns a structure containing information about the interface. The description structure associated with each interface is provided below with the interface's definition. The kind of definition described by the structure returned is provided with the returned structure. For example, if the **describe** operation is invoked on an attribute object, the **kind** field contains **dk_Attribute** and the **value** field contains an **any**, which contains the **AttributeDescription** structure.

Write Interface

Setting the **id** attribute changes the global identity of this definition. An error is returned if an object with the specified **id** attribute already exists within this object's **Repository**.

Setting the **name** attribute changes the identity of this definition within its **Container**. An error is returned if an object with the specified **name** attribute already exists within the this object's **Container**. The **absolute_name** attribute is also updated, along with any other attributes that reflect the name of the object. If this object is a **Container**, the **absolute_name** attribute of any objects it contains are also updated.

The **move** operation atomically removes this object from its current **Container**, and adds it to the **Container** specified by **new_container**, which must:

- Be in the same **Repository,**
- Be capable of containing this object's type (see FIGURE 13. on page 6-7); and
- Not already contain an object with this object's name (unless multiple versions are supported by the IR).

The **name** attribute is changed to **new_name**, and the **version** attribute is changed to **new_version**.

The **defined_in** and **absolute_name** attributes are updated to reflect the new container and **name**. If this object is also a **Container**, the **absolute_name** attributes of any objects it contains are also updated.

6.5.4 Container

The **Container** interface is used to form a containment hierarchy in the Interface Repository. A **Container** can contain any number of objects derived from the **Contained** interface. All **Container**s, except for **Repository**, are also derived from **Contained**.

```
module CORBA {
    typedef sequence <Contained> ContainedSeq;

    interface Container : IRObject {
        // read interface

        Contained lookup (in ScopedName search_name);

        ContainedSeq contents (
            in DefinitionKind        limit_type,
            in boolean               exclude_inherited
        );

        ContainedSeq lookup_name (
            in Identifier            search_name,
            in long                  levels_to_search,
            in DefinitionKind        limit_type,
            in boolean               exclude_inherited
        );

        struct Description {
            Contained            contained_object;
            DefinitionKind       kind;
            any                  value;
        };

        typedef sequence<Description> DescriptionSeq;

        DescriptionSeq describe_contents (
            in DefinitionKind        limit_type,
            in boolean               exclude_inherited,
            in long                  max_returned_objs
        );

        // write interface

        ModuleDef create_module (
            in RepositoryId          id,
            in Identifier            name,
            in VersionSpec           version
        );

        ConstantDef create_constant (
            in RepositoryId          id,
            in Identifier            name,
            in VersionSpec           version,
            in IDLType               type,
            in any                   value
        );

        StructDef create_struct (
            in RepositoryId       id,
            in Identifier         name,
            in VersionSpec        version,
            in StructMemberSeq    members
        );

        UnionDef create_union (
            in RepositoryId       id,
            in Identifier         name,
            in VersionSpec        version,
            in IDLType            discriminator_type,
            in UnionMemberSeq     members
        );

        EnumDef create_enum (
            in RepositoryId       id,
            in Identifier         name,
            in VersionSpec        version,
            in EnumMemberSeq      members
        );

        AliasDef create_alias (
            in RepositoryId       id,
            in Identifier         name,
            in VersionSpec        version,
            in IDLType            original_type
        );

        InterfaceDef create_interface (
            in RepositoryId       id,
            in Identifier         name,
            in VersionSpec        version,
            in InterfaceDefSeq    base_interfaces
        );
    };
};
```

Read Interface

The **lookup** operation locates a definition relative to this container given a scoped name using OMG IDL’s name scoping rules. An absolute scoped name (beginning with “::”) locates the definition relative to the enclosing **Repository**. If no object is found, a nil object reference is returned.

The **contents** operation returns the list of objects directly contained by or inherited into the object. The operation is used to navigate through the hierarchy of objects. Starting with the Repository object, a client uses this operation to list all of the objects contained by the Repository, all of the objects contained by the modules within the Repository, and then all of the interfaces within a specific module, and so on.

limit_type	If **limit_type** is set to **dk_all**, objects of all interface types are returned. For example, if this is an **InterfaceDef**, the attribute, operation, and exception objects are all returned. If **limit_type** is set to a specific interface, only objects of that interface type are returned. For example, only attribute objects are returned if **limit_type** is set to **dk_Attribute**.
exclude_inherited	If set to TRUE, inherited objects (if there are any) are not returned. If set to FALSE, all contained objects—whether contained due to inheritance or because they were defined within the object—are returned.

The **lookup_name** operation is used to locate an object by name within a particular object or within the objects contained by that object.

search_name	Specifies which name is to be searched for.
levels_to_search	Controls whether the lookup is constrained to the object the operation is invoked on or whether it should search through objects contained by the object as well.

Setting **levels_to_search** to -1 searches the current object and all contained objects. Setting **levels_to_search** to 1 searches only the current object.

limit_type	If **limit_type** is set to **dk_all**, objects of all interface types are returned (e.g., attributes, operations, and exceptions are all returned). If **limit_type** is set to a specific interface, only objects of that interface type are returned. For example, only attribute objects are returned if **limit_type** is set to **dk_Attribute**.
exclude_inherited	If set to TRUE, inherited objects (if there are any) are not returned. If set to FALSE, all contained objects (whether contained due to inheritance or because they were defined within the object) are returned.

The **describe_contents** operation combines the **contents** operation and the **describe** operation. For each object returned by the **contents** operation, the description of the object is returned (i.e., the object's **describe** operation is invoked and the results returned).

max_returned_objs	Limits the number of objects that can be returned in an invocation of the call to the number provided. Setting the parameter to -1 means return all contained objects.

Write Interface

The **Container** interface provides operations to create **ModuleDef**s, **Constant-Def**s, **StructDef**s, **UnionDef**s, **EnumDef**s, **AliasDef**s, and **InterfaceDef**s as contained objects. The **defined_in** attribute of a definition created with any of these operations is initialized to identify the **Container** on which the operation is invoked, and the **containing_repository** attribute is initialized to its **Repository**.

The **create_<type>** operations all take **id** and **name** parameters which are used to initialize the identity of the created definition. An error is returned if an object with the specified **id** already exists within this object's **Repository**, or, assuming multiple versions are not supported, if an object with the specified **name** already exists within this **Container**.

The **create_module** operation returns a new empty **ModuleDef**. Definitions can be added using **Container::create_<type>** operations on the new module, or by using the **Contained::move** operation.

The **create_constant** operation returns a new **ConstantDef** with the specified **type** and **value**.

The **create_struct** operation returns a new **StructDef** with the specified **members**. The **type** member of the **StructMember** structures is ignored, and should be set to **TC_void**. See "StructDef" on page 6-19 for more information.

The **create_union** operation returns a new **UnionDef** with the specified **discriminator_type** and **members**. The **type** member of the **UnionMember** structures is ignored, and should be set to **TC_void**. See "UnionDef" on page 6-19 for more information.

The **create_enum** operation returns a new **EnumDef** with the specified **members**. See "EnumDef" on page 6-20 for more information.

The **create_alias** operation returns a new **AliasDef** with the specified **original_type**.

The **create_interface** operation returns a new empty **InterfaceDef** with the specified **base_interfaces**. Type, exception, and constant definitions can be added using **Container::create_<type>** operations on the new **InterfaceDef**. **OperationDefs** can be added using **InterfaceDef::create_operation** and **AttributeDefs** can be added using **Interface::create_attribute**. Definitions can also be added using the **Contained::move** operation.

6.5.5 IDLType

The **IDLType** interface is an abstract interface inherited by all IR objects that represent OMG IDL types. It provides access to the **TypeCode** describing the type, and is used in defining other interfaces wherever definitions of IDL types must be referenced.

```
module CORBA {
    interface IDLType : IRObject {
        readonly    attribute TypeCode      type;
    };
};
```

The **type** attribute describes the type defined by an object derived from **IDLType**.

6.5.6 Repository

Repository is an interface that provides global access to the Interface Repository. The **Repository** object can contain constants, typedefs, exceptions, interfaces, and modules. As it inherits from **Container**, it can be used to look up any definition (whether globally defined or defined within a module or interface) either by **name** or by **id**.

There may be more than one Interface Repository in a particular ORB environment (although some ORBs might require that definitions they use be registered with a particular repository). Each ORB environment will provide a means for obtaining object references to the Repositories available within the environment.

```
module CORBA {
    interface Repository : Container {
        // read interface

        Contained lookup_id (in RepositoryId search_id);

        PrimitiveDef get_primitive (in PrimitiveKind kind);

        // write interface

        StringDef create_string (in unsigned long bound);

        SequenceDef create_sequence (
                in unsigned long    bound,
                in IDLType          element_type
        );

        ArrayDef create_array (
                in unsigned long    length,
                in IDLType          element_type
        );
    };
};
```

Read Interface

The **lookup_id** operation is used to lookup an object in a **Repository** given its **RepositoryId**. If the **Repository** does not contain a definition for **search_id**, a nil object reference is returned.

The **get_primitive** operation returns a reference to a **PrimitiveDef** with the specified kind attribute. All **PrimitiveDef**s are immutable and owned by the **Repository**.

Write Interface

The three **create_<type>** operations create new objects defining anonymous types. As these interfaces are not derived from **Contained**, it is the caller's responsibility to invoke **destroy** on the returned object if it is not successfully used in creating a definition that is derived from **Contained**. Each anonymous type definition must be used in defining exactly one other object.

The **create_string** operation returns a new **StringDef** with the specified **bound**, which must be non-zero. The **get_primitive** operation is used for unbounded strings.

The **create_sequence** operation returns a new **SequenceDef** with the specified **bound** and **element_type**.

The **create_array** operation returns a new **ArrayDef** with the specified **length** and **element_type**.

6.5.7 ModuleDef

A **ModuleDef** can contain constants, typedefs, exceptions, interfaces and other module objects.

```
module CORBA {
    interface ModuleDef : Container, Contained {
    };

    struct ModuleDescription {
        Identifier        name;
        RepositoryId      id;
        RepositoryId      defined_in;
        VersionSpec       version;
    };
};
```

The inherited **describe** operation for a **ModuleDef** object returns a **ModuleDescription**.

6.5.8 ConstantDef Interface

A **ConstantDef** object defines a named constant.

```
module CORBA {
    interface ConstantDef : Contained {
        readonly    attribute TypeCode      type;
                    attribute IDLType       type_def;
                    attribute any           value;
    };

    struct ConstantDescription {
        Identifier          name;
        RepositoryId        id;
        RepositoryId        defined_in;
        VersionSpec         version;
        TypeCode            type;
        any                 value;
    };
};
```

Read Interface

The **type** attribute specifies the **TypeCode** describing the type of the constant. The type of a constant must be one of the simple types (long, short, float, char, string, octet, etc.). The **type_def** attribute identifies the definition of the type of the constant.

The **value** attribute contains the value of the constant, not the computation of the value (e.g., the fact that it was defined as “1+2”).

The **describe** operation for a **ConstantDef** object returns a **ConstantDescription**.

Write Interface

Setting the **type_def** attribute also updates the **type** attribute.

When setting the **value** attribute, the **TypeCode** of the supplied any must be equal to the **type** attribute of the **ConstantDef**.

6.5.9 TypedefDef Interface

TypedefDef is an abstract interface used as a base interface for all named non-object types (structures, unions, enumerations, and aliases). The **TypedefDef** interface is not inherited by the definition objects for primitive or anonymous types.

```
module CORBA {
    interface TypedefDef : Contained, IDLType {
    };

    struct TypeDescription {
        Identifier          name;
        RepositoryId        id;
        RepositoryId        defined_in;
        VersionSpec         version;
        TypeCode            type;
    };
};
```

The inherited **describe** operation for interfaces derived from **TypedefDef** returns a **TypeDescription**.

6.5.10 StructDef

A **StructDef** represents an OMG IDL structure definition.

```
module CORBA {
    struct StructMember {
        Identifier      name;
        TypeCode        type;
        IDLType         type_def;
    };
    typedef sequence <StructMember> StructMemberSeq;

    interface StructDef : TypedefDef {
        attribute StructMemberSeq        members;
    };
};
```

Read Interface

The **members** attribute contains a description of each structure member.

The inherited **type** attribute is a **tk_struct TypeCode** describing the structure.

Write Interface

Setting the **members** attribute also updates the **type** attribute. When setting the **members** attribute, the **type** member of the **StructMember** structure is ignored and should be set to **TC_void**.

6.5.11 UnionDef

A **UnionDef** represents an OMG IDL union definition.

```
module CORBA {
    struct UnionMember {
        Identifier        name;
        any               label;
        TypeCode          type;
        IDLType           type_def;
    };
    typedef sequence <UnionMember> UnionMemberSeq;

    interface UnionDef : TypedefDef {
        readonly    attribute TypeCode              discriminator_type;
                    attribute IDLType               discriminator_type_def;
                    attribute UnionMemberSeq        members;
    };
};
```

Read Interface

The **discriminator_type** and **discriminator_type_def** attributes describe and identify the union’s discriminator type.

The **members** attribute contains a description of each union member. The **label** of each **UnionMemberDescription** is a distinct value of the **discriminator_type**. Adjacent members can have the same **name**. Members with the same **name** must also have the same **type**. A **label** with type **octet** and value 0 indicates the default union member.

The inherited **type** attribute is a **tk_union TypeCode** describing the union.

Write Interface

Setting the **discriminator_type_def** attribute also updates the **discriminator_type** attribute and setting the **discriminator_type_def** or **members** attribute also updates the **type** attribute.

When setting the **members** attribute, the **type** member of the **UnionMember** structure is ignored and should be set to **TC_void**.

6.5.12 EnumDef

An **EnumDef** represents an OMG IDL enumeration definition.

```
module CORBA {
    typedef sequence <Identifier> EnumMemberSeq;

    interface EnumDef : TypedefDef {
        attribute EnumMemberSeq     members;
    };
};
```

Read Interface

The **members** attribute contains a distinct name for each possible value of the enumeration.

The inherited **type** attribute is a **tk_enum TypeCode** describing the enumeration.

Write Interface

Setting the **members** attribute also updates the **type** attribute.

6.5.13 AliasDef

An **AliasDef** represents an OMG IDL typedef that aliases another definition.

```
module CORBA {
    interface AliasDef : TypedefDef {
        attribute IDLType    original_type_def;
    };
};
```

6.5.14 Read Interface

The **original_type_def** attribute identifies the type being aliased.

The inherited **type** attribute is a **tk_alias TypeCode** describing the alias.

Write Interface

Setting the **original_type_def** attribute also updates the **type** attribute.

6.5.15 PrimitiveDef

A **PrimitiveDef** represents one of the IDL primitive types. As primitive types are unnamed, this interface is not derived from **TypedefDef** or **Contained**.

```
module CORBA {
    enum PrimitiveKind {
        pk_null, pk_void, pk_short, pk_long, pk_ushort, pk_ulong,
        pk_float, pk_double, pk_boolean, pk_char, pk_octet,
        pk_any, pk_TypeCode, pk_Principal, pk_string, pk_objref
    };

    interface PrimitiveDef: IDLType {
        readonly attribute PrimitiveKind    kind;
    };
};
```

The **kind** attribute indicates which primitive type the **PrimitiveDef** represents. There are no **PrimitiveDef**s with kind **pk_null**. A **PrimitiveDef** with kind **pk_string** represents an unbounded string. A **PrimitiveDef** with kind **pk_objref** represents the IDL type **Object**.

The inherited **type** attribute describes the primitive type.

All **PrimitiveDef**s are owned by the Repository. References to them are obtained using **Repository::get_primitive**.

6.5.16 StringDef

A **StringDef** represents an IDL bounded string type. The unbounded string type is represented as a **PrimitiveDef**. As string types are anonymous, this interface is not derived from **TypedefDef** or **Contained**.

```
module CORBA {
    interface StringDef : IDLType {
        attribute unsigned long    bound;
    };
};
```

The **bound** attribute specifies the maximum number of characters in the string, and must not be zero.

The inherited **type** attribute is a **tk_string TypeCode** describing the string.

6.5.17 SequenceDef

A **SequenceDef** represents an IDL sequence type. As sequence types are anonymous, this interface is not derived from **TypedefDef** or **Contained**.

```
module CORBA {
    interface SequenceDef : IDLType {
                    attribute unsigned long    bound;
        readonly    attribute TypeCode         element_type;
                    attribute IDLType          element_type_def;
    };
};
```

Read Interface

The **bound** attribute specifies the maximum number of elements in the sequence. A **bound** of zero indicates an unbounded sequence.

The type of the elements is described by **element_type** and identified by **element_type_def**.

The inherited **type** attribute is a **tk_sequence TypeCode** describing the sequence.

Write Interface

Setting the **element_type_def** attribute also updates the **element_type** attribute.

Setting the **bound** or **element_type_def** attribute also updates the **type** attribute.

6.5.18 ArrayDef

An **ArrayDef** represents an IDL array type. As array types are anonymous, this interface is not derived from **TypedefDef** or **Contained**.

```
module CORBA {
    interface ArrayDef : IDLType {
                        attribute unsigned long     length;
        readonly        attribute TypeCode          element_type;
                        attribute IDLType           element_type_def;
    };
};
```

Read Interface

The **length** attribute specifies the number of elements in the array.

The type of the elements is described by **element_type** and identified by **element_type_def**. Since an **ArrayDef** only represents a single dimension of an array, multi-dimensional IDL arrays are represented by multiple **ArrayDef** objects, one per array dimension. The **element_type_def** attribute of the **ArrayDef** representing the leftmost index of the array, as defined in IDL, will refer to the **ArrayDef** representing the next index to the right, and so on. The innermost **ArrayDef** represents the rightmost index and the element type of the multi-dimensional OMG IDL array.

The inherited **type** attribute is a **tk_array TypeCode** describing the array.

Write Interface

Setting the **element_type_def** attribute also updates the **element_type** attribute.

Setting the **bound** or **element_type_def** attribute also updates the **type** attribute.

6.5.19 ExceptionDef

An **ExceptionDef** represents an exception definition.

```
module CORBA {
    interface ExceptionDef : Contained {
        readonly    attribute TypeCode            type;
                    attribute StructMemberSeq     members;
    };

    struct ExceptionDescription {
        Identifier      name;
        RepositoryId    id;
        RepositoryId    defined_in;
        VersionSpec     version;
        TypeCode        type;
    };
};
```

Read Interface

The **type** attribute is a **tk_except TypeCode** describing the exception.

The members **attribute** describes any exception members.

The **describe** operation for a **ExceptionDef** object returns an **ExceptionDescription**.

Write Interface

Setting the **members** attribute also updates the **type** attribute. When setting the **members** attribute, the **type** member of the **StructMember** structure is ignored and should be set to **TC_void**.

6.5.20 AttributeDef

An **AttributeDef** represents the information that defines an attribute of an interface.

```
module CORBA {
    enum AttributeMode {ATTR_NORMAL, ATTR_READONLY};

    interface AttributeDef : Contained {
        readonly    attribute TypeCode          type;
                    attribute IDLType           type_def;
                    attribute AttributeMode     mode;
    };

    struct AttributeDescription {
        Identifier          name;
        RepositoryId        id;
        RepositoryId        defined_in;
        VersionSpec         version;
        TypeCode            type;
        AttributeMode       mode;
    };
};
```

Read Interface

The **type** attribute provides the **TypeCode** describing the type of this attribute. The **type_def** attribute identifies the object defining the type of this attribute.

The **mode** attribute specifies read only or read/write access for this attribute.

Write Interface

Setting the **type_def** attribute also updates the **type** attribute.

6.5.21 OperationDef

An **OperationDef** represents the information needed to define an operation of an interface.

```
module CORBA {
    enum OperationMode {OP_NORMAL, OP_ONEWAY};

    enum ParameterMode {PARAM_IN, PARAM_OUT, PARAM_INOUT};
    struct ParameterDescription {
        Identifier          name;
        TypeCode            type;
        IDLType             type_def;
        ParameterMode       mode;
    };
    typedef sequence <ParameterDescription> ParDescriptionSeq;

    typedef Identifier ContextIdentifier;
    typedef sequence <ContextIdentifier> ContextIdSeq;

    typedef sequence <ExceptionDef> ExceptionDefSeq;
    typedef sequence <ExceptionDescription> ExcDescriptionSeq;

    interface OperationDef : Contained {
        readonly    attribute TypeCode              result;
                    attribute IDLType               result_def;
                    attribute ParDescriptionSeq     params;
                    attribute OperationMode         mode;
                    attribute ContextIdSeq          contexts;
                    attribute ExceptionDefSeq       exceptions;
    };

    struct OperationDescription {
        Identifier              name;
        RepositoryId            id;
        RepositoryId            defined_in;
        VersionSpec             version;
        TypeCode                result;
        OperationMode           mode;
        ContextIdSeq            contexts;
        ParDescriptionSeq       parameters;
        ExcDescriptionSeq       exceptions;
    };
};
```

Read Interface

The **result** attribute is a **TypeCode** describing the type of the value returned by the operation. The **result_def** attribute identifies the definition of the returned type.

The **params** attribute describes the parameters of the operation. It is a sequence of **ParameterDescription** structures. The order of the **ParameterDescription**s in the sequence is significant. The **name** member of each structure provides the parameter name. The **type** member is a **TypeCode** describing the type of the parameter. The

type_def member identifies the definition of the type of the parameter. The **mode** member indicates whether the parameter is an in, out, or inout parameter.

The operation's **mode** is either oneway (i.e., no output is returned) or normal.

The **contexts** attribute specifies the list of context identifiers that apply to the operation.

The **exceptions** attribute specifies the list of exception types that can be raised by the operation.

The inherited **describe** operation for an **OperationDef** object returns an **OperationDescription**.

The inherited **describe_contents** operation provides a complete description of this operation, including a description of each parameter defined for this operation.

Write Interface

Setting the **result_def** attribute also updates the **result** attribute.

The **mode** attribute can only be set to **OP_ONEWAY** if the result is **TC_void** and all elements of **params** have a **mode** of **PARAM_IN**.

6.5.22 InterfaceDef

An **InterfaceDef** object represents an interface definition. It can contain constants, typedefs, exceptions, operations, and attributes.

```
module CORBA {
    interface InterfaceDef;
    typedef sequence <InterfaceDef> InterfaceDefSeq;
    typedef sequence <RepositoryId> RepositoryIdSeq;
    typedef sequence <OperationDescription> OpDescriptionSeq;
    typedef sequence <AttributeDescription> AttrDescriptionSeq;

    interface InterfaceDef : Container, Contained, IDLType {
        // read/write interface

        attribute InterfaceDefSeq       base_interfaces;

        // read interface

        boolean is_a (in RepositoryId interface_id);

        struct FullInterfaceDescription {
            Identifier                  name;
            RepositoryId                id;
            RepositoryId                defined_in;
            VersionSpec                 version;
            OpDescriptionSeq            operations;
            AttrDescriptionSeq          attributes;
            RepositoryIdSeq             base_interfaces;
            TypeCode                    type;
        };

        FullInterfaceDescription describe_interface();

        // write interface

        AttributeDef create_attribute (
                in RepositoryId         id,
                in Identifier           name,
                in VersionSpec          version,
                in IDLType              type,
                in AttributeMode        mode
        );

        OperationDef create_operation (
                in RepositoryId         id,
                in Identifier           name,
                in VersionSpec          version,
                in IDLType              result,
                in OperationMode        mode,
                in ParDescriptionSeq    params,
                in ExceptionDefSeq      exceptions,
                in ContextIdSeq         contexts
        );
    };
```

```
        struct InterfaceDescription {
            Identifier          name;
            RepositoryId        id;
            RepositoryId        defined_in;
            VersionSpec         version;
            RepositoryIdSeq     base_interfaces;
        };
    };
```

Read Interface

The **base_interfaces** attribute lists all the interfaces from which this interface inherits. The **is_a** operation returns TRUE if the interface on which it is invoked either is identical to or inherits, directly or indirectly, from the interface identified by its **interface_id** parameter. Otherwise it returns FALSE.

The **describe_interface** operation returns a **FullInterfaceDescription** describing the interface, including its operations and attributes.

The inherited **describe** operation for an **InterfaceDef** returns an **InterfaceDescription**.

The inherited **contents** operation returns the list of constants, typedefs, and exceptions defined in this InterfaceDef and the list of attributes and operations either defined or inherited in this InterfaceDef. If the **exclude_inherited** parameter is set to TRUE, only attributes and operations defined within this interface are returned. If the **exclude_inherited** parameter is set to FALSE, all attributes and operations are returned.

Write Interface

Setting the **base_interfaces** attribute returns an error if the **name** attribute of any object contained by this **InterfaceDef** conflicts with the **name** attribute of any object contained by any of the specified base **InterfaceDef**s.

The **create_attribute** operation returns a new **AttributeDef** contained in the **InterfaceDef** on which it is invoked. The **id**, **name**, **version, type_def**, and **mode** attributes are set as specified. The **type** attribute is also set. The **defined_in** attribute is initialized to identify the containing **InterfaceDef**. An error is returned if an object with the specified **id** already exists within this object's **Repository**, or if an object with the specified **name** already exists within this **InterfaceDef**.

The **create_operation** operation returns a new **OperationDef** contained in the **InterfaceDef** on which it is invoked. The **id**, **name**, **version**, **result_def**, **mode**, **params**, **exceptions**, and **contexts** attributes are set as specified. The **result** attribute is also set. The **defined_in** attribute is initialized to identify the containing **InterfaceDef**. An error is returned if an object with the specified **id** already exists within this object's **Repository**, or if an object with the specified **name** already exists within this **InterfaceDef**.

6.6 RepositoryIds

RepositoryIds are values that can be used to establish the identity of information in the repository. A **RepositoryId** is represented as a string, allowing programs to store, copy, and compare them without regard to the structure of the value. It does not matter what format is used for any particular **RepositoryId**. However, conventions are used to manage the name space created by these IDs.

RepositoryIds may be associated with OMG IDL definitions in a variety of ways. Installation tools might generate them, they might be defined with pragmas in OMG IDL source, or they might be supplied with the package to be installed.

The format of the id is a short format name followed by a colon (":") followed by characters according to the format. This specification defines three formats: one derived from OMG IDL names, one that uses DCE UUIDs, and another intended for short-term use, such as in a development environment.

6.6.1 OMG IDL Format

The OMG IDL format for **RepositoryIds** primarily uses OMG IDL scoped names to distinguish between definitions. It also includes an optional unique prefix, and major and minor version numbers.

The **RepositoryId** consist of three components, separated by colons, (":")

The first component is the format name, "IDL".

The second component is a list of identifiers, separated by "/" characters. These identifiers are arbitrarily long sequences of alphabetic, digit, underscore ("_"), hyphen ("-"), and period (".") characters. Typically, the first identifier is a unique prefix, and the rest are the OMG IDL Identifiers that make up the scoped name of the definition.

The third component is made up of major and minor version numbers, in decimal format, separated by a ".". When two interfaces have **RepositoryId**s differing only in minor version number it can be assumed that the definition with the higher version number is upwardly compatible with (i.e. can be treated as derived from) the one with the lower minor version number.

6.6.2 DCE UUID Format

DCE UUID format **RepositoryId**s start with the characters "DCE:" and are followed by the printable form of the UUID, a colon, and a decimal minor version number, for example: "DCE:700dc518-0110-11ce-ac8f-0800090b5d3e:1".

6.6.3 LOCAL Format

Local format **RepositoryId**s start with the characters "LOCAL:" and are followed by an arbitrary string. Local format IDs are not intended for use outside a particular repository, and thus do not need to conform to any particular convention. Local IDs that are just consecutive integers might be used within a development environment to have a very cheap way to manufacture the IDs while avoiding conflicts with well-known interfaces.

6.6.4 Pragma Directives for RepositoryId

Three pragma directives (id, prefix, and version), are specified accommodate arbitrary **RepositoryId** formats and still support the OMG IDL **RepositoryId** format with minimal annotation. The pragma directives can be used with the OMG IDL, DCE UUID, and LOCAL formats. An IDL compiler must either interpret these annotations as specified, or ignore them completely.

The ID Pragma

An OMG IDL pragma of the format

#pragma ID <name> "<id>"

associates an arbitrary **RepositoryId** string with a specific OMG IDL name. The **<name>** can be a fully or partially scoped name or a simple identifier, interpreted according to the usual OMG IDL name lookup rules relative to the scope within which the pragma is contained.

The Prefix Pragma

An OMG IDL pragma of the format

#pragma prefix "<string>"

sets the current prefix used in generating OMG IDL format **RepositoryId**s. The specified prefix applies to **RepositoryId**s generated after the pragma until the end of the current scope is reached or another prefix pragma is encountered.

For example, the **RepositoryId** for the initial version of interface **Printer** defined on module **Office** by an organization known as "SoftCo" might be "IDL:SoftCo/Office/Printer:1.0".

This format makes it convenient to generate and manage a set of IDs for a collection of OMG IDL definitions. The person creating the definitions sets a prefix ("SoftCo"), and the IDL compiler or other tool can synthesize all the needed IDs.

Because **RepositoryId**s may be used in many different computing environments and ORBs, as well as over a long period of time, care must be taken in choosing them. Prefixes that are distinct, such as trademarked names, domain names, UUIDs, and so forth, are preferable to generic names such as "document."

The Version Pragma

An OMG IDL pragma of the format

#pragma version <name> <major>.<minor>

provides the version specification used in generating an OMG IDL format **RepositoryId** for a specific OMG IDL name. The **<name>** can be a fully or partially scoped name or a simple identifier, interpreted according to the usual OMG IDL name lookup rules relative to the scope within which the pragma is contained. The **<major>** and **<minor>** components are decimal unsigned shorts.

If no version pragma is supplied for a definition, version 1.0 is assumed.

Generation of OMG IDL - Format IDs

A definition is globally identified by an OMG IDL - format **RepositoryId** if no ID pragma is encountered for it.

The ID string can be generated by starting with the string "IDL:". Then, if any prefix pragma applies, it is appended, followed by a "/" character. Next, the components of the scoped name of the definition, relative to the scope in which any prefix that applies was encountered, are appended, separated by "/" characters. Finally, a ":" and the version specification are appended.

For example, the following OMG IDL:

```
module M1 {
    typedef long T1;
    typedef long T2;
    #pragma ID T2 "DCE:d62207a2-011e-11ce-88b4-0800090b5d3e:3"

};

#pragma prefix "P1"

module M2 {
    module M3 {
        #pragma prefix "P2"
        typedef long T3;
};
        typedef long T4;
#pragma version T4 2.4
};
```

specifies types with the following scoped names and **RepositoryId**s:

::M1::T1	IDL:M1/T1:1.0
::M1::T2	DCE:d62207a2-011e-11ce-88b4-0800090b5d3e:3
::M2::M3::T3	IDL:P2/T3:1.0
::M2::T4	IDL:P1/M2/T4:2.4

For this scheme to provide reliable global identity, the prefixes used must be unique. Two non-colliding options are suggested: Internet domain names and DCE UUIDs.

Furthermore, in a distributed world, where different entities independently evolve types, a convention must be followed to avoid the same **RepositoryId** being used for two different types. Only the entity that created the prefix has authority to create new IDs by simply incrementing the version number. Other entities must use a new prefix, even if they are only making a minor change to an existing type.

Prefix pragmas can be used to preserve the existing IDs when a module or other container is renamed or moved.

```
module M4 {
    #pragma prefix “P1/M2”
module M3 {
    #pragma prefix “P2”
        typedef long T3;
            };
        typedef long T4;
        #pragma version T4 2.4
};
```

This OMG IDL declares types with the same global identities as those declared in module M2 above.

For More Information

Section 6.8, “OMG IDL for Interface Repository,” on page 6-41 shows the OMG IDL specification of the IR, including the #pragma directive; Section 3.3, “Preprocessing,” on page 3-8 contain additional, general information on the pragma directive.

6.7 TypeCodes

TypeCodes are values that represent invocation argument types and attribute types. They can be obtained from the Interface Repository or from IDL compilers.

TypeCodes have a number of uses. They are used in the dynamic invocation interface to indicate the types of the actual arguments. They are used by an Interface Repository to represent the type specifications that are part of many OMG IDL declarations. Finally, they are crucial to the semantics of the **any** type.

TypeCodes are themselves values that can be passed as invocation arguments. To allow different ORB implementations to hide extra information in **TypeCode**s, the representation of **TypeCode**s will be opaque (like object references). However, we will assume that the representation is such that **TypeCode** “literals” can be placed in C include files.

Abstractly, **TypeCode**s consist of a “kind” field, and a set of parameters appropriate for that kind. For example, the **TypeCode** describing OMG IDL type **long** has kind **tk_long** and no parameters. The **TypeCode** describing OMG IDL type

sequence<boolean,10> has kind **tk_sequence** and two parameters: **10** and **boolean**.

6.7.1 The TypeCode Interface

The PIDL interface for **TypeCodes** is

```
module CORBA {
    enum TCKind {
        tk_null, tk_void,
        tk_short, tk_long, tk_ushort, tk_ulong,
        tk_float, tk_double, tk_boolean, tk_char,
        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
    };

    interface TypeCode {
        exception       Bounds {};
        exception       BadKind {};

        // for all TypeCode kinds
        boolean         equal (in TypeCode tc);
        TCKind          kind ();

        // for tk_objref, tk_struct, tk_union, tk_enum, tk_alias, and tk_except
        RepositoryId    id () raises (BadKind);

        // for tk_objref, tk_struct, tk_union, tk_enum, tk_alias, and tk_except
        Identifier      name () raises (BadKind);

        // for tk_struct, tk_union, tk_enum, and tk_except
        unsigned long   member_count () raises (BadKind);
        Identifier      member_name (in unsigned long index) raises (BadKind, Bounds);

        // for tk_struct, tk_union, and tk_except
        TypeCode        member_type (in unsigned long index) raises (BadKind, Bounds);

        // for tk_union
        any             member_label (in unsigned long index) raises (BadKind, Bounds);
        TypeCode        discriminator_type () raises (BadKind);
        long            default_index () raises (BadKind);

        // for tk_string, tk_sequence, and tk_array
        unsigned long   length () raises (BadKind);

        // for tk_sequence, tk_array, and tk_alias
        TypeCode        content_type () raises (BadKind);

        // deprecated interface
        long            param_count ();
        any             parameter (in long index) raises (Bounds);
    };
};
```

With the above operations, any **TypeCode** can be decomposed into its constituent parts. The **BadKind** exception is raised if an operation is not appropriate for the **TypeCode** kind is invoked.

The **equal** operation can be invoked on any **TypeCode**. Equal **TypeCode**s are interchangeable, and give identical results when **TypeCode** operations are applied to them.

The **kind** operation can be invoked on any **TypeCode**. Its result determines what other operations can be invoked on the **TypeCode**.

The **id** operation can be invoked on object reference, structure, union, enumeration, alias, and exception **TypeCode**s. It returns the **RepositoryId** globally identifying the type. Object reference and exception **TypeCode**s always have a **RepositoryId**. Structure, union, enumeration, and alias **TypeCode**s obtained from the Interface Repository or the **ORB::create_operation_list** operation also always have a **RepositoryId**. Otherwise, the **id** operation can return an empty string.

The **name** operation can also be invoked on object reference, structure, union, enumeration, alias, and exception **TypeCode**s. It returns the simple name identifying the type within its enclosing scope. Since names are local to a **Repository**, the name returned from a **TypeCode** may not match the name of the type in any particular **Repository**, and may even be an empty string.

The **member_count** and **member_name** operations can be invoked on structure, union, and enumeration **TypeCode**s. **Member_count** returns the number of members constituting the type. **Member_name** returns the simple name of the member identified by **index**. Since names are local to a **Repository**, the name returned from a **TypeCode** may not match the name of the member in any particular **Repository**, and may even be an empty string.

The **member_type** operation can be invoked on structure and union **TypeCode**s. It returns the **TypeCode** describing the type of the member identified by **index**.

The **member_label**, **discriminator_type**, and **default_index** operations can only be invoked on union **TypeCode**s. **Member_label** returns the label of the union member identified by **index**. For the default member, the label is the zero octet. The **discriminator_type** operation returns the type of all non-default member labels. The **default_index** operation returns the index of the default member, or -1 if there is no default member.

The **member_name**, **member_type**, and **member_label** operations raise **Bounds** if the index parameter is greater than or equal to the number of members constituting the type.

The **length** operation can be invoked on string, sequence, and array **TypeCode**s. For strings and sequences, it returns the bound, with zero indicating an unbounded string or sequence. For arrays, it returns number of elements in the array.

The **content_type** operation can be invoked on sequence, array, and alias **TypeCode**s. For sequences and arrays, it returns the element type. For aliases, it returns the original type.

An array **TypeCode** only describes a single dimension of an OMG IDL array. Multidimensional arrays are represented by nesting **TypeCode**s, one per dimension. The outermost **tk_array Typecode** describes the leftmost array index of the array as defined in IDL. Its **content_type** describes the next index. The innermost nested **tk_array TypeCode** describes the rightmost index and the array element type.

The deprecated **param_count** and **parameter** operations provide access to those parameters that were present in previous versions of *CORBA*. Some information available via other **TypeCode** operations is not visible via the **parameter** operation. The meaning of the indexed parameters for each **TypeCode** kind are listed in TABLE 12. on page 6-37, along with the information that is not visible via the **parameter** operation.

TABLE 12. **Legal TypeCode Kinds and Parameters**

KIND	PARAMETER LIST	NOT VISIBLE
tk_null	*NONE*	
tk_void	*NONE*	
tk_short	*NONE*	
tk_long	*NONE*	
tk_ushort	*NONE*	
tk_ulong	*NONE*	
tk_float	*NONE*	
tk_double	*NONE*	
tk_boolean	*NONE*	
tk_char	*NONE*	
tk_octet	*NONE*	
tk_any	*NONE*	
tk_TypeCode	*NONE*	
tk_Principal	*NONE*	
tk_objref	{ interface-id }	interface name
tk_struct	{ struct-name, member-name, TypeCode, ... (repeat pairs) }	RepositoryId
tk_union	{ union-name, discriminator-TypeCode, label-value, member-name, TypeCode, ... (repeat triples) }	RepositoryId
tk_enum	{ enum-name, enumerator-name, ... }	RepositoryId
tk_string	{ maxlen-integer }	
tk_sequence	{ TypeCode, maxlen-integer }	
tk_array	{ TypeCode, length-integer }	
tk_alias	{ alias-name, TypeCode }	Repositoryid
tk_except	{ except-name, member-name, TypeCode, ... (repeat pairs) }	RepositoryId

The **tk_objref TypeCode** represents an interface type. Its parameter is the **RepositoryId** of that interface.

A structure with N members results in a **tk_struct TypeCode** with 2N+1 parameters: first, the simple name of the struct; the rest are member names alternating with the corresponding member **TypeCode**. Member names are represented as strings.

A union with N members results in a **tk_union TypeCode** with 3N+2 parameters: the simple name of the union, the discriminator **TypeCode** followed by a label value, member name, and member **TypeCode** for each of the N members. The label values are all values of the data type designated by the discriminator **TypeCode**, with one exception. The default member (if present) is marked with a label value consisting of the 0 **octet**. Recall that the operation "parameter(tc,i)" returns an **any**, and that anys themselves carry a **TypeCode** that can distinguish an octet from any of the legal switch types.

The **tk_enum TypeCode** has the simple name of the enum followed by the enumerator names as parameters. Enumerator names are represented as strings.

The **tk_string TypeCode** has 1 parameter: an integer giving the maximum string length. A maximum of 0 denotes unbounded.

The **tk_sequence TypeCode** has 2 parameters: a **TypeCode** for the sequence elements, and an integer giving the maximum sequence. Again, 0 denotes unbounded.

The **tk_array TypeCode** has 2 parameters: a **TypeCode** for the array elements, and an integer giving the array length. Arrays are never unbounded.

The **tk_alias TypeCode** has 2 parameters: the name of the alias followed by the **TypeCode** of the type being aliased.

The **tk_except TypeCode** has the same format as the **tk_struct TypeCode**, except that exceptions with no members are allowed.

6.7.2 TypeCode Constants

If "**typedef ... FOO;**" is an IDL type declaration, the IDL compiler will (if asked) produce a declaration of a **TypeCode** constant named TC_FOO for the C language mapping. In the case of an unnamed, bounded string type used directly in an operation or attribute declaration, a **TypeCode** constant named TC_string_n, where n is the bound of the string is produced. (For example, "string<4> op1();" produces the constant "TC_string_4".) These constants can be used with the dynamic invocation interface, and any other routines that require **TypeCode**s. The predefined **TypeCode** constants, named according to the C language mapping, are:

TC_null
TC_void
TC_short
TC_long
TC_ushort
TC_ulong
TC_float
TC_double
TC_boolean
TC_char
TC_octet
TC_any
TC_TypeCode

```
TC_Principal
TC_Object = tk_objref { Object }
TC_string= tk_string { 0 } // unbounded
TC_CORBA_NamedValue= tk_struct { ... }
TC_CORBA_InterfaceDescription= tk_struct { ... }
TC_CORBA_OperationDescription= tk_struct { ... }
TC_CORBA_AttributeDescription= tk_struct { ... }
TC_CORBA_ParameterDescription= tk_struct { ... }
TC_CORBA_ModuleDescription= tk_struct { ... }
TC_CORBA_ConstantDescription= tk_struct { ... }
TC_CORBA_ExceptionDescription= tk_struct { ... }
TC_CORBA_TypeDescription= tk_struct { ... }
TC_CORBA_InterfaceDef_FullInterfaceDescription= tk_struct { ... }
```

The exact form for **TypeCode** constants is language mapping, and possibly implementation, specific.

6.7.3 Creating TypeCodes

When creating type definition objects in an Interface Repository, types are specified in terms of object references, and the **TypeCode**s describing them are generated automatically.

In some situations, such as bridges between ORBs, **TypeCode**s need to be constructed outside of any Interface Repository. This can be done using operations on the **ORB** pseudo-object.

```
module CORBA {
    interface ORB {
        // other operations ...

        TypeCode  create_struct_tc (
                in RepositoryId         id,
                in Identifier           name,
                in StructMemberSeq      members
        );

        TypeCode  create_union_tc (
                in RepositoryId         id,
                in Identifier           name,
                in TypeCode             discriminator_type,
                in UnionMemberSeq       members
        );

        TypeCode  create_enum_tc (
                in RepositoryId         id,
                in Identifier           name,
                in EnumMemberSeq        members
        );

        TypeCode create_alias_tc (
                in RepositoryId         id,
                in Identifier           name,
                in TypeCode             original_type
        );

        TypeCode  create_exception_tc (
                in RepositoryId         id,
                in Identifier           name,
                in StructMemberSeq      members
        );

        TypeCode  create_interface_tc (
                in RepositoryId         id,
                in Identifier           name
        );

        TypeCode  create_string_tc (
                in unsigned long        bound
        );

        TypeCode  create_sequence_tc (
                in unsigned long        bound,
                in TypeCode             element_type
        );

        TypeCode  create_recursive_sequence_tc (
                in unsigned long        bound,
                in unsigned long        offset
        );

        TypeCode  create_array_tc (
                in unsigned long        length,
                in TypeCode             element_type
        );
    };
};
```

Most of these operations are similar to corresponding IR operations for creating type definitions. **TypeCode**s are used here instead of **IDLType** object references to refer to other types. In the **StructMember** and **UnionMember** structures, only the **type** is used, and the **type_def** should be set to nil.

The **create_recursive_sequence_tc** operation is used to create **TypeCode**s describing recursive sequences (see See "Constructed Types" on page 22.) The result of this operation is used in constructing other types, with the **offset** parameter determining which enclosing **TypeCode** describes the elements of this sequence. For instance, to construct a **TypeCode** for the following OMG IDL structure, the offset used when creating its sequence **TypeCode** would be one:

```
struct foo {
    long value;
    sequence <foo> chain;
};
```

Operations to create primitive **TypeCode**s are not needed, since **TypeCode** constants for these are available.

6.8 OMG IDL for Interface Repository

This section contains the complete OMG IDL specification for the Interface Repository.

```
#pragma prefix “omg.org”

module CORBA {
   typedef string Identifier;
   typedef string ScopedName;
   typedef string RepositoryId;

   enum DefinitionKind {
     dk_none, dk_all,
     dk_Attribute, dk_Constant, dk_Exception, dk_Interface,
     dk_Module, dk_Operation, dk_Typedef,
     dk_Alias, dk_Struct, dk_Union, dk_Enum,
     dk_Primitive, dk_String, dk_Sequence, dk_Array,
     dk_Repository
   };



   interface IRObject {
     // read interface
     readonly attribute DefinitionKind def_kind;

     // write interface
     void destroy ();
   };



   typedef string VersionSpec;

   interface Contained;
   interface Repository;
   interface Container;

   interface Contained : IRObject {
     // read/write interface

     attribute RepositoryId id;
     attribute Identifier name;
     attribute VersionSpec version;

     // read interface

     readonly attribute Container defined_in;
     readonly attribute ScopedName absolute_name;
     readonly attribute Repository containing_repository;

     struct Description {
        DefinitionKind kind;
        any value;
```

```
  };

  Description describe ();

  // write interface

  void move (
      in Container new_container,
      in Identifier new_name,
      in VersionSpec new_version
      );
};


interface ModuleDef;
interface ConstantDef;
interface IDLType;
interface StructDef;
interface UnionDef;
interface EnumDef;
interface AliasDef;
interface InterfaceDef;
typedef sequence <InterfaceDef> InterfaceDefSeq;

typedef sequence <Contained> ContainedSeq;

struct StructMember {
  Identifier name;
  TypeCode type;
  IDLType type_def;
};
typedef sequence <StructMember> StructMemberSeq;

struct UnionMember {
  Identifier name;
  any label;
  TypeCode type;
  IDLType type_def;
};
typedef sequence <UnionMember> UnionMemberSeq;

typedef sequence <Identifier> EnumMemberSeq;

interface Container : IRObject {
  // read interface

  Contained lookup ( in ScopedName search_name);

  ContainedSeq contents (
      in DefinitionKind limit_type,
      in boolean exclude_inherited
```

```
   );

ContainedSeq lookup_name (
   in Identifier search_name,
   in long levels_to_search,
   in DefinitionKind limit_type,
   in boolean exclude_inherited
   );

struct Description {
   Contained contained_object;
   DefinitionKind kind;
   any value;
};

typedef sequence<Description> DescriptionSeq;

DescriptionSeq describe_contents (
   in DefinitionKind limit_type,
   in boolean exclude_inherited,
   in long max_returned_objs
   );

// write interface

ModuleDef create_module (
   in RepositoryId id,
   in Identifier name,
   in VersionSpec version
   );

ConstantDef create_constant (
   in RepositoryId id,
   in Identifier name,
   in VersionSpec version,
   in IDLType type,
   in any value
   );

StructDef create_struct (
   in RepositoryId id,
   in Identifier name,
   in VersionSpec version,
   in StructMemberSeq members
   );

UnionDef create_union (
   in RepositoryId id,
   in Identifier name,
   in VersionSpec version,
   in IDLType discriminator_type,
        in UnionMemberSeq members
        );

    EnumDef create_enum (
        in RepositoryId id,
        in Identifier name,
        in VersionSpec version,
        in EnumMemberSeq members
        );

    AliasDef create_alias (
        in RepositoryId id,
        in Identifier name,
        in VersionSpec version,
        in IDLType original_type
        );

    InterfaceDef create_interface (
        in RepositoryId id,
        in Identifier name,
        in VersionSpec version,
        in InterfaceDefSeq base_interfaces
        );
};



interface IDLType : IRObject {
    readonly attribute TypeCode type;
};



interface PrimitiveDef;
interface StringDef;
interface SequenceDef;
interface ArrayDef;

enum PrimitiveKind {
    pk_null, pk_void, pk_short, pk_long, pk_ushort, pk_ulong,
    pk_float, pk_double, pk_boolean, pk_char, pk_octet,
    pk_any, pk_TypeCode, pk_Principal, pk_string, pk_objref
};

interface Repository : Container {
    // read interface

    Contained lookup_id (in RepositoryId search_id);

    PrimitiveDef get_primitive (in PrimitiveKind kind);
```

```
  // write interface

  StringDef create_string (in unsigned long bound);

  SequenceDef create_sequence (
     in unsigned long bound,
     in IDLType element_type
     );

  ArrayDef create_array (
     in unsigned long length,
     in IDLType element_type
     );
};



interface ModuleDef : Container, Contained {
};

struct ModuleDescription {
  Identifier name;
  RepositoryId id;
  RepositoryId defined_in;
  VersionSpec version;
};



interface ConstantDef : Contained {
  readonly attribute TypeCode type;
  attribute IDLType type_def;
  attribute any value;
};

struct ConstantDescription {
  Identifier name;
  RepositoryId id;
  RepositoryId defined_in;
  VersionSpec version;
  TypeCode type;
  any value;
};



interface TypedefDef : Contained, IDLType {
};

struct TypeDescription {
  Identifier name;
  RepositoryId id;
  RepositoryId defined_in;
  VersionSpec version;
  TypeCode type;
};


interface StructDef : TypedefDef {
  attribute StructMemberSeq members;
};


interface UnionDef : TypedefDef {
  readonly attribute TypeCode discriminator_type;
  attribute IDLType discriminator_type_def;
  attribute UnionMemberSeq members;
};


interface EnumDef : TypedefDef {
  attribute EnumMemberSeq members;
};


interface AliasDef : TypedefDef {
  attribute IDLType original_type_def;
};


interface PrimitiveDef: IDLType {
  readonly attribute PrimitiveKind kind;
};


interface StringDef : IDLType {
  attribute unsigned long bound;
};


interface SequenceDef : IDLType {
  attribute unsigned long bound;
  readonly attribute TypeCode element_type;
  attribute IDLType element_type_def;
};
```

```
interface ArrayDef : IDLType {
  attribute unsigned long length;
  readonly attribute TypeCode element_type;
  attribute IDLType element_type_def;
};



interface ExceptionDef : Contained {
  readonly attribute TypeCode type;
  attribute StructMemberSeq members;
};
struct ExceptionDescription {
  Identifier name;
  RepositoryId id;
  RepositoryId defined_in;
  VersionSpec version;
  TypeCode type;
};



enum AttributeMode {ATTR_NORMAL, ATTR_READONLY};

interface AttributeDef : Contained {
  readonly attribute TypeCode type;
  attribute IDLType type_def;
  attribute AttributeMode mode;
};

struct AttributeDescription {
  Identifier name;
  RepositoryId id;
  RepositoryId defined_in;
  VersionSpec version;
  TypeCode type;
  AttributeMode mode;
};



enum OperationMode {OP_NORMAL, OP_ONEWAY};

enum ParameterMode {PARAM_IN, PARAM_OUT, PARAM_INOUT};
struct ParameterDescription {
  Identifier name;
  TypeCode type;
  IDLType type_def;
  ParameterMode mode;
};
typedef sequence <ParameterDescription> ParDescriptionSeq;

typedef Identifier ContextIdentifier;
typedef sequence <ContextIdentifier> ContextIdSeq;

typedef sequence <ExceptionDef> ExceptionDefSeq;
typedef sequence <ExceptionDescription> ExcDescriptionSeq;

interface OperationDef : Contained {
  readonly attribute TypeCode result;
  attribute IDLType result_def;
  attribute ParDescriptionSeq params;
  attribute OperationMode mode;
  attribute ContextIdSeq contexts;
  attribute ExceptionDefSeq exceptions;
};

struct OperationDescription {
  Identifier name;
  RepositoryId id;
  RepositoryId defined_in;
  VersionSpec version;
  TypeCode result;
  OperationMode mode;
  ContextIdSeq contexts;
  ParDescriptionSeq parameters;
  ExcDescriptionSeq exceptions;
};



typedef sequence <RepositoryId> RepositoryIdSeq;
typedef sequence <OperationDescription> OpDescriptionSeq;
typedef sequence <AttributeDescription> AttrDescriptionSeq;

interface InterfaceDef : Container, Contained, IDLType {
  // read/write interface

  attribute InterfaceDefSeq base_interfaces;

  // read interface

  boolean is_a (in RepositoryId interface_id);

  struct FullInterfaceDescription {
     Identifier name;
     RepositoryId id;
     RepositoryId defined_in;
     VersionSpec version;
```

```
        OpDescriptionSeq operations;
        AttrDescriptionSeq attributes;
        RepositoryIdSeq base_interfaces;
        TypeCode type;
    };

    FullInterfaceDescription describe_interface();

    // write interface

    AttributeDef create_attribute (
        in RepositoryId id,
        in Identifier name,
        in VersionSpec version,
        in IDLType type,
        in AttributeMode mode
        );

    OperationDef create_operation (
        in RepositoryId id,
        in Identifier name,
        in VersionSpec version,
        in IDLType result,
        in OperationMode mode,
        in ParDescriptionSeq params,
        in ExceptionDefSeq exceptions,
        in ContextIdSeq contexts
        );
};

struct InterfaceDescription {
    Identifier name;
    RepositoryId id;
    RepositoryId defined_in;
    VersionSpec version;
    RepositoryIdSeq base_interfaces;
};



enum TCKind {
    tk_null, tk_void,
    tk_short, tk_long, tk_ushort, tk_ulong,
    tk_float, tk_double, tk_boolean, tk_char,
    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
};

interface TypeCode { // PIDL
    exception Bounds {};
```

```
  exception BadKind {};

  // for all TypeCode kinds
  boolean equal (in TypeCode tc);
  TCKind kind ();

  // for tk_objref, tk_struct, tk_union, tk_enum, tk_alias, and tk_except
  RepositoryId id () raises (BadKind);

  // for tk_objref, tk_struct, tk_union, tk_enum, tk_alias, and tk_except
  Identifier name () raises (BadKind);

  // for tk_struct, tk_union, tk_enum, and tk_except
  unsigned long member_count () raises (BadKind);
  Identifier member_name (in unsigned long index) raises (BadKind, Bounds);

  // for tk_struct, tk_union, and tk_except
  TypeCode member_type (in unsigned long index) raises (BadKind, Bounds);

  // for tk_union
  any member_label (in unsigned long index) raises (BadKind, Bounds);
  TypeCode discriminator_type () raises (BadKind);
  long default_index () raises (BadKind);

  // for tk_string, tk_sequence, and tk_array
  unsigned long length () raises (BadKind);

  // for tk_sequence, tk_array, and tk_alias
  TypeCode content_type () raises (BadKind);

  // deprecated interface
  long param_count ();
  any parameter (in long index) raises (Bounds);
};


interface ORB {
  // other operations ...

  TypeCode create_struct_tc (
     in RepositoryId id,
     in Identifier name,
     in StructMemberSeq members
     );

  TypeCode create_union_tc (
     in RepositoryId id,
     in Identifier name,
     in TypeCode discriminator_type,
     in UnionMemberSeq members
```

```
            );

        TypeCode create_enum_tc (
            in RepositoryId id,
            in Identifier name,
            in EnumMemberSeq members
            );

        TypeCode create_alias_tc (
            in RepositoryId id,
            in Identifier name,
            in TypeCode original_type
            );

        TypeCode create_exception_tc (
            in RepositoryId id,
            in Identifier name,
            in StructMemberSeq members
            );

        TypeCode create_interface_tc (
            in RepositoryId id,
            in Identifier name
            );

        TypeCode create_string_tc (
            in unsigned long bound
            );

        TypeCode create_sequence_tc (
            in unsigned long bound,
            in TypeCode element_type
            );

        TypeCode create_recursive_sequence_tc (
            in unsigned long bound,
            in unsigned long offset
            );

        TypeCode create_array_tc (
            in unsigned long length,
            in TypeCode element_type
            );
    };
};
```

ORB Interface 7

The ORB interface is the interface to those ORB functions that do not depend on which object adapter is used. These operations are the same for all ORBs and all object implementations, and can be performed either by clients of the objects or implementations. Some of these operations appear to be on the ORB, others appear to be on the object reference. Because the operations in this section are implemented by the ORB itself, they are not in fact operations on objects, although they may be described that way and the language binding will, for consistency, make them appear that way.

The ORB interface also defines operations for creating lists and determining the default context used in the Dynamic Invocation Interface. Those operations are described in Chapter 4.

All types defined in this chapter are part of the CORBA module. When referenced in OMG IDL, the type names must be prefixed by “CORBA::”.

7.1 Converting Object References to Strings

Because an object reference is opaque and may differ from ORB to ORB, the object reference itself is not a convenient value for storing references to objects in persistent storage or communicating references by means other than invocation. Two problems must be solved: allowing an object reference to be turned into a value that a client can store in some other medium, and ensuring that the value can subsequently be turned into the appropriate object reference.

An object reference may be translated into a string by the operation **object_to_string**. The value may be stored or communicated in whatever ways strings may be manipulated. Subsequently, the **string_to_object** operation will accept a string produced by **object_to_string** and return the corresponding object reference.

```
module CORBA {

interface ORB {                                                      // PIDL
      string object_to_string (in Object obj);
      Object  string_to_object (in string str);

      Status   create_list (
      in long                 count,
      out NVList              new_list
);
      Status   create_operation_list (
      in OperationDef     oper,
      out NVList              new_list
);

Status get_default_context (   out   Context ctx);

};
      };
```

To guarantee that an ORB will understand the string form of an object reference, that ORB's **object_to_string** operation must be used to produce the string. Since in general a client does not know or care which ORB is used for a particular object reference, the client can choose whatever ORB is convenient.

For a description of the **create_list** and **create_operation_list** operations, see "List Operations" on page 4-10. The **get_default_context** operation is described in the section "get_default_context" on page 4-14.

7.2 Object Reference Operations

There are some operations that can be done on any object. These are not operations in the normal sense, in that they are implemented directly by the ORB, not passed on to the object implementation. We will describe these as being operations on the object reference, although the interfaces actually depend on the language binding. As above, where we used interface Object to represent the object reference, we will define an interface for Object:

```
module CORBA {

    interface Object {                                          // PIDL

            ImplementationDef get_implementation ();
            InterfaceDef get_interface ();
            boolean is_nil();
            Object duplicate ();
            void    release ();
            boolean    is_a (in string logical_type_id);
            boolean    non_existent();
            boolean    is_equivalent (in Object other_object);
            unsigned   long hash(in unsigned long maximum);


            Status create_request (
            in Context  ctx,
            in Identifieroperation,
            in NVList   arg_list,
            inout NamedValueresult,
            out Requestrequest,
            in Flags    req_flags
            );
            };
            };
```

The **create_request** operation is part of the Object interface because it creates a pseudo-object (a Request) for an object. It is described with the other Request operations in the section “Request Operations” on page 4-4.

7.2.1 Determining the Object Implementation and Interface

An operation on the object reference, **get_interface**, returns an object in the Interface Repository, which provides type information that may be useful to a program. See Chapter 6 for a definition of operations on the Interface Repository. An operation on the Object called **get_implementation** will return an object in an implementation repository that describes the implementation of the object. See the Basic Object Adapter chapter for information about the Implementation Repository.

```
InterfaceDef get_interface ();                          // PIDL
ImplementationDef get_implementation ();
```

7.2.2 Duplicating and Releasing Copies of Object References

Because object references are opaque and ORB-dependent, it is not possible for clients or implementations to allocate storage for them. Therefore, there are operations defined to copy or release an object reference.

```
Object duplicate ();                                    // PIDL
void release ();
```

If more than one copy of an object reference is needed, the client may create a **duplicate**. Note that the object implementation is not involved in creating the duplicate, and that the implementation cannot distinguish whether the original or a duplicate was used in a particular request.

When an object reference is no longer needed by a program, its storage may be reclaimed by use of the **release** operation. Note that the object implementation is not involved, and that neither the object itself nor any other references to it are affected by the **release** operation.

7.2.3 Nil Object References

An object reference whose value is OBJECT_NIL denotes no object. An object reference can be tested for this value by the **is_nil** operation. The object implementation is not involved in the nil test.

```
boolean is_nil ();                                      // PIDL
```

7.2.4 Equivalence Checking Operation

An operation is defined to facilitate maintaining type-safety for object references over the scope of an ORB.

```
boolean is_a(in string logical_type_id);                // PIDL
```

The **logical_type_id** is a string denoting a shared type identifier (RepositoryId). The operation returns true if the object is really an instance of that type, including if that type is an ancestor of the “most derived” type of that object.

This operation exposes to application programmers functionality that must already exist in ORBs which support “type safe narrow”, and allows programmers working in environments that do not have compile time type checking to explicitly maintain type safety.

7.2.5 Probing for Object Non-Existence

```
boolean non_existent ();                                // PIDL
```

The **non_existent** operation may be used to test whether an object (e.g. a proxy object) has been destroyed. It does this without invoking any application level operation on the object, and so will never affect the object itself. It returns true (rather than raising CORBA::OBJECT_NOT_EXIST) if the ORB knows authoritatively that the object does not exist, and otherwise it returns false.

Services that maintain state that includes object references, such as bridges, event channels, and base relationship services, might use this operation in their “idle time” to sift through object tables for objects that no longer exist, deleting them as they go, as a form of garbage collection. In the case of proxies, this kind of activity can cascade, such that cleaning up one table allows others then to be cleaned up.

7.2.6 Object Reference Identity

In order to efficiently manage state that include large numbers of object references, services need to support a notion of object reference identity. Such services include not just bridges, but relationship services and other layered facilities.

```
unsigned long hash(in unsigned long maximum);                // PIDL
boolean is_equivalent(in Object other_object);
```

Two identity-related operations are provided. One maps object references into disjoint groups of potentially equivalent references, and the other supports more expensive pairwise equivalence testing. Together, these operations support efficient maintenance and search of tables keyed by object references.

Hashing: Object Identifiers

Object references are associated with ORB-internal identifiers which may indirectly be accessed by applications using the **hash()** operation. The value of this identifier does not change during the lifetime of the object reference, and so neither will any hash function of that identifier.

The value of this operation is not guaranteed to be unique; that is, another object reference may return the same hash value. However, if two object references hash differently, applications can determine that the two object references are *not* identical.

The **maximum** parameter to the **hash** operation specifies an upper bound on the hash value returned by the ORB. The lower bound of that value is zero. Since a typical use of this feature is to construct and access a collision chained hash table of object references, the more randomly distributed the values are within that range, and the cheaper those values are to compute, the better.

For bridge construction, note that proxy objects are themselves objects, so there could be many proxy objects representing a given “real” object. Those proxies would not necessarily hash to the same value.

Equivalence Testing

The **is_equivalent()** operation is used to determine if two object references are equivalent, so far as the ORB can easily determine. It returns TRUE if the target object reference is known to be equivalent to the other object reference passed as its parameter, and FALSE otherwise.

If two object references are identical, they are equivalent. Two different object references which in fact refer to the same object are also equivalent.

ORBs are allowed, but not required, to attempt determination of whether two distinct object references refer to the same object. In general, the existence of reference translation and encapsulation, in the absence of an omniscient topology service, can make such determination impractically expensive. This means that a FALSE return from **is_equivalent()** should be viewed as only indicating that the object references are distinct, and not necessarily an indication that the references indicate distinct objects.

A typical application use of this operation is be to match object references in a hash table. Bridges could use it to shorten the lengths of chains of proxy object references. Externalization services could use it to “flatten” graphs that represent cyclical relationships between objects. Some might do this as they construct the table, others during idle time.

7.3 Overview: ORB and OA Initialization and Initial References

Before an application can enter the CORBA environment, it must first:

- Be initialized into the ORB and object adapter (BOA and OA) environments.
- Get references to ORB and OA (including BOA) pseudo-objects—and sometimes to other objects—for use in future ORB and OA operations.

CORBA V2.0 provides operations, specified in PIDL, to initialize applications and obtain the appropriate object references. The following is provided:

- Operations providing access to the ORB. These operations reside in CORBA module, but not in the ORB interface and are described in Section 7.4, “ORB Initialization,” on page 7-6.
- Operations providing access to the Basic Object Adapter (BOA) and other object adapters (OAs) These operations reside in the ORB interface and are described in Section 7.5, “OA and BOA Initialization,” on page 7-8.
- Operations providing access to the Interface Repository, Naming Service, and other Object Services. These operations reside in the ORB interface and are described in Section 7.6, “Obtaining Initial Object References,” on page 7-10.

In addition, this manual provides a mapping of the PIDL initialization and object reference operations to the C and C++ programming languages. For mapping information, refer to Section 14.26, “ORB and OA/BOA Initialization Operations,” on page 14-31 and to Section 17.12, “ORB,” on page 17-11.

7.4 ORB Initialization

When an application requires a CORBA environment it needs a mechanism to get ORB and OA pseudo-object references. This serves two purposes. First, it initializes an application into the ORB and OA environments. Second, it returns the ORB and OA pseudo-object references to the application for use in future ORB and OA operations.

The ORB and BOA initialization operations must be ordered with ORB occurring before OA: an application cannot call OA initialization routines until ORB initialization routines have been called for the given ORB.

The operation to initialize an application in the ORB and get its pseudo-object reference is not performed on an object. This is because applications do not initially have an object on which to invoke operations. The ORB initialization operation is an application's bootstrap call into the CORBA world. The PIDL for the call (Figure 7-1) shows that the **ORB_init** call is part of the CORBA module but not part of the ORB interface.

Applications can be initialized in one or more ORBs. When an ORB initialization is complete, its pseudo reference is returned and can be used to obtain OA references for that ORB.

In order to obtain an ORB pseudo-object reference, applications call the **ORB_init** operation. The parameters to the call comprise an identifier for the ORB for which the pseudo-object reference is required, and an **arg_list,** which is used to allow environment-specific data to be passed into the call. PIDL for the ORB initialization is as follows:

```
// PIDL
module CORBA {

 typedef string ORBid;
 typedef sequence <string> arg_list;
 ORB ORB_init (inout arg_list argv, in ORBid orb_identifier);
 };
```

Figure 7-1

The identifier for the ORB will be a name of type ORBid (string). The allocation of ORBids is the responsibility of ORB administrators and is not intended to be managed by the OMG. Names are locally scoped and the ORB administrator is responsible for ensuring that the names are unambiguous. Examples of potential ORBids are "Internet ORB," "BNR_private," "BNR_interop_1_2." If a NULL ORBid is used then arg_list arguments can be used to determine which ORB should be returned. This is achieved by searching the arg_list parameters for one tagged ORBid, for example, –ORBid "ORBid_example." Other parameters of significance to the ORB can be identified, for example, "Hostname," "SpawnedServer," and so forth. To allow for other parameters to be specified without causing applications to be re-written, it is necessary to specify the format that ORB parameters may take. The format of those parameters will be

–ORB<suffix> <value>.

The **ORB_init** operation can be called any number of times and is expected to return the same pseudo-object reference for the same parameters. Calling the **ORB_init** function multiples times for the same ORB may be required where an ORB is implemented as a shared library, or where several threads of a multi-threaded application require to use the same ORB and all wish to call the **ORB_init** operation.

7.5 OA and BOA Initialization

An ORB may have zero or more object adaptors associated with it. Servers must have a reference to an OA pseudo-object in order to access its functionality.

The only object adaptor defined in *CORBA* is the Basic Object Adaptor (BOA). However other adaptors such as the Library Object Adaptor (LOA) are also mentioned. Given an ORB reference, an application must be able to initialize itself in an OA environment and get the pseudo reference of the OA from the ORB.

Because OAs are pseudo-objects and therefore do not necessarily share a common interface, it is not possible to have a generic OA_init operation that returns an object type which is then explicitly narrowed or widened to the correct pseudo-object type. It is therefore necessary to provide an initialization function for each OA type separately. To achieve this a template is suggested for OA initialization, and the BOA initialization operation is generated from that template.

The operation to get the OA pseudo object reference is part of the ORB interface. The **<OA>_init** operation is therefore an operation on the ORB pseudo object. Figure 7-2 shows the PIDL for the for the **<OA>_init** (specifically BOA_init) operation.

```
// PIDL
module CORBA {

    interface ORB
    {

        typedef sequence <string> arg_list;
        typedef string OAid;

        // Template for OA initialization operations
        // <OA> <OA>_init (inout arg_list argv,
//                              in OAid oa_identifier);

BOA BOA_init (inout arg_list argv,
                    in OAid boa_identifier);

};

}
```

Figure 7-2

The identifier for the OA will be a name of the type **OAid** (string). The allocation of OAids is the responsibility of ORB administrators and is not intended to be managed by the OMG. Names are locally scoped and the ORB administrator is responsible for ensuring that the names are unambiguous. Examples of potential **OAid**s are “BOA,” “BNR_BOA,” “HP_LOA.”

If a NULL OAid is used then arg_list arguments can be used to determine which OA should be returned. This is achieved by searching the **arg_list** parameters for one tagged OAid, e.g. -OAid “OAid_example”.

In order to allow for other OA parameters to be specified in the future without causing applications to be re-written it is necessary to specify the format parameters may take. The format of OA specific parameters will be **- OA<suffix> <value>**.

The BOA_init function may be called any number of times and is expected to return the same pseudo object reference for the same parameters. Calling the **BOA_init** operation multiples times for the same BOA may be required where several threads of a multi-threaded application require to use the same BOA and therefore need to the **BOA_init** operation.

The **BOA_init** operation returns a BOA. Once the operation has returned the BOA is assumed to be initialized for the application object.

7.6 Obtaining Initial Object References

Applications require a portable means by which to obtain their initial object references. References are required for the Interface Repository and Object Services. (The Interface Repository is described in Chapter 6 of this manual; Object Services are described in *CORBAservices*.) The functionality required by the application is similar to that provided by the Naming Service. However, the OMG does not want to mandate that the Naming Service be made available to all applications in order that they may be portably initialized. Consequently, the operations shown in this section provide a simplified, local version of the Naming Service that applications can use to obtain a small, defined set of object references which are essential to its operation. Because only a small well defined set of objects are expected with this mechanism, the naming context can be flattened to be a single-level name space. This simplification results in only two operations being defined to achieve the functionality required.

Initial references are not obtained via a new interface; instead two new operations are added to the ORB pseudo-object interface, providing facilities to list and resolve initial object references. Figure 7-3 on page 7-10 shows the PIDL for these operations.

```
// PIDL interface for getting initial object references
module CORBA {
    interface ORB {
    typedef string ObjectId;
    typedef sequence <ObjectId> ObjectIdList;

    exception InvalidName {};

    ObjectIdList list_initial_services ();

    Object resolve_initial_references (in ObjectId identifier)
        raises (InvalidName);
  }

}
```

Figure 7-3

The **resolve_initial_references o**peration is an operation on the ORB rather than the Naming Service's NamingContext. The interface differs from the Naming Service's resolve in that **ObjectId** (a string) replaces the more complex Naming Service construct (a sequence of structures containing string pairs for the components of the name). This simplification reduces the name space to one context.

ObjectIds are strings that identify the object whose reference is required. To maintain the simplicity of the interface for obtaining initial references, only a limited set of objects are expected to have their references found via this route. Unlike the ORB and BOA identifiers, the **ObjectId** name space requires careful management. To achieve this. the OMG may, in the future, define which services are required by applications through this interface and specify names for those services.

Currently, reserved **ObjectIds** are **InterfaceRepository** and **NameService.**

To allow an application to determine which objects have references available via the initial references mechanism, the **list_initial_services** operation (also a call on the ORB) is provided. It returns an **ObjectIdList,** which is a sequence of **ObjectIds**. **ObjectIds** are typed as strings. Each object, which may need to be made available at initialization time, is allocated a string value to represent it. In addition to defining the id, the type of object being returned must be defined, i.e. "InterfaceRepository" returns a object of type Repository, and “NameService" returns a CosNamingContext object.

The application is responsible for narrowing the object reference returned from **resolve_initial_references** to the type which was requested in the ObjectId. E.g. for InterfaceRepository the object returned would be narrowed to **Repository** type.

In the future, specifications for Object Services (in *CORBAservices*) will state whether it is expected that a service’s initial reference be made available via the **resolve_initial_references** operation or not, i.e. whether the service is necessary or desirable for bootstrap purposes.

The Basic Object Adapter 8

An object adapter is the primary interface that an implementation uses to access ORB functions. The *Basic Object Adapter* (BOA) is an interface intended to be widely available and to support a wide variety of common object implementations. It includes convenient interfaces for generating object references, registering implementations that consist of one or more programs, activating implementations, and authenticating requests. It also provides a limited amount of persistent storage for objects that can be used for connecting to a larger or more general storage facility, for storing access control information, or other purposes.

Most of the *Basic Object Adapter* interface can be expressed in OMG IDL, since the interface is to the operations on the object adapter. Some of the operations to bind the implementation to the object adapter depend on the language mapping. Such dependencies are noted in this chapter, but OMG IDL will be used to describe the interface.

All types defined in this chapter are part of the CORBA module. When referenced in OMG IDL, the type names must be prefixed by “CORBA::”.

8.1 Role of the Basic Object Adapter

One object adapter, called the Basic Object Adapter, should be available in every ORB implementation; although the BOA will generally have an ORB-dependent implementation, object implementations that use it should be able to run on any ORB that supports the required language mapping, assuming they have been installed appropriately.

Other Object Adapters are likely to be created. Ordinarily, it is not necessary for a client of an object to be concerned about which Object Adapter is used by the implementation.

The following functions are provided through the Basic Object Adapter:

- Generation and interpretation of object references
- Authentication of the principal making the call
- Activation and deactivation of the implementation
- Activation and deactivation of individual objects
- Method invocation through skeletons

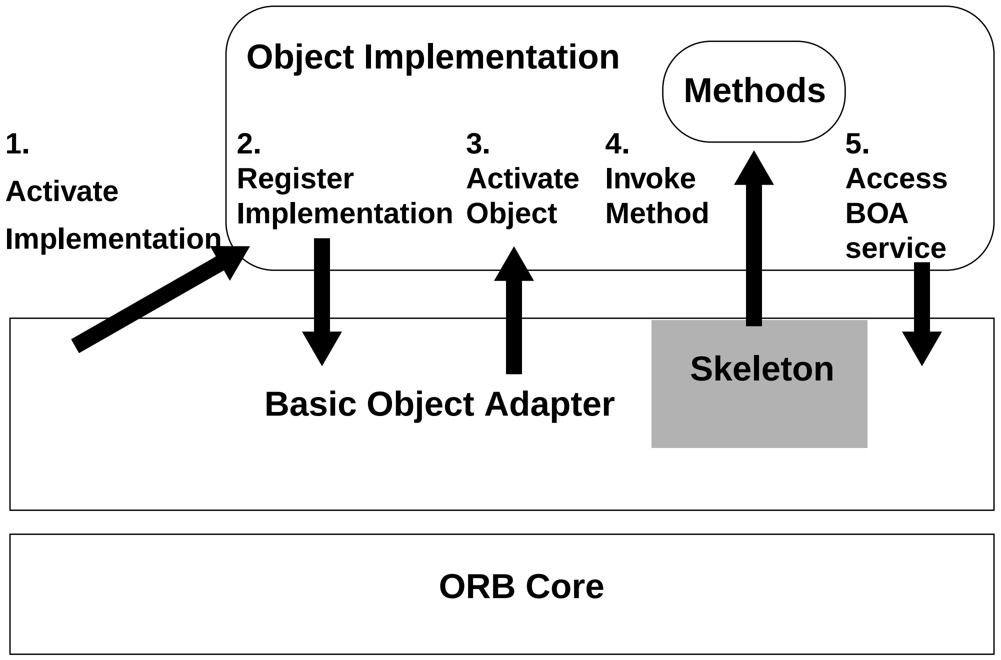
The Basic Object Adapter supports object implementations that are constructed from one or more programs[1]. The BOA activates and communicates with these programs using operating system facilities that are not part of the ORB. Therefore the BOA requires some information that is inherently non-portable. Although not defining this information, the BOA does define the concept of an Implementation Repository which can hold this information, allowing each system to install and start implementations in the way that is appropriate for that system.

The mechanism for binding the program to the BOA and ORB is also not specified because it is inherently system and language-dependent. We assume that the BOA can connect the methods to the skeleton by some means, whether at the time the implementation is compiled, installed, or activated, etc. Subsequent to activation, the BOA can make calls on routines in the implementation and the implementation can make calls on the BOA.

Figure 14 on page 8-3 shows the structure of the Basic Object Adapter, and some of the interactions between the BOA and an Object Implementation. The Basic Object Adapter will start a program to provide the Object Implementation, in this example, a per-class server (1). The Object Implementation notifies the BOA that it has finished initializing and is prepared to handle requests (2). When the first request for a particular object arrives, the implementation is notified to activate the object (3). On subsequent requests, the BOA calls the appropriate method using the per-interface skeleton (4). At various times, the implementation may access BOA services such as object creation, deactivation, and so forth. (5).

1.The term “program” is meant to include a wide range of possible constructs, including scripts, loadable modules, etc., in addition to the traditional notions of an application or server.

FIGURE 14. **The Structure and Operation of the Basic Object Adapter**



The BOA exports operations that are accessed by the Object Implementation. The BOA also calls the Object Implementation under certain circumstances. The interface between a particular version of the BOA and the ORB Core it runs on is private, as is the interface between the BOA and the skeletons. Thus, the BOA can exploit features or overcome limitations of a specific ORB Core, and can cooperate with the ORB Core and skeletons to provide a set of portable interfaces for the object implementation.

8.2 Basic Object Adapter Interface

The BOA interface is specified in OMG IDL, so that the way it is accessed in any programming language is specified by the client side language mapping for that language. Some data structures used by the BOA are specific to a given language mapping, so most IDL compilers will not be able to accept this definition literally.

In practice, the BOA is most likely to be implemented partially as a separate component and partially as a library in the Object Implementation. The separate component is required to do activation when the implementation is not present. The library portion is needed to establish the linkage between the methods and the skeleton. The exact partitioning of functionality between these parts is implementation dependent. Generally, there will appear to be a BOA object in the object implementation. When it is invoked, some operations are satisfied in the library, some in an external server, and some in the ORB Core.

The following is the approximate interface definition for the BOA object. More details will be provided as the operations are discussed.

```
module CORBA {

    interface InterfaceDef;             // from Interface Repository   // PIDL
    interface ImplementationDef;        // from Implementation Repository
    interface Object;                   // an object reference
    interface Principal;                // for the authentication service
    typedef sequence <octet, 1024> ReferenceData;

    interface BOA {
        Object create (
            in ReferenceData            id,
            in InterfaceDef             intf,
            in ImplementationDef        impl
        );
        void dispose (in Object obj);
        ReferenceData get_id (in Object obj);

        void change_implementation (
            in Object                   obj,
            in ImplementationDef        impl
        );

        Principal get_principal (
            in Object                   obj,
            in Environment              ev
        );

        void set_exception (
            in exception_type           major,      // NO, USER,
                                                    //or SYSTEM_EXCEPTION
            in string                   userid,     // exception type id
            in void                     *param      // pointer to associated data
        );

        void impl_is_ready (in ImplementationDef impl);
        void deactivate_impl (in ImplementationDef impl);
        void obj_is_ready (in Object obj, in ImplementationDef impl);
        void deactivate_obj (in Object obj);
};

            };
```

Requests by an implementation on the BOA are of the following kinds:

- Operations to create or destroy object references, or query or update the information the BOA maintains for an object reference.
- Operations associated with a particular request.
- Operations to maintain a registry of active objects and implementations.

Requests by the BOA to an implementation are made with skeletons or using an implementation's run-time language mapping information, and are of these kinds:

- Activating an implementation.
- Activating an object.
- Performing an operation (through a skeleton method).

Each of the BOA operations is described in detail later in this section; the requests of the BOA to an implementation are described in the language mapping section.

8.2.1 Registration of Implementations

The Basic Object Adapter expects information describing the implementations to be stored in an *Implementation Repository.* The Implementation Repository ordinarily is updated at program installation time, but may be set up incrementally or otherwise. There are objects with an OMG IDL interface called **ImplementationDef**, which capture this information. The Implementation Repository may contain additional information for debugging, administration, etc. Note that the Implementation Repository is logically distinct from the Interface Repository, although they may in fact be implemented together.

The *Interface Repository* contains information about interfaces. There are objects with an OMG IDL interface called **InterfaceDef**, which capture this information. The Interface Repository may contain additional information for debugging, administration, browsing, etc. The ORB Core may or may not make use of the Interface Repository or the Implementation Repository, but the ORB and BOA use these objects to associate object references with their interfaces and implementations.

8.2.2 Activation and Deactivation of Implementations

There are two kinds of activation that a BOA needs to perform as part of operation invocation. The first, discussed in this section, is *implementation activation*, which occurs when no implementation for an object is currently available to handle the request. The second, discussed later, is *object activation*, which occurs when no instance of the object is available to handle the request.

Implementation activation requires coordination between the BOA and the program(s) containing the implementation. This manual uses the term *server* as the separately executable entity that the BOA can start on a particular system. In a POSIX environment, a server would be a process. In most systems, a server corresponds to the notion of a program, but it can correspond to whatever the appropriate system facility is in a particular environment.

The BOA initiates activity by the implementation by starting the appropriate server, probably in an operating system-dependent way. The implementation initializes itself, then notifies the BOA that it is prepared to handle requests by calling **impl_is_ready** or **obj_is_ready**[2].

Between the time that the program is started and it indicates it is ready, the BOA will prevent any other requests from being delivered to the server. After that point, the BOA, through the skeletons, will make calls on the methods of the implementation.

```
void impl_is_ready (in ImplementationDef impl);                    // PIDL
void obj_is_ready (
    in Object               obj,
    in ImplementationDef    impl

                ); 
```

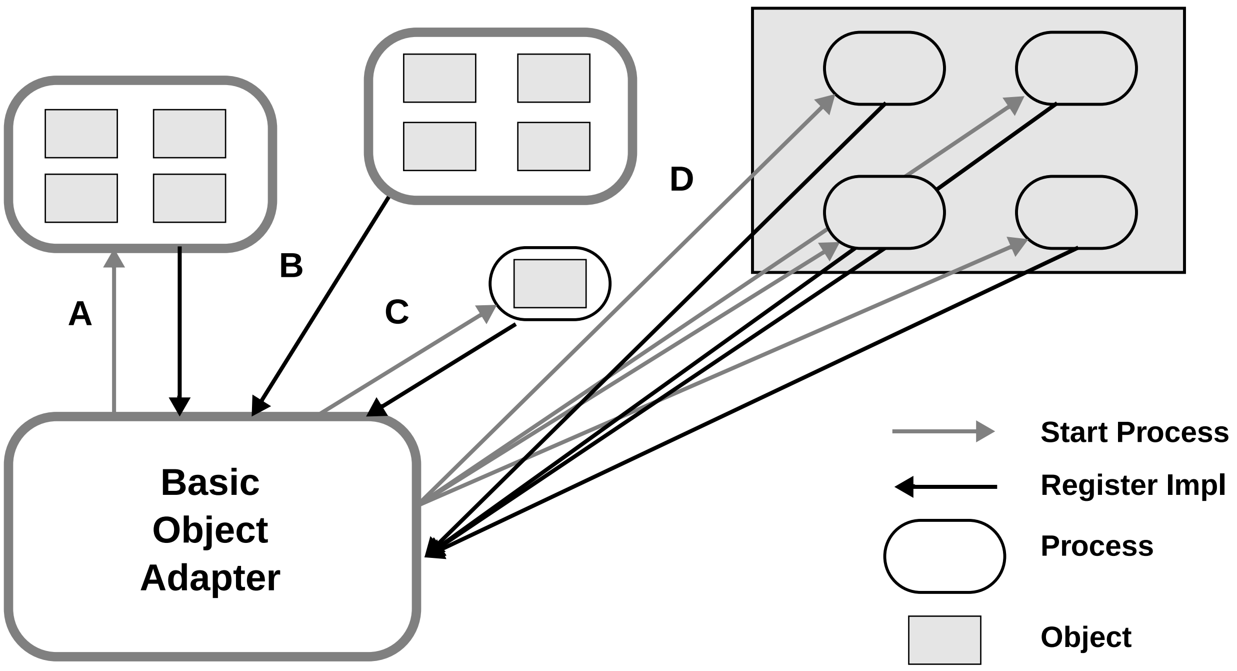
An *activation policy* describes the rules that a given implementation follows when there are multiple objects or implementations active. There are four policies that all BOA implementations support for implementation activation:

- A *shared server* policy, where multiple active objects of a given implementation share the same server.
- An *unshared server* policy, where only one object of a given implementation at a time can be active in one server.
- A *server-per-method* policy, where each invocation of a method is implemented by a separate server being started, with the server terminating when the method completes.
- *Persistent server* policy, where the server is activated by something outside the BOA. The server nonetheless must register with the BOA to receive invocations. A persistent server is assumed to be shared by multiple active objects.

These kinds of implementation activation are illustrated in Figure 15 on page 8-7. Case A is a shared server, where the BOA starts a process which then registers itself with the BOA. Case B is the case of a persistent server, which is very similar but just registers itself with the BOA, without the BOA having had to start a process. An unshared server is illustrated in case C, where the process started by the BOA can only hold one object; the server-per-method policy in case D causes each method invocation to be done by starting a process.

2.The latter is for per-object servers.

FIGURE 15. **Implementation Activation Policies**



Shared Server Activation Policy

In a shared server, multiple objects may be implemented by the same program. This is likely to be the most common kind of server. The server is activated the first time a request is performed on any object implemented by that server. When the server has initialized itself, it notifies the BOA that it is ready by calling **impl_is_ready**. Subsequently, the BOA will deliver requests or object activations for any objects implemented by that server. The server remains active and will receive requests until it calls **deactivate_impl**. The BOA will not activate another server for that implementation if one is active.

Before the first request is delivered for a particular object, the object activate routine of the server is called. An object remains active as long as its server is active, unless the server calls **deactivate_obj** for that object.

Unshared Server Activation Policy

In an unshared server, each object is implemented in a different server. This kind of server is convenient if a object is intended to encapsulate an application or if the server requires exclusive access to a resource such as a printer. A new server is activated the first time a request is performed on the object. When the server has initialized itself, it notifies the BOA that it is ready by calling **obj_is_ready**. Subsequently, the BOA will deliver requests for that object. The server remains active and will receive requests until it calls **deactivate_obj**.

A new server is started whenever a request is made for an object that is not yet active, even if a server for another object with the same implementation is active.

Server-per-Method Activation Policy

Under the server-per-method policy, a new server is always started each time a request is made. The server runs only for the duration of the particular method. Several servers for the same object or even the same method of the same object may be active simultaneously. Because a new server is started for each request, it is not necessary for the implementation to notify the BOA when an object is ready or deactivated.

The BOA activates an implementation for each request, whether or not another request for that operation, object, or implementation is active at the same time.

Persistent Server Activation Policy

Persistent servers are those servers which are activated by means outside the BOA. Such implementations notify the BOA that they are available using the **impl_is_ready** operation. Once the BOA knows about a persistent server, it treats the server as a shared server, sending it activations for individual objects and method calls. If no implementation is ready when a request arrives, an error is returned for that request.

8.2.3 Generation and Interpretation of Object References

Object references are generated by the BOA using the ORB Core when requested by an implementation. The BOA and the ORB Core work together to associate some information with a particular object reference. This information is later provided to the implementation upon the activation of an object. Note that this is the only information an implementation may use portably to distinguish different object references. The BOA operation used to create a new object reference is:

```
Object create (                                              // PIDL
    in ReferenceData                    id,
    in InterfaceDef                     intf,
    in ImplementationDef                impl

            );
```

The **id** is immutable identification information, chosen by the implementation at object creation time, and never changed during the lifetime of the object. The **intf** is the Interface Repository object that specifies the complete set of interfaces implemented by the object. The **impl** is the Implementation Repository object that specifies the implementation to be used for the object.

A typical implementation will use the **id** value to distinguish different objects, but it is free to use it in any way it chooses or to assign the same value to different object references. Two object references created with the same parameters are *not* the same object reference as far as the ORB is concerned, although the implementation may or may not treat them as references to the same object. Note that the object reference itself is opaque and may be different for different ORBs, but the **id** value is available portably in all ORBs. Only the implementation can normally interpret the **id** value. The operation to get the **id** is a BOA operation:

```
ReferenceData get_id (in Object obj);                    // PIDL
```

It is possible for the implementation associated with an object reference to be changed. This will cause subsequent requests to be handled according to the information in the new implementation. The operation to set the implementation is a BOA operation:

```
void change_implementation (                             // PIDL
    in Object                   obj,
    in ImplementationDef        impl

                );
```

Note – Care must be taken in order to change the implementation after the object has been created. There are issues of synchronization with activation, security, and whether or not the new implementation is prepared to handle requests for that object. The **change_implementation** operation affects all copies of that particular object reference.

If an object reference is copied, all copies have the same **id**, **intf**, and **impl.**

An implementation is allowed to dispose of an object it has created by asking the BOA to invalidate the object reference. The implementation is responsible for deallocating all other information about the object. After a **dispose** is done, the ORB Core and BOA act as if the object had never been created, and attempts to issue requests on any existing object references for that object will fail.

```
void dispose (in Object obj);                            // PIDL
```

Note that all of the operations on object references in this section may be done whether or not the object is active.

8.2.4 Authentication and Access Control

The BOA does not enforce any specific style of security management. It guarantees that for every method invocation (or object activation) it will identify the principal on whose behalf the request is performed. The object implementation can obtain this principal by the operation:

```
Principal get_principal (                                // PIDL
    in Object           obj,
    in Environment      ev

            );
```

The **obj** parameter is the object reference passed to the method. If another object is used the result is undefined. The **ev** parameter is the language-mapping-specific request environment passed to the method.

The meaning of the principal depends on the security environment that the implementation is running in. The decision of whether or not to permit a particular operation is left up to the implementation. Typically, an implementation will associate access rights with particular objects and principals, and will examine those access rights to determine if the principal making the request has the privileges required by the particular method. An implementation could store a reference to the access control information for an object in the **id** for the object.

8.2.5 Persistent Storage

Objects (or, more precisely, object references) are made persistent by the BOA and the ORB Core, in that a client that has an object reference can use it at any time without warning, even if the implementation has been deactivated or the system has been restarted. Although the ORB Core and BOA maintain the persistence of object references, the implementation must participate in keeping any data outside the ORB Core and BOA persistent.

Toward this end, the BOA provides a small amount of storage for an object in the **id** value. In most cases, this storage is insufficient and inconvenient for the complete state of the object. Instead, the implementation provides and manages that storage, using the **id** value to locate the actual storage. For example, the **id** value might contain the name of a file, or a key for a database system that holds the persistent state.

Standard OMG IDL Types A

The OMG IDL types listed in this appendix are available in all ORB implementations. IDL specifications that incorporate these types are therefore portable across ORB implementations.

TBL. 13 Types Defined by IDL

Type	**Described In**
short	“Integer Types” on page 3-21
long	“Integer Types” on page 3-21
unsigned short	“Integer Types” on page 3-21
unsigned long	“Integer Types” on page 3-21
float	“Floating-Point Types” on page 3-21
double	“Floating-Point Types” on page 3-21
char	“Char Type” on page 3-22
boolean	“Boolean Type” on page 3-22
octet	“Octet Type” on page 3-22
struct	“Structures” on page 3-23
union	“Discriminated Unions” on page 3-23
enum	“Enumerations” on page 3-24
sequence	“Sequences” on page 3-25
string	“String Literals” on page 3-8
array	“Arrays” on page 3-26
any	“Any Type” on page 3-22
Object	“Object Reference Operations” on page 7-2

TBL. 14 on page A-2 lists the ORB pseudo-objects that should be available in any language mapping; in the C mapping, these definitions are contained in the file orb.h. Pseudo-objects cannot be invoked with the dynamic interface, and do not have object references. Those pseudo-objects that cannot be used as general arguments (passed as arguments in requests on real objects) are identified in the table. The definitions of pseudo-objects that can be used as general arguments are contained in the file orb.idl, and can be **#included** into IDL specifications.

TBL. 14 Pseudo-objects

Name	General Argument?	In orb.idl?	Described In
Environment	No	No	[insert new c map ref]
Request	No	No	Section 4.2 on page 4-4
Context	No	No	Section 4.5 on page 4-12
ORB	No	No	Section 7.1 on page 7-1
BOA	No	No	Section 8.2 on page 8-3
TypeCode	Yes	Yes	Section 6.4.2 on page 6-5
Principal	Yes	Yes	Section 8.2.4 on page 8-9
NVList	No	No	Section 4.1.1 on page 4-1

Types used with the Interface Repository are shown in TBL. 14 on page A-2. They are contained in orb.idl.

TBL. 15 Interface Repository Types

Name	Type	Described In
Identifier	string	Section 6.6 on page 6-30
RepositoryId	string	Section 6.6 on page 6-30
OperationMode	enum	Section 6.5.21 on page 6-25
ParameterMode	enum	Section 6.5.21 on page 6-25
AttributeMode	enum	Section 6.5.20 on page 6-24
InterfaceDescription	struct	Section 6.5.22 on page 6-27
OperationDescription	struct	Section 6.5.21 on page 6-25
AttributeDescription	struct	Section 6.5.20 on page 6-24
ParameterDescription	struct	Section 6.5.21 on page 6-25
RepositoryDescription	struct	Section 6.5.6 on page 6-16
ModuleDescription	struct	Section 6.5.7 on page 6-17
ConstDescription	struct	Section 6.5.8 on page 6-17
ExceptionDescription	struct	Section 6.5.19 on page 6-23
TypeDescription	struct	Section 6.5.6 on page 6-16
FullInterfaceDescription	struct	Section 6.5.22 on page 6-27
InterfaceDef	interface	Section 6.5.22 on page 6-27
OperationDef	interface	Section 6.5.21 on page 6-25
AttributeDef	interface	Section 6.5.20 on page 6-24
ParameterDef	interface	Section 6.7 on page 6-33
RepositoryDef	interface	Section 6.5.6 on page 6-16
ModuleDef	interface	Section 6.5.7 on page 6-17
TypeDef	interface	Section 6.5.6 on page 6-16
ConstDef	interface	Section 6.5.8 on page 6-17

TBL. 15 Interface Repository Types *(Continued)*

Name	**Type**	**Described In**
ExceptionDef	interface	Section 6.5.19 on page 6-23
ImplementationDef	interface	Section 8.2.1 on page 8-5

The **any** type can be used to represent a variety of types of values. All ORB implementations must support all data types expressible in OMG IDL as **any** values.

Interoperability Overview 9

ORB interoperability specifies a comprehensive, flexible approach to supporting networks of objects that are distributed across and managed by multiple, heterogeneous CORBA-compliant ORBs. The approach to “interORBability” is universal, because its elements can be combined in many ways to satisfy a very broad range of needs.

9.1 Elements of Interoperability

The elements of interoperability are as follows:

- ORB interoperability architecture
- Inter-ORB bridge support
- General and Internet inter-ORB Protocols (GIOPs and IIOPs)

In addition, the architecture accommodates **environment-specific inter-ORB protocols (ESIOPs)** that are optimized for particular environments such as DCE.

9.1.1 ORB Interoperability Architecture

The ORB Interoperability Architecture provides a conceptual framework for defining the elements of interoperability and for identifying its compliance points. It also characterizes new mechanisms and specifies conventions necessary to achieve interoperability between independently produced ORBs.

Specifically, the architecture introduces the concepts of *immediate* and *mediated bridging* of ORB domains. The Internet inter-ORB Protocol (IIOP) forms the common basis for broad-scope mediated bridging. The inter-ORB bridge support can be used to implement both immediate bridges and to build “half-bridges” to mediated bridge domains.

By use of bridging techniques, ORBs can interoperate without knowing any details of that ORB's implementation, such as what particular IPC or protocols (such as ESIOPs) are used to implement the *CORBA* specification.

The IIOP may be used in bridging two or more ORBs by implementing "half bridges" which communicate using the IIOP. This approach works both for stand-alone ORBs, and for networked ones which use an ESIOP.

The IIOP may also be used to implement an ORB's internal messaging, if desired. Since ORBs are not required to use the IIOP internally, the goal of not requiring prior knowledge of each others' implementation is fully satisfied.

9.1.2 Inter-ORB Bridge Support

The interoperability architecture clearly identifies the role of different kinds of domains for ORB-specific information. Such domains can include object reference domains, type domains, security domains (e.g. the scope of a *Principal* identifier), a transaction domain, and more.

Where two ORBs are in the same domain, they can communicate directly. In many cases, this is the preferable approach. This is not always true, however, since organizations often need to establish local control domains.

When information in an invocation must leave its domain, the invocation must traverse a bridge. The role of a bridge is to ensure that content and semantics are mapped from the form appropriate to one ORB to that of another, so that users of any given ORB only see their appropriate content and semantics.

The inter-ORB bridge support element specifies ORB APIs and conventions to enable the easy construction of interoperability bridges between ORB domains. Such bridge products could be developed by ORB vendors, Sieves, system integrators or other third-parties.

Because the extensions required to support Inter-ORB Bridges are largely general in nature, do not impact other ORB operation, and can be used for many other purposes besides building bridges, they are appropriate for all ORBs to support. Other applications include debugging, interposing of objects, implementing objects with interpreters and scripting languages and dynamically generating implementations.

The inter-ORB bridge support can also be used to provide interoperability with non-CORBA systems, such as Microsoft's Component Object Model (COM). The ease of doing this will depend on the extent that those systems conform to the CORBA Object Model.

9.1.3 General Inter-ORB Protocol (GIOP)

The General Inter-ORB Protocol (GIOP) element specifies a standard transfer syntax (low-level data representation) and a set of message formats for communications between ORBs. The GIOP is specifically built for ORB to ORB interactions and is designed to work directly over any connection-oriented transport protocol that meets a minimal set of assumptions. It does not require or rely on the use of higher level RPC mechanisms. The protocol is simple (as simple as possible, but not simpler), scalable and relatively easy to implement. It is designed to allow portable implementations with small memory footprints and reasonable performance, with minimal dependencies on supporting software other than the underlying transport layer.

While versions of the GIOP running on different transports would not be directly interoperable, their commonality would allow easy and efficient bridging between such networking domains.

9.1.4 Internet Inter-ORB Protocol (IIOP)

The Internet Inter-ORB Protocol (IIOP) element specifies how GIOP messages are exchanged using TCP/IP connections. The IIOP specifies a standardized interoperability protocol for the Internet, providing “out of the box” interoperation with other compatible ORBs based on the most popular product- and vendor-neutral transport layer. It can also be used as the protocol between half-bridges (see below).

The protocol is designed to be suitable and appropriate for use by any ORB to interoperate in Internet Protocol domains unless an alternative protocol is necessitated by the specific design center or intended operating environment of the ORB. In that sense it represents the basic inter-ORB protocol for TCP/IP environments, a most pervasive transport layer.

The IIOP’s relationship to the GIOP is similar to that of a specific language mapping to OMG IDL; the GIOP may be mapped onto a number of different transports, and specifies the protocol elements that are common to all such mappings. The GIOP by itself, however, does not provide complete interoperability, just as IDL cannot be used to built complete programs. The IIOP, and other similar mappings to different transports, are concrete realizations of the abstract GIOP definitions, as shown in Figure 2-1.

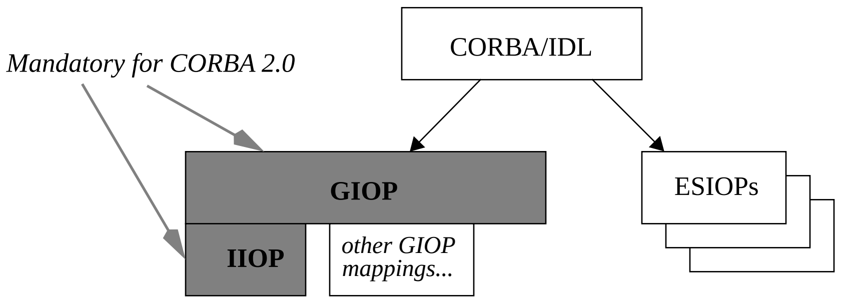


Figure 9-1 Inter-ORB Protocol Relationships.

9.1.5 Environment-Specific Inter-ORB Protocols (ESIOPs)

This specification also makes provision for an open ended set of Environment-Specific Inter-ORB Protocols (ESIOPs) Such protocols would be used for “out of the box” interoperation at user sites where a particular networking or distributing computing infrastructure is already in general use.

Because of the opportunity to leverage and build on facilities provided by the specific environment, ESIOPs might support specialized capabilities such as those relating to security and administration.

While ESIOPs may be optimized for particular environments, all ESIOP specifications will be expected to conform to the general ORB interoperability architecture conventions to enable easy bridging. The inter-ORB bridge support enables bridges to be built between ORB domains that use the IIOP and ORB domains that use a particular ESIOP.

9.2 Relationship to Previous Versions of CORBA

The ORB Interoperability Architecture builds on Common Object Request Broker Architecture by adding the notion of ORB Services, and their domains. (ORB Services are described in Section 10.2, ORBS and ORB Services. The architecture defines the problem of ORB interoperability in terms of bridging between those domains, and defines several ways in which those bridges can be constructed: the bridges can be internal (in-line) and external (request-level) to ORBs.

APIs included in the interoperability specifications include compatible extensions to previous versions of *CORBA* to support request level bridging:

- A Dynamic Skeleton Interface (DSI) is the basic support needed for building request level bridges; it is the server side analogue of the Dynamic Invocation Interface, and in the same way it has general applicability beyond bridging. For information about the Dynamic Skeleton Interface, refer to Chapter 5.
- APIs for managing object references have been defined, building on the support identified for the Relationship Service. The APIs are defined in “Object Reference Operations” on page 7-2. The Relationship Service is described in *CORBAservices: Common Object Service Specifications*; refer to Section 9.3.6, “The CosObjectIdentity Module.

9.3 Examples of Interoperability Solutions

The elements of interoperability (Inter-ORB Bridges, General and Internet Inter-ORB Protocols, Environment-Specific Inter-ORB Protocols) can be combined in a variety of ways to satisfy particular product and customer needs. This section provides some examples.

9.3.1 Example 1

ORB product A is designed to support objects distributed across a network and provide "out of the box" interoperability with compatible ORBs from other vendors. In addition it allows for bridges to be built between it and other ORBs that use environment-specific or proprietary protocols. To accomplish this, ORB A uses the IIOP and provides inter-ORB bridge support.

9.3.2 Example 2

ORB product B is designed to provide highly optimized, very high speed support for objects located on a single machine; for example, to support thousands of Fresco GUI objects operated on at near function-call speeds. In addition, some of the objects will need to be accessible from other machines and objects on other machines will need to be infrequently accessed. To accomplish this, ORB A provides a half-bridge to support the Internet IOP for communication with other "distributed" ORBs.

9.3.3 Example 3

ORB product C is optimized to work in a particular operating environment. It uses a particular environment-specific protocol based on distributed computing services that are commonly available at the target customer sites. In addition, ORB C is expected to interoperate with arbitrary other ORBs from other vendors. To accomplish this, ORB C provides inter-ORB bridge support and a companion half-bridge product (supplied by the ORB vendor or some third-party) provides the connection to other ORBs. The half-bridge uses the IIOP to enable interoperability with other compatible ORBs.

9.3.4 Interoperability Compliance

An ORB is considered to be interoperability-compliant when it meets the following requirements:

- In the CORBA Core part of this specification, standard APIs are provided by an ORB to enable the construction of request level inter-ORB bridges. APIs are defined by the Dynamic Invocation Interface, the Dynamic Skeleton Interface, and by the object identity operations, which are described in Chapter 7.
- An Internet Inter-ORB Protocol (IIOP) (explained in Chapter 12) defines a transfer syntax and message formats (described independently as the General Inter-ORB Protocol), and defines how to transfer messages via TCP/IP connections. The IIOP can be supported natively or via a half-bridge.

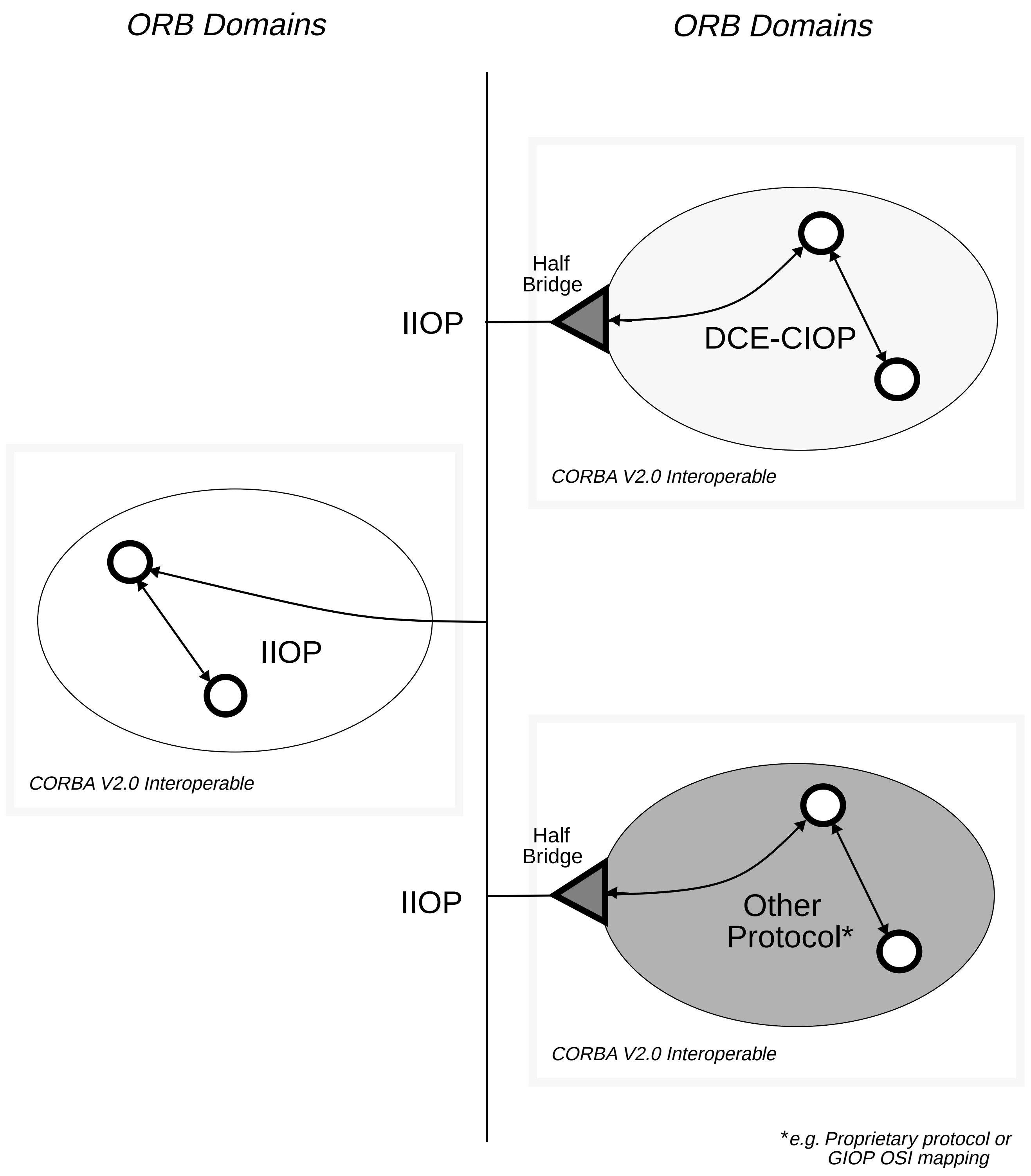
Support for additional ESIOPs and other proprietary protocols is optional in a interoperability-compliant system. However, any implementation that chooses to use the other protocols defined by the CORBA interoperability specifications (Chapters 9 - 13) must adhere to those specifications to be compliant with CORBA interoperability.

The illustration on page 9-7 shows examples of interoperable ORB domains that are CORBA-compliant.

These compliance points support a range of interoperability solutions. For example, the standard APIs may be used to construct “half bridges” to the IIOP, relying on another “half bridge” to connect to another ORB. The standard APIs also support construction of “full bridges”, without using the Internet IOP to mediate between separated bridge components. ORBs may also use the Internet IOP internally. In addition, ORBs may use GIOP messages to communicate over other network protocol families (such as Novell or OSI), and provide transport-level bridges to the IIOP.

The GIOP is described separately from the IIOP to allow future specifications to treat it as an independent compliance point. For additional information on CORBA compliance, refer to Definition of CORBA Compliance on page vii.

Examples of CORBA Version 2.0 Interoperability Compliance



9.4 Motivating Factors

This section explains the factors that motivated the creation of interoperability specifications.

9.4.1 ORB Implementation Diversity

Today, there are many different ORB products that address a variety of user needs. A large diversity of implementation techniques is evident. For example, the time for a request ranges over at least 5 orders of magnitude, from a few microseconds to several seconds. The scope ranges from a single application to enterprise networks. Some ORBs have high levels of security, others are more open. Some ORBs are layered on a particular widely used protocols, others use highly optimized, proprietary protocols.

The market for object systems and applications that use them will grow as object systems are able to be applied to more kinds computing. From application integration to process control, from loosely coupled operating systems to the information superhighway, CORBA-based object systems can be the common infrastructure.

9.4.2 ORB Boundaries

Even when it is not required by implementation differences, there are other reasons to partition an environment into different ORBs.

For security reasons, it may be important to know that it not generally possible to access objects in one domain from another. For example, an “internet ORB” may make public information widely available, but a “company ORB” will want to restrict what information can get out. Even if they used the same ORB implementation, these two ORBs would be separate, so that the company could allow access to public objects from inside the company without allowing access to private objects from outside. Even though individual objects should protect themselves, prudent system administrators will want to avoid exposing sensitive objects to attacks from outside the company.

Supporting multiple ORBs also helps handle the difficult problem of testing and upgrading the object system. It would be unwise to test new infrastructure without limiting the set of objects that might be damaged by bugs, and it may be impractical to replace “the ORB” everywhere simultaneously. A new ORB might be tested and deployed in the same environment, interoperating with the existing ORB until it either a complete switch is made or it incrementally displaces the existing one.

Management issues may also motivate partitioning an ORB. Just as networks are subdivided into domains to allow decentralized control of databases, configurations, resources, etc., management of the state in an ORB (object reference location and translation information, interface repositories, per-object data, etc.) might also be done by creating sub-ORBs.

9.4.3 ORBs Vary in Scope, Distance, and Lifetime

Even in a single computing environment produced by a single vendor, there are reasons why some of the objects an application might use would be in one ORB, and others in another ORB. Some objects and services are accessed over long distances, with more global visibility, longer delays, and less reliable communication. Other objects are nearby, are not accessed from elsewhere, and provide higher quality service. By deciding which ORB to use, an implementer sets expectations for the clients of the objects.

One ORB might be used to retain links to information that is expected to accumulate over decades, such as a library archives. Another ORB might be used to manage a distributed chess program in which the objects should all be destroyed when the game is over. Although while it is running, it makes sense for “chess ORB” objects to access the “archives ORB”, we would not expect the archives to try to keep a reference to the current board position.

9.5 Interoperability Design Goals

Because of the diversity in ORB implementations, multiple approaches to interoperability are required. Options identified in previous versions of *CORBA* include:

- *Protocol Translation*, where a gateway residing somewhere in the system maps requests from the format used by one ORB to that used by another;
- *Reference Embedding*, where invocation using a native object reference delegates to a special object whose job it is to forward that invocation to another ORB;
- *Alternative ORBs*, where ORB implementations agree to coexist in the same address space so easily that a client or implementation can transparently use any of them, and pass object references created by one ORB to another ORB without losing functionality.

In general, there is no single protocol that can meet everyone's needs, and there is no single means to interoperate between two different protocols. There are many environments in which multiple protocols exist, and there are ways to bridge between environments that share no protocols.

This specification adopts a flexible architecture that allows a wide variety of ORB implementations to interoperate and that includes both bridging and common protocol elements.

The following goals guided the creation of interoperability specifications:

- The architecture and specifications should allow high performance, small footprint, lightweight interoperability solutions.
- The design should scale, should be not unduly difficult to implement and should not unnecessarily restrict implementation choices.
- Interoperability solutions should be able to work with any vendors’ existing ORB implementations, with respect to their CORBA compliant core feature set; those implementations are diverse.

- All operations implied by the CORBA object model (i.e. the stringify and destringify operations defined on the `CORBA:ORB` pseudo-object, and all the operations on `CORBA:Object`) as well as type management (e.g. narrowing, as needed by the C++ mapping) should be supported.

9.5.1 Non-Goals

The following were taken into account, but were not goals:

- Support for security
- Support for future ORB Services

ORB Interoperability Architecture 10

This chapter provides the architectural framework used in the interoperability specifications (Chapters 9–13).

10.1 Overview

The original Request for Proposal on Interoperability (OMG Document 93-9-15) defines interoperability as the ability for a client on ORB A to invoke an OMG IDL-defined operation on an object on ORB B, where ORB A and ORB B are independently developed. It further identifies general requirements including in particular:

- Ability for two vendors' ORBs to interoperate without prior knowledge of each other's implementation.
- Support of all ORB functionality.
- Preservation of content and semantics of ORB-specific information across ORB boundaries (for example, security).

In effect, the requirement is for invocations between client and server objects to be independent of whether they are on the same or different ORBs, and not to mandate fundamental modifications to existing ORB products.

10.1.1 Domains

The CORBA Object Model identifies various distribution transparencies that must be supported within a single ORB environment, such as location transparency. Elements of ORB functionality often correspond directly to such transparencies. Interoperability can be viewed as extending transparencies to span multiple ORBs.

In this architecture a *domain* is a distinct scope, within which certain common characteristics are exhibited and common rules are observed: over which a distribution transparency is preserved. Thus, interoperability is fundamentally involved with transparently crossing such domain boundaries.

Domains tend to be either administrative or technological in nature, and need not correspond to the boundaries of an ORB installation. Administrative domains include naming domains, trust groups, resource management domains and other `run-time' characteristics of a system. Technology domains identify common protocols, syntaxes and similar `build-time' characteristics. In many cases, the need for technology domains derives from basic requirements of administrative domains.

Within a single ORB, most domains are likely to have similar scope to that of the ORB itself: common object references, network addresses, security mechanisms, and more. However, it is possible for there to be multiple domains of the same type supported by a given ORB: internal representation on different machine types, or security domains. Conversely, a domain may span several ORBs: similar network addresses may be used by different ORBs, type identifiers may be shared.

10.1.2 Bridging Domains

The abstract architecture describes ORB interoperability in terms of the translation required when an object request traverses domain boundaries. Conceptually, a mapping or *bridging mechanism* resides at the boundary between the domains, transforming requests expressed in terms of one domain's model into the model of the destination domain.

The concrete architecture identifies two approaches to inter-ORB bridging:

- At application level, allowing flexibility and portability
- At ORB level, built into the ORB itself

10.2 ORBs and ORB Services

The ORB Core is that part of the ORB which provides the basic representation of objects and the communication of requests. The ORB Core therefore supports the minimum functionality to enable a client to invoke an operation on a server object, with (some of) the distribution transparencies required by *CORBA 2.0*.

An object request may have implicit attributes which affect the way in which it is communicated - though not the way in which a client makes the request. These attributes include security, transactional capabilities, recovery and replication. These features are provided by "ORB Services", which will in some ORBs be layered as internal services over the core, or in other cases incorporated directly into an ORB's core. It is an aim of this specification to allow for new ORB Services to be defined in the future, without the need to modify or enhance this architecture.

Within a single ORB, ORB services required to communicate a request will be implemented and (implicitly) invoked in a private manner. For interoperability between ORBs, the ORB services used in the ORBs, and the correspondence between them, must be identified.

10.2.1 The Nature of ORB Services

ORB Services are invoked implicitly in the course of application-level interactions. ORB Services range from fundamental mechanisms such as reference resolution and message encoding to advanced features such as support for security, transactions or replication.

An ORB Service is often related to a particular transparency. For example, message encoding – the marshaling and unmarshaling of the components of a request into and out of message buffers – provides transparency of the representation of the request. Similarly, reference resolution supports location transparency. Some transparencies, such as security, are supported by a combination of ORB Services and Object Services while others, such as replication, may involve interactions between ORB Services themselves.

ORB Services differ from Object Services in that they are positioned below the application and are invoked transparently to the application code. However, many ORB Services include components which correspond to conventional Object Services in that they are invoked explicitly by the application.

Security is an example of service with both ORB Service and normal Object Service components, the ORB components being those associated with transparently authenticating messages and controlling access to objects while the necessary administration and management functions resemble conventional Object Services.

10.2.2 ORB Services and Object Requests

Interoperability between ORBs extends the scope of distribution transparencies and other request attributes to span multiple ORBs. This requires the establishment of relationships between supporting ORB Services in the different ORBs.

In order to discuss how the relationships between ORB Services are established, it is necessary to describe an abstract view of how an operation invocation is communicated from client to server object.

- The client generates an operation request, using a reference to the server object, explicit parameters, and an implicit invocation context.
- This is processed by certain ORB Services on the client path;
- On the server side, corresponding ORB Services process the incoming request, transforming it into a form directly suitable for invoking the operation on the server object.
- The server object performs the requested operation.
- Any result of the operation is returned to the client in a similar manner.

The correspondence between client-side and server-side ORB Services need not be one-to-one and in some circumstances may be far more complex. For example, if a client application requests on operation on a replicated server, there may be multiple server-side ORB service instances, possibly interacting with each other.

In other cases, such as security, client-side or server-side ORB Services may interact with Object Services such as authentication servers.

10.2.3 Selection of ORB Services

The ORB Services used are determined by:

- Static properties of both client and server objects; for example, whether a server is replicated;
- Dynamic attributes determined by a particular invocation context; for example, whether a request is transactional;
- Administrative policies; for example, security.

Within a single ORB, private mechanisms (and optimizations) can be used to establish which ORB Services are required and how they are provided. Service selection might in general require negotiation to select protocols or protocol options. The same is true between different ORBs: it is necessary to agree which ORB Services are used, and how each transforms the request. Ultimately, these choices become manifest as one or more protocols between the ORBs, or as transformations of requests.

In principle, agreement on the use of each ORB Service can be independent of the others and, in appropriately constructed ORBs, services could be layered in any order or in any grouping. This potentially allows applications to specify selective transparencies according to their requirements, although at this time CORBA provides no ways to penetrate its transparencies.

A client ORB must be able to determine which ORB Services must be used in order to invoke operations on a server object. Correspondingly, where a client requires dynamic attributes to be associated with specific invocations, or administrative policies dictate, it must be possible to cause the appropriate ORB Services to be used on client and server sides of the invocation path. Where this is not possible - because, for example, one ORB does not support the full set of services required - either the interaction cannot proceed or it can only do so with reduced facilities or transparencies.

10.3 Domains

From a computational viewpoint, the OMG Object Model identifies various distribution transparencies which ensure that client and server objects are presented with a uniform view of a heterogeneous distributed system. From an engineering viewpoint, however, the system is not wholly uniform. There may be distinctions of location and possibly many others such as processor architecture, networking mechanisms and data representations. Even when a single ORB implementation is used throughout the system, local instances may represent distinct, possibly optimized scopes for some aspects of ORB functionality.

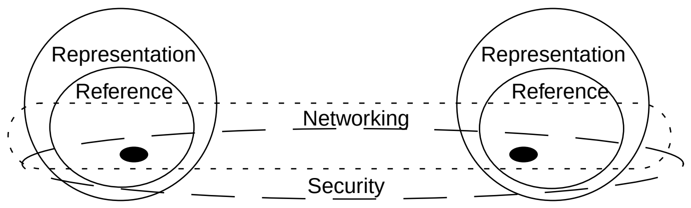


Figure 10-1 Different Kinds of Domains can Coexist.

Interoperability, by definition, introduces further distinctions, notably between the scopes associated with each ORB. To describe both the requirements for interoperability and some of the solutions, this architecture introduces the concept of *domains* to describe the scopes and their implications.

Informally, a domain is a set of objects sharing a common characteristic or abiding by common rules. It is a powerful modelling concept which can simplify the analysis and description of complex systems. There may be many types of domains, for example, management domains, naming domains, language domains, technology domains.

10.3.1 Definition of a Domain

Domains allow partitioning of systems into collections of components which have some characteristic in common. In this architecture a domain is a scope in which a collection of objects, said to be members of the domain, is associated with some common characteristic; any object for which the association does not exist, or is undefined, is not a member of the domain. A domain can be modelled as an object and may be itself a member of other domains.

It is the scopes themselves and the object associations or bindings defined within them which characterize a domain.This information is disjoint between domains. However, an object may be a member of several domains, of similar kinds as well as of different kinds, and so the sets of members of domains may overlap.

The concept of a domain boundary is defined as the limit of the scope in which a particular characteristic is valid or meaningful. When a characteristic in one domain is translated to an equivalent in another domain, it is convenient to consider it as traversing the boundary between the two domains.

Domains are generally either administrative or technological in nature. Examples of domains related to ORB interoperability issues are:

- Referencing domain – the scope of an object reference
- Representation domain – the scope of a message transfer syntax and protocol
- Network addressing domain – the scope of a network address
- Network connectivity domain – the potential scope of a network message
- Security domain – the extent of a particular security policy

- Type domain – the scope of a particular type identifier
- Transaction domain – the scope of a given transaction service

Domains can be related in two ways: containment, where a domain is contained within another domain, and federation, where two domains are joined in a manner agreed and set up by their administrators.

10.3.2 Mapping Between Domains: Bridging

Interoperability between domains is only possible if there is a well-defined mapping between the behaviors of the domains being joined. Conceptually, a mapping mechanism or bridge resides at the boundary between the domains, transforming requests expressed in terms of one domain's model into the model of the destination domain. Note that the use of the term "bridge" in this context is conceptual and refers only to the functionality which performs the required mappings between distinct domains. There are several implementation options for such bridges and these are discussed elsewhere.

For full interoperability, it is essential that all the concepts used in one domain are transformable into concepts in other domains with which interoperability is required, or that if the bridge mechanism filters such a concept out, nothing is lost as far as the supported objects are concerned. In other words, one domain may support a superior service to others, but such a superior functionality will not be available to an application system spanning those domains.

A special case of this requirement is that the object models of the two domains need to be compatible. This specification assumes that both domains are strictly compliant with the CORBA Object Model and the *CORBA V2.0* Core specifications. This includes the use of OMG IDL when defining interfaces, the use of the CORBA Core Interface Repository, and other modifications that were made to *CORBA V1.2*. Variances from this model could easily compromise some aspects of interoperability.

10.4 Interoperability Between ORBs

An ORB "provides the mechanisms by which objects transparently make and receive requests and responses. In so doing, the ORB provides interoperability between applications on different machines in heterogeneous distributed environments..." ORB interoperability extends this definition to cases in which client and server objects on different ORBs "transparently make and receive requests..."

Note that a direct consequence of this transparency requirement is that bridging must be bidirectional: that is, it must work as effectively for object references passed as parameters as for the target of an object invocation. Were bridging unidirectional (e.g. if one ORB could only be a client to another) then transparency would not have been provided, because object references passed as parameters would not work correctly: ones passed as "callback objects", for example, could not be used.

Without loss of generality, most of this specification focuses on bridging in only one direction. This is purely to simplify discussions, and does not imply that unidirectional connectivity satisfies basic interoperability requirements.

10.4.1 ORB Services and Domains

In this architecture, different aspects of ORB functionality - ORB Services - can be considered independently and associated with different domain types. The architecture does not, however, prescribe any particular decomposition of ORB functionality and interoperability into ORB Services and corresponding domain types. There is a range of possibilities for such a decomposition:

1. The simplest model, for interoperability, is to treat all objects supported by one ORB (or, alternatively, all ORBs of a given type) as comprising one domain. Interoperability between any pair of different domains (or domain types) is then achieved by a specific all-encompassing bridge between the domains. (This is all *CORBA V2.0* implies.)
2. More detailed decompositions would identify particular domain types - such as referencing, representation, security and networking. A core set of domain types would be pre-determined and allowance made for additional domain types to be defined as future requirements dictate (e.g. for new ORB Services).

10.4.2 ORBs and Domains

In many respects, issues of interoperability between ORBs are similar to those which can arise with a single type of ORB (e.g. a product). For example:

- Two installations of the ORB may be installed in different security domains, with different Principal identifiers. Requests crossing those security domain boundaries will need to establish locally meaningful Principals for the caller identity, and for any Principals passed as parameters.
- Different installations might assign different type identifiers for equivalent types, and so requests crossing type domain boundaries would need to establish locally meaningful type identifiers (and perhaps more).

Conversely, not all of these problems need to appear when connecting two ORBs of a different type (e.g. two different products). Examples include:

- They could be administered to share user visible naming domains, so that naming domains do not need bridging.
- They might reuse the same networking infrastructure, so that messages could be sent without needing to bridge different connectivity domains.

Additional problems can arise with ORBs of different types. In particular, they may support different concepts or models, between which there are no direct or natural mappings. CORBA only specifies the application level view of object interactions, and requires that distribution transparencies conceal a whole range of lower level issues. It follows that within any particular ORB, the mechanisms for supporting transparencies are not visible at the application level and are entirely a matter of implementation choice. So there is no guarantee that any two ORBs support similar internal models or that there is necessarily a straightforward mapping between those models.

These observations suggest that the concept of an ORB (instance) is too coarse or superficial to allow detailed analysis of interoperability issues between ORBs. Indeed, it becomes clear that an ORB instance is an elusive notion: it can perhaps best be characterized as the intersection or coincidence of ORB Service domains.

10.4.3 Interoperability Approaches

When an interaction takes place across a domain boundary, a mapping mechanism, or bridge, is required to transform relevant elements of the interaction as they traverse the boundary. There are essentially two approaches to achieving this: mediated bridging and immediate bridging. These approaches are described in the following subsections.

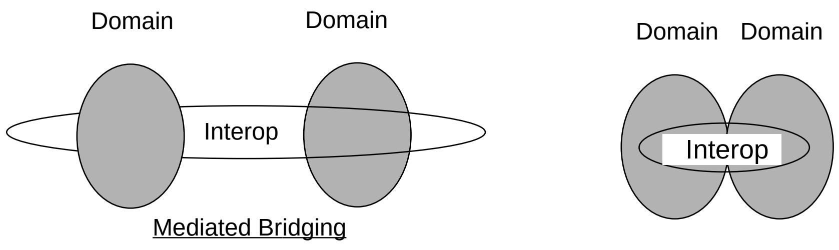


Figure 10-2 Two bridging techniques, different uses of an intermediate form agreed on between the two domains.

Mediated Bridging

With mediated bridging, elements of the interaction relevant to the domain are transformed, at the boundary of each domain, between the internal form of that domain and an agreed, common form.

Observations on mediated bridging are as follows:

- The scope of agreement of a common form can range from a private agreement between two particular ORB/domain implementations to a universal standard;
- There can be more than one common form, each oriented or optimized for a different purpose;
- If there is more than one possible common form, then selection of which is used can be static (e.g. administrative policy agreed between ORB vendors, or between system administrators) or dynamic (e.g. established separately for each object, or on each invocation);
- Engineering of this approach can range from in-line specifically compiled (compare to stubs) or generic library code (such as encryption routines) code, to intermediate bridges to the common form.

Immediate Bridging

With immediate bridging, elements of the interaction relevant to the domain are transformed, at the boundary of each domain, directly between the internal form of one domain and the internal form of the other.

Observations on immediate bridging are as follows:

- This approach has the potential to be optimal (in that the interaction is not mediated via a third party, and can be specifically engineered for each pair of domains) but sacrifices flexibility and generality of interoperability to achieve this;
- This approach is often applicable when crossing domain boundaries which are purely administrative (i.e. there is no change of technology). For example, when crossing security administration domains between similar ORBs, it is not necessary to use a common intermediate standard.

As a general observation, the two approaches can become almost indistinguishable when private mechanisms are used between ORB/domain implementations.

Location of Inter-Domain Functionality

Logically, an inter-domain bridge has components in both domains, whether the mediated or immediate bridging approach is used. However, domains can span ORB boundaries and ORBs can span machine and system boundaries; conversely, a machine may support, or a process may have access to more than one ORB (or domain of a given type). From an engineering viewpoint, this means that the components of an inter-domain bridge may be dispersed or co-located, with respect to ORBs or systems. It also means that the distinction between an ORB and a bridge can be a matter of perspective: there is a duality between viewing inter-system messaging as belonging to ORBs, or to bridges.

For example, if a single ORB encompasses two security domains, the inter-domain bridge could be implemented wholly within the ORB and thus be invisible as far as ORB interoperability is concerned. A similar situation arises when a bridge between two ORBs or domains is implemented wholly within a process or system which has access to both. In such cases, the engineering issues of inter-domain bridging are confined, possibly to a single system or process. If it were practical to implement all bridging in this way, then interactions between systems or processes would be solely within a single domain or ORB.

Bridging Level

As noted at the start of this section, bridges may be implemented both internally to an ORB and as layers above it. These are called respectively “in-line” and “request-level” bridges.

Request level bridges use the CORBA APIs, including the Dynamic Skeleton Interface, to receive and issue requests. However, there is an emerging class of “implicit context” which may be associated with some invocations, holding ORB

Service information such as transaction and security context information, which is not at this time exposed through general purpose public APIs. (Those APIs expose only OMG IDL-defined operation parameters, not implicit ones.) Rather, the precedent set with the Transaction Service is that special purpose APIs are defined to allow bridging of each kind of context. This means that request level bridges must be built to specifically understand the implications of bridging such ORB Service domains, and to make the appropriate API calls.

10.4.4 Policy-Mediated Bridging

An assumption made through most of this specification is that the existence of domain boundaries should be transparent to requests: that the goal of interoperability is to hide such boundaries. However, if this were always the goal, then there would be no real need for those boundaries in the first place.

Realistically, administrative domain boundaries exist because they reflect ongoing differences in organizational policies or goals. Bridging the domains will in such cases require *policy mediation*. That is, inter-domain traffic will need to be constrained, controlled, or monitored; fully transparent bridging may be highly undesirable. Resource management policies may even need to be applied, restricting some kinds of traffic during certain periods.

Security policies are a particularly rich source of examples: a domain may need to audit external access, or to provide domain-based access control. Only a very few objects, types of objects, or classifications of data might be externally accessible through a “firewall”.

Such policy-mediated bridging requires a bridge that knows something about the traffic being bridged. It could in general be an application-specific policy, and many policy-mediated bridges could be parts of applications. Those might be organization-specific, off-the-shelf, or anywhere in between.

Request-level bridges, which use only public ORB APIs, easily support the addition of policy mediation components, without loss of access to any other system infrastructure that may be needed to identify or enforce the appropriate policies.

10.4.5 Configurations of Bridges in Networks

In the case of network-aware ORBs, we anticipate that some ORB protocols will be more frequently bridged to than others, and so will begin to serve the role of “backbone ORBs”. (This is a role that the IIOP is specifically expected to serve.) This use of “backbone topology” is true both on a large scale and a small scale. While a large scale public data network provider could define its own backbone ORB, on a smaller scale, any given institution will probably designate one commercially available ORB as its backbone.

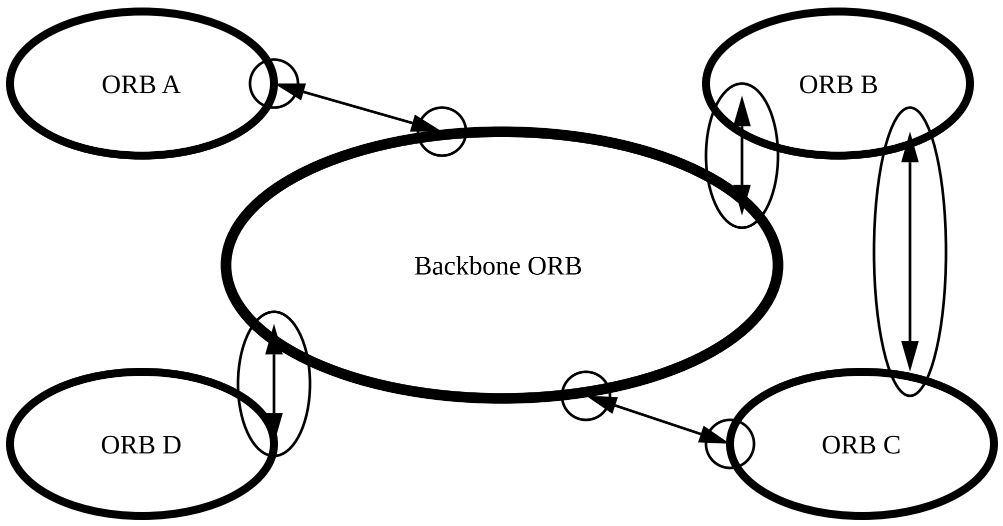


Figure 10-3 An ORB chosen as a backbone will connect other ORBs through bridges, both full-bridges and half-bridges.

Adopting a backbone style architecture is a standard administrative technique for managing networks. It has the consequence of minimizing the number of bridges needed, while at the same time making the ORB topology match typical network organizations. (That is, it allows the number of bridges to be proportional to the number of protocols, rather than combinatorial.)

In large configurations, it will be common to notice that adding ORB bridges doesn't even add any new "hops" to network routes, because the bridges naturally fit in locations where connectivity was already indirect, and augment or supplant the existing network firewalls.

10.5 Object Addressing

The Object Model, in "Requests" on page 1-2, defines an object reference as an object name that reliably denotes a particular object. An object reference identifies the same object each time the reference is used in a request, and an object may be denoted by multiple, distinct references.

The fundamental ORB interoperability requirement is to allow clients to use such object names to invoke operations on objects in other ORBs. Clients do not need to distinguish between references to objects in a local ORB or in a remote one. Providing this transparency can be quite involved, and naming models are fundamental to it.

This section of this specification discusses models for naming entities in multiple domains, and transformations of such names as they cross the domain boundaries. That is, it presents transformations of object reference information as it passes through networks of inter-ORB bridges. It uses the word “ORB” as synonymous with referencing domain; this is purely to simplify the discussion. In other contexts, “ORB” can usefully denote other kinds of domain.

10.5.1 Domain-relative Object Referencing

Since CORBA does not require ORBs to understand object references from other ORBs, when discussing object references from multiple ORBs one must always associate the object reference’s domain (ORB) with the object reference. We use the notation *D0.R0* to denote an object reference *R0* from domain *D0*; this is itself an object reference. This is called “domain-relative” referencing (or addressing), and need not reflect the implementation of object references within any ORB.

At an implementation level, associating an object reference with an ORB is only important at an inter-ORB boundary; that is, inside a bridge. This is simple, since the bridge knows from which ORB each request (or response) came, including any object references embedded in it.

10.5.2 Handling of Referencing Between Domains

When a bridge hands an object reference to an ORB, it must do so in a form understood by that ORB: the object reference must be in the recipient ORB’s native format. Also, in cases where that object originated from some other ORB, the bridge must associate each newly created “proxy” object reference with (what it sees as) the original object reference.

Several basic schemes to solve these two problems exist. These all have advantages in some circumstances; all can be used, and in arbitrary combination with each other, since CORBA object references are opaque to applications. The ramifications of each scheme merits attention, with respect to scaling and administration. The schemes include:

1. *Object Reference Translation Reference Embedding)*: The bridge can store the original object reference itself, and pass an entirely different proxy reference into the new domain. The bridge must then manage state on behalf of each bridged object reference, map these references from one ORB’s format to the other’s, and vice versa.

2. *Reference Encapsulation:* The bridge can avoid holding any state at all by conceptually concatenating a domain identifier to the object name. Thus if a reference *D0.R*, originating in domain *D0*, traversed domains *D1... D4* it could be identified in *D4* as proxy reference *d3.d2.d1.d0.R*, where *dn* is the address of *Dn* relative to *Dn+1*.

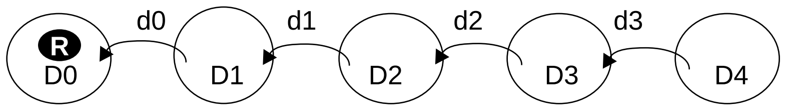


Figure 10-4 Reference encapsulation adds domain information during bridging.

3. *Domain Reference Translation:* Like object reference translation, this scheme holds some state in the bridge. However, it supports sharing that state between multiple object references by adding a domain-based route identifier to the proxy (which still holds the original reference, as in the reference encapsulation scheme).

   It achieves this by providing encoded domain route information each time a domain boundary is traversed; thus if a reference *D0.R*, originating in domain *D0*, traversed domains *D1...D4* it would be identified in *D4* as *(d3, x3).R*, and in *D2* as *(d1,x1).R*, and so on, where *dn* is the address of *Dn* relative to *Dn+1*, and *xn* identifies the pair *(dn-1, xn-1)*.

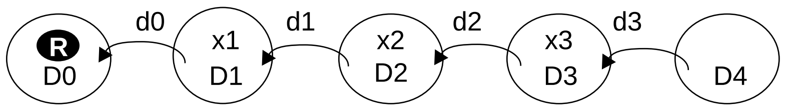


Figure 10-5 Domain Reference Translation *substitutes domain references* during bridging.

4. *Reference Canonicalization:* This scheme is like domain reference translation, except that the proxy uses a “well known” (e.g. global) domain identifier rather than an encoded path. Thus a reference *R*, originating in domain *D0* would be identified in other domains as *D0.R*.

Observations about these approaches to inter-domain reference handling are as follows:

- Naive application of reference encapsulation could lead to arbitrarily large references. A “topology service” could optimize cycles within any given encapsulated reference, and eliminate the appearance of references to local objects as alien references.
- A topology service could also optimize the chains of routes used in the domain reference translation scheme. Since the links in such chains are re-used by any path traversing the same sequence of domains, such optimization has particularly high leverage.

- With the general purpose APIs defined in *CORBA 2.0*, object reference translation can be supported even by ORBs not specifically intended to support efficient bridging, but this approach involves the most state in intermediate bridges. As with reference encapsulation, a topology service could optimize individual object references. (APIs are defined by the Dynamic Skeleton Interface, Dynamic Invocation Interface, and by the object identity operations described in Chapter 7.)
- The chain of addressing links established with both object and domain reference translation schemes must be represented as state within the network of bridges. There are issues associated with managing this state.
- Reference canonicalization can also be performed with managed hierarchical name spaces such as those now in use on the Internet, and X.500 naming.

10.6 An Information Model for Object References

This section provides a simple, powerful information model for the information found in an object reference. That model is intended to be used directly by developers of bridging technology, and is used in that role by the IIOP, described in “Object References” on page 12-15.

10.6.1 What Information Do Bridges Need?

The following potential information about object references has been identified as critical for use in bridging technologies:

- *Is it null?* Nulls only need to be transmitted, and never support operation invocation.
- *What type is it?* Many ORBs require knowledge of an object’s type in order to efficiently preserve the integrity of their type systems.
- *What protocols are supported?* Some ORBs support objrefs that in effect live in multiple referencing domains, to allow clients the choice of the most efficient communications facilities available.
- *What ORB Services are available?* As noted in Section 10.2.3, Selection of ORB Services, several different ORB Services might be involved in an invocation, and providing information about those services in a standardized way could in many cases reduce or eliminate negotiation overhead in selecting them.

10.6.2 Interoperable Object References: IORs

To provide the information above, an “Interoperable Object Reference”, or IOR, data structure has been provided. This data structure need not be used internally to any given ORB, and is not intended to be visible to application-level ORB programmers. It should be used only when crossing object reference domain boundaries, within bridges.

This data structure is designed to be efficient in typical single-protocol configurations, while not penalizing multiprotocol ones.

```
module IOP{                                             // IDL
    //
    // Standard Protocol Profile tag values
    //
    typedef unsigned long                   ProfileId;
    const ProfileId                         TAG_INTERNET_IOP = 0;
    const ProfileId                         TAG_MULTIPLE_COMPONENTS = 1;

    struct TaggedProfile {
        ProfileId                           tag;
        sequence <octet>                    profile_data;
    };

    //
    // an Interoperable Object Reference is a sequence of
    // object-specific protocol profiles, plus a type ID.
    //
    struct IOR {
        string                              type_id;
        sequence <TaggedProfile>            profiles;
    };

    //
    // Standard way of representing multicomponent profiles.
    // This would be encapsulated in a TaggedProfile.
    //
    typedef unsigned long ComponentId;
    struct TaggedComponent {
        ComponentId                             tag;
        sequence <octet>                    component_data;
    };
    typedef sequence <TaggedComponent> MultipleComponentProfile;
};
```

Object references have at least one *tagged profile* per protocol supported. Those profiles encapsulate all the basic information that protocol needs to identify an object. Any single profile holds enough information to drive a complete invocation using that protocol; the content and structure of those profile entries are wholly specified by that protocol. A bridge between two domains may need to know the detailed content of the profile for those domains' profiles, depending on the technique it uses to bridge the domains[1].

Each profile has a unique numeric tag, assigned by OMG. The ones defined here are for the IIOP (see Chapter 12, “General Inter-ORB Protocol”) and for use in “multi component protocol profiles.”

1.Based on topology and policy information available to it, a bridge may find it prudent to add or remove some profiles as it forwards an object reference. For example, a bridge acting as a firewall might remove all profiles except ones that make such profiles, letting clients that understand the profiles make routing choices.

The **TAG_MULTIPLE_COMPONENTS** tag indicates that the value encapsulated is of type **MultipleComponentProfile**. In this case, the profile consists of a list of protocol components, indicating ORB services accessible using that protocol. ORB services are assigned component identifiers in a name space that is distinct from the profile identifiers. Note that protocols may use the **MultipleComponentProfile** data structure to hold profile components even without using **TAG_MULTIPLE_COMPONENTS** to indicate that particular protocol profile, and need not use a **MultipleComponentProfile** to hold sets of profile components.

Null object references are indicated by an empty set of profiles, and by a Null type ID (a string which contains only a single terminating character). Type IDs may only be Null when the object reference is Null.The type ID is provided to allow ORBs to preserve strong typing; it is further explained in the description in the Interface Repository chapter. This identifier is agreed on within the bridge and, for reasons outside the scope of this interoperability specification, needs to have a much broader scope to address various problems in system evolution and maintenance. Type IDs support detection of type equivalence, and in conjunction with an Interface Repository, allow processes to reason about the relationship of the type of the object referred to and any other type.

The type ID is provided by the server and indicates the most derived type at the time the reference is generated.

10.6.3 Profile and Component Composition in IORs

The following rules augment the preceding discussion:

1. Profiles must be independent, complete, and self-contained. Their use shall not depend on information contained in another profile.
2. Any invocation uses information from exactly one profile.
3. Information used to drive multiple inter-ORB protocols may coexist within a single profile, possibly with some information (e.g. components) shared between the protocols, as specified by the specific protocols.
4. Unless otherwise specified in the definition of a particular profile, multiple profiles with the same profile tag may be included in an IOR.
5. Unless otherwise specified in the definition of a particular component, multiple components with the same component tag may be part of a given profile within an IOR.
6. A **TAG_MULTIPLE_COMPONENTS** profile may hold components shared between multiple protocols. Multiple such profiles may exist in an IOR.
7. The definition of each protocol using a **TAG_MULTIPLE_COMPONENTS** profile must specify which components it uses, and how it uses them.

8. Profile and component definitions can be either public or private. Public definitions are those whose tag and data format is specified in OMG documents. For private definitions, only the tag is registered with OMG.
9. Public component definitions shall state whether or not they are intended for use by protocols other than the one(s) for which they were originally defined, and dependencies on other components.

The OMG is responsible for allocating and registering protocol and component tags. Neither allocation nor registration indicates any "standard" status, only that the tag will not be confused with other tags. Requests to allocate tags should be sent to tag_request@omg.org

10.6.4 IOR Creation and Scope

IORs are created from object references when required to cross some kind of referencing domain boundary. ORBs will implement object references in whatever form they find appropriate, including possibly using the IOR structure. Bridges will normally use IORs to mediate transfers where that standard is appropriate.

10.6.5 Stringified Object References

Object references can be "stringified" (turned into an external string form) by the **ORB::object_to_string** operation, and then "destringified" (turned back into a programming environment's object reference representation) using the **ORB::string_to_object** operation.

There can be a variety of reasons why being able to parse this string form might *not* help make an invocation on the original object reference:

- Identifiers embedded in the string form can belong to a different domain than the ORB attempting to destringify the object reference.
- The ORBs in question might not share a network protocol, or be connected.
- Security constraints may be placed on object reference destringification.

Nonetheless, there is utility in having a defined way for ORBs to generate and parse stringified IORs, so that in some cases an object reference stringified by one ORB could be destringified by another.

To allow a stringified object reference to be internalized by what may be a different ORB, a stringified IOR representation is specified. This representation instead establishes that ORBs could parse stringified object references using that format. This helps address the problem of bootstrapping, allowing programs to obtain and use object references, even from different ORBs.

The following is the representation of the stringified (externalized) IOR:

```
<oref>          ::= <prefix> <hex_Octets>
<prefix>        ::= "IOR:"
<hex_Octets>    ::= <hex_Octet> {<hex_Octet>}*
```

```
<hex_Octet>     ::= <hexDigit> <hexDigit>
<hexDigit>      ::= <digit> | <a> | <b> | <c> | <d> | <e> | <f>
<digit>         ::= "0" | "1" | "2" | "3" | "4" | "5" |
                        "6" | "7" | "8" | "9"
<a>             ::= "a" | "A"
<b>             ::= "b" | "B"
<c>             ::= "c" | "C"
<d>             ::= "d" | "D"
<e>             ::= "e" | "E"
<f>             ::= "f" | "F"
```

The hexadecimal strings are generated by first turning an object reference into an IOR, and then encapsulating the IOR using the encoding rules of CDR. (See "CDR Transfer Syntax" on page 12-4 for more information.) The content of the encapsulated IOR is then turned into hexadecimal digit pairs, starting with the first octet in the encapsulation and going until the end. The high four bits of each octet are encoded as a hexadecimal digit, then the low four bits.

10.6.6 Object Service Context

Emerging specifications for Object Services occasionally require service-specific context information to be passed implicitly with requests and replies. (Specifications for OMG's Object Services are contained in *CORBAservices: Common Object Service Specifications.*) The Interoperability specifications define a mechanism for identifying and passing this service-specific context information as "hidden" parameters. The specification makes the following assumptions:

- Object Service specifications that need additional context passed will completely specify that context as an OMG IDL data type.
- ORB APIs will be provided that will allow services to supply and consume context information at appropriate points in the process of sending and receiving requests and replies.
- It is an ORB's responsibility to determine when to send service-specific context information, and what to do with such information in incoming messages. It may be possible, for example, for a server receiving a request to be unable to de-encapsulate and use a certain element of service-specific context, but nevertheless still be able to successfully reply to the message.

As shown in the following OMG IDL specification, the IOP module provides the mechanism for passing Object Service–specific information. It does not describe any service-specific information. It only describes a mechanism for transmitting it in the most general way possible. The mechanism is currently used by the DCE ESIOP and could also be used by the Internet Inter-ORB protocol (IIOP) General Inter_ORB Protocol (GIOP).

Each Object Service requiring implicit service-specific context to be passed through GIOP will be allocated a unique service context ID value by OMG. Service context ID values are of type **unsigned long**. Object service specifications are responsible for describing their context information as single OMG IDL data types, one data type associated with each service context ID.

The marshaling of Object Service data is described by the following OMG IDL:

```
module IOP  {                                   // IDL


        typedef unsigned long           ServiceID;

        struct ServiceContext {
            ServiceID           context_id;
            sequence <octet>context_data;
        };
        typedef sequence <ServiceContext>ServiceContextList;

        const ServiceID                 TransactionService = 0;
};
```

The context data for a particular service will be encoded as specified for its service-specific OMG IDL definition, and that encoded representation will be encapsulated in the **context_data** member of **IOP::ServiceContext**. (See Section 12.3.3, Encapsulation.) The **context_id** member contains the service ID value identifying the service and data format. Context data is encapsulated in octet sequences to permit ORBs to handle context data without unmarshaling, and to handle unknown context data types.

During request and reply marshaling, ORBs will collect all service context data associated with the *Request* or *Reply* in a **ServiceContextList**, and include it in the generated messages. No ordering is specified for service context data within the list. The list is placed at the beginning of those messages to support security policies that may need to apply to the majority of the data in a request (including the message headers).

Note – The only *ServiceID* currently defined is *TransactionService*, for a CDR encapsulation of the Cos*TSInteroperation::PropogationContext* defined in *CORBAservices*, Section 10.5.2, "ORB/TS Implementation Considerations," on page 10-56.

Building Inter-ORB Bridges 11

This chapter provides an implementation-oriented conceptual framework for the construction of bridges to provide interoperability between ORBs. It focuses on the layered *request level bridges* that the CORBA Core specifications facilitate, although ORBs may always be internally modified to support bridges. Specifications for the CORBA Core are contained in Chapters 1 - 8 in this manual.

Key feature of the specifications for inter-ORB bridges are as follows:

- Enables requests from one ORB to be translated to requests on another
- Provides support for managing tables keyed by object references

The OMG IDL specification for interoperable object references, which are important to inter-ORB bridging, is shown in Section 10.6.2, “Interoperable Object References: IORs,” on page 10-14.

11.1 In-Line and Request-Level Bridging

Bridging of an invocation between a client in one domain and a server object in another domain can be mediated through a standardized mechanism, or done immediately using nonstandard ones.

The question of how this bridging is constructed is broadly independent of whether the bridging uses a standardized mechanism. There are two possible options for where the bridge components are located:

- Code inside the ORB may perform the necessary translation or mappings; this is termed *in-line bridging*.
- Application style code outside the ORB can perform the translation or mappings; this is termed *request level bridging*.

Request level bridges which mediate through a common protocol (using networking, shared memory, or some other IPC provided by the host operating system) between distinct execution environments will involve components, one in each ORB, known as “half bridges”.

When that mediation is purely internal to one execution environment, using a shared programming environment’s binary interfaces to CORBA- and OMG-IDL-defined data types, this is known as a “full bridge”[1]. From outside the execution environment this will appear identical to some kinds of in-line bridging, since only that environment knows the construction techniques used. However, full bridges more easily support portable policy mediation components, because of their use of only standard CORBA programming interfaces.

Network protocols may be used immediately “in-line”, or to mediate between request-level half bridges. The Chapter 12, “General Inter-ORB Protocol” can be used in either manner. In addition, this specification provides for Environment Specific Inter-ORB Protocols (ESIOP), allowing for alternative mediation mechanisms.

Note that mediated, request level half-bridges can be built by anyone who as access to an ORB, without needing information about the internal construction of that ORB. Immediate-mode request level half-bridges (i.e., ones using nonstandard mediation mechanisms) can similarly be built without needing information about ORB internals. Only in-line bridges (using either standard or nonstandard mediation mechanisms) need potentially proprietary information about ORB internals.

11.1.1 In-line Bridging

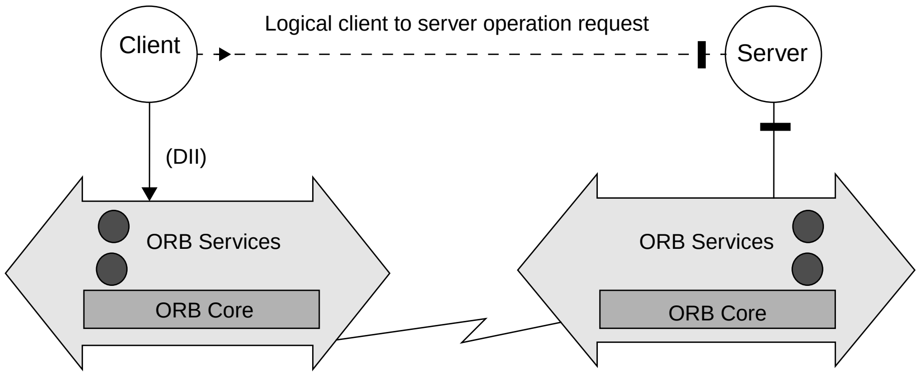


Figure 11-1 In-Line bridges are built using *ORB internal APIs*.

In this approach, the required bridging functionality can be provided by a combination of software components at various levels:

1.Special initialization supporting object referencing domains (e.g. two protocols) to be exposed to application programmers to support construction of this style bridge.

- As additional or alternative services provided by the underlying ORBs
- As additional or alternative stub and skeleton code.

In-line bridging is in general the most direct method of bridging between ORBs. It is structurally similar to the engineering commonly used to bridge between systems within a single ORB (e.g. mediating using some common inter-process communications scheme, such as a network protocol). This means that implementing in-line bridges involves as fundamental a set of changes to an ORB as adding a new inter-process communications scheme. (Some ORBs may be designed to facilitate such modifications, though.)

11.1.2 Request-level Bridging

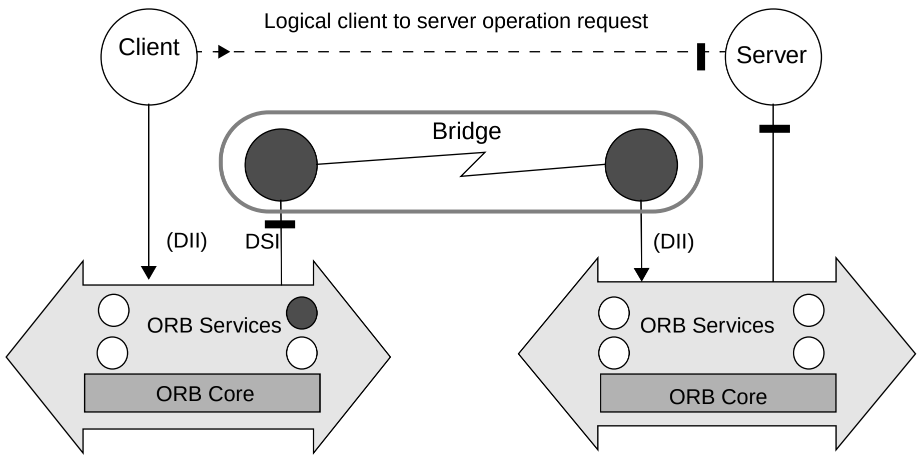


Figure 11-2 Request-Level bridges are built using *public ORB APIs*.

The general principle of request-level bridging is as follows:

1. the original request is passed to a proxy object in the client ORB;
2. the proxy object translates the request contents (including the target object reference) to a form that will be understood by the server ORB;
3. the proxy invokes the required operation on the apparent server object;
4. any operation result is passed back to the client via a complementary route.

The request translation involves performing object reference mapping for all object references involved in the request (the target, explicit parameters, and perhaps implicit ones such as transaction context). As elaborated later, this translation may also involve mappings for other domains: the security domain of **CORBA::Principal** parameters, type identifiers, and so on.

It is a language mapping requirement of the CORBA Core specification that all dynamic typing APIs (e.g. **Any, NamedValue**) support such manipulation of parameters even when the bridge was not created with compile-time knowledge of the data types involved.

11.1.3 Collocated ORBs

In the case of immediate bridging (i.e. not via a standardized, external protocol) the means of communication between the client-side bridge component and that on the server-side is an entirely private matter. One possible engineering technique optimizes this communication by coalescing the two components into the same system or even the same address space. In the latter case, accommodations must be made by both ORBs to allow them to share the same execution environment.

Similar observations apply to request level bridges, which in the case of collocated ORBs use a common binary interface to all OMG IDL-defined data as their mediating data format.

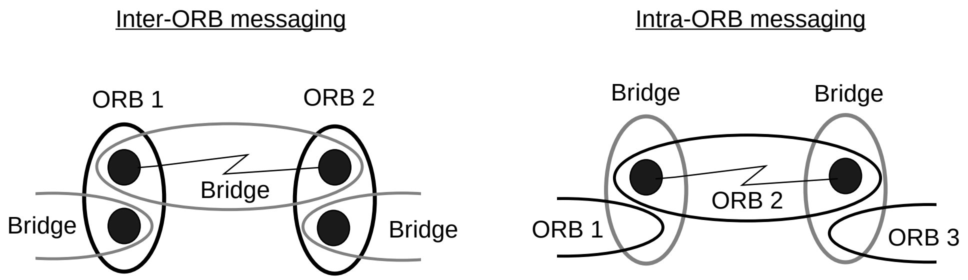


Figure 11-3 When the two ORBs are collocated in a bridge execution environment, network communications will be purely intra-ORB. If the ORBs are not collocated, such communications must go between ORBs.

An advantage of using bridges spanning collocated ORBs is that all external messaging can be arranged to be intra-ORB, using whatever message passing mechanisms each ORB uses to achieve distribution within a single ORB, multiple machine system. That is, for bridges between networked ORBs such a bridge would add only a single "hop," a cost analogous to normal routing.

11.2 Proxy Creation and Management

Bridges need to support arbitrary numbers of proxy objects, because of the (bidirectional) object reference mappings required. The key schemes for creating and managing proxies are *reference translation* and *reference encapsulation*, as discussed in "Handling of Referencing Between Domains" on page 10-12.

- Reference translation approaches are possible with CORBA V2.0 Core APIs. Proxies themselves can be created as normal objects using the Basic Object Adapter (BOA) and the Dynamic Skeleton Interface (DSI).
- Reference Encapsulation is not supported by the BOA, since it would call for knowledge of more than one ORB. Some ORBs could provide other object adapters which support such encapsulation.

Note that from the perspective of clients, they only ever deal with local objects; clients do not need to distinguish between proxies and other objects. Accordingly, all CORBA operations supported by the local ORB are also supported through a bridge. The ORB used by the client might, however, be able to recognize that encapsulation is in use, depending on how the ORB is implemented.

Also, note that the **CORBA::InterfaceDef** used when creating proxies (e.g. the one passed to **CORBA::BOA::create**) could be either a proxy to one in the target ORB, or could be an equivalent local one. When the domains being bridged include a type domain, then the **InterfaceDef** objects cannot be proxies since type descriptions will not have the same information. When bridging CORBA V2.0 compliant ORBs, type domains by definition do not need to be bridged.

11.3 Interface-specific Bridges and Generic Bridges

Request-level bridges may be:

- *Interface-specific*: they support predetermined IDL interfaces only, and are built using IDL-compiler generated stub and skeleton interfaces;
- *Generic*: capable of bridging requests to server objects of arbitrary IDL interfaces, using the interface repository and other dynamic invocation support (DII and DSI).

Interface-specific bridges may be more efficient in some cases (a generic bridge could conceivably create the same stubs and skeletons using the interface repository), but the requirement for prior compilation means that this approach offers less flexibility than use of generic bridges.

11.4 Building Generic Request-Level Bridges

The CORBA Core specifications (Chapters 1 - 8) define the following interfaces. These interfaces are of particular significance when building a generic request-level bridge:

- ***Dynamic Invocation Interface (DII)*** lets the bridge make arbitrary invocations on object references whose types may not have been known when the bridge was developed or deployed.
- ***Dynamic Skeleton Interface (DSI****)* lets the bridge handle invocations on proxy object references which it implements, even when their types may not have been known when the bridge was developed or deployed.
- ***Interface Repositories*** are consulted by the bridge to acquire the information used to drive DII and DSI, such as the type codes for operation parameters, return values, and exceptions.
- ***Object Adapters*** *(*such as the Basic Object Adapter*)* are used to create proxy object references both when bootstrapping the bridge and when mapping object references which are dynamically passed from one ORB to the other.
- ***CORBA Object References*** support operations to fully describe their interfaces and to create tables mapping object references to their proxies (and vice versa).

Interface repositories accessed on either side of a half bridge need not have the same information, though of course the information associated with any given repository ID (e.g. an interface type ID, exception ID) or operation ID must be the same.

Using these interfaces and an interface to some common transport mechanism such as TCP, portable request-level half bridges connected to an ORB can:

- Use DSI to translate all CORBA invocations on proxy objects to the form used by some mediating protocol such as IIOP (see Chapter 12, “General Inter-ORB Protocol”).
- Translate requests made using such a mediating protocol into DII requests on objects in the ORB.

As noted in “In-Line and Request-Level Bridging” on page 11-1, translating requests and responses (including exceptional responses) involves mapping object references (and other explicit and implicit parameter data) from the form used by the ORB to the form used by the mediating protocol, and vice versa. Explicit parameters, which are defined by an operation’s OMG-IDL definition, are presented through DII or DSI and are listed in the Interface Repository entry for any particular operation.

Operations on object references such as **hash()** and **is_equivalent()** may be used to maintain tables that support such mappings. When such a mapping does not exist, an object adapter is used to create a ORB-specific proxy object references, and bridge-internal interfaces are used to create the analogous data structure for the mediating protocol.

11.5 Bridging Non-Referencing Domains

In the simplest form of request-level bridging, the bridge operates only on IDL-defined data, and bridges only object reference domains. In this case, a proxy object in the client ORB acts as a representative of the target object and is, in almost any practical sense, indistinguishable from the target server object - indeed, even the client ORB will not be aware of the distinction.

However, as alluded to above, there may be multiple domains that need simultaneous bridging. The transformation and encapsulation schemes described above may not apply in the same way to Principal or type identifiers. Request level bridges may need to translate such identifiers, in addition to object references, as they are passed as explicit operation parameters.

Moreover, there is an emerging class of “implicit context” information that ORBs may need to convey with any particular request, such as transaction and security context information. Such parameters are not defined as part of an operation’s OMG-IDL signature, hence are “implicit” in the invocation context. Bridging the domains of such implicit parameters could involve additional kinds of work, needing to mediate more policies, than bridging the object reference, Principal, and type domains directly addressed by CORBA.

CORBA does not yet have a generic way (including support for both static and dynamic invocations) to expose such implicit context information.

11.6 Bootstrapping Bridges

A particularly useful policy for setting up bridges is to create a pair of proxies for two Naming Service naming contexts (one in each ORB) and then install those proxies as naming contexts in the other ORB's naming service. (The Naming Service is described in *CORBAservices*.) This will allow clients in either ORB to transparently perform naming context lookup operations on the other ORB, retrieving (proxy) object references for other objects in that ORB. In this way, users can access facilities that have been selectively exported from another ORB, through a naming context, with no administrative action beyond exporting those initial contexts. (See "Obtaining Initial Object References" on page 7-10 for additional information).

This same approach may be taken with other discovery services, such as a trading service or any kind of object that could provide object references as operation results (and in "out" parameters). While bridges can be established which only pass a predefined set of object references, this kind of minimal connectivity policy is not always desirable.

General Inter-ORB Protocol 12

This chapter specifies a General Inter-ORB Protocol (GIOP) for ORB interoperability, which can be mapped onto any connection-oriented transport protocol that meets a minimal set of assumptions. This chapter also defines a specific mapping of the GIOP which runs directly over TCP/IP connections, called the Internet Inter-ORB Protocol (IIOP). The IIOP must be supported by conforming networked ORB products regardless of other aspects of their implementation. Such support does not require using it internally; conforming ORBs may also provide bridges to this protocol.

12.1 Goals of the General Inter-ORB Protocol

The GIOP and IIOP support protocol-level ORB interoperability in a general, low-cost manner. The following objectives were pursued vigorously in the GIOP design:

- ***Widest possible availability*** The GIOP and IIOP are based on the most widely-used and flexible communications transport mechanism available (TCP/IP), and defines the minimum additional protocol layers necessary to transfer CORBA requests between ORBs.
- ***Simplicity*** The GIOP is intended to be as simple as possible, while meeting other design goals. Simplicity is deemed the best approach to ensure a variety of independent, compatible implementations.
- ***Scalability*** The GIOP/IIOP protocol should support ORBs, and networks of bridged ORBs, to the size of today's Internet, and beyond.
- ***Low cost*** Adding support for GIOP/IIOP to an existing or new ORB design should require small engineering investment. Moreover, the run-time costs required to support IIOP in deployed ORBs should be minimal.
- ***Generality*** While the IIOP is initially defined for TCP/IP, GIOP message formats are designed to be used with any transport layer that meets a minimal set of assumptions; specifically, the GIOP is designed to be implemented on other connection-oriented transport protocols.

- ***Architectural neutrality*** The GIOP specification makes minimal assumptions about the architecture of agents that will support it. The GIOP specification treats ORBs as opaque entities with unknown architectures.

The approach a particular ORB takes to providing support for the GIOP/IIOP is undefined. For example, an ORB could choose to use the IIOP as its internal protocol, it could choose to externalize IIOP as much as possible by implementing it in a half-bridge, or it could choose a strategy between these two extremes. All that is required of a conforming ORB is that some entity or entities in or associated with the ORB be able to send and receive IIOP messages.

12.2 General Inter-ORB Protocol Overview

The GIOP specification consists of the following elements:

- *The Common Data Representation (CDR) definition*. CDR is a transfer syntax mapping OMG IDL data types into a bicanonical low-level representation for “on-the-wire” transfer between ORBs and Inter-ORB bridges (agents).
- *GIOP Message Formats*. GIOP messages are exchanged between agents to facilitate object requests, locate object implementations, and manage communication channels.
- *GIOP Transport Assumptions*. The GIOP specification describes general assumptions made concerning any network transport layer that may be used to transfer GIOP messages. The specification also describes how connections may be managed, and constraints on GIOP message ordering.

The IIOP specification adds the following element to the GIOP specification:

- *Internet IOP Message Transport*. The IIOP specification describes how agents open TCP/IP connections and use them to transfer GIOP messages.

The IIOP is not a separate specification; it is a specialization, or mapping, of the GIOP to a specific transport (TCP/IP). The GIOP specification (without the transport-specific IIOP element) may be considered as a separate conformance point for future mappings to other transport layers.

The complete OMG IDL specifications for the GIOP and IOP are shown in Section 12.8, “OMG IDL for the GIOP and IIOP Specifications,” on page 12-29. Fragments of the specification are used throughout this chapter as necessary.

12.2.1 Common Data Representation (CDR)

CDR is a transfer syntax, mapping from data types defined in OMG IDL to a bicanonical, low-level representation for transfer between agents. CDR has the following features:

- ***Variable byte ordering.*** Machines with a common byte order may exchange messages without byte swapping. When communicating machines have different byte order, the message originator determines the message byte order, and the receiver is responsible for swapping bytes to match its native ordering. Each GIOP message (and CDR encapsulation) contains a flag that indicates the appropriate byte order.

- ***Aligned primitive types.*** Primitive OMG IDL data types are aligned on their natural boundaries within GIOP messages, permitting data to be handled efficiently by architectures that enforce data alignment in memory.
- ***Complete OMG IDL Mapping.*** CDR describes representations for all OMG IDL data types, including transferable pseudo-objects such as TypeCodes. Where necessary, CDR defines representations for data types whose representations are undefined or implementation-dependent in the CORBA Core specifications.

12.2.2 GIOP Message Overview

The GIOP specifies formats for messages that are exchanged between inter- operating ORBs. GIOP message formats have the following features:

- ***Few, simple messages.*** With only seven message formats, the GIOP supports full CORBA functionality between ORBs, with extended capabilities supporting object location services, dynamic migration, and efficient management of communication resources. GIOP semantics require no format or binding negotiations. In most cases, clients can send requests to objects immediately upon opening a connection.
- ***Dynamic object location.*** Many ORBs' architectures allow an object implementation to be activated at different locations during its lifetime, and may allow objects to migrate dynamically. GIOP messages provide support for object location and migration, without requiring ORBs to implement such mechanisms when unnecessary or inappropriate to an ORB's architecture.
- ***Full CORBA support.*** GIOP messages directly support all functions and behaviors required by CORBA, including exception reporting, passing operation context, and remote object reference operations (such as **CORBA::Object::get_interface**).

GIOP also supports passing service-specific context, such as the transaction context defined by the Transaction Service (the Transaction Service is described in *CORBAservices*). This mechanism is designed to support any service that requires service related context to be implicitly passed with requests.

12.2.3 GIOP Message Transfer

The GIOP specification is designed to operate over any connection-oriented transport protocol that meets a minimal set of assumptions (described in "GIOP Message Transport" on page 12-23). GIOP uses underlying transport connections in the following ways:

- ***Asymmetrical connection usage.*** The GIOP defines two distinct roles with respect to connections, client and server. The client side of a connection originates the connection, and sends object requests over the connection. The server side receives requests and sends replies. The server side of a connection may not send object requests. This restriction allows the GIOP specification to be much simpler and avoids certain race conditions.

- ***Request multiplexing***. If desirable, multiple clients within an ORB may share a connection to send requests to a particular ORB or server. Each request uniquely identifies its target object. Multiple independent requests for different objects, or a single object, may be sent on the same connection.
- ***Overlapping requests***. In general, GIOP message ordering constraints are minimal. GIOP is designed to allow overlapping asynchronous requests; it does not dictate the relative ordering of requests or replies. Unique request/reply identifiers provide proper correlation of related messages. Implementations are free to impose any internal message ordering constraints required by their ORB architectures.
- ***Connection management***. GIOP defines messages for request cancellation and orderly connection shutdown. These features allow ORBs to reclaim and reuse idle connection resources.

12.3 CDR Transfer Syntax

The Common Data Representation (CDR) transfer syntax is the format in which the GIOP represents OMG IDL data types in an octet stream.

An octet stream is an abstract notion that typically corresponds to a memory buffer that is to be sent to another process or machine over some IPC mechanism or network transport. For the purposes of this discussion, an octet stream is an arbitrarily long (but finite) sequence of eight-bit values (octets) with a well-defined beginning. The octets in the stream are numbered from *0* to *n-1*, where *n* is the size of the stream. The numeric position of an octet in the stream is called its *index*. Octet indices are used to calculate alignment boundaries, as described in “Alignment” on page 12-5.

GIOP defines two distinct kinds of octet streams, messages and encapsulations. Messages are the basic units of information exchange in GIOP, described in detail in “GIOP Message Formats” on page 12-15.

Encapsulations are octet streams into which OMG IDL data structures may be marshaled independently, apart from any particular message context. Once a data structure has been encapsulated, the octet stream can be represented as the OMG IDL opaque data type **sequence<octet>**, which can subsequently marshaled into a message or another encapsulation. Encapsulations allow complex constants (such as **TypeCodes**) to be pre-marshaled; they also allow certain message components to be handled without requiring full unmarshaling. Whenever encapsulations are used in CDR or the GIOP, they are clearly noted.

12.3.1 Primitive Types

Primitive data types are specified for both big-endian and little-endian orderings. The message formats (see “GIOP Message Formats” on page 12-15) include tags in message headers that indicate the byte ordering in the message. Encapsulations include an initial flag that indicates the byte ordering within the encapsulation, described in “Encapsulation” on page 12-9. The byte ordering of any encapsulation may be different from the message or encapsulation within which it is nested. It is the responsibility of the message recipient to translate byte ordering if necessary.

Primitive data types are encoded in multiples of octets. An octet is an 8-bit value.

Alignment

In order to allow primitive data to be moved into and out of octet streams with instructions specifically designed for those primitive data types, in CDR all primitive data types must be aligned on their natural boundaries; i.e., the alignment boundary of a primitive datum is equal to the size of the datum in octets. Any primitive of size *n* octets must start at an octet stream index that is a multiple of *n*. In CDR, *n* is one of 1, 2, 4, or 8.

Where necessary, an alignment gap precedes the representation of a primitive datum. The value of octets in alignment gaps is undefined. A gap must be the minimum size necessary to align the following primitive. Table 12-1 gives alignment boundaries for CDR/OMG-IDL primitive types.

Table 12-1 Alignment requirements for OMG IDL primitive data types

TYPE	OCTET ALIGNMENT
char	1
octet	1
short	2
unsigned short	2
long	4
unsigned long	4
float	4
double	8
boolean	1
enum	4

Alignment is defined above as being relative to the beginning of an octet stream. The first octet of the stream is octet index zero (0); any data type may be stored starting at this index. Such octet streams begin at the start of an GIOP message header (see "GIOP Message Header" on page 12-15) and at the beginning of an encapsulation, even if the encapsulation itself is nested in another encapsulation. (See "Encapsulation" on page 12-9).

Integer Data Types

Figure 12-1 illustrates the representations for OMG IDL integer data types, including the following data types:

- short

- unsigned short
- long
- unsigned long

The figure illustrates bit ordering and size. Signed types (**short** and **long**) are represented as two's complement numbers; unsigned versions of these types are represented as unsigned binary numbers.

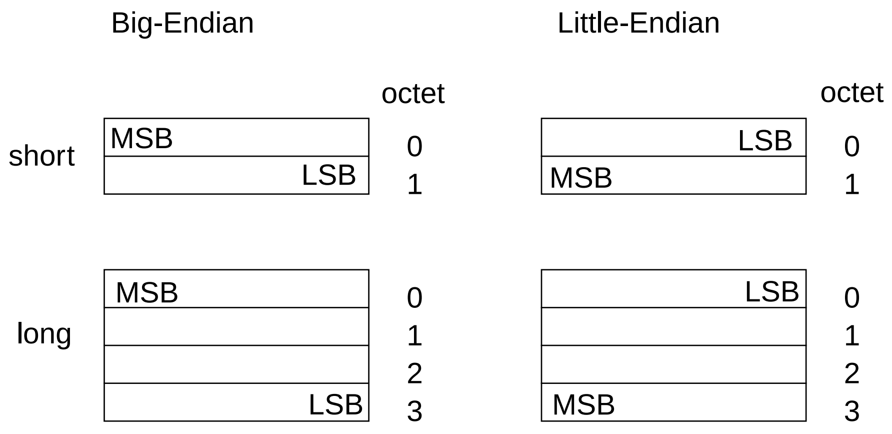


Figure 12-1 Sizes and bit ordering in big-endian and little-endian encodings of OMG IDL integer data types, both signed and unsigned.

Floating Point Data Types

Figure 12-2 on page 7 illustrates the representation of floating point numbers. These exactly follow the IEEE standard formats for floating point numbers[1], selected parts of which are abstracted here for explanatory purposes. The diagram shows three different components for floating points numbers, the sign bit (s), the exponent (e) and the fractional part (f) of the mantissa. The sign bit has values of 0 or 1, representing positive and negative numbers, respectively.

For single-precision float values the exponent is 8 bits long, comprising e1 and e2 in the figure, where the 7 bits in e1 are most significant. The exponent is represented as excess 127. The fractional mantissa (f1 - f3) is a 23-bit value f where 1.0 <= f < 2.0, f1 being most significant and f3 being least significant. The value of a normalized number is described by:

$$-1^{sign} \times 2^{(exponent - 127)} \times (1 + fraction)$$

1."IEEE Standard for Binary Floating-Point Arithmetic", ANSI/IEEE Standard 754-1985, Institute of Electrical and Electronics Engineers, August 1985.

For double-precision values the exponent is 11 bits long, comprising e1 and e2 in the figure, where the 7 bits in e1 are most significant. The exponent is represented as excess 1023. The fractional mantissa (f1 - f7) is a 52-bit value m where 1.0 <= m < 2.0, f1 being most significant and f7 being least significant. The value of a normalized number is described by:

$$-1^{sign} \times 2^{(exponent - 1023)} \times (1 + fraction)$$

Octet

Octets are uninterpreted 8-bit values whose contents are guaranteed not to undergo any conversion during transmission. For the purposes of describing possible octet values in this specification, octets may be considered as unsigned 8-bit integer values.

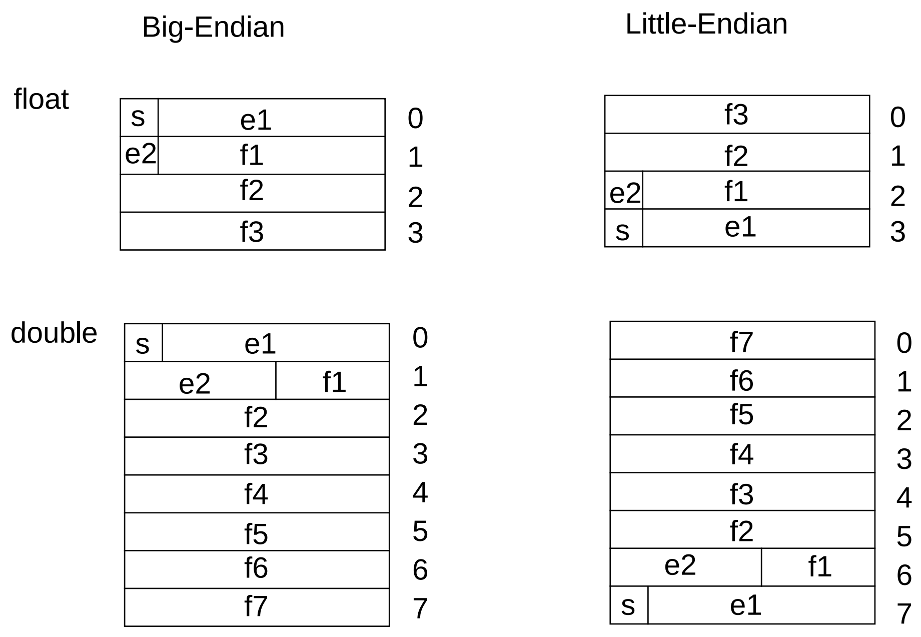
Boolean

Boolean values are encoded as single octets, where TRUE is the value 1, and FALSE as 0.

Character Types

OMG IDL characters are represented single octets, encoded as defined by ISO Latin-1 (8859.1).

Figure 12-2 Sizes and bit ordering in big-endian and little-endian representations of OMG IDL single and double precision floating point numbers.



12.3.2 OMG IDL Constructed Types

Constructed types are built from OMG IDL's data types using facilities defined by the OMG IDL language.

Alignment

Constructed type have no alignment restrictions beyond those of their primitive components; the alignment of those primitive types is not intended to support use of marshaling buffers as equivalent to the implementation of constructed data types within any particular language environment. GIOP assumes that agents will usually construct structured data types by copying primitive data between the marshaled buffer and the appropriate in-memory data structure layout for the language mapping implementation involved.

Struct

The components of a structure are encoded in their order of their declaration in the structure. Each component is encoded as defined for its data type.

Union

Unions are encoded as the discriminant tag of the type specified in the union declaration, followed by the representation of any selected member, encoded as its type indicates.

Array

Arrays are encoded as the array elements in sequence. As the array length is fixed, no length values are encoded. Each element is encoded as defined for the type of the array. In multidimensional arrays, the elements are ordered so the index of the first dimension varies most slowly, and the index of the last dimension varies most quickly.

Sequence

Sequences are encoded as an unsigned long value, followed by the elements of the sequence. The initial unsigned long contains the number of elements in the sequence. The elements of the sequence are encoded as specified for their type.

String

Strings are encoded as an unsigned long containing the length of the string, followed by the individual characters in the string, encoded in ISO Latin-1 (8859.1). The length (initial unsigned long) and string representation include a terminating null character, so that conventional C-string handling library routines (e.g., `strcpy`) may be used in the encoded message buffer.

Enum

Enum values are encoded as unsigned longs. The numeric values associated with enum identifiers are determined by the order in which the identifiers appear in the enum declaration. The first enum identifier has the numeric value zero (0). Successive enum identifiers are take ascending numeric values, in order of declaration from left to right.

12.3.3 Encapsulation

As described above, OMG IDL data types may be independently marshaled into encapsulation octet streams. The octet stream is represented as the OMG IDL type **sequence<octet>,** which may be subsequently included in a GIOP message or nested in another encapsulation.

The GIOP and IIOP explicitly use encapsulations in three places: *TypeCodes* (see “TypeCode” on page 12-10), the IIOP protocol profile inside an IOR (see “Object References” on page 12-15), and in service-specific context (see “Object Service Context” on page 10-18). In addition, some ORBs may use choose to use an encapsulation to hold **Principal** identification information (see “Principal” on page 12-14), the **object_key** (see “IIOP IOR Profiles” on page 12-27), or in other places that a **sequence<octet>** data type is in use.

When encapsulating OMG IDL data types, the first octet in the stream (index 0) contains a boolean value indicating the byte ordering of the encapsulated data. If the value is FALSE (0), the encapsulated data is encoded in big-endian order; if TRUE (1), the data is encoded in little-endian order, exactly like the byte order flag in GIOP message headers (see “GIOP Message Header” on page 12-15). This value is not part of the data being encapsulated, but is part of the octet stream holding the encapsulation. Following the byte order flag, the data to be encapsulated is marshaled into the buffer as defined by CDR encoding rules. Marshaled data are aligned relative to the beginning of the octet stream (the first octet of which is occupied by the byte order flag).

When the encapsulation is encoded as type **sequence<octet>** for subsequent marshaling, an unsigned long value containing the sequence length is prefixed to the octet stream, as prescribed for sequences (see “Sequence” on page 12-8). The length value is not part of the encapsulation’s octet stream, and does not affect alignment of data within the encapsulation. Note that this guarantees a four octet alignment of the start of all encapsulated data within GIOP messages and nested encapsulations.[2]

12.3.4 Pseudo-Object Types

CORBA defines some kinds of entities that are neither primitive types (integral or floating point) nor constructed ones.

2.Accordingly, in cases where encapsulated data holds data with natural alignment of greater than four octets, some processors may need to copy the octet data before removing it from the encapsulation. The GIOP protocol itself does not require encapsulation of such data.

TypeCode

In general, TypeCodes are encoded as the **TCKind** enum value, potentially followed by values that represent the TypeCode parameters. Unfortunately, TypeCodes cannot be expressed simply in OMG IDL, since their definitions are recursive. The basic TypeCode representations are given in Table 12-2. The enum value column this table gives the **TCKind** enum value corresponding to the given TypeCode, and lists the parameters associated with such a TypeCode. The rest of this section presents the details of the encoding.

Basic TypeCode Encoding Framework

The encoding of a TypeCode is the **TCKind** enum value (encoded, like all enum values, using four octets), followed by zero or more parameter values. The encodings of the parameter lists fall into three general categories, and differ in order to conserve space and to support efficient traversal of the binary representation:

- Typecodes with an *empty parameter list* are encoded simply as the corresponding **TCKind** enum value.

- Ones with *simple parameter lists* are encoded as the **TCKind** enum value followed by the parameter value(s), encoded as indicated in Table 12-2. A “simple” parameter list has a fixed number of fixed length entries, or a single parameter which is has its length encoded first. Currently, only the **TCKind** value **tk_string** has such a parameter list.

- All other typecodes have *complex parameter lists*, which are encoded as the **TCKind** enum value followed by a CDR encapsulation octet sequence (see “Encapsulation” on page 12-9) containing the encapsulated, marshaled parameters. The order of these parameters is shown in the fourth column of Table 12-2.

The third column of Table 12-2 shows whether each parameter list is *empty*, *simple*, or *complex*. Also, note that an internal indirection facility is needed to represent some kinds of typecodes; this is explained in “Indirection: Recursive and Repeated TypeCodes” on page 12-13. This indirection does not need to be exposed to application programmers.

TypeCode Parameter Notation

TypeCode parameters are specified in the fourth column of Table 12-2. The ordering and meaning of parameters is a superset of those given in the Interface Repository specification (Chapter 6); more information is needed by CDR’s representation in order to provide the full semantics of TypeCodes as shown by the API.

- Each parameter is written in the form *type (name),* where *type* describes the parameter’s type, and *name* describes the parameter’s meaning.
- The encoding of some parameter lists (specifically, **tk_struct, tk_union, tk_enum, tk_except**) contain a counted sequence of tuples.

Such counted tuple sequences are written in the form *count {parameters}*, where *count* is the number of tuples in the encoded form, and the *parameters* enclosed in braces are available in each tuple instance. First the *count*, which is an `unsigned long`, and then each *parameter* in each tuple (using the noted type), is encoded in the CDR representation of the typecode. Each tuple is encoded, first parameter followed by second etc., before the next tuple is encoded (first, then second, etc.).

Note that the tuples identifying struct, exception and enum members must be in the order defined in the OMG IDL definition text. Also, that the types of discriminant values in encoded `tk_union` TypeCodes are established by the second encoded parameter (*discriminant type*), and cannot be specified except with reference to a specific OMG IDL definition.[3]

Table 12-2 TypeCode enum values, parameter list types, and parameters

TCKind	**Integer Value**	**Type**	**Parameters**
`tk_null`	0	empty	– none –
`tk_void`	1	empty	– none –
`tk_short`	2	empty	– none –
`tk_long`	3	empty	– none –
`tk_ushort`	4	empty	– none –
`tk_ulong`	5	empty	– none –
`tk_float`	6	empty	– none –
`tk_double`	7	empty	– none –
`tk_boolean`	8	empty	– none –
`tk_char`	9	empty	– none –
`tk_octet`	10	empty	– none –
`tk_any`	11	empty	– none –
`tk_TypeCode`	12	empty	– none –
`tk_Principal`	13	empty	– none –
`tk_objref`	14	complex	string (repository ID), string(name)
`tk_struct`	15	complex	string (repository ID), string (name), ulong (count) {string (member name), TypeCode (member type)}

3.This means that, for example, two OMG IDL unions that are textually equivalent, except that one uses a “char” discriminant, and the other uses a “long” one, would have different size encoded TypeCodes.

Table 12-2 TypeCode enum values, parameter list types, and parameters

TCKind	Integer Value	Type	Parameters
`tk_union`	16	complex	string (repository ID), string(name), TypeCode (discriminant type), long (default used), ulong (count) {*discriminant type*[1] (label value), string (member name), TypeCode (member type)}
`tk_enum`	17	complex	string (repository ID), string (name), ulong (count) {string (member name)}
`tk_string`	18	simple	ulong (max length[2])
`tk_sequence`	19	complex	TypeCode (element type), ulong (max length[3])
`tk_array`	20	complex	TypeCode (element type), ulong (length)
tk_alias	21	complex	string (repository ID), string (name), TypeCode
tk_except	22	complex	string (repository ID), string (name), ulong (count) {string (member name), TypeCode (member type)}
– none –	0xffffffff	simple	long (indirection[4])

1. The type of union label values is determined by the second parameter, discriminant type.

2. For unbounded strings, this value is zero.

3. For unbounded sequences, this value is zero.

4. See "Indirection: Recursive and Repeated TypeCodes" on page 12-13.

Encoded Identifiers and Names

The Repository ID parameters in **tk_objref, tk_struct, tk_union, tk_enum, tk_alias**, and **tk_except** TypeCodes are Interface Repository `RepositoryId` values, whose format is described in the specification of the Interface Repository. RepositoryId values are required for **tk_objref** and **tk_except** TypeCodes; for other TypeCodes they are optional and are encoded as empty strings if omitted.

The **name** parameters in **tk_objref, tk_struct, tk_union, tk_enum, tk_alias,** and **tk_except** TypeCodes and the **member name** parameters in **tk_struct, tk_union, tk_enum** and **tk_except** TypeCodes are not specified by (or significant in) GIOP. Agents should not make assumptions about type equivalence based on these name values; only the structural information (including **RepositoryId** values, if provided) is significant. If provided, the strings should be the simple, unscoped names supplied in the OMG IDL definition text. If omitted, they are encoded as empty strings.

Encoding the tk_union Default Case

In **tk_union** TypeCodes, the long **default used** value is used to indicate which tuple in the sequence describes the union's `default` case. If this value is less than zero, then the union contains no default case. Otherwise, the value contains the zero based index of the default case in the sequence of tuples describing union members.

TypeCodes for Multi-Dimensional Arrays

The **tk_array** TypeCode only describes a single dimension of any array. TypeCodes for multi-dimensional arrays are constructed by nesting **tk_array** TypeCodes within other **tk_array** TypeCodes, one per array dimension. The outermost (or top-level) **tk_array** TypeCode describes the leftmost array index of the array as defined in IDL; the innermost nested **tk_array** TypeCode describes the rightmost index.

Indirection: Recursive and Repeated TypeCodes

The typecode representation of OMG IDL data types that can indirectly contain instances of themselves (e.g **struct foo {sequence <foo> bar;}**) must also contain an indirection. Such an indirection is also useful to reduce the size of encodings; for example, unions with many cases sharing the same value.

CDR provides a constrained indirection to resolve this problem:

- The indirection applies only to TypeCodes nested within some "top level" TypeCode. Indirected TypeCodes are not "freestanding", but only exist inside some other encoded TypeCode.
- Only the second (and subsequent) references to a given TypeCode in that scope may use the indirection facility. The first reference to that TypeCode must be encoded using the normal rules; in the case of a recursive TypeCode, this means that the first instance will not have been fully encoded before a second one must be completely encoded.

The indirection is a numeric octet offset within the scope of the "top level" TypeCode and points to the TCKind value for the typecode. (Note that the byte order of the TCKind value can be determined by its encoded value.) This indirection may well cross encapsulation boundaries, but this is not problematic because of first constraint identified above. Because of the second constraint, the value of the offset will always be negative.

The encoding of such an indirection is as a TypeCode with an "TCKind value" that has the special value 2^{32}-1 (0xffffffff, all ones). Such typecodes have a single (simple) parameter, which is the `long` offset (in units of octets) from the simple parameter. (As an example, that this means offset of negative four (-4) is illegal, because it be self-indirecting.)

Any

Any values are encoded as a TypeCode (encoded as described above) followed by the encoded value.

Principal

Principal pseudo objects are encoded as **sequence<octet>**. In the absence of a Security service specification, **Principal** values have no standard format or interpretation, beyond (as described in the CORBA CORE) serving to identify callers (and potential callers). This specification does not define any inter-ORB security mechanisms, or prescribe any usage of **Principal** values.

By representing Principal values as **sequence<octet>**, GIOP guarantees that ORBs may use domain-specific principal identification schemes; such values undergo no translation or interpretation during transmission. This allows bridges to translate or interpret these identifiers as needed when forwarding requests between different security domains.

Context

Context pseudo objects are encoded as **sequence<string>**. The strings occur in pairs. The first string in each pair is the context property name, and the second string in each pair is the associated value.

Exception

Exceptions are encoded as a string followed by exception members, if any. The string contains the RepositoryId for the exception, as defined in the Interface Repository chapter. Exception members (if any) are encoded in the same manner as a struct.[4]

Note – A catalog of minor codes for CORBA's System Exceptions will be provided, based on implementers' agreements. Such an agreement is needed to support complete interoperability, since otherwise applications could distinguish ORBs based on the diagnostics they report, and could not reliably assign meanings to system exceptions reported to them.

4.Compiled stubs are guaranteed to know how to unmarshal all exceptions. As of this writing, there are recognized problems with the language mappings for the DII in that they can not provide the ORB core with the same amount of information that can be provided to it by compiled stubs, unless an implementation of the DII consults an Interface Repository. Those mappings are being revised to address this issue.

12.3.5 Object References

Object references are encoded in OMG IDL as described in “Object Addressing” on page 10-11. IOR profiles contain transport-specific addressing information, so there is no general-purpose IOR profile format defined for GIOP. Instead, this specification describes the general information model for GIOP profiles and provides a specific format for the IIOP (see “IIOP IOR Profiles” on page 12-27).

In general, GIOP profiles shall include at least these three elements:

- The version number of the transport-specific protocol specification that the server supports,
- The address of an endpoint for the transport protocol being used, and
- An opaque datum (an **object_key**, in the form of an octet sequence) used exclusively by the agent at the specified endpoint address to identify the object.

12.4 GIOP Message Formats

In describing GIOP messages, it is necessary to define client and server roles. For the purpose of this discussion, a client is the agent that opens a connection (see more details in “Connection Management” on page 12-24) and originates requests. A server is an agent that accepts connections and receives requests.

GIOP message types are summarized in Table 12-3, which lists the message type names, whether the message is originated by client, server, or both, and the value used to identify the message type in GIOP message headers.

Table 12-3 GIOP Message Types and originators

Message Type	**Originator**	**Value**
`Request`	Client	0
`Reply`	Server	1
`CancelRequest`	Client	2
`LocateRequest`	Client	3
`LocateReply`	Server	4
`CloseConnection`	Server	5
`MessageError`	Both	6

12.4.1 GIOP Message Header

All GIOP messages begin with the following header, defined in OMG IDL:

```
module GIOP {
    enum MsgType {
    Request, Reply, CancelRequest,
    LocateRequest, LocateReply,
    CloseConnection, MessageError
    };

    struct MessageHeader {
    char                magic [4];
    Version             GIOP_version;
    boolean             byte_order;
    octet               message_type;
    unsigned long       message_size;
    };
};
```

The message header clearly identifies GIOP messages, but is defined to be byte-ordering independent, since the header itself defines the byte ordering of subsequent message elements. The members of the header are:

- **magic** identifies GIOP messages. The value of this member is always the four (upper case) characters "GIOP", encoded in ISO Latin-1 (8859.1).
- **GIOP_version** contains the version number of the GIOP protocol being used in the message. The version number applies to the transport-independent elements of this specification (i.e., the CDR and message formats) which constitute the GIOP. This is not equivalent to the IIOP version number as described in "Object References" on page 12-15, though it has the same structure. The major GIOP version number of this specification is one (1); the minor version is zero (0).
- **byte_order** indicates the byte ordering used in subsequent elements of the message (including *message_size*). A value of FALSE (0) indicates big-endian byte ordering, and TRUE (1) indicates little-endian byte ordering.
- **message_type** indicates the type of the message, according to Table 12-3; these correspond to enum values of type MsgType.
- **message_size** contains the length of the message following the message header, in octets. This count includes any alignment gaps. The use of a message size of 0 with a Request, LocateRequest, Reply, or LocateReply message is reserved for future use.

Request Message

Request messages encode CORBA object invocations, including attribute accessor operations, and **CORBA::Object** operations **get_interface and get_implementation**. Requests flow from client to server.

Request messages have three elements, encoded in this order:

- A GIOP message header

- A Request Header
- The Request Body

Request Header

The request header is specified as follows:

```
module GIOP {                                   // IDL
    struct RequestHeader {
    IOP::ServiceContextList     service_context;
    unsigned long               request_id;
    boolean                     response_expected;
    sequence <octet>            object_key;
    string                      operation;
    Principal                   requesting_principal;
    };
};
```

The members have the following definitions:

- **service_context** contains ORB service data being passed from the client to the server, encoded as described in "Object Service Context" on page 10-18.
- **request_id** is used to associate reply messages with request messages (including LocateRequest messages). The client (requester) is responsible for generating values so that ambiguity is eliminated; specifically, a client must not re-use request_id values during a connection if: *(a)* the previous request containing that ID is still pending, or *(b)* if the previous request containing that ID was canceled and no reply was received. (See the semantics of the "CancelRequest Message" on page 12-20).
- **response_expected** is set to TRUE if the request is expected to have an associated reply. The value is FALSE if the operation is defined as oneway, or if the operation is invoked with the DII and the invocation flags include the INV_NO_RESPONSE flag.
- **object_key** identifies the object which is the target of the invocation. It is the **object_key** field from the transport-specific GIOP profile, e.g. from the encapsulated IIOP profile of the IOR for the target object. This value is only meaningful to the server and is not interpreted or modified by the client.
- **operation** contains the name of the operation being invoked. In the case of attribute accessors, the names are `_get_<attribute>` and `_set_<attribute>`. The case of the operation or attribute name must match the case of the operation name specified in the OMG IDL source for the interface being used.

  In the case of `CORBA::Object` operations that are defined in the CORBA Core ("Object Reference Operations" on page 7-2) and that correspond to GIOP request messages, the operation names are `_interface`, `_implementation`[5], `_is_a` and `_not_existent`.

- **requesting_principal** contains a value identifying the requesting principal. It is provided to support the `BOA::get_principal` operation.

Request Body

The request body includes the following items encoded in this order:

- All **in** and **inout** parameters, in the order in which they are specified in the operation's OMG IDL definition, from left to right.
- An optional **Context** pseudo object, encoded as described in "Context" on page 12-14. This item is only included if the operation's OMG IDL definition includes a context expression, and only includes context members as defined in that expression.

For example, the request body for the following OMG IDL operation

```
double example (in short m, out string str, inout Principal p);
```

would be equivalent to this structure:

```
struct example_body {
short                  m;          // leftmost in or inout parameter
Principal              p;          // ... to the rightmost
};
```

12.4.2 Reply Message

Reply messages are sent in response to Request messages. Replies include inout and out parameters, operation results, and may include exception values. In addition, Reply messages may provide object location information. Replies flow from server to client.

Reply messages have three elements, encoded in this order:

- A GIOP message header
- A ReplyHeader structure
- The reply body

Reply Header

The reply header is defined as follows:

5.Since CORBA::Object::get_implementation is a null interface, clients must narrow the object reference they get to some ORB-specific kind of ImplementationDef.

```
module GIOP {                              // IDL
    enum ReplyStatusType {
    NO_EXCEPTION,
    USER_EXCEPTION,
    SYSTEM_EXCEPTION,
    LOCATION_FORWARD
    };

    struct ReplyHeader {
    IOP::ServiceContextList     service_context;
    unsigned long               request_id;
    ReplyStatusType             reply_status;
    };
};
```

The members have the following definitions:

- **service_context** contains ORB service data being passed from the server to the client, encoded as described in "GIOP Message Transfer" on page 12-3.
- **request_id** is used to associate replies with requests. It contains the same request_id value as the corresponding request.
- **reply_status** indicates the completion status of the associated request, and also determines part of the reply body contents. If no exception occurred and the operation completed successfully, the value is **NO_EXCEPTION** and the body contains return values. Otherwise the body contains an exception, or else directs the client to reissue the request to an object at some other location.

Reply Body

The reply body format is controlled by the value of reply_status. There are three types of reply body:

- If the **reply_status** value is **NO_EXCEPTION**, the body is encoded as if it were a structure holding first any operation return value, then any `inout` and `out` parameters in the order in which they appear in the operation's OMG IDL definition, from left to right. (That structure could be empty.)
- If the **reply_status** value is **USER_EXCEPTION** or **SYSTEM_EXCEPTION**, then the body contains the exception that was raised by the operation, encoded as described in "Exception" on page 12-14. (Only the user defined exceptions listed in the operation's OMG IDL definition may be raised.)
- If the **reply_status** value is **LOCATION_FORWARD**, then the body contains an object reference (IOR) encoded as described in "Object References" on page 12-15. The client ORB is responsible for re-sending the original request to that (different) object. This resending is transparent to the client program making the request.

For example, the reply body for a successful response (the value of **reply_status** is **NO_EXCEPTION**) to the Request example shown on page 12-18 would be equivalent to the following structure:

```
struct example_reply {
    double        return_value;     // return value
    string        str;              // leftmost inout or out parameter
    Principal     p;                // ... to the rightmost
};
```

Note that the **object_key** field in any specific GIOP profile is server-relative, not absolute. Specifically, when a new object reference is received in a **LOCATION_FORWARD** Reply or in a LocateReply message, the **object_key** field embedded in the new object reference's GIOP profile may not have the same value as the **object_key** in the GIOP profile of the original object reference. For details on location forwarding, see "Object Location" on page 12-25.

12.4.3 CancelRequest Message

CancelRequest messages may be sent from clients to servers. **CancelRequest** messages notify a server that the client is no longer expecting a reply for a specified pending **Request** or **LocateRequest** message.

CancelRequest messages have two elements, encoded in this order:

- A GIOP message header
- A CancelRequestHeader

Cancel Request Header

The cancel request header is defined as follows:

```
module GIOP {                        // IDL
    struct CancelRequestHeader {
    unsigned long        request_id;
    };
};
```

The **request_id** member identifies the **Request** or **LocateRequest** message to which the cancel applies. This value is the same as the **request_id** value specified in the original Request or LocateRequest message.

When a client issues a cancel request message, it serves in an advisory capacity only. The server is not required to acknowledge the cancellation, and may subsequently send the corresponding reply. The client should have no expectation about whether a reply (including an exceptional one) arrives.

12.4.4 LocateRequest Message

LocateRequest messages may be sent from a client to a server to determine the following regarding a specified object reference: *(a)* whether the object reference is valid, (b) whether the current server is capable of directly receiving requests for the object reference, and if not, *(c)* to what address requests for the object reference should be sent.

Note that this information is also provided through the **Request** message, but that some clients might prefer not to support retransmission of potentially large messages that might be implied by a **LOCATION_FORWARD** status in a **Reply** message. That is, client use of this represents a potential optimization.

LocateRequest messages have two elements, encoded in this order:

- A GIOP message header
- A LocateRequestHeader

LocateRequest Header.

The **LocateRequest** header is defined as follows:

```
module GIOP {                                   // IDL
     struct LocateRequestHeader {
     unsigned long          request_id;
     sequence <octet>       object_key;
     };
};
```

The members are defined as follows:

- **request_id** is used to associate *LocateReply* messages with *LocateRequest* ones. The client (requester) is responsible for generating values; see “Request Message” on page 12-16 for the applicable rules.
- **object_key** identifies the object being located. In an IIOP context, this value is obtained from the **object_key** field from the encapsulated **IIOP::ProfileBody** in the IIOP profile of the IOR for the target object. When GIOP is mapped to other transports, their IOR profiles must also contain an appropriate corresponding value. This value is only meaningful to the server and is not interpreted or modified by the client.

See “Object Location” on page 12-25 for details on the use of **LocateRequest.**

12.4.5 LocateReply Message

LocateReply messages are sent from servers to clients in response to **LocateRequest** messages.

A LocateReply message has three elements, encoded in this order:

- A GIOP message header
- A LocateReplyHeader
- The locate reply body

Locate Reply Header

The locate reply header is defined as follows:

```
module GIOP {                          // IDL
    enum LocateStatusType {
        UNKNOWN_OBJECT,
        OBJECT_HERE,
        OBJECT_FORWARD
    };

    struct LocateReplyHeader {
        unsigned long           request_id;
        LocateStatusType        locate_status;
    };
};
```

The members have the following definitions:

- **request_id** is used to associate replies with requests. This member contains the `same request_id` value as the corresponding *LocateRequest* message.
- **locate_status.** The value of this member is used to determine whether a **LocateReply** body exists. Values are:
    - **UNKNOWN_OBJECT** The object specified in the corresponding **LocateRequest** message is unknown to the server; no body exists.
    - **OBJECT_HERE** This server (the originator of the **LocateReply** message) can directly receive requests for the specified object; no body exists.
    - **OBJECT_FORWARD** A **LocateReply** body exists.

LocateReply Body

The body is empty unless the **LocateStatus** value is **OBJECT_FORWARD**, in which case the body contains an object reference (IOR) that may be used as the target for requests to the object specified in the LocateRequest message.

12.4.6 CloseConnection Message

CloseConnection messages are sent only by servers. They inform clients that the server intends to close the connection and must not be expected to provide further responses. Moreover, clients know that any requests for which they awaiting replies will never be processed, and may safely be reissued (on another connection).

The **CloseConnection** message consists only of the GIOP message header, identifying the message type.

For details on the usage of **CloseConnection** messages, see “Connection Management” on page 12-24.

12.4.7 MessageError Message

The **MessageError** message is sent in response to any GIOP message whose version number or message type is unknown to the recipient, or any message is received whose header is not properly formed (e.g., has the wrong magic value). Error handling is context-specific.

The **MessageError** message consists only of the GIOP message header, identifying the message type.

12.5 GIOP Message Transport

The GIOP is designed to be implementable on a wide range of transport protocols. The GIOP definition makes the following assumptions regarding transport behavior:

- The transport is connection-oriented. GIOP uses connections to define the scope and extent of request IDs.
- The transport is reliable. Specifically, the transport guarantees that bytes are delivered in the order they are sent, at most once, and that some positive acknowledgment of delivery is available.
- The transport can be viewed as a byte stream. No arbitrary message size limitations, fragmentation, or alignments are enforced.
- The transport provides some reasonable notification of disorderly connection loss. If the peer process aborts, the peer host crashes, or network connectivity is lost, a connection owner should receive some notification of this condition.
- The transport’s model for initiating connections can be mapped onto the general connection model of TCP/IP. Specifically, an agent (described herein as a server) publishes a known network address in an IOR, which is used by the client when initiating a connection.

The server does not actively initiate connections, but is prepared to accept requests to connect (i.e., it *listens* for connections in TCP/IP terms). Another agent that knows the address (called a client) can attempt to initiate connections by sending *connect* requests to the address. The listening server may *accept* the request, forming a new, unique connection with the client, or it may *reject* the request (e.g., due to lack of resources).

Once a connection is open, either side may *close* the connection. (However, see "Connection Management" on page 12-24 for semantic issues related to connection closure.) A candidate transport might not directly support this specific connection model; it is only necessary that the transport's model can be mapped onto this view.

12.5.1 Connection Management

For the purposes of this discussion, the roles client and server are defined as follows:

- A client initiates the connection, presumably using addressing information found in an object reference (IOR) for an object to which it intends to send requests.
- A server accepts connections, but does not initiate them.

These terms only denote roles with respect to a connection. They do not have any implications for ORB or application architectures.

Connections are not symmetrical. Only clients can send *Request*, *LocateRequest*, and *CancelRequest* messages over a connection. Only a server can send *Reply, LocateReply* and *CloseConnection* messages over a connection. Either client or server can send *MessageError* messages.

Only GIOP messages are sent over GIOP connections.

Request IDs must unambiguously associate replies with requests within the scope and lifetime of a connection. Request IDs may be re-used if there is no possibility that the previous request using the ID may still have a pending reply. Note that cancellation does not guarantee no reply will be sent. It is the responsibility of the client to generate and assign request IDs. Request IDs must be unique among both *Request* and *LocateRequest* messages.

Connection Closure

Connections can be closed in two ways: orderly shutdown, or abortive disconnect. Orderly shutdown is initiated by servers reliably sending a **CloseConnection** message, or by clients just closing down a connection. Orderly shutdown may be initiated by the client at any time. If there are pending requests when a client shuts down a connection, the server should consider all such requests canceled. A server may not initiate shutdown if it has begun processing any requests for which it has not either received a *CancelRequest* or sent a corresponding reply.

If a client receives an **CloseConnection** message from the server, it should assume that any outstanding messages (i.e., without replies) were received after the server sent the CloseConnection message, were not processed, and may be safely resent on a new connection.

After reliably issuing a **CloseConnection** message, the server may close the connection. Some transport protocols (not including TCP) do not provide an "orderly disconnect" capability, guaranteeing reliable delivery of the last message sent. When

GIOP is used with such protocols an additional handshake needs to be provided to guarantee that both ends of the connection understand the disposition of any outstanding GIOP requests.

If a client detects connection closure without receiving a **CloseConnection** message, it should assume an abortive disconnect has occurred, and treat the condition as an error. Specifically, it should report COMM_FAILURE exceptions for all pending requests on the connection, with completion_status values set to COMPLETED_MAYBE.

Multiplexing Connections

A client, if it chooses, may send requests to multiple target objects over the same connection, provided that the connection's server side is capable of responding to requests for the objects. It is the responsibility of the client to optimize resource usage by re-using connections, if it wishes. If not, the client may open a new connection for each active object supported by the server, although this behavior should be avoided.

12.5.2 Message Ordering

Only the client (connection originator) may send **Request, LocateRequest, and CancelRequest** messages. Connections are not fully symmetrical.

Clients may have multiple pending requests. A client need not wait for a reply from a previous request before sending another request.

Servers may reply to pending requests in any order. **Reply** messages are not required to be in the same order as the corresponding **Requests**.

The ordering restrictions regarding connection closure mentioned in Connection Management, above, are also noted here. Servers may only issue **CloseConnection** messages when **Reply** messages have been sent in response to all received **Request** messages that require replies.

12.6 Object Location

The GIOP is defined to support object migration and location services without dictating the existence of specific ORB architectures or features. The protocol features are based on the following observations:

A given transport address does not necessarily correspond to any specific ORB architectural component (such as an object adapter, object server process, Inter-ORB bridge, and so forth). It merely implies the existence of some agent with which a connection may be opened, and to which requests may be sent.

The “agent” (owner of the server side of a connection) may have one of the following roles with respect to a particular object reference:

- The agent may be able to accept object requests directly for the object and return replies. The agent may or may not own the actual object implementation; it may be an Inter-ORB bridge that transforms the request and passes it on to another process or ORB. From GIOP's perspective, it is only important that requests can be sent directly to the agent.
- The agent may not be able to accept direct requests for any objects, but acts instead as a location service. Any Request messages sent to the agent would result in either exceptions or replies with LOCATION_FORWARD status, providing new addresses to which requests may be sent. Such agents would also respond to *LocateRequest* messages with appropriate LocateReply messages.
- The agent may directly respond to some requests (for certain objects) and provide forwarding locations for other objects.
- The agent may directly respond to requests for a particular object at one point in time, and provide a forwarding location at a later time (perhaps during the same connection).

Agents are not required to implement location forwarding mechanisms. An agent can be implemented with the policy that a connection either supports direct access to an object, or returns exceptions. Such an ORB (or inter-ORB bridge) always return LocateReply messages with either OBJECT_HERE or UNKNOWN_OBJECT status, and never OBJECT_FORWARD status.

Clients must, however, be able to accept and process Reply messages with LOCATION_FORWARD status, since any ORB may choose to implement a location service. Whether a client chooses to send LocationRequest messages is at the discretion of the client. For example, if the client routinely expected to see LOCATION_FORWARD replies when using the address in an object reference, it might always send LocateRequest messages to objects for which it has no recorded forwarding address. If a client sends LocateRequest messages, it should (obviously) be prepared to accept LocateReply messages.

A client should not make any assumptions about the longevity of object addresses returned by location forwarding mechanisms. Once a connection based on location forwarding information is closed, subsequent attempts to send requests to the same object should start with the original address specified in the initial object reference.

Even after performing successful invocations using an address, a client should be prepared to be forwarded. The only object address that a client should expect to continue working reliably is the one in the initial object reference. If an invocation using that address returns UNKNOWN_OBJECT, the object should be deemed non-existent.

In general, the implementation of location forwarding mechanisms is at the discretion of ORBs, available to be used for optimization and to support flexible object location and migration behaviors.

12.7 Internet Inter-ORB Protocol (IIOP)

The baseline transport specified for GIOP is TCP/IP[6]. Specific APIs for libraries supporting TCP/IP may vary, so this discussion is limited to an abstract view of TCP/IP and management of its connections. The mapping of GIOP message transfer to TCP/IP connections is called the Internet Inter-ORB Protocol (IIOP).

12.7.1 TCP/IP Connection Usage

Agents that are capable of accepting object requests or providing locations for objects (i.e., servers) publish TCP/IP addresses in IORs, as described in "IIOP IOR Profiles" on page 12-27. A TCP/IP address consists of an IP host address, typically represented by a host name, and a TCP port number. Servers must listen for connection requests.

A client needing a object's services must initiate a connection with the address specified in the IOR, with a connect request.

The listening server may accept or reject the connection. In general, servers should accept connection requests if possible, but ORBs are free to establish any desired policy for connection acceptance (e.g., to enforce fairness or optimize resource usage).

Once a connection is accepted, the client may send **Request, LocateRequest**, or **CancelRequest** messages by writing to the TCP/IP socket it owns for the connection. The server may send **Reply, LocateReply**, and **CloseConnection** messages by writing to its TCP/IP connection.

After sending (or receiving) a **CloseConnection** message, both client or server must close the TCP/IP connection.

Given TCP/IP's flow control mechanism, it is possible to create deadlock situations between clients and servers if both sides of a connection send large amounts of data on a connection (or two different connections between the same processes) and do not read incoming data. Both processes may block on write operations, and never resume. It is the responsibility of both clients and servers to avoid creating deadlock by reading incoming messages and avoiding blocking when writing messages, by providing separate threads for reading and writing, or any other workable approach. ORBs are free to adopt any desired implementation strategy, but should provide robust behavior.

12.7.2 IIOP IOR Profiles

IIOP profiles, identifying individual objects accessible through the Internet Inter_ORB Protocol, have the following form:

6.Postel, J., "Transmission Control Protocol – DARPA Internet Program Protocol Specification", RFC-793, Information Sciences Institute, September 1981

```
module IIOP {                  // IDL
    struct Version {
        char              major;
        char              minor;
    };

    struct ProfileBody {
    Version           iiop_version;
        string            host;
        unsigned short    port;
        sequence <octet>object_key;
    };
};
```

An instance of the **IIOP::ProfileBody** type is marshaled into an encapsulation octet stream. This encapsulation (a **sequence<octet>**) becomes the **profile_data** member of the **IOR::TaggedProfile** structure representing the IIOP profile in an IOR, and the `tag` has the value **TAG_INTERNET_IOP** (as defined earlier).

The members of **IIOP::ProfileBody** are defined as follows:

- **iiop_version** describes the version of IIOP that the agent at the specified address is prepared to receive. When an agent generates IIOP profiles specifying a particular version, it must be able to accept messages complying with the specified version or any previous minor version (i.e., any smaller version number). The major version number of this specification is one (1); the minor version is zero (0). Note that this value is not equivalent to the GIOP version number specified in GIOP message headers. Transport-specific elements of the IIOP specification may change independently from the GIOP specification.
- **host** identifies the Internet host to which GIOP messages for the specified object may be sent. In order to promote a very large (Internet-wide) scope for the object reference, this will typically be the fully qualified domain name of the host, rather than an unqualified (or partially qualified) name. However, per Internet standards, the host string may also contain a host address expressed in standard “dotted decimal” form (e.g., “192.231.79.52”).
- **port** contains the TCP/IP port number (at the specified host) where the target agent is listening for connection requests. The agent must be ready to process IIOP messages on connections accepted at this port.
- **object_key** is an opaque value supplied by the agent producing the IOR. This value will be used in request messages to identify the object to which the request is directed. An agent that generates an object key value must be able to map the value unambiguously onto the corresponding object when routing requests internally.

Note that host addresses are restricted in this version of IIOP to be Class A, B, or C Internet addresses. That is, Class D (multi-cast) addresses are not allowed. Such addresses are reserved for use in future versions of IIOP.

Also, note that at this time no “well known” port number has been allocated, so individual agents will need to assign previously unused ports as part of their installation procedures. IIOP supports multiple such agents per host.

12.8 OMG IDL for the GIOP and IIOP Specifications

This section contains the OMG IDL for the GIOP and IIOP modules.

12.8.1 GIOP Module

```
module GIOP {                       // IDL
    enum MsgType {
            Request, Reply, CancelRequest,
            LocateRequest, LocateReply,
            CloseConnection, MessageError
    };

    struct MessageHeader {
            char                        magic [4];
            Version                     GIOP_version;
            boolean                     byte_order;
            octet                       message_type;
            unsigned long               message_size;
    };

    struct RequestHeader {
            IOP::ServiceContextList     service_context;
            unsigned long               request_id;
            boolean                     response_expected;
            sequence <octet>            object_key;
            string                      operation;
            Principal                   requesting_principal;
    };
```

```
        enum ReplyStatusType {
                NO_EXCEPTION,
                USER_EXCEPTION,
                SYSTEM_EXCEPTION,
                LOCATION_FORWARD
        };

        struct ReplyHeader {
                IOP::ServiceContextList              service_context;
                unsigned long                        request_id;
                ReplyStatusType                      reply_status;
    };

        struct CancelRequestHeader {
                unsigned long                        request_id;
    };

        struct LocateRequestHeader {
                unsigned long                        request_id;
                sequence <octet>                     object_key;
        };

        enum LocateStatusType {
                UNKNOWN_OBJECT,
                OBJECT_HERE,
                OBJECT_FORWARD
        };

        struct LocateReplyHeader {
                unsigned long                        request_id;
                LocateStatusType                     locate_status;
        };
};
```

12.8.2 IIOP Module

```
module IIOP {                          \\ IDL
    struct Version {
        char                   major;
        char                   minor;
    };

    struct ProfileBody {
        Version                iiop_version;
        string                 host;
        unsigned short         port;
        sequence <octet>       object_key;
    };


};
```

The DCE ESIOP 13

This chapter specifies an Environment Specific Inter-ORB Protocol (ESIOP) for the OSF DCE environment, the DCE Common Inter-ORB Protocol (DCE-CIOP).

13.1 Goals of the DCE Common Inter-ORB Protocol

DCE CIOP was designed to meet the following goals:

- Support multi-vendor, mission-critical, enterprise-wide, ORB-based applications.
- Leverage services provided by DCE wherever appropriate.
- Allow efficient and straightforward implementation using public DCE APIs.
- Preserve ORB implementation freedom.

DCE CIOP achieves these goals by using DCE-RPC to provide message transport, while leaving the ORB responsible for message formatting, data marshaling, and operation dispatch.

13.2 DCE Common Inter-ORB Protocol Overview

The DCE Common Inter-ORB Protocol uses the wire format and RPC packet formats defined by DCE-RPC to enable independently implemented ORBs to communicate. It defines the message formats that are exchanged using DCE-RPC, and specifies how information in object references is used to establish communication between client and server processes.

The full OMG IDL for the DCE ESIOP specification is shown in Section 13.7, “OMG IDL for the DCE CIOP Module,” on page 13-24. Fragments are used throughout this chapter as necessary.

13.2.1 DCE-CIOP RPC

DCE-CIOP requires an RPC which is interoperable with the DCE connection-oriented and/or connectionless protocols as specified in the X/Open *CAE Specification C309* and the OSF *AES/Distributed Computing RPC Volume*. Some of the features of the DCE-RPC are as follows:

- Defines connection-oriented and connectionless protocols for establishing the communication between a client and server.
- Supports multiple underlying transport protocols including TCP/IP.
- Supports multiple outstanding requests to multiple CORBA objects over the same connection.
- Supports fragmentation of messages. This provides for buffer management by ORBs of CORBA requests which contain a large amount of marshaled data.

All interactions between ORBs take the form of remote procedure calls on one of two well-known DCE-RPC interfaces. Two DCE operations are provided in each interface:

- *invoke* - for invoking CORBA operation requests, and
- *locate* - for locating server processes.

Each DCE operation is a synchronous remote procedure call[1,2], consisting of a request message being transmitted from the client to the server, followed by a response message being transmitted from the server to the client.

Using one of the DCE-RPC interfaces, the messages are transmitted as pipes of uninterpreted bytes. By transmitting messages via DCE pipes, the following characteristics are achieved:

- Large amounts of data can be transmitted efficiently.
- Buffering of complete messages is not required.
- Marshaling and demarshaling can take place concurrently with message transmission.
- Encoding of messages and marshaling of data is completely under the control of the ORB.
- DCE client and server stubs can be used to implement DCE-CIOP.

1. DCE *maybe* operation semantics cannot be used for CORBA *oneway* operations because they are idempotent as opposed to at-most-once.

2. The deferred synchronous DII API can be implemented on top of synchronous RPCs by using threads.

Using the other DCE-RPC interface, the messages are transmitted as conformant arrays of uninterpreted bytes. This interface does not offer all the advantages of the pipe-based interface, but is provided to enable interoperability with ORBs using DCE-RPC implementations that do not adequately support pipes.

13.2.2 DCE-CIOP Data Representation

DCE-CIOP messages represent OMG IDL types by using the CDR transfer syntax, which is defined in “CDR Transfer Syntax” on page 12-4. DCE-CIOP message headers and bodies are specified as OMG IDL types. These are encoded using CDR, and the resulting messages are passed between client and server processes via DCE-RPC pipes or conformant arrays.

NDR is the transfer syntax used by DCE-RPC for operations defined in DCE IDL. CDR, used to represent messages defined in OMG IDL on top of DCE RPCs, represents the OMG IDL primitive types identically to the NDR representation of corresponding DCE IDL primitive types. The corresponding OMG IDL and DCE IDL primitive types are shown in table Table 13-1.

Table 13-1 Relationship between CDR and NDR primitive data types

OMG IDL type	**DCE IDL type with NDR representation equivalent to CDR representation of OMG IDL type**
char	byte
octet	byte
short	short
unsigned short	unsigned short
long	long
unsigned long	unsigned long
float	float[1]
double	double[2]
boolean	byte[3]
enum	unsigned long

1. Restricted to IEEE format.
2. Restricted to IEEE format.
3. Values restricted to 0 and 1.

The CDR representation of OMG IDL constructed types and pseudo-object types does not correspond to the NDR representation of types describable in DCE IDL.

As new data types are added to OMG IDL, NDR can be used as a model for their CDR representations.

13.2.3 DCE-CIOP Messages

The following request and response messages are exchanged between ORB clients and servers via the `invoke` and `locate` RPCs:

- *Invoke Request* identifies the target object and the operation and contains the principal, the operation context, a **ServiceContext**, and the in and inout parameter values.
- *Invoke Response* indicates whether the operation succeeded, failed, or needs to be reinvoked at another location, and returns a **ServiceContext**. If the operation succeeded, the result and the out and inout parameter values are returned. If it failed, an exception is returned. If the object is at another location, new RPC binding information is returned.
- *Locate Request* identifies the target object and the operation.
- *Locate Response* indicates whether the location is in the current process, is elsewhere, or is unknown. If the object is at another location, new RPC binding information is returned.

All message formats begin with a field that indicates the byte order used in the CDR encoding of the remainder of the message. The CDR byte order of a message is required to match the NDR byte order used by DCE-RPC to transmit the message.

13.2.4 Interoperable Object Reference (IOR)

For DCE-CIOP to be used to invoke operations on an object, the information necessary to reference an object via DCE-CIOP must be included in an IOR. This information can coexist with the information needed for other protocols such as IIOP. DCE-CIOP information is stored in an IOR as a set of components in a profile identified by **TAG_MULTIPLE_COMPONENTS**. Components are defined for the following purposes:

- To identify a server process via a DCE string binding, which can be either fully or partially bound. This process may be a server process implementing the object, or it may be an agent capable of locating the object implementation.
- To identify a server process via a name that can be resolved using a DCE nameservice. Again, this process may implement the object or may be an agent capable of locating it.
- To identify the target object when request messages are sent to the server.
- To enable a DCE-CIOP client to recognize objects that share an endpoint.
- To indicate whether a DCE-CIOP client should send a locate message or an invoke message.
- To indicate if the pipe-based DCE-RPC interface is not available.

The IOR is created by the server ORB to provide the information necessary to reference the CORBA object.

13.3 DCE-CIOP Message Transport

DCE-CIOP defines two DCE-RPC interfaces for the transport of messages between client ORBs and server ORBs. One interface uses pipes to convey the messages, while the other uses conformant arrays.

The pipe-based interface is the preferred interface, since it allows messages to be transmitted without precomputing the message length. But not all DCE-RPC implementations adequately support pipes, so this interface is optional. All client and server ORBs implementing DCE-CIOP must support the array-based interface[3].

While server ORBs may provide both interfaces or just the array-based interface, it is up to the client ORB to decide which to use for an invocation. If a client ORB tries to use the pipe-based interface and receives a `rpc_s_unknown_if` error, it should fall back to the array-based interface.

13.3.1 Pipe-based Interface

The `dce_ciop_pipe` interface is defined by the DCE IDL specification shown below:

```
[                       /* DCE IDL */
uuid(0e07f95c-37b0-11ce-90a7-0800090b5d3e),
version(1.0)
]
interface dce_ciop_pipe
{
    typedef pipe byte message_type;

    void invoke ([in] handle_t binding_handle,
                 [in] message_type *request_message,
                 [out] message_type *response_message);

    void locate ([in] handle_t binding_handle,
                 [in] message_type *request_message,
                 [out] message_type *response_message);
}
```

ORBs can implement the `dce_ciop_pipe` interface by using DCE stubs generated from this IDL specification, or by using lower-level APIs provided by a particular DCE-RPC implementation.

3. A future DCE-CIOP revision may eliminate the array-based interface and require support of the pipe-based interface.

The `dce_ciop_pipe` interface is identified by the UUID and version number shown. To provide maximal performance, all server ORBs and location agents implementing DCE-CIOP should listen for and handle requests made to this interface. To maximize the chances of interoperating with any DCE-CIOP client, servers should listen for requests arriving via all available DCE protocol sequences.

Client ORBs can invoke OMG IDL operations over DCE-CIOP by performing DCE RPCs on the `dce_ciop_pipe` interface.

The `dce_ciop_pipe` interface is made up of two DCE-RPC operations, `invoke` and `locate`. The first parameter of each of these RPCs is a DCE binding handle, which identifies the server process on which to perform the RPC. See “DCE-CIOP String Binding Component” on page 13-16, “DCE-CIOP Binding Name Component” on page 13-17, and “DCE-CIOP Object Location” on page 13-21 for discussion of how these binding handles are obtained. The remaining parameters of the `dce_ciop_pipe` RPCs are pipes of uninterpreted bytes. These pipes are used to convey messages encoded using CDR. The `request_message` input parameters send a request message from the client to the server, while the `response_message` output parameters return a response message from the server to the client.

Figure 13-1 below illustrates the layering of DCE-CIOP messages on the DCE-RPC protocol as NDR pipes:

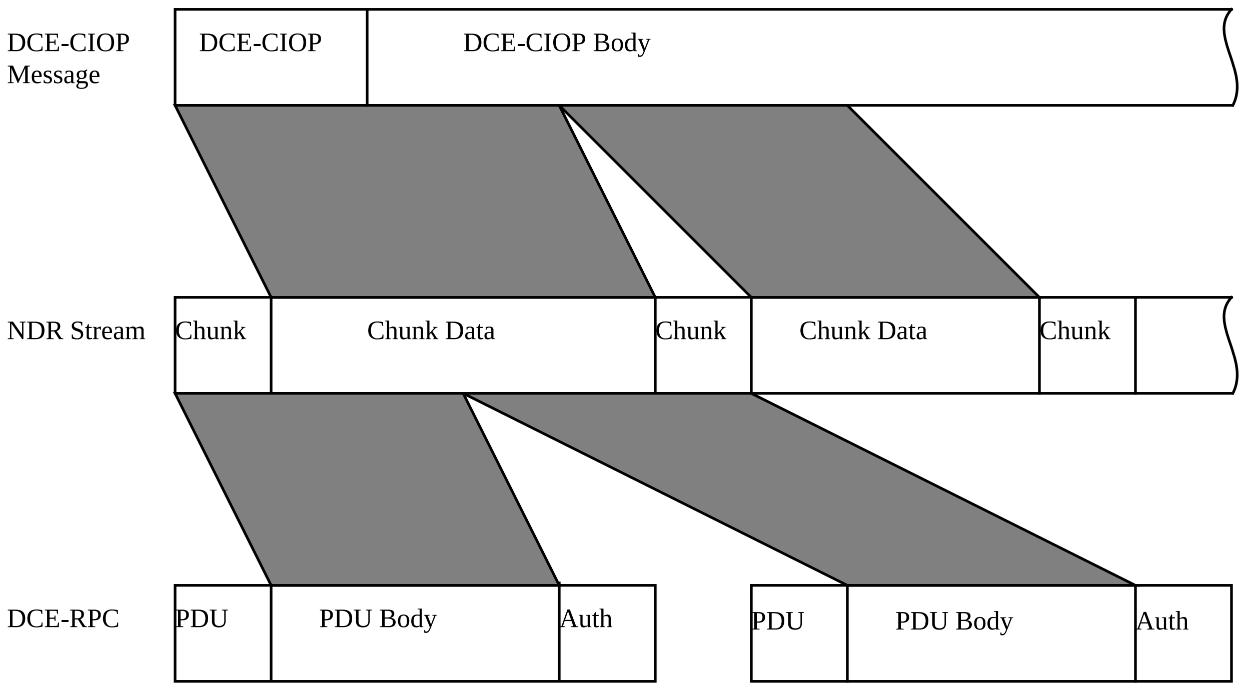


Figure 13-1 Pipe-based interface protocol layering.

The DCE-RPC protocol data unit (PDU) bodies, after any appropriate authentication is performed[4], are concatenated by the DCE-RPC run-time to form an NDR stream. This stream is then interpreted as the NDR representation of a DCE IDL pipe.

A pipe is made up of chunks, where each chunk consists of a chunk length and chunk data. The chunk length is an unsigned long indicating the number of pipe elements that make up the chunk data. The pipe elements are DCE IDL bytes, which are uninterpreted by NDR. A pipe is terminated by a chunk length of zero. The pipe chunks are concatenated to form a DCE-CIOP message.

Invoke

The `invoke` RPC is used by a DCE-CIOP client process to attempt to invoke a CORBA operation in the server process identified by the `binding_handle` parameter. The `request_message` pipe transmits a DCE-CIOP invoke request message, encoded using CDR, from the client to the server. See "DCE_CIOP Invoke Request Message" on page 13-10 for a description of its format. The `response_message` pipe transmits a DCE-CIOP invoke response message, also encoded using CDR, from the server to the client. See "DCE-CIOP Invoke Response Message" on page 13-11 for a description of the response format.

Locate

The `locate` RPC is used by a DCE-CIOP client process to query the server process identified by the `binding_handle` parameter for the location of the server process where requests should be sent. The `request_message` and `response_message` parameters are used similarly to the parameters of the `invoke` RPC. See "DCE-CIOP Locate Request Message" on page 13-13 and "DCE-CIOP Locate Response Message" on page 13-14 for descriptions of their formats. Use of the `locate` RPC is described in detail in "DCE-CIOP Object Location" on page 13-21.

13.3.2 Array-based Interface

The `dce_ciop_array` interface is defined by the DCE IDL specification shown below:

```
[                       /* DCE IDL */
uuid(8108ae54-4cd9-11ce-acca-0800090b5d3e),
version(1.0)
]
interface dce_ciop_array
{
    typedef struct {
```

4. The use of authentication, or other DCE security services, has not yet been defined for DCE-CIOP.

```
        unsigned long length;
        [size_is(length),ptr] byte *data;
    } message_type;

    void invoke ([in] handle_t binding_handle,
                 [in] message_type *request_message,
                 [out] message_type *response_message);

    void locate ([in] handle_t binding_handle,
                 [in] message_type *request_message,
                 [out] message_type *response_message);
}
```

ORBs can implement the `dce_ciop_array` interface, identified by the UUID and version number shown, by using DCE stubs generated from this IDL specification, or by using lower-level APIs provided by a particular DCE-RPC implementation.

All server ORBs and location agents implementing DCE-CIOP must listen for and handle requests made to the `dce_ciop_array` interface, and to maximize interoperability, should listen for requests arriving via all available DCE protocol sequences.

Client ORBs can invoke OMG IDL operations over DCE-CIOP by performing `locate` and `invoke` RPCs on the `dce_ciop_array` interface.

As with the `dce_ciop_pipe` interface, the first parameter of each `dce_ciop_array` RPC is a DCE binding handle that identifies the server process on which to perform the RPC. The remaining parameters are structures containing CDR-encoded messages. The `request_message` input parameters send a request message from the client to the server, while the `response_message` output parameters return a response message from the server to the client.

The `message_type` structure used to convey messages is made up of a `length` member and a `data` member:

- *length* - This member indicates the number of bytes in the message.
- *data* - This member is a full pointer to the first byte of the conformant array containing the message.

The layering of DCE-CIOP messages on DCE-RPC using NDR arrays is illustrated in Figure 13-2 below:

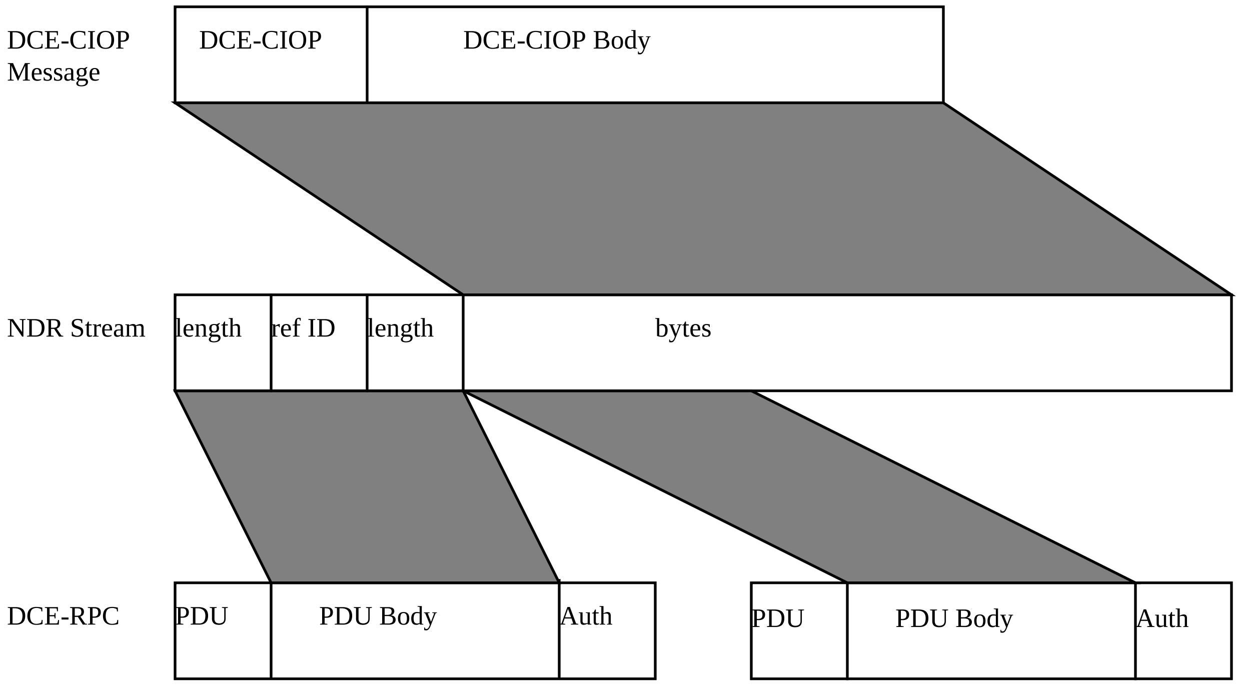


Figure 13-2 Array-based interface protocol layering.

The NDR stream, formed by concatenating the PDU bodies, is interpreted as the NDR representation of the DCE IDL `message_type` structure. The `length` member is encoded first, followed by the `data` member. The `data` member is a full pointer, which is represented in NDR as a referent ID. In this case, this non-NULL pointer is the first (and only) pointer to the referent, so the referent ID is 1 and it is followed by the representation of the referent. The referent is a conformant array of bytes, which is represented in NDR as an unsigned long indicating the length, followed by that number of bytes. The bytes form the DCE-CIOP message.

Invoke

The `invoke` RPC is used by a DCE-CIOP client process to attempt to invoke a CORBA operation in the server process identified by the `binding_handle` parameter. The `request_message` input parameter contains a DCE-CIOP invoke request message. The `response_message` output parameter returns a DCE-CIOP invoke response message from the server to the client.

Locate

The `locate` RPC is used by a DCE-CIOP client process to query the server process identified by the `binding_handle` parameter for the location of the server process where requests should be sent. The `request_message` and `response_message` parameters are used similarly to the parameters of the `invoke` RPC.

13.4 DCE-CIOP Message Formats

The section defines the message formats used by DCE-CIOP. These message formats are specified in OMG IDL, are encoded using CDR, and are transmitted over DCE-RPC as either pipes or arrays of bytes as described in “DCE-CIOP Message Transport” on page 13-5.

13.4.1 DCE_CIOP Invoke Request Message

DCE-CIOP invoke request messages encode CORBA object requests, including attribute accessor operations and `CORBA::Object` operations such as `get_interface` and `get_implementation`. Invoke requests are passed from client to server as the `request_message` parameter of an `invoke` RPC.

A DCE-CIOP invoke request message is made up of a header and a body. The header has a fixed format, while the format of the body is determined by the operation’s IDL definition.

Invoke Request Header

DCE-CIOP request headers have the following structure:

```
module DCE_CIOP {                          // IDL
        struct InvokeRequestHeader {
            boolean byte_order;
            IOP::ServiceContextList service_context;
            sequence <octet> object_key;
            string endpoint_id;
            string operation;
            CORBA::Principal principal;
            sequence <string> client_context;

            // in and inout parameters follow
        };
};
```

The members have the following definitions:

- **byte_order** indicates the byte ordering used in the representation of the remainder of the message. A value of FALSE indicates big-endian byte ordering, and TRUE indicates little-endian byte ordering.

- **service_context** contains any ORB service data that needs to be sent from the client to the server.
- **object_key** contains opaque data used to identify the object that is the target of the operation. See “Object Key Component” on page 13-19.
- **endpoint_id** contains an identifier for the endpoint at which the object is located, if this was included in the IOR profile. If not, an empty string is used. See “Endpoint ID Component” on page 13-19.
- **operation** contains the name of the CORBA operation being invoked. The case of the operation name must match the case of the operation name specified in the OMG IDL source for the interface being used.

  Attribute accessors have names as follows:
  - Attribute selector: operation name is "_get_<attribute>"
  - Attribute mutator: operation name is "_set_<attribute>"

  CORBA::Object pseudo-operations have operation names as follows:
  - **get_interface** - operation name is "_interface"
  - **get_implementation** - operation name is "_implementation"
  - **is_a** - operation name is "_is_a"
  - **non_existent** - operation name is "_non_existent"
- **Principal** contains a value identifying the requesting principal. No particular meaning or semantics are associated with this value. It is provided to support the `BOA::get_principal` operation.
- **client_context** contains any context properties associated with the request. Each property is encoded as a pair of strings; the first naming the property and the second containing its value.

  Clients are not required to include all properties listed in the context expression of the operation’s OMG IDL definition. They are allowed to include properties not listed in the OMG IDL definition.

Invoke Request Body

The invoke request body contains all in and inout parameters, in the order in which they are specified in the operation definition, from left to right.

13.4.2 DCE-CIOP Invoke Response Message

Invoke response messages are returned from servers to clients as the `response_message` parameter of an `invoke` RPC.

Like invoke request messages, an invoke response message is made up of a header and a body. The header has a fixed format, while the format of the body depends on the operation’s OMG IDL definition and the outcome of the invocation.

Invoke Response Header

DCE-CIOP invoke response headers have the following structure:

```
module DCE_CIOP {                          // IDL
        enum InvokeResponseStatus {
            INVOKE_NO_EXCEPTION,
            INVOKE_USER_EXCEPTION,
            INVOKE_SYSTEM_EXCEPTION,
            INVOKE_LOCATION_FORWARD,
            INVOKE_TRY_AGAIN
        };

        struct InvokeResponseHeader {
            boolean byte_order;
            IOP::ServiceContextList service_context;
            InvokeResponseStatus status;

            // if status = INVOKE_NO_EXCEPTION,
            // result then inouts and outs follow

            // if status = INVOKE_USER_EXCEPTION or
            // INVOKE_SYSTEM_EXCEPTION, an exception follows

            // if status = INVOKE_LOCATION_FORWARD, an
            // IOP::MultipleComponentsProfile follows
        };
};
```

The members have the following definitions:

- **byte_order** indicates the byte ordering used in the representation of the remainder of the message. A value of FALSE indicates big-endian byte ordering, and TRUE indicates little-endian byte ordering.
- **service_context** contains any ORB service data that needs to be sent from the client to the server.
- **status** indicates the completion status of the associated request, and also determines the contents of the body.

Invoke Response Body

The contents of the invoke response body depends on the value of the `status` member of the invoke response header, as well as the OMG IDL definition of the operation being invoked. Its format is one of the following:

- If the **status** value is INVOKE_NO_EXCEPTION, then the body contains the operation result value (if any), followed by all inout and out parameters, in the order in which they appear in the operation signature, from left to right.
- If the `status` value is INVOKE_USER_EXCEPTION or INVOKE_SYSTEM_EXCEPTION, then the body contains the exception, encoded as in GIOP.
- If the `status` value is INVOKE_LOCATION_FORWARD, then the body contains a new **MultipleComponentProfile** structure containing components that can be used to communicate with the object specified in the invoke request message. This profile must provide at least one new DCE-CIOP binding component. The client ORB is responsible for re-sending the request to the server identified by the new profile. This operation should be transparent to the client program making the request. See “DCE-CIOP Object Location” on page 13-21 for more details.
- If the `status` value is INVOKE_TRY_AGAIN, then the body is empty and the client should reissue the `invoke` RPC, possibly after a short delay[5].

13.4.3 DCE-CIOP Locate Request Message

Locate request messages may be sent from a client to a server, as the `request_message` parameter of a `locate` RPC, to determine the following regarding a specified object reference:

- Whether the object reference is valid
- Whether the current server is capable of directly receiving requests for the object reference
- If not capable, to solicit an address to which requests for the object reference should be sent.

For details on the usage of the `locate` RPC, see “DCE-CIOP Object Location” on page 13-21.

Locate request messages contain a fixed-format header, but no body.

Locate Request Header

DCE-CIOP locate request headers have the following format:

5. An exponential back-off algorithm is recommended, but not required.

```
module DCE_CIOP {                              // IDL
        struct LocateRequestHeader {
            boolean byte_order;
            sequence <octet> object_key;
            string endpoint_id;
            string operation;

            // no body follows
        };
};
```

The members have the following definitions:

- **byte_order** indicates the byte ordering used in the representation of the remainder of the message. A value of FALSE indicates big-endian byte ordering, and TRUE indicates little-endian byte ordering.
- **object_key** contains opaque data used to identify the object that is the target of the operation. See “Object Key Component” on page 13-19.
- **endpoint_id** member contains an identifier for the endpoint at which the object is located, if this was included in the IOR profile. If not, an empty string is used. See “Endpoint ID Component” on page 13-19.
- **operation** contains the name of the CORBA operation being invoked. It is encoded as in the invoke request header.

13.4.4 DCE-CIOP Locate Response Message

Locate response messages are sent from servers to clients as the `response_message` parameter of a `locate` RPC. They consist of a fixed-format header, and a body whose format depends on information in the header.

Locate Response Header

DCE-CIOP locate response headers have the following format:

```
module DCE_CIOP {                           // IDL
        enum LocateResponseStatus {
            LOCATE_UNKNOWN_OBJECT,
            LOCATE_OBJECT_HERE,
            LOCATE_LOCATION_FORWARD,
            LOCATE_TRY_AGAIN
        };

        struct LocateResponseHeader {
            boolean byte_order;
            LocateResponseStatus status;

            // if status = LOCATE_LOCATION_FORWARD, an
            // IOP::MultipleComponentProfile follows
        };
};
```

The members have the following definitions:

- **byte_order** indicates the byte ordering used in the representation of the remainder of the message. A value of FALSE indicates big-endian byte ordering, and TRUE indicates little-endian byte ordering.
- **status** indicates whether the object is valid and whether it is located in this server. It determines the contents of the body.

Locate Response Body

The contents of the locate response body depends on the value of the `status` member of the locate response header. Its format is one of the following:

- If the `status` value is LOCATE_UNKNOWN_OBJECT, then the object specified in the corresponding locate request message is unknown to the server. The locate reply body is empty in this case.
- If the `status` value is LOCATE_OBJECT_HERE, then this server (the originator of the locate response message) can directly receive requests for the specified object. The locate response body is also empty in this case.
- If the `status` value is LOCATE_LOCATION_FORWARD, then the locate response body contains a new **MultipleComponentProfile** structure containing components that can be used to communicate with the object specified in the locate request message. This profile must provide at least one new DCE-CIOP binding component.
- If the status value is LOCATE_TRY_AGAIN, the locate response body is empty and the client should reissue the `locate` RPC, possibly after a short delay[6].

6. An exponential back-off algorithm is recommended, but not required.

13.5 DCE-CIOP Object References

The information necessary to invoke operations on objects using DCE-CIOP is encoded in an IOR in a profile identified by **TAG_MULTIPLE_COMPONENTS**. The `profile_data` for this profile is a CDR encapsulation of the **MultipleComponentProfile** type, which is a sequence of **TaggedComponent** structures. These types are described in "An Information Model for Object References" on page 10-14.

DCE-CIOP defines a number of IOR components that can be included in a **MultipleComponentProfile**. Each is identified by a unique tag, and the encoding and semantics of the associated **component_data** are specified.

An IOR profile identified by **TAG_MULTIPLE_COMPONENTS** can contain components for other protocols in addition to DCE-CIOP, and can contain components used by other kinds of ORB services. For example, an ORB vendor can define its own private components within this profile to support the vendor's native protocol. Several of the components defined for DCE-CIOP may be of use to other protocols as well. The following component descriptions will note whether the component is intended solely for DCE-CIOP or can be used by other protocols, whether the component is required or optional for DCE-CIOP, and whether more than one instance of the component can be included in a profile.

A conforming implementation of DCE-CIOP is only required to generate and recognize the components defined here. Unrecognized components should be preserved but ignored. Implementations should also be prepared to encounter profiles identified by **TAG_MULTIPLE_COMPONENTS** that do not support DCE-CIOP.

13.5.1 DCE-CIOP String Binding Component

A DCE-CIOP string binding component, identified by **TAG_DCE_STRING_BINDING**, contains a fully or partially bound string binding. A string binding provides the information necessary for DCE-RPC to establish communication with a server process that can either service the client's requests itself, or provide the location of another process that can. The DCE API routine `rpc_binding_from_string_binding` can be used to convert a string binding to the DCE binding handle required to communicate with a server as described in "DCE-CIOP Message Transport" on page 13-5.

This component is intended to be used only by DCE-CIOP. At least one string binding or binding name component must be present for an IOR profile to support DCE-CIOP.

Multiple string binding components can be included in a profile to define endpoints for different DCE protocols, or to identify multiple servers or agents capable of servicing the request.

The string binding component is defined as follows:

```
module DCE_CIOP {                                              \\ IDL
        const IOP::ComponentId TAG_DCE_STRING_BINDING = 100;
};
```

A **TaggedComponent** structure is built for the string binding component by setting the tag member to **TAG_DCE_STRING_BINDING**, and setting the **component_data** member to the value of a DCE string binding. The string is represented directly in the sequence of octets, including the terminating NUL, without further encoding.

The format of a string binding is defined in Chapter 3 of the OSF *AES/Distributed Computing RPC Volume*. The DCE API function `rpc_binding_from_string_binding` converts a string binding into a binding handle that can be used by a client ORB as the first parameter to the `invoke` and `locate` RPCs.

A string binding contains:

- A protocol sequence
- A network address
- An optional endpoint
- An optional object UUID

DCE object UUIDs are used to identify server process endpoints, which can each support any number of CORBA objects. DCE object UUIDs do not necessarily correspond to individual CORBA objects.

A partially bound string binding does not contain an endpoint. Since the DCE-RPC run-time uses an endpoint mapper to complete a partial binding, and multiple ORB servers might be located on the same host, partially bound string bindings must contain object UUIDs to distinguish different endpoints at the same network address.

13.5.2 DCE-CIOP Binding Name Component

A DCE-CIOP binding name component is identified by **TAG_DCE_BINDING_NAME**. It contains a name that can be used with a DCE nameservice such as CDS or GDS to obtain the binding handle needed to communicate with a server process.

This component is intended for use only by DCE-CIOP. Multiple binding name components can be included to identify multiple servers or agents capable of handling a request. At least one binding name or string binding component must be present for a profile to support DCE-CIOP.

The binding name component is defined by the following OMG IDL:

```
module DCE_CIOP {                                              // IDL
        const IOP::ComponentId TAG_DCE_BINDING_NAME = 101;

        struct BindingNameComponent {
            unsigned long entry_name_syntax;
            string entry_name;
            string object_uuid;
        };
};
```

A **TaggedComponent** structure is built for the binding name component by setting the tag member to **TAG_DCE_BINDING_NAME**, and setting the **component_data member** to a CDR encapsulation of a **BindingNameComponent** structure.

BindingNameComponent

The **BindingNameComponent** structure contains the information necessary to query a DCE nameservice such as CDS. A client ORB can use the **entry_name_syntax, entry_name,** and **object_uuid** members of the **BindingName** structure with the `rpc_ns_binding_import_*` or `rpc_ns_binding_lookup_*` families of DCE API routines to obtain binding handles to communicate with a server. If the `object_uuid` member is an empty string, a nil object UUID should be passed to these DCE API routines.

13.5.3 DCE-CIOP No Pipes Component

The optional component identified by **TAG_DCE_NO_PIPES** indicates to an ORB client that the server does not support the `dce_ciop_pipe` DCE-RPC interface. It is only a hint, and can be safely ignored. As described in “DCE-CIOP Message Transport” on page 13-5, the client must fall back to the array-based interface if the pipe-based interface is not available in the server.

```
module DCE_CIOP {
        const IOP::ComponentId TAG_DCE_NO_PIPES = 102;
};
```

A **TaggedComponent** structure with a `tag` member of **TAG_DCE_NO_PIPES** must have an empty **component_data** member.

This component is intended for use only by DCE-CIOP, and a profile should not contain more than one component with this tag.

13.5.4 Object Key Component

An ORB server must include a single object key component, identified by **TAG_OBJECT_KEY**, in a DCE-CIOP IOR profile to hold the data it uses to identify the object. Its **component_data** value is used as the **object_key** member in invoke and locate request message headers.

The object key component is available for use by all protocols that use the **TAG_MULTIPLE_COMPONENTS** profile. By sharing this component, protocols can avoid duplicating object identity information.

```
module IOP {                                        \\ IDL
        const ComponentId TAG_OBJECT_KEY = 10;


};
```

The **component_data** of this component is not interpreted by the client process. Its format only needs to be understood by the server process and any location agent that it uses.

13.5.5 Endpoint ID Component

An optional endpoint ID component can be included in IOR profiles to enable client ORBs to minimize resource utilization and to avoid redundant locate messages. It can be used by other protocols as well as by DCE-CIOP. No more than one endpoint ID component should be included in a profile.

```
module IOP {                                        \\ IDL
        const ComponentId TAG_ENDPOINT_ID = 11;
};
```

An endpoint ID component, identified by **TAG_ENDPOINT_ID**, provides an identifier for the endpoint at which operations on an object can be invoked. The **component_data** is a NUL-terminated globally unique string identifying the endpoint. The recommended format for the **component_data** is a stringified UUID.

If multiple objects have the same endpoint ID, they can be messaged to at a single endpoint, avoiding the need to locate each object individually. DCE-CIOP clients can use a single binding handle to invoke requests on all of the objects with a common endpoint ID. See “Use of the Location Policy and the Endpoint ID” on page 13-23.

The endpoint ID component, if present in the IOR profile, is included in invoke and locate request message headers as the **endpoint_id** member. The server or agent can use the endpoint ID in conjunction with the object key to identify the object and its implementation. If no endpoint ID is included in the profile, an empty string is used as the `endpoint_id` member of the request messages.

13.5.6 Location Policy Component

An optional location policy component can be included in IOR profiles to specify when a DCE-CIOP client ORB should perform a `locate` RPC before attempting to perform an `invoke` RPC. No more than one location policy component should be included in a profile, and it can be used by other protocols that have location algorithms similar to DCE-CIOP.

```
module IOP {                                        \\ IDL
        const ComponentId TAG_LOCATION_POLICY = 12;

        const octet LOCATE_NEVER = 0;
        const octet LOCATE_OBJECT = 1;
        const octet LOCATE_OPERATION = 2;
        const octet LOCATE_ALWAYS = 3;
};
```

A **TaggedComponent** structure for a location policy component is built by setting the tag member to **TAG_LOCATION_POLICY**, and setting the **component_data** member to a sequence containing a single octet, whose value is **LOCATE_NEVER**, **LOCATE_OBJECT, LOCATE_OPERATION,** or **LOCATE_ALWAYS**.

If a location policy component is not present in a profile, the client should assume a location policy of **LOCATE_OBJECT**.

A client should interpret the location policy as follows:

- LOCATE_NEVER Perform only the `invoke` RPC. No `locate` RPC is necessary.
- LOCATE_OBJECT Perform a `locate` RPC once per object. The `operation` member of the locate request message will be ignored.
- LOCATE_OPERATION Perform a separate `locate` RPC for each distinct operation on the object. This policy can be used when different methods of an object are located in different processes.
- LOCATE_ALWAYS Perform a separate `locate` RPC for each invocation on the object. This policy can be used to support server-per-method activation.

The location policy is a hint that enables a client to avoid unnecessary `locate` RPCs and to avoid `invoke` RPCs that return **INVOKE_LOCATION_FORWARD** status. It is not needed to provide correct semantics, and can be ignored. Even when this hint is utilized, an `invoke` RPC might result in an **INVOKE_LOCATION_FORWARD** response. See “DCE-CIOP Object Location” on page 13-21 for more detail.

A client does not need to implement all location policies to make use of this hint. A location policy with a higher value can be substituted for one with a lower value. For instance, a client might treat **LOCATE_OPERATION** as **LOCATE_ALWAYS** to avoid having to keep track of binding information for each operation on an object.

When combined with an endpoint ID component, a location policy of **LOCATE_OBJECT** indicates that the client should perform a `locate` RPC for the first object with a particular endpoint ID, and then just perform an `invoke` RPC for other objects with the same endpoint ID. When a location policy of **LOCATE_NEVER** is combined with an endpoint ID component, only `invoke` RPCs need be performed. The **LOCATE_ALWAYS** and **LOCATE_OPERATION** policies should not be combined with an endpoint ID component in a profile.

13.6 DCE-CIOP Object Location

This section describes how DCE-CIOP client ORBs locate the server ORBs that can perform operations on an object via the `invoke` RPC.

13.6.1 Location Mechanism Overview

DCE-CIOP is defined to support object migration and location services without dictating the existence of specific ORB architectures or features. The protocol features are based on the following observations:

A given transport address does not necessarily correspond to any specific ORB architectural component (such as an object adapter, server process, ORB process, locator, etc.). It merely implies the existence of some agent to which requests may be sent.

The "agent" (receiver of an RPC) may have one of the following roles with respect to a particular object reference:

- The agent may be able to accept object requests directly for the object and return replies. The agent may or may not own the actual object implementation; it may be a gateway that transforms the request and passes it on to another process or ORB. From DCE-CIOP’s perspective, it is only important that invoke request messages can be sent directly to the agent.
- The agent may not be able to accept direct requests for any objects, but acts instead as a location service. Any invoke request messages sent to the agent would result in either exceptions or replies with **INVOKE_LOCATION_FORWARD** status, providing new addresses to which requests may be sent. Such agents would also respond to locate request messages with appropriate locate response messages.
- The agent may directly respond to some requests (for certain objects) and provide forwarding locations for other objects.

- The agent may directly respond to requests for a particular object at one point in time, and provide a forwarding location at a later time.

Server ORBs are not required to implement location forwarding mechanisms. An ORB can be implemented with the policy that servers either support direct access to an object, or return exceptions. Such a server ORB would always return locate response messages with either **LOCATE_OBJECT_HERE** or **LOCATE_UNKNOWN_OBJECT** status, and never **LOCATE_LOCATION_FORWARD** status. It would also never return invoke response messages with **INVOKE_LOCATION_FORWARD** status.

Client ORBs must, however, be able to accept and process invoke response messages with **INVOKE_LOCATION_FORWARD** status, since any server ORB may choose to implement a location service. Whether a client ORB chooses to send locate request messages is at the discretion of the client.

Client ORBs that send locate request messages can use the location policy component found in DCE-CIOP IOR profiles to decide whether to send a locate request message before sending an invoke request message. See "Location Policy Component" on page 13-20. This hint can be safely ignored by a client ORB.

A client should not make any assumptions about the longevity of addresses returned by location forwarding mechanisms. If a binding handle based on location forwarding information is used successfully, but then fails, subsequent attempts to send requests to the same object should start with the original address specified in the object reference.

In general, the use of location forwarding mechanisms is at the discretion of ORBs, available to be used for optimization and to support flexible object location and migration behaviors.

13.6.2 Activation

Activation of ORB servers is transparent to ORB clients using DCE-CIOP. Unless an IOR refers to a transient object, the agent addressed by the IOR profile should either be permanently active, or should be activated on demand by DCE's endpoint mapper.

The current DCE endpoint mapper, rpcd, does not provide activation. In ORB server environments using rpcd, the agent addressed by an IOR must not only be capable of locating the object, it must also be able to activate it if necessary. A future DCE endpoint mapper may provide automatic activation, but client ORB implementations do not need to be aware of this distinction.

13.6.3 Basic Location Algorithm

ORB clients can use the following algorithm to locate the server capable of handling the `invoke` RPC for a particular operation:

1. Pick a profile with **TAG_MULTIPLE_COMPONENTS** from the IOR. Make the **MultipleComponentProfile** structure encoded in the **profile_data** of this the *original* profile and the *current* profile. If no profiles with **TAG_MULTIPLE_COMPONENTS** are available, operations cannot be invoked using DCE-CIOP with this IOR.

2. Get a binding handle to try from the *current* profile. See "DCE-CIOP String Binding Component" on page 13-16 and "DCE-CIOP Binding Name Component" on page 13-17. If no binding handles can be obtained, the server cannot be located using the *current* profile, so go to step 1.

3. Perform either a `locate` or `invoke` RPC using the **TAG_OBJECT_KEY** component and optional **TAG_ENDPOINT_ID** component from the *original* profile.
   - If the RPC fails, go to step 2 to try a different binding handle.
   - If the RPC returns **INVOKE_TRY_AGAIN** or **LOCATE_TRY_AGAIN**, try the same RPC again, possibly after a delay.
   - If the RPC returns either **INVOKE_LOCATION_FORWARD** or **LOCATE_LOCATION_FORWARD**, make the new **MultipleComponentProfile** structure returned in the response message body the *current* profile and go to step 2.
   - If the RPC returns **LOCATE_UNKNOWN_OBJECT**, the object no longer exists.
   - Otherwise, the server has been successfully located.

Any `invoke` RPC might return **INVOKE_LOCATION_FORWARD**, in which case the client ORB should make the returned **MultipleComponentProfile** structure the *current* profile, and re-enter the location algorithm at step 2.

If an RPC on a binding handle fails after it has been used successfully, the client ORB should start over at step 1.

Note that the **TAG_OBJECT_KEY** and **TAG_ENDPOINT_ID** components for all `invoke` and `locate` RPCs are taken from the *original* profile. These components should not be included in the **MultipleComponentProfile** structure returned in **INVOKE_LOCATION_FORWARD** and **LOCATE_LOCATION_FORWARD** response messages. Only the **TAG_DCE_STRING_BINDING** and **TAG_DCE_BINDING_NAME** components, and possibly the optional **TAG_LOCATION_POLICY** and **TAG_DCE_NO_PIPES** components are taken from the *current* profile.

13.6.4 Use of the Location Policy and the Endpoint ID

The algorithm above will allow a client ORB to successfully locate a server ORB, if possible, so that operations can be invoked using DCE-CIOP. But unnecessary `locate` RPCs may be performed, and `invoke` RPCs may be performed when `locate` RPCs would be more efficient. The optional location policy and endpoint ID components can be used by the client ORB, if present in the IOR profile, to optimize this algorithm.

Current Location Policy

The client ORB can decide whether to perform a `locate` RPC or an `invoke` RPC in step 3 based on the location policy of the *current* IOR profile. If the *current* profile has a **TAG_LOCATION_POLICY** component with a value of **LOCATE_NEVER**, the client should perform an `invoke` RPC. Otherwise, it should perform a `locate` RPC.

Original Location Policy

The client ORB can use the location policy of the *original* IOR profile as follows to determine whether it is necessary to perform the location algorithm for a particular invocation:

- LOCATE_OBJECT or LOCATE_NEVER A binding handle previously used successfully to invoke an operation on an object can be reused for all operations on the same object. The client only needs to perform the location algorithm once per object.
- LOCATE_OPERATION A binding handle previously used successfully to invoke an operation on an object can be reused for that same operation on the same object. The client only needs to perform the location algorithm once per operation.
- LOCATE_ALWAYS Binding handles should not be reused. The client needs to perform the location algorithm once per invocation.

Original Endpoint ID

If a component with **TAG_ENDPOINT_ID** is present in the *original* IOR profile, the client ORB can reuse a binding handle that was successfully used to perform an operation on another object with the same endpoint ID `component_data` value. The client only needs to perform the location algorithm once per endpoint.

An endpoint ID component should never be combined in the same profile with a location policy of **LOCATE_OPERATION** or **LOCATE_ALWAYS**.

13.7 OMG IDL for the DCE CIOP Module

This section shows the DCE_CIOP module.

```
module DCE_CIOP {
        struct InvokeRequestHeader {
            boolean byte_order;
            IOP::ServiceContextList service_context;
            sequence <octet> object_key;
            string endpoint_id;
            string operation;
            CORBA::Principal principal;
            sequence <string> client_context;

            // in and inout parameters follow
        };

module DCE_CIOP {
        enum InvokeResponseStatus {
            INVOKE_NO_EXCEPTION,
            INVOKE_USER_EXCEPTION,
            INVOKE_SYSTEM_EXCEPTION,
            INVOKE_LOCATION_FORWARD,
            INVOKE_TRY_AGAIN
        };

        struct InvokeResponseHeader {
            boolean byte_order;
            IOP::ServiceContextList service_context;
            InvokeResponseStatus status;

            // if status = INVOKE_NO_EXCEPTION,
            // result then inouts and outs follow

            // if status = INVOKE_USER_EXCEPTION or
            // INVOKE_SYSTEM_EXCEPTION, an exception follows

            // if status = INVOKE_LOCATION_FORWARD, an
            // IOP::MultipleComponentsProfile follows
        };


module DCE_CIOP {
        struct LocateRequestHeader {
            boolean byte_order;
            sequence <octet> object_key;
            string endpoint_id;
            string operation;

            // no body follows
        };
```

```
module IOP {
        const ComponentId TAG_OBJECT_KEY = 10;
        const ComponentId TAG_ENDPOINT_ID = 11;
        const ComponentId TAG_LOCATION_POLICY = 12;
        const octet LOCATE_NEVER = 0;
        const octet LOCATE_OBJECT = 1;
        const octet LOCATE_OPERATION = 2;
        const octet LOCATE_ALWAYS = 3;
};
```

13.8 References for this Chapter

AES/Distributed Computing RPC Volume, P T R Prentice Hall, Englewood Cliffs, New Jersey, 1994

CAE Specification C309 X/Open DCE: Remote Procedure Call, X/Open Company Limited, Reading, UK

Interworking Architecture 13A

The Interworking chapters describe a specification for communication between two similar but very distinct object management systems: Microsoft's COM (including OLE) and the OMG's CORBA. An optimal specification would allow objects from either system to make their key functionality visible to clients using the other system as transparently as possible. The architecture for Interworking is designed to meet this goal.

13.1 Purpose of the Interworking Architecture

The purpose of the Interworking architecture is to specify support for two-way communication between CORBA objects and COM objects. The goal is that objects from one object model should be able to be viewed as if they existed in the other object model. For example, a client working in a CORBA model should be able to view a COM object as if it were a CORBA object. Likewise, a client working in a COM object model should be able to view a CORBA object as if it were a COM object.

There are many similarities between the two systems. In particular, both are centered around the idea that an object is a discrete unit of functionality that presents its behavior through a set of fully-described interfaces. Each system hides the details of implementation from its clients. To a large extent COM and CORBA are semantically isomorphic. Much of the COM/CORBA Interworking specification simply involves a mapping of the syntax, structure and facilities of each to the other — a straightforward task.

There are, however, differences in the CORBA and COM object models. COM and CORBA each have a different way of describing what an object is, how it is typically used, and how the components of the object model are organized. Even among largely isomorphic elements, these differences raise a number of issues as to how to provide the most transparent mapping.

13.1.1 Comparing COM Objects to CORBA Objects

From a COM point of view, an object is typically a subcomponent of an application, which represents a point of exposure to other parts of the application, or to other applications. Many OLE objects are document-centric and are often (though certainly not exclusively) tied to some visual presentation metaphor. Historically, the typical domain of an COM object is a single-user, multitasking visual desktop such as a Microsoft Windows desktop. Currently, the main goal of COM and OLE is to expedite collaboration- and information-sharing among applications using the same desktop, largely through user manipulation of visual elements (for example, drag-and-drop, cut-and-paste).

From a CORBA point of view, an object is an independent component providing a related set of behaviors. An object is expected to be available transparently to any CORBA client regardless of the location (or implementation) of either the object or the client. Most CORBA objects focus on distributed control in a heterogeneous environment. Historically, the typical domain of a CORBA object is an arbitrarily scalable distributed network. In its current form, the main goal of CORBA is to allow these independent components to be shared among a wide variety of applications (and other objects), any of which may be otherwise unrelated.

Of course, CORBA is already used to define desktop objects, and COM can be extended to work over a network. Also, both models are growing and evolving, and will probably overlap in functionally in the future. Therefore, a good interworking model must map the functionality of two systems to each other while preserving the flavor of each system as it is typically presented to a developer.

The most obvious similarity between these two systems is that they are both based architecturally on *objects*. The Interworking Object Model describes the overlap between the features of the CORBA and COM object models, and how the common features map between the two models.

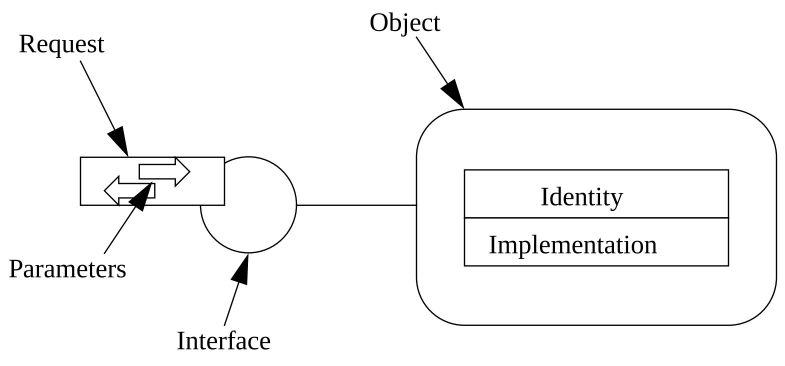


Figure 13-1 Interworking Object Model

13.2 Interworking Object Model

13.2.1 Relationship to CORBA Object Model

In the Interworking Object Model, each object is simply a discrete unit of functionality that presents itself through a published interface described in terms of a well-known, fully-described set of interface semantics. An interface (and its underlying functionality) is accessed through at least one well-known, fully described form of request. Each request in turn targets a specific object—an object instance—based on a reference to its identity. That target object is then expected to service the request by invoking the expected behavior in its own particular implementation. Request parameters are object references or nonobject data values described in the object model's data type system. Interfaces may be composed by combining other interfaces according to some well-defined composition rules. In each object system, interfaces are described in a specialized language or can be represented in some repository or library.

In CORBA, the Interworking Object Model is mapped to an architectural abstraction known as the Object Request Broker (ORB). Functionally, an ORB provides for the registration of the following:

- Types and their interfaces, as described in the OMG Interface Definition Language (OMG IDL).
- Instance identities, from which the ORB can then construct appropriate references to each object for interested clients.

A CORBA object may thereafter receive requests from interested clients that hold its object reference and have the necessary information to make a properly-formed request on the object's interface. This request can be statically defined at compile time or dynamically created at run-time based upon type information available through an interface type repository.

While CORBA specifies the existence of an implementation type description called ImplementationDef (and an Implementation Repository, which contains these type descriptions), CORBA does not specify the interface or characteristics of the Implementation Repository or the ImplementationDef. As such, implementation typing and descriptions vary from ORB to ORB and are not part of this specification.

13.2.2 Relationship to the OLE/COM Model

In OLE, the Interworking Object Model is principally mapped to the architectural abstraction known as the Component Object Model (COM). Functionally, COM allows an object to expose its interfaces in a well-defined binary form (that is, a virtual function table) so that clients with static compile-time knowledge of the interface's structure, and with a reference to an instance offering that interface, can send it appropriate requests. Most COM interfaces are described in Microsoft Interface Definition Language (MIDL).

COM supports an implementation typing mechanism centered around the concept of a COM class. A COM class has a well-defined identity and there is a repository (known as the system registry) that maps implementations (identified by class IDs) to specific executable code units that embody the corresponding implementation realizations.

COM also provides an extension called OLE Automation. Interfaces that are Automation-compatible can be described in Object Definition Language (ODL) and can optionally be registered in a binary Type Library. Automation interfaces can be invoked dynamically by a client having no compile-time interface knowledge through a special COM interface (IDispatch). Run-time type checking on invocations can be implemented when a Type Library is supplied. Automation interfaces have properties and methods, whereas COM interfaces have only methods. The data types that may be used for properties and as method parameters comprise a subset of the types supported in COM, with no support for user-defined constructed types.

Thus, use of and interoperating with objects exposing OLE Automation interfaces is considerably different from other COM objects. Although Automation is implemented through COM, for the purposes of this document, OLE Automation and COM are considered to be distinct object models. Interworking between CORBA and OLE Automation will be described separately from interworking with the basic COM model.

13.2.3 Basic Description of the Interworking Model

Viewed at this very high level, Microsoft's COM and OMG's CORBA appear quite similar. Roughly speaking, COM interfaces (including Automation interfaces) are equivalent to CORBA interfaces. In addition, COM interface pointers are very roughly equivalent to CORBA object references. Assuming that lower-level design details (calling conventions, data types, and so forth) are more or less semantically isomorphic, a reasonable level of interworking is probably possible between the two systems through straightforward mappings.

How such interworking can be practically achieved is illustrated in an Interworking Model, shown in Figure 13-2. It shows how an object in Object System B can be mapped and represented to a client in Object System A. From now on, this will be called a B/A mapping. For example, mapping a CORBA object to be visible to a COM client is a CORBA/COM mapping.

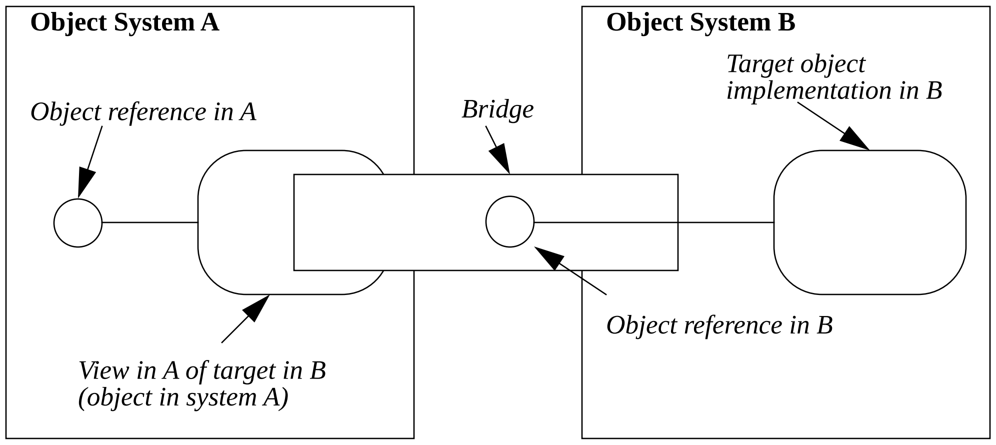


Figure 13-2 B/A Interworking Model

On the left is a client in object system A, that wants to send a request to a target object in system B, on the right. We refer to the entire conceptual entity that provides the mapping as a bridge. The goal is to map and deliver any request from the client transparently to the target.

To do so, we first provide an object in system A called a View. The View is an object in system A that presents the identity and interface of the target in system B mapped to the vernacular of system A, and is described as an A View of a B target.

The View exposes an interface, called the View Interface, which is isomorphic to the target's interface in system B. The methods of the View Interface convert requests from system A clients into requests on the target's interface in system B. The View is a component of the bridge. A bridge may be composed of many Views.

The bridge maps interface and identify forms between different object systems. Conceptually, the bridge holds a reference in B for the target (although this is not physically required). The bridge must provide a point of rendezvous between A and B, and may be implemented using any mechanism that permits communication between the two systems (IPC, RPC, network, shared memory, and so forth) sufficient to preserve all relevant object semantics.

The client treats the View as though it is the real object in system A, and makes the request in the vernacular request form of system A. The request is translated into the vernacular of object system B, and delivered to the target object. The net effect is that a request made on an interface in A is transparently delivered to the intended instance in B.

The Interworking Model works in either direction. For example, if system A is COM, and system B is CORBA, then the View is called the COM View of the CORBA target. The COM View presents the target's interface to the COM client. Similarly if system A is CORBA and system B is COM, then the View is called the *CORBA View* of the COM target. The CORBA View presents the target's interface to the CORBA client.

Figure 13-3 shows the interworking mappings discussed in the Interworking chapters. They represent the following:

- The mapping providing a COM View of a CORBA target
- The mapping providing a CORBA View of a COM target
- The mapping providing an Automation View of a CORBA target
- The mapping providing a CORBA View of an Automation target

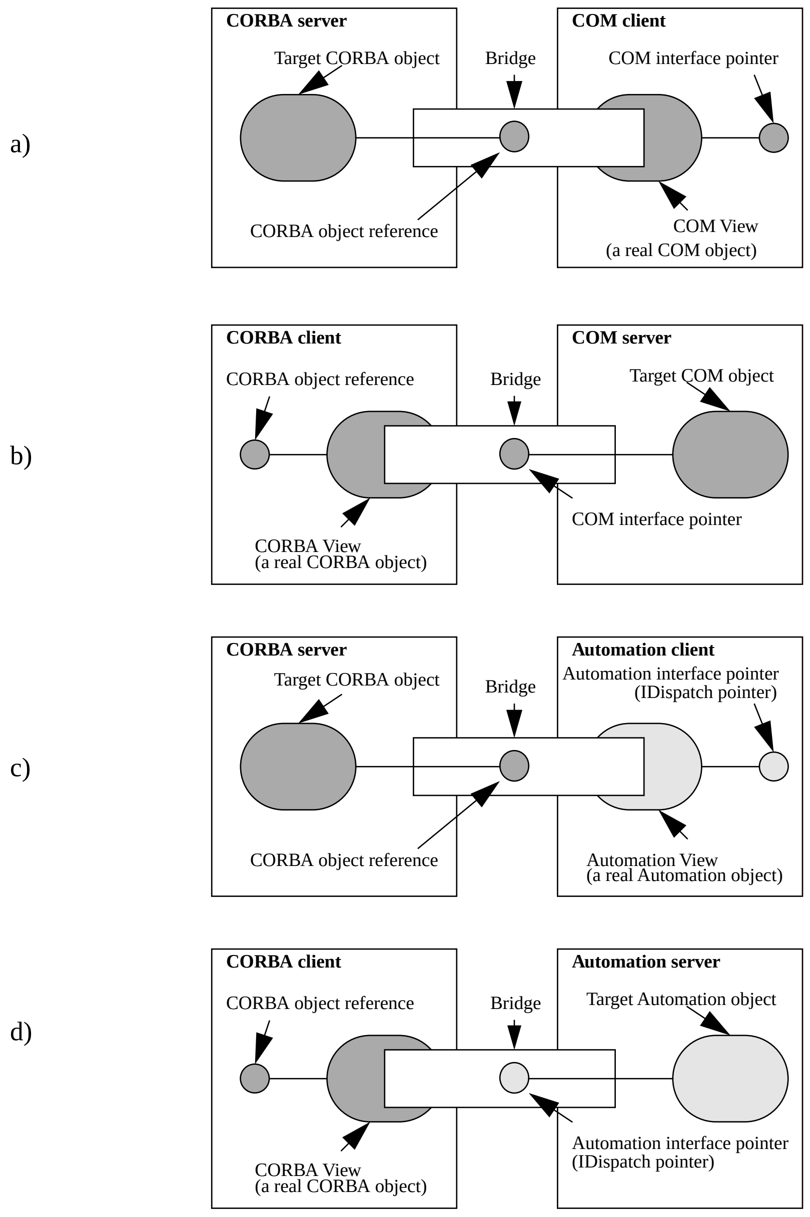


Figure 13-3 Interworking Mapping

Note that the division of the mapping process into these architectural components does not infer any particular design or implementation strategy. For example, a COM View and its encapsulated CORBA reference could be implemented in COM as a single component or as a system of communicating components on different hosts.

The architecture allows for a range of implementation strategies, including, but not limited to generic and interface-specific mapping.

- **Generic Mapping** assumes that all interfaces can be mapped through a dynamic mechanism supplied at run-time by a single set of bridge components. This allows automatic access to new interfaces as soon as they are registered with the target system. This approach generally simplifies installation and change management, but may incur the run-time performance penalties normally associated with dynamic mapping.
- **Interface-Specific Mapping** assumes that separate bridge components are generated for each interface or for a limited set of related interfaces (for example, by a compiler). This approach generally improves performance by "precompiling" request mappings, but may create installation and change management problems.

13.3 Interworking Mapping Issues

The goal of the Interworking specification is to achieve a straightforward two-way (COM/CORBA and CORBA/COM) mapping in conformance with the previously described Interworking Model. However, despite many similarities, there are some significant differences between CORBA and COM that complicate achieving this goal. The most important areas involve:

- **Interface Mapping**. A CORBA interface must be mapped to and from two distinct forms of interfaces, OLE Automation and COM.
- **Interface Composition Mapping**. CORBA multiple inheritance must be mapped to COM single inheritance/aggregation. COM interface aggregation must be mapped to the CORBA multiple inheritance model.
- **Identity Mapping**. The explicit notion of an instance identity in CORBA must be mapped to the more implicit notion of instance identity in COM.
- **Mapping Invertibility**. It may be desirable for the object model mappings to be invertible, but the Interworking specification does not guarantee invertibility in all situations.

13.4 Interface Mapping

The CORBA standard for describing interfaces is OMG IDL. It describes the requests that an object supports. OLE provides two distinct and somewhat disjointed interface models: COM and Automation. Each has its own respective request form, interface semantics, and interface syntax.

Therefore, we must consider the problems and benefits of four distinct mappings:

- CORBA/COM
- CORBA/Automation
- COM/CORBA
- Automation/CORBA

We must also consider the bidirectional impact of a third, hybrid form of interface, the Dual Interface, which supports both an Automation and a COM-like interface. The succeeding sections summarize the main issues facing each of these mappings.

13.4.1 CORBA/COM

There is a reasonably good mapping from CORBA objects to COM Interfaces; for instance:

- OMG IDL primitives map closely to COM primitives.
- Constructed data types (structs, unions, arrays, strings, and enums) also map closely.
- CORBA object references map closely to COM interface pointers.
- Inherited CORBA interfaces may be represented as multiple COM interfaces.
- CORBA attributes may be mapped to get and set operations in COM interfaces.

This mapping is perhaps the most natural way to represent the interfaces of CORBA objects in the COM environment. In practice, however, many COM clients (for example, Visual Basic applications) can only bind to Automation Interfaces and cannot bind to the more general COM Interfaces. Therefore, providing only a mapping of CORBA to the COM Interfaces would not satisfy many COM/OLE clients.

13.4.2 CORBA/Automation

There is a limited fit between OLE Automation objects and CORBA objects:

- Some OMG IDL primitives map directly to Automation primitives. However, there are primitives in both systems (for example, the OLE CURRENCY type and the CORBA unsigned integral types) that must be mapped as special cases (possibly with loss of range or precision).
- OMG IDL constructed types do not map naturally to any Automation constructs. Since such constructed types cannot be passed as argument parameters in Automation interfaces, these must be simulated by providing specially constructed interfaces (for example, viewing a struct as an OLE object with its own interface).
- CORBA Interface Repositories can be mapped dynamically to Automation Type Libraries.
- CORBA object references map to Automation interface pointers.
- There is no clean mapping for multiple inheritance to OLE Automation interfaces. All methods of the multiply-inherited interfaces could be expanded to a single Automation interface; however, this approach would require a total ordering over the methods if [dual] interfaces are to be supported. An alternative approach would be to map multiple inheritance to multiple Automation interfaces. This mapping, however, would require that an interface navigation mechanism be exposed to OLE Automation controllers. Currently OLE Automation does not provide a canonical way for clients (such as Visual Basic) to navigate between multiple interfaces.

- CORBA attributes may be mapped to get and put properties in Automation interfaces.

This form of interface mapping will place some restrictions on the types of argument passing that can be mapped, and/or the cost (in terms of run-time translations) incurred in those mappings. Nevertheless, it is likely to be the most popular form of CORBA-to-COM interworking, since it will provide dynamic access to CORBA objects from Visual Basic and other OLE Automation client development environments.

13.4.3 COM/CORBA

This mapping is similar to CORBA/COM, except for the following:

- Some COM primitive data types (for example, UNICODE long, unsigned long long, and wide char) and constructed types (for example, wide string) are not currently supported by OMG IDL. (These data types may be added to OMG IDL in the future.)
- Some unions, pointer types and the SAFEARRAY type require special handling.

The COM/CORBA mapping is somewhat further complicated, by the following issues:

- Though it is less common, COM objects may be built directly in C and C++ (without exposing an interface specification) by providing custom marshaling implementations. If the interface can be expressed precisely in some COM formalism (MIDL, ODL, or a Type Library), it must first be hand-translated to such a form before any formal mapping can be constructed. If not, the interworking mechanism (such as the View, request transformation, and so forth) must be custom-built.
- MIDL, ODL, and Type Libraries are somewhat different, and some are not supported on certain Windows platforms; for example, MIDL is not available on Win16 platforms.

13.4.4 Automation/CORBA

The Automation interface model and type system are markedly constrained, bounding the size of the problem of mapping from OLE Automation interfaces to CORBA interfaces.

- Automation interfaces and references (IDispatch pointers) map directly to CORBA interfaces and object references.
- Automation request signatures map directly into CORBA request signatures.
- Most of the Automation data types map directly to CORBA data types. Certain Automations types (for example, CURRENCY) do not have corresponding predefined CORBA types, but can easily be mapped onto isomorphic constructed types.
- Automation properties map to CORBA attributes.

13.5 Interface Composition Mappings

CORBA provides a multiple inheritance model for aggregating and extending object interfaces. Resulting CORBA interfaces are, essentially, statically defined either in OMG IDL files or in the Interface Repository. Run-time interface evolution is possible by deriving new interfaces from existing ones. Any given CORBA object reference refers to a CORBA object that exposes, at any point in time, a single most-derived interface in which all ancestral interfaces are joined. The CORBA object model does not support objects with multiple, disjoint interfaces.[1]

In contrast, COM objects expose aggregated interfaces by providing a uniform mechanism for navigating among the interfaces that a single object supports (that is, the QueryInterface method). In addition, COM anticipates that the set of interfaces that an object supports will vary at run-time. The only way to know if an object supports an interface at a particular instant is to ask the object.

OLE Automation objects typically provide all Automation operations in a single "flattened" IDispatch interface. While an analogous mechanism to QueryInterface could be supported in OLE Automation as a standard method, it is not the current use model for OLE Automation services.[2]

13.5.1 CORBA/COM

CORBA multiple inheritance maps into COM interfaces with some difficulty. Examination of object-oriented design practice indicates two common uses of interface inheritance, extending and mixing in. Inheritance may be used to extend an interface linearly, creating a specialization or new version of the inherited interface. Inheritance (particularly multiple inheritance) is also commonly used to mix in a new capability (such as the ability to be stored or displayed) that may be orthogonal to the object's basic application function.

Ideally, extension maps well into a single inheritance model, producing a single linear connection of interface elements. This usage of CORBA inheritance for specialization maps directly to COM; a unique CORBA interface inheritance path maps to a single COM interface vtable that includes all of the elements of the CORBA interfaces in the inheritance path.[3] The use of inheritance to mix in an interface maps well into COM's aggregation mechanism; each mixed-in inherited interface (or interface graph) maps to a separate COM interface, which can be acquired by invoking QueryInterface with the interface's specific UUID.

Unfortunately, with CORBA multiple inheritance there is no syntactic way to determine whether a particular inherited interface is being extended or being mixed in (or used with some other possible design intent). Therefore it is not possible to make

1. This is established in the CORBA 2.0 specification, Section 1.2.5, and in the Object Management Architecture Guide, Section 4.4.7.
2. One can use [dual] interfaces to expose multiple IDispatch interfaces for a given COM co-class. The "Dim A as new Z" statement in Visual Basic 4.0 can be used to invoke a QueryInterface for the Z interface. Many OLE Automation controllers, however, do not use the dual interface mechanism.

ideal mappings mechanically from CORBA multiply-inherited interfaces to collections of COM interfaces without some additional annotation that describes the intended design. Since extending OMG IDL (and the CORBA object model) to support distinctions between different uses of inheritance is undesirable, alternative mappings require arbitrary decisions about which nodes in a CORBA inheritance graph map to which aggregated COM interfaces, and/or an arbitrary ordering mechanism. The mapping described in Section 13.5.2, Ordering Rules for the CORBA->MIDL Transformation, describes a compromise that balances the need to preserve linear interface extensions with the need to keep the number of resulting COM interfaces manageably small. It satisfies the primary requirement for interworking in that it describes a uniform, deterministic mapping from any CORBA inheritance graph to a composite set of COM interfaces.

COM/CORBA

The features of COM's interface aggregation model can be preserved in CORBA by providing a set of CORBA interfaces that can be used to manage a collection of multiple CORBA objects with different disjoint interfaces as a single composite unit. The mechanism described in OMG IDL in Section 13.2.10, Interface Mapping, is sufficiently isomorphic to allow composite COM interfaces to be uniformly mapped into composite OMG IDL interfaces with no loss of capability.

CORBA/Automation

OLE Automation (as exposed through the IDispatch interface) does not rely on ordering in a virtual function table. The target object implements the IDispatch interface as a mini interpreter and exposes what amounts to a flattened single interface for all operations exposed by the object. The object is not required to define an ordering of the operations it supports.

An ordering problem still exists, however, for dual interfaces. Dual interfaces are COM interfaces whose operations are restricted to the Automation data types. Since these are COM interfaces, the client can elect to call the operations directly by mapping the operation to a predetermined position in a function dispatch table. Since the interpreter is being bypassed, the same ordering problems discussed in the previous section apply for OLE Automation dual interfaces.

3. An ordering is needed over the CORBA operations in an interface to provide a deterministic mapping from the OMG IDL interface to a COM vtable. The current ordering is lexicographical by bytes in machine-collating sequence.

Automation/CORBA

OLE Automation interfaces are simple collections of operations, with no inheritance or aggregation issues. Each IDispatch interface maps directly to an equivalent OMG IDL-described interface.

13.5.2 Detailed Mapping Rules

Ordering Rules for the CORBA->MIDL Transformation

- Each OMG IDL interface that does not have a parent is mapped to an MIDL interface deriving from IUnknown.
- Each OMG IDL interface that inherits from a single parent interface is mapped to an MIDL interface that derives from the mapping for the parent interface.
- Each OMG IDL interface that inherits from multiple parent interfaces is mapped to an MIDL interface deriving from IUnknown.
- For each CORBA interface, the mapping for operations precede the mapping for attributes.
- The resulting mapping of operations within an interface are ordered based upon the operation name. The ordering is lexicographic by bytes in machine-collating order.
- The resulting mapping of attributes within an interface are ordered based upon the attribute name. The ordering is lexicographic by bytes in machine-collating order. If the attribute is not read-only, the get_<attribute name> method immediately precedes the set_<attribute name> method.

Ordering Rules for the CORBA->OLE Automation Transformation

- Each OMG IDL interface that does not have a parent is mapped to an ODL interface deriving from IDispatch.
- Each OMG IDL interface that inherits from a single parent interface is mapped to an ODL interface that derives from the mapping for the parent interface.
- Each OMG IDL interface that inherits from multiple parent interfaces is mapped to an ODL interface which derives using single inheritance from the mapping for the first parent interface. The first parent interface is defined as the first interface when the immediate parent interfaces are sorted based upon interface repository id. The order of sorting is lexicographic by bytes in machine-collating order.
- Within an interface, the mapping for operations precede the mapping for attributes.
- An OMG IDL interface’s operations are ordered in the resulting mapping based upon the operation name. The ordering is lexicographic by bytes in machine-collating order.

- An OMG IDL interface's attributes are ordered in the resulting mapping based upon the attribute name. The ordering is lexicographic by bytes in machine-collating order. For non-read-only attributes, the [propget] method immediately precedes the [propput] method.
- For OMG IDL interfaces that multiply inherit from parent interfaces, the operations introduced in the current interface are mapped first and ordered based on the above rules. After the interface's operations are mapped, the operations are followed by the ordered operations from the mapping of the parent interfaces (excluding the first interface which was mapped using inheritance).

13.5.3 Example of Applying Ordering Rules

Consider the OMG IDL description shown in Figure 13-4.

```
interface A {// OMG IDL
    void opA();
    attribute long val;
};
interface B : A {
    void opB();
};
interface C: A {
    void opC();
};
interface D : B, C {
    void opD();
};
interface E {
    void opE();
};
interface F : D, E {
    void opF();
};
```

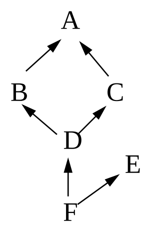


Figure 13-4 OMG IDL Description with Multiple Inheritance

Following the rules in Section 13.5.2, Ordering Rules for the CORBA->MIDL Transformation, the interface description would map to the Microsoft MIDL definition shown in Figure 13-5 and would map to the ODL definition shown in Figure 13-6.

```
[object, uuid(7fc56270-e7a7-0fa8-1d59-35b72eacbe29)]
interface IA : IUnknown{// Microsoft MIDL
    HRESULT opA();
    HRESULT get_val([out] long * val);
    HRESULT set_val([in] long val);
};
[object, uuid(9d5ed678-fe57-bcca-1d41-40957afab571)]
interface IB : IA {
    HRESULT opB();
```

$$\begin{tikzcd}
IU & IU & IU & IU & IU \\
A \arrow[u] & A \arrow[u] & D \arrow[u] & E \arrow[u] & F \arrow[u] \\
B \arrow[u] & C \arrow[u] & & &
\end{tikzcd}$$

```
};
[object,uuid(0d61f837-0cad-1d41-1d40-b84d143e1257)]
interface IC: IA {
    HRESULT opC();
};
[object, uuid(f623e75a-f30e-62bb-1d7d-6df5b50bb7b5)]
interface ID : IUnknown {
    HRESULT opD();
};
[object, uuid(3a3ea00c-fc35-332c-1d76-e5e9a32e94da)]
interface IE : IUnknown{
    HRESULT opE();
};
[object, uuid(80061894-3025-315f-1d5e-4e1f09471012)]
interface IF : IUnknown {
    HRESULT opF();
};
```

Figure 13-5 MIDL Description

```
[uuid(7fc56270-e7a7-0fa8-1dd9-35b72eacbe29),
oleautomation, dual]
interface DA : IDispatch {                                      //
Microsoft ODL
    HRESULT opA([out, optional] VARAINT* v);
    [propget]
    HRESULT val([out] long *val);
    [propset]
    HRESULT val([in] long val);
};
[uuid(9d5ed678-fe57-bcca-1dc1-40957afab571),
oleautomation,dual]
interface DB : DA {
    HRESULT opB([out, optional]VARIANT * v);
};
[uuid(0d61f837-0cad-1d41-1dc0-b84d143e1257),
oleautomation, dual]
interface DC: DA {
    HRESULT opC([out, optional]VARIANT *v);
};
[uuid(f623e75a-f30e-62bb-1dfd-6df5b50bb7b5),
oleautomation, dual]
interface DD : DB {
    HRESULT opD([out, optional]VARIANT *v);
    HRESULT opC([out, optional] VARIANT *v);
};
[uuid(3a3ea00c-fc35-332c-1df6-e5e9a32e94da),
oleautomation, dual]
interface DE : IDispatch{
    HRESULT opE([out, optional] VARIANT *v);
};
[uuid(80061894-3025-315f-1dde-4e1f09471012)
oleautomation, dual]
interface DF : DD {
    HRESULT opF([out, optional] VARIANT *v);
    HRESULT opE([out, optional] VARIANT *v);
};
```

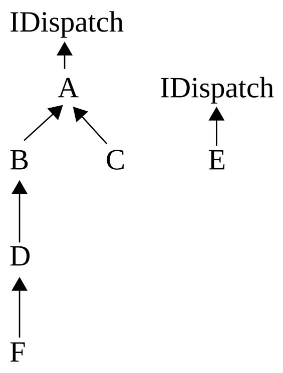


Figure 13-6 Example: ODL Mapping for Multiple Inheritance

13.5.4 Mapping Interface Identity

This specification enables interworking solutions from different vendors to interoperate across client/server boundaries (for example, a COM View created by product A can invoke a CORBA server created with product B, given that they both share the same IDL interface). To interoperate in this way, all COM Views mapped from a particular CORBA interface must share the same COM Interface IDs. This section describes a uniform mapping from CORBA Interface Repository IDs to COM Interface IDs.

Mapping Interface Repository IDs to COM IIDs

A CORBA Repository ID is mapped to a corresponding COM Interface ID using a derivative of the RSA Data Security, Inc. MD5 Message-Digest algorithm.[4,5] The repository ID of the CORBA interface is fed into the MD5 algorithm to produce a 128-bit hash identifier. The least significant byte is byte 0 and the most significant byte is byte 8. The resulting 128 bits are modified as follows.

Note – The DCE UUID space is currently divided into four main groups:

byte 8 =	0xxxxxxx (the NCS1.4 name space)
	10xxxxxx (A DCE 1.0 UUID name space)
	110xxxxx (used by Microsoft)
	1111xxxx (Unspecified)

For NCS1.5, the other bits in byte 8 specify a particular family. Family 29 will be assigned to ensure that the autogenerated IIDs do not interfere with other UUID generation techniques.

The upper two bits of byte 9 will be defined as follows.

```
00 unspecified
01 generated COM IID
10 generated Automation IID
11 generated dual interface Automation ID
```

Note – These bits should never be used to determine the type of interface. They are used only to avoid collisions in the name spaces when generating IIDs for multiple types of interfaces — dual, COM, or Automation.

The other bits in the resulting key are taken from the MD5 message digest (stored in the UUID with little endian ordering).

The IID generated from the CORBA repository ID will be used for a COM view of a CORBA interface except when the repository ID is a DCE UUID and the IID being generated is for a COM interface (not Automation or dual). In this case, the DCE UUID will be used as the IID instead of the IID generated from the repository ID (this is done to allow CORBA server developers to implement existing COM interfaces).

4. Rivest, R. “The MD5 Message-Digest Algorithm,” RFC 1321, MIT and RSA Data Security, Inc., April 1992.

5. MD5 was chosen as the hash algorithm because of its uniformity of distribution of bits in the hash value and its popularity for creating unique keys for input text. The algorithm is designed such that on average, half of the output bits change for each bit change in the input. The original algorithm provides a key with uniform distribution in 128 bits. The modification used in this specification selects 118 bits. With a uniform distribution, the probability of drawing k distinct keys (using k distinct inputs) is $n!/((n-k)!*n^k)$, where n is the number of distinct key values (i.e., $n=2^{118}$). If a million (i.e., $k=10^6$) distinct interface repository IDs are passed through the algorithm, the probability of a collision in any of the keys is less than 1 in 10^{23}.

This mechanism requires no change to IDL. However, there is an implicit assumption that repository IDs should be unique across ORBs for different interfaces and identical across ORBs for the same interface.

Note – This assumption is also necessary for IIOP to function correctly across ORBs.

Mapping COM IIDs to CORBA Interface IDs

The mapping of a COM IID to the CORBA interface ID is vendor specific. However, the mapping should be the same as if the CORBA mapping of the COM interface were defined with the #pragma ID <interface_name> = “DCE:...”.

Thus, the MIDL definition

```
[uuid(f4f2f07c-3a95-11cf-affb-08000970dac7), object]
interface A: IUnknown {
...
}
```

maps to this OMG IDL definition:

```
interface A {
#pragma ID A=”DCE:f4f2f07c-3a95-11cf-affb-08000970dac7”
...
};
```

13.6 Object Identity, Binding, and Life Cycle

The interworking model illustrated in Figure 13-2 and Figure 13-3 maps a View in one object system to a reference in the other system. This relationship raises questions:

- How do the concepts of object identity and object life cycle in different object models correspond, and to the extent that they differ, how can they be appropriately mapped?
- How is a View in one system bound to an object reference (and its referent object) in the other system?

13.6.1 Object Identity Issues

COM and CORBA have different notions of what object identity means. The impact of the differences between the two object models affects the transparency of presenting CORBA objects as COM objects or COM objects as CORBA objects. The following sections discuss the issues involved in mapping identities from one system to another. They also describe the architectural mechanics of identity mapping and binding.

CORBA Object Identity and Reference Properties

CORBA defines an object as a combination of state and a set of methods that explicitly embodies an abstraction characterized by the behavior of relevant requests. An object reference is defined as a name that reliably and consistently denotes a particular object. A useful description of a particular object in CORBA terms is an entity that exhibits a consistency of interface, behavior, and state over its lifetime. This description may fail in many boundary cases, but seems to be a reasonable statement of a common intuitive notion of object identity.

Other important properties of CORBA objects include the following:

- Objects have opaque identities that are encapsulated in object references.
- Object identities are unique within some definable reference domain, which is at least as large as the space spanned by an ORB instance.
- Object references reliably denote a particular object; that is, they can be used to identify and locate a particular object for the purposes of sending a request.
- Identities are immutable, and persist for the lifetime of the denoted object.
- Object references can be used as request targets irrespective of the denoted object's state or location; if an object is passively stored when a client makes a request on a reference to the object, the ORB is responsible for transparently locating and activating the object.
- There is no notion of "connectedness" between object reference and object, nor is there any notion of reference counting.
- Object references may be externalized as strings and reinternalized anywhere within the ORB's reference domain.
- Two object references may be tested for equivalence (that is, to determine whether both references identify the same object instance), although only a result of TRUE for the test is guaranteed to be reliable.

COM Object Identity and Reference Properties

The notion of what it means to be "a particular COM object" is somewhat less clearly defined than under CORBA. In practice, this notion typically corresponds to an active instance of an implementation, but not a particular persistent state. A COM instance can be most precisely defined as "the entity whose interface (or rather, one of whose interfaces) is returned by an invocation of **`IClassFactory::CreateInstance`**." The following observations may be made regarding COM instances:

- COM instances are either initialized with a default "empty" state (e.g., a document or drawing with no contents), or they are initialized to arbitrary states; **`IClassFactory::CreateInstance`** has no parameters for describing initial state.
- The only inherently available identity or reference for a COM instance is its collection of interface pointers. Their usefulness for determining identity equivalence is limited to the scope and extent of the process they live in. There is

no canonical information model, visible or opaque, that defines the identity of a COM object. Individual COM class types may establish a strong notion of persistent identity, but this is not the responsibility of the COM model itself.

- There is no inherent mechanism to determine whether two interface pointers belong to the same COM class or not.
- The identity and management of state are generally independent of the identity and life cycle of COM class instances. Files that contain document state are persistent, and are identified within the file system's name space. A single COM instance of a document type may load, manipulate, and store several different document files during its lifetime; a single document file may be loaded and used by multiple COM class instances, possibly of different types. Any relationship between a COM instance and a state vector is either an artifact of the particular class type, or the user's imagination.

13.6.2 Binding and Life Cycle

The identity-related issues previously discussed emerge as practical problems in defining binding and life cycle management mechanisms in the Interworking models. Binding refers to the way in which an existing object in one system can be located by clients in the other system and associated with an appropriate View. Life cycle, in this context, refers to the way objects in one system are created and destroyed by clients in the other system.

Lifetime Comparison

The in-memory lifetime of COM (including Automation) objects is bounded by the lifetimes of its clients. That is, in COM, when there are no more clients attached to an object, it is destroyed. If clients remain, the object cannot be removed from memory. Unfortunately, a reference counted lifecycle model such as COM's has problems when applied to wide area networks, when network traffic is heavy, and when networks and routers are not fault tolerant (and thus not 100% reliable). For example, if the network connection between clients and the server object were down, the server would think that its clients had died, and would delete itself (if there were no local references to it). When the network connection was later restored, even just seconds later, the clients would then have invalid object references and would need to be restarted, or be prepared to handle invalid interface reference errors for the previously valid references. In addition, if clients exist for a server object but rarely use it, the server object is still required to be in memory. In large, long-running distributed systems, this type of memory consuming behavior is not typically acceptable.

In contrast, the CORBA Life Cycle model decouples the lifetime of the clients from the lifetime of the active (in-memory) representation of the persistent server object. The CORBA model allows clients to maintain references to CORBA server objects even when the clients are no longer running. Server objects can deactivate and remove themselves from memory whenever no clients are currently using them. This behavior avoids the problems and limitations introduced by distributed reference counting. Clients can be started and stopped without incurring expensive data reloads in the server. Servers can relinquish memory (but can later be restored) if they have not been used recently or if the network connection is down. In addition, since the client and server lifetimes are decoupled, CORBA, unlike COM, has no requirement for the servers to constantly “ping” their clients -- a requirement of distributed reference counting which can become expensive across local networks and impractical across wide area networks.

Binding Existing CORBA Objects to COM Views

COM and Automation have limited mechanisms for registering and accessing active objects. A single instance of a COM class can be registered in the active object registry. COM or Automation clients can obtain an IUnknown pointer for an active object with the COM GetActiveObject function or the Automation GetObject function. The most natural way for COM or Automation clients to access existing CORBA objects is through this (or some similar) mechanism.

Interworking solutions can, if desirable, create COM Views for any CORBA object and place them in the active object registry, so that the View (and thus, the object) can be accessed through GetActiveObject or GetObject.

The resources associated with the system registry are limited; some interworking solutions will not be able to map objects efficiently through the registry. This submission defines an interface, ICORBAFactory, which allows interworking solutions to provide their own name spaces through which CORBA objects can be made available to COM and Automation clients in a way that is similar to OLE’s native mechanism (GetObject). This interface is described fully in Section 13.7.3, ICORBAFactory Interface.

Binding COM Objects to CORBA Views

As described in Section 13.6.1, Object Identity Issues, COM class instances are inherently transient. Clients typically manage COM and Automation objects by creating new class instances and subsequently associating them with a desired stored state. Thus, COM objects are made available through factories. The SimpleFactory OMG IDL interface (described next in Section 13.7.1, SimpleFactory Interface) is designed to map onto COM class factories, allowing CORBA clients to create (and bind to) COM objects. A single CORBA SimpleFactory maps to a single COM class factory. The manner in which a particular interworking solution maps SimpleFactories to COM class factories is not specified. Moreover, the manner in which mapped SimpleFactory objects are presented to CORBA clients is not specified.

COM View of CORBA Life Cycle

The SimpleFactory interface in Section 13.7.1, SimpleFactory Interface, provides a create operation without parameters. CORBA SimpleFactory objects can be wrapped with COM IClassFactory interfaces and registered in the Windows registry. The process of building, defining, and registering the factory is implementation-specific.

To allow COM and Automation developers to benefit from the robust CORBA lifecycle model, the following rules apply to COM and Automation Views of CORBA objects. When a COM or Automation View of a CORBA object is dereferenced and there are no longer any clients for the View, the View may delete itself. It should not, however, delete the CORBA object that it refers to. The client of the View may call the **`LifeCycleObject::remove`** operation (if the interface is supported) on the CORBA object to remove it. Otherwise, the lifetime of the CORBA object is controlled by the implementation-specific lifetime management process.

COM currently provides a mechanism for client-controlled persistence of COM objects (equivalent to CORBA externalization). However, unlike CORBA, COM currently provides no general-purpose mechanism for clients to deal with server objects, such as databases, which are inherently persistent (i.e. they store their own state -- their state is not stored through an outside interface such as IPersistStorage). COM does provide monikers, which are conceptually equivalent to CORBA persistent object references. However, monikers are currently only used for OLE graphical linking. To enable COM developers to use CORBA objects to their fullest extent, the submission defines a mechanism that allows monikers to be used as persistent references to CORBA objects, and a new COM interface, IMonikerProvider, that allows clients to obtain an IMoniker interface pointer from COM and Automation Views. The resulting moniker encapsulates, stores, and loads the externalized string representation of the CORBA reference managed by the View from which the moniker was obtained. The IMonkierProvider interface and details of object reference monikers are described in Section 13.7.2, IMonikerProvider Interface and Moniker Use.

CORBA View of COM/Automation Life Cycle

Initial references to COM and Automation objects can be obtained in the following way: COM IClassFactories can be wrapped with CORBA SimpleFactory interfaces. These SimpleFactory Views of COM IClassFactories can then be installed in the naming service or used via factory finders. The mechanisms used to register or dynamically look up these factories is beyond the scope of this specification.

All CORBA Views for COM and Automation objects support the LifeCycleObject interface. In order to destroy a View for a COM or Automation object, the remove method of the LifeCycleObject interface must be called. Once a CORBA View is instantiated, it must remain active (in memory) for the lifetime of the View unless the COM or Automation objects supports the IMonikerProvider interface. If the COM or Automation object supports the IMonikerProvider interface, then the CORBA View can safely be deactivated and reactivated provided it stores the object's moniker in persistent storage between activations. Interworking solutions are not required to support deactivation and activation of CORBA View objects, but are enabled to do so by the IMonikerProvider interface.

13.7 Interworking Interfaces

13.7.1 SimpleFactory Interface

CORBA allows object factories to be arbitrarily defined. In contrast, COM IClassFactory is limited to having only one object constructor and the object constructor method (called CreateInstance) has no arguments for passing data during the construction of the instance. The SimpleFactory interface allows CORBA objects to be created under the rigid factory model of COM. The interface also supports CORBA Views of COM class factories.

```
module CosLifeCycle
{
    interface SimpleFactory
    {
        Object create_object();
    };
};
```

SimpleFactory provides a generic object constructor for creating instances with no initial state. In the future, CORBA objects, which can be created with no initial state, should provide factories, which implement the SimpleFactory interface.

13.7.2 IMonikerProvider Interface and Moniker Use

COM or Automation Views for CORBA objects may support the IMonikerProvider interface. COM clients may use QueryInterface for this interface.

```
[object, uuid(ecce76fe-39ce-11cf-8e92-08000970dac7)] // MIDL
interface IMonikerProvider: IUnknown {
   HRESULT get_moniker([out] IMoniker ** val);
}
```

This allows COM clients to persistently save the object reference for later use without needing to keep the View in memory. The moniker returned by IMonikerProvider must support at least the IMoniker and IPersistStorage interfaces. To allow CORBA object reference monikers to be created with one COM/CORBA interworking solution and later restored using another, **`IPersist::GetClassID`** must return the following CLSID:

```
{a936c802-33fb-11cf-a9d1-00401c606e79}
```

In addition, the data stored by the moniker's IPersistStorage interface must be four 0 (null) bytes followed by the length in bytes of the stringified IOR (stored as a little endian 4-byte unsigned integer value) followed by the stringified IOR itself (without null terminator).

13.7.3 ICORBAFactory Interface

All interworking solutions that expose COM Views of CORBA objects shall expose the ICORBAFactory interface. This interface is designed to support general, simple mechanisms for creating new CORBA object instances and binding to existing CORBA object references by name.

```
interface ICORBAFactory: IUnknown
{
   HRESULT CreateObject( [in] LPTSTR factoryName, [out,
retval] IUknown ** val);
   HRESULT GetObject([in] LPTSTR objectName, [out, retval]
IUknown ** val);
}
```

The UUID for the ICORBAFactory interface is:

```
{204F6240-3AEC-11cf-BBFC-444553540000}
```

A COM class implementing ICORBAFactory must be registered in the Windows System Registry on the client machine using the following class id, class id tag, and Program Id respectively:

```
{913D82C0-3B00-11cf-BBFC-444553540000}
   DEFINE_GUID(IID_ICORBAFactory,
   0x913d82c0, 0x3b00, 0x11cf, 0xbb, 0xfc, 0x44, 0x45, 0x53,
0x54, 0x0, 0x0);
   “CORBA.Factory.COM”
```

The CORBA factory object may be implemented as a singleton object, i.e., subsequent calls to create the object may return the same interface pointer.

We define a similar interface, DICORBAFactory, that supports creating new CORBA object instances and binding to existing CORBA objects for OLE Automation clients. DICORBAFactory is an Automation Dual Interface. (For an explanation of Automation Dual interfaces, see Chapter 13C, Mapping: OLE Automation and CORBA.)

```
interface DICORBAFactory: IDispatch
{
   HRESULT CreateObject( [in] BSTR factoryName, [out,
      retval] IDispatch ** val);
   HRESULT GetObject([in] BSTR objectName, [out, retval]
      IDispatch ** val);
}
```

The UUID for the DICORBAFactory interface is:

```
{204F6241-3AEC-11cf-BBFC-444553540000}
```

An instance of this class must be registered in the Windows System Registry by calling on the client machine using the Program Id “CORBA.Factory.”

The CreateObject and GetObject methods are intended to approximate the usage model and behavior of the Visual Basic CreateObject and GetObject functions.

The first method, CreateObject, causes the following actions:

- A COM View is created. The specific mechanism by which it is created is undefined. We note here that one possible (and likely) implementation is that the View delegates the creation to a registered COM class factory.
- A CORBA object is created and bound to the View. The argument, factoryName, identifies the type of CORBA object to be created. Since the CreateObject method does not accept any parameters, the CORBA object must either be created by a null factory (a factory whose creation method requires no parameters), or the View must supply its own factory parameters internally.
- The bound View is returned to the caller.

The factoryName parameter identifies the type of CORBA object to be created, and thus implicitly identifies (directly or indirectly) the interface supported by the View. In general, the factoryName string takes the form of a sequence of identifiers separated by period characters (“.”), such as “personnel.record.person”. The intent of this name form is to provide a mechanism that is familiar and natural for COM and OLE Automation programmers by duplicating the form of OLE ProgIDs. The specific semantics of name resolution are determined by the implementation of the interworking solution. The following examples illustrate possible implementations:

- The factoryName sequence could be interpreted as a key to a CosNameService-based factory finder. The CORBA object would be created by invoking the factory create method. Internally, the interworking solution would map the factoryName onto the appropriate COM class ID for the View, create the View, and bind it to the CORBA object.
- The creation could be delegated directly to a COM class factory by interpreting the factoryName as a COM ProgID. The ProgID would map to a class factory for the COM View, and the View’s implementation would invoke the appropriate CORBA factory to create the CORBA server object.

The GetObject method has the following behavior:

- The objectName parameter is mapped by the interworking solution onto a CORBA object reference. The specific mechanism for associating names with references is not specified. In order to appear familiar to COM and Automation users, this parameter shall take the form of a sequence of identifiers separated by periods (.), in the same manner as the parameter to CreateObject. An implementation could, for example, choose to map the objectName parameter to a name in the OMG Naming Service implementation. Alternatively, an interworking solution could choose to put precreated COM Views bound to specific CORBA object references in the active object registry, and simply delegate GetObject calls to the registry.
- The object reference is bound to an appropriate COM or Automation View and returned to the caller.

Another name form that is specialized to CORBA is a single name with a preceding period, such as ".NameService". When the name takes this form, the Interworking solution shall interpret the identifier (without the preceding period) as a name in the ORB Initialization interface. Specifically, the name shall be used as the parameter to an invocation of the **`CORBA::ORB::ResolveInitialReferences`** method on the ORB pseudo-object associated with the ICORBAFactory. The resulting object reference is bound to an appropriate COM or Automation View, which is returned to the caller.

13.7.4 IForeignObject Interface

As object references are passed back and forth between two different object models through a bridge, and the references are mapped through Views (as is the case in this specification), the potential exists for the creation of indefinitely long chains of Views that delegate to other Views, which in turn delegate to other Views, and so on. To avoid this, the Views of at least one object system must be able to expose the reference for the "foreign" object managed by the View. This exposure allows other Views to determine whether an incoming object reference parameter is itself a View and, if so, obtain the "foreign" reference that it manages. By passing the foreign reference directly into the foreign object system, the bridge can avoid creating View chains.

This problem potentially exists for any View representing an object in a foreign object system. The IForeignObject interface is specified to provide bridges access to object references from foreign object systems that are encapsulated in proxies.

```
typedef struct {
   unsigned long cbMaxSize;
   unsigned long cbLengthUsed;
   [ size_is(cbMaxSize), length_is(cbLengthUsed), unique ]
long *pValue;
} objSystemIDs;
interface IForeignObject : IUnknown {
   HRESULT GetForeignReference([in[ objSystemIDs systemIDs,
                        [out] long *systemID,
                        [out] LPSTR* objRef);
   HRESULT GetRepositoryId([out] RepositoryId
      *repositoryId);
}
```

The UUID for IForeignObject is:

`{204F6242-3AEC-11cf-BBFC-444553540000}`

The first parameter (systemIDs) is an array of long values that correspond to specific object systems. These values must be positive, unique, and publicly known. The OMG will manage the allocation of identifier values in this space to guarantee uniqueness. The value for the CORBA object system is the long value 1. The systemIDs array contains a list of IDs for object systems for which the caller is interested in obtaining a reference. The order of IDs in the list indicates the caller's order of preference. If the View can produce a reference for at least one of the specified object systems, then the second parameter (systemID) is the ID of the first object system in the incoming array that it can satisfy. The objRef out parameter will contain the object reference converted to a string form. Each object system is responsible for providing a mechanism to convert its references to strings, and back into references. For the CORBA object system, the string contains the IOR string form returned by **`CORBA::ORB::object_to_string`**, as defined in the CORBA 2.0 specification.

The choice of object reference strings is motivated by the following observations:

- Language mappings for object references do not prescribe the representation of object references. Therefore, it is impossible to reliably map any given ORB's object references onto a fixed OLE Automation parameter type.
- The object reference being returned from GetForeignObject may be from a different ORB than the caller. IORs in string form are the only externalized standard form of object reference supported by CORBA.

The purpose of the GetRepositoryID method is to support the ability of DICORBAAny (see Section 13.1.13, Mapping for **`any`**s) when it wraps an object reference, to produce a type code for the object when asked to do so via DICORBAAny's readonly typeCode property.

It is not possible to provide a similar inverse interface exposing COM references to CORBA clients through CORBA Views, because of limitations imposed by COM's View of object identity and use of interface pointer as references.

13.7.5 ICORBAObject Interface

The ICORBAObject interface is a COM interface that is exposed by COM Views, allowing COM clients to have access to operations on the CORBA object references, defined on the **`CORBA::Object`** pseudo-interface. The ICORBAObject interface can be obtained by COM clients through QueryInterface. ICORBAObject is defined as follows:

```
interface ICORBAObject: IUnknown
{
   HRESULT GetInterface([out] IUnknown ** val);
   HRESULT GetImplementation([out] IUnknown ** val);
   HRESULT IsA([in] LPTSTR repositoryID, [out] boolean);
   HRESULT IsNil([out] boolean *val);
   HRESULT IsEquivalent([in] IUnknown* obj,[out] boolean *
val);
   HRESULT NonExistent([out] boolean *val);
   HRESULT Hash([out] long *val);
}
```

The UUID for ICORBAObject is:

```
{204F6243-3AEC-11cf-BBFC-444553540000}
```

Automation controllers gain access to operations on the CORBA object reference interface through the Dual Interface **DIORBObject::GetCORBAObject** method described next.

```
interface DICORBAObject: IDispatch
{
   HRESULT GetInterface([out, retval] IDispatch ** val);
   HRESULT GetImplementation([out, retval] IDispatch **
                                    val);
   HRESULT IsA([in] BSTR repositoryID, [out, retval]
                     boolean);
   HRESULT IsNil([out, retval] boolean *val);
   HRESULT IsEquivalent([in] IDispatch* obj,[out,retval]
                        boolean * val);
   HRESULT NonExistent([out,retval] boolean *val);
   HRESULT Hash([out, retval] long *val);
}
```

The UUID for DICORBAObject is:

```
{204F6244-3AEC-11cf-BBFC-444553540000}
```

13.7.6 IORBObject Interface

The IORBObject interface provides Automation and COM clients with access to the operations on the ORB pseudo-object.

The IORBObject is defined as follows:

```
typedef struct {
   unsigned long cbMaxSize;
   unsigned long cbLengthUsed;
   [ size_is(cbMaxSize), length_is(cbLengthUsed), unique ]
   LPSTR *pValue;
} CORBA_ORBObjectIdList;
interface IORBObject : IUnknown
   HRESULT ObjectToString([in] IUnknown* obj, [out] LPSTR
   *val);
   HRESULT StringToObject([in] LPTSTR ref, [out] IUnknown
   *val);
   HRESULT GetInitialReferences([out], CORBA_ORBObjectIdList
   *val);
   HRESULT ResolveInitialReference([in] LPTSTR name, [out]
   IUnknown ** val));
}
```

The UUID for IORBObject is:

```
{204F6245-3AEC-11cf-BBFC-444553540000}
```

A reference to this interface is obtained by calling ICORBAFactory::GetObject(”CORBA.ORB.2”).

The methods of DIORBObject delegate their function to the similarly-named operations on the ORB pseudo-object associated with the IORBObject.

Automation clients access operations on the ORB via the following Dual Interface:

```
interface DIORBObject: IDispatch {
   HRESULT ObjectToString([in] IDispatch* obj, [out,retval]
   BSTR *val);
   HRESULT StringToObject([in] BSTR ref, [out,retval]
   IDispatch * val);
   HRESULT GetInitialReferences([out, retval]
   SAFEARRAY(IDispatch *) *val);
   HRESULT ResolveInitialReference([in] BSTR name, [out,
   retval] IDispatch ** val));
   HRESULT GetCORBAObject([in] IDispatch* obj, [out, retval]
   DICORBAObject * val);
}
```

The UUID for DIORBObject is:

```
{204F6246-3AEC-11cf-BBFC-444553540000}
```

A reference to this interface is obtained by calling DICORBAFactory::GetObject(”CORBA.ORB.2”).

This interface is very similar to IORBObject, except for the additional method GetCORBAObject. This method returns an IDispatch pointer to the DICORBAObject interface associated with the parameter Object. This operation is primarily provided to allow Automation controllers (i.e. Automation clients) that cannot invoke QueryInterface on the View object to obtain the ICORBAObject interface.

13.7.7 Naming Conventions for View Components

Naming the COM View Interface Id

The default tag for the COM View’s Interface Id (IID) should be:

```
IID_I<module name>_<interface name>
```

For example, if the module name is “MyModule” and the interface name is “MyInterface” then the default IID tag should be:

```
IID_IMyModule_MyInterface
```

If the module containing the interface is itself nested within other modules, the default tag should be:

IID_I<module name>_<module name>_...<module name>_<interface name>

where the module names read from outermost on the left to innermost on the right. Extending our example, if module "MyModule" were nested within module "OuterModule," then the default IID tag shall be:

IID_IOuterModule_MyModule_MyInterface

Tag for the Automation Interface Id

No standard tag is required for Automation and Dual Interface IDs because client programs written in Automation controller environments such as Visual Basic are not expected to explicitly use the UUID value.

Naming the COM View Interface

The default name of the COM View's Interface should be:

I<module name>_<interface name>

For example, if the module name is "MyModule" and the interface name is "MyInterface," then the default name should be:

IMyModule_MyInterface

If the module containing the interface is itself nested within other modules, the default name should be:

I<module name>_<module name>_...<module name>_<interface name>

where the module names read from outermost on the left to innermost on the right. Extending our example, if module "MyModule" were nested within module "OuterModule," then the default name shall be:

IOuterModule_MyModule_MyInterface

Naming the Automation View Dispatch Interface

The default name of the Automation View's Interface should be:

D<module name>_<interface name>

For example, if the module name is "MyModule" and the interface name is "MyInterface," then the default name should be:

DMyModule_MyInterface

If the module containing the interface is itself nested within other modules, the default name should be:

D<module name>_<module name>_...<module name>_<interface name>

where the module names read from outermost on the left to innermost on the right. Extending our example, if module "MyModule" were nested within module "OuterModule," then the default name shall be:

DOuterModule_MyModule_MyInterface

Naming the Automation View Dual Interface

The default name of the Automation Dual View's Interface should be:

DI<module name>_<interface name>

For example, if the module name is "MyModule" and the interface name is "MyInterface," then the default name should be:

DIMyModule_MyInterface

If the module containing the interface is itself nested within other modules, the default name should be:

DI<module name>_<module name>_...<module name>_<interface name>

where the module names read from outermost on the left to innermost on the right. Extending our example, if module "MyModule" were nested within module "OuterModule," then the default name shall be:

DIOuterModule_MyModule_MyInterface

Naming the Program Id for the COM Class

If a separate COM class is registered for each View Interface, then the default Program Id for that class shall be:

<module name> "." <module name> "." ...<module name> "." <interface name>

where the module names read from outermost on the left to innermost on the right. In our example, the default Program Id shall be:

"OuterModule.MyModule.MyInterface"

Naming the Class Id for the COM Class

If a separate COM co-class is registered for each Automation View Interface, then the default tag for the COM Class Id (CLSID) for that class should be:

```
CLSID_<module name>_<module name>_...<module name>_
<interface name>
```

where the module names read from outermost on the left to innermost on the right. In our example, the default CLSID tag should be:

```
CLSID_OuterModule_MyModule_MyInterface
```

13.8 Distribution

The version of COM (and OLE) that is addressed in this specification (OLE 2.0 in its currently released form) does not include any mechanism for distribution. CORBA specifications define a distribution architecture, including a standard protocol (IIOP) for request messaging. Consequently, the CORBA architecture, specifications, and protocols shall be used for distribution.

13.8.1 Bridge Locality

One of the goals of this specification is to allow any compliant interworking mechanism delivered on a COM client node to interoperate correctly with any CORBA 2.0-compliant components that use the same interface specifications. Compliant interworking solutions must appear, for all intents and purposes, to be CORBA object implementations and/or clients to other CORBA clients, objects, and services on an attached network.

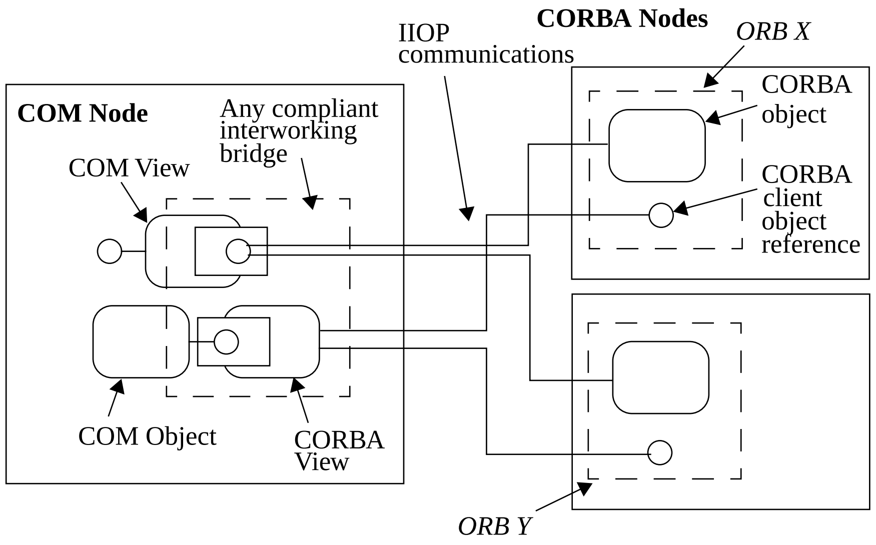


Figure 13-7 Bridge Locality

Figure 13-7 illustrates the required locality for interworking components. All of the transformations between CORBA interfaces and COM interfaces described in this submission will take place on the node executing the COM environment. Mapping agents (COM views, CORBA views, and bridging elements) will reside and execute on the COM client node. This requirement allows compliant interworking solutions to be localized to a COM client node, and to interoperate with any CORBA V2.0-compliant networking ORB that shares the same view of interfaces with the interworking solution.

13.8.2 Distribution Architecture

External communications between COM client machines, and between COM client machines and machines executing CORBA environments and services, will follow specifications contained in *CORBA V2.0*. Figure 13-7 illustrates the required distribution architecture. The following statements articulate the responsibilities of compliant solutions.

- All externalized CORBA object references will follow *CORBA V2.0* specifications for Interoperable Object References (IORs). Any IORs generated by components performing mapping functions must include a valid IIOP profile.
- The mechanisms for negotiating protocols and binding references to remote objects will follow the architectural model described in *CORBA V2.0*.
- A product component acting as a CORBA client may bind to an object by using any profile contained in the object’s IOR. The client must, however, be capable of binding with an IIOP profile.
- Any components that implement CORBA interfaces for remote use must support the IIOP.

13.9 Interworking Targets

This specification is targeted specifically at interworking between the following systems and versions:

- CORBA as described in *CORBA V2.0: Common Object Request Broker Architecture and Specification.*
- OLE as embodied in version 2.03 of the OLE run-time libraries.
- Microsoft Object Description Language (ODL) as supported by MKTYPELIB version 2.03.3023.
- Microsoft Interface Description Language (MIDL) as supported by the MIDL Compiler version 2.00.0102.

In determining which features of Automation to support, the expected usage model for Automation Views follows the Automation controller behavior established by Visual Basic 4.0.

13.10 Compliance to COM/CORBA Interworking

This section explains which software products are subject to compliance to the Interworking specification, and provides compliance points. For general information about compliance to CORBA specifications, refer to Section 0.6, Definition of CORBA Compliance.

13.10.1 Products Subject to Compliance

COM/CORBA interworking covers a wide variety of software activities and a wide range of products. This specification is not intended to cover all possible products that facilitate or use COM and CORBA mechanisms together. This Interworking specification defines three distinct categories of software products, each of which are subject to a distinct form of compliance. The categories are:

- Interworking Solutions
- Mapping Solutions
- Mapped Components

Interworking Solutions

Products that facilitate the development of software that will bidirectionally transform COM and/or Automation invocations into isomorphic CORBA invocations (and vice versa) in a generic way are Interworking Solutions. An example of this kind of software would be a language processor that parses OMG IDL specifications and automatically generates code for libraries that map the OMG IDL interfaces into OLE Automation interfaces and which also parses OLE Automation ODL and automatically generates code for libraries that map the OLE Automation interfaces into CORBA interfaces. Another example would be a generic bridging component that, based on run-time interface descriptions, interpretively maps both COM and CORBA invocations onto CORBA and COM objects (respectively).

A product of this type is a **compliant** Interworking Solution if the resulting mapped interfaces are transformed as described in this specification, and if the mapped interfaces support all of the features and interface components required by this specification.

A compliant Interworking Solution must designate whether it is a compliant COM/CORBA Interworking Solution and/or a compliant Automation/CORBA Interworking Solution.

Mapping Solutions

Products that facilitate the development of software that will unidirectionally transform COM and/or Automation invocations into isomorphic CORBA invocations (and vice versa) in a generic way are described as *Mapping Solutions*. An example of this kind of software would be a language processor that parses OMG IDL specifications and automatically generates code for libraries that map the OMG IDL interfaces into OLE Automation interfaces. Another example would be a generic bridging component that interpretively maps OLE Automation invocations onto CORBA objects based on run-time interface descriptions.

A product of this type will be considered a **compliant** Mapping Solution if the resulting mapped interfaces are transformed as described in this specification, and if the mapped interfaces support all of the features and interface components required in this specification.

A compliant Mapping Solution must designate whether it is a compliant COM to CORBA Mapping Solution, a compliant Automation to CORBA Mapping Solution, a compliant CORBA to COM Mapping Solution, and/or a compliant CORBA to Automation Mapping Solution.

Mapped Components

Applications, components or libraries that expose a specific, fixed set of interfaces mapped from CORBA to COM or Automation (and/or vice versa) are described as Mapped Components. An example of this kind of product would be a set of business objects defined and implemented in CORBA that also expose isomorphic OLE Automation interfaces.

This type of product will be considered a **compliant** Mapped Component if the interfaces it exposes are mapped as described in this specification, and if the mapped interfaces support all of the features and interface components required in this specification.

13.10.2 Compliance Points

The intent of this submission is to allow the construction of implementations that fit in the design space described in Section 13.2, Interworking Object Model, and yet guarantee interface uniformity among implementations with similar or overlapping design centers. This goal is achieved by the following compliance statements:

- When a product offers the mapping of CORBA interfaces onto isomorphic COM and/or Automation interfaces, the mapping of COM and/or Automation interfaces onto isomorphic CORBA interfaces, or when a product offers the ability to automatically generate components that perform such mappings, then the product must use the interface mappings defined in this specification. Note that products may offer custom, nonisomorphic interfaces that delegate some or all of their behavior to CORBA, COM, or Automation objects. These interfaces are not in the scope of this specification, and are neither compliant nor noncompliant.
- Interworking solutions that expose COM Views of CORBA objects are required to expose the CORBA-specific COM interfaces ICORBAObject and IORBObject, defined in Section 13.7.5, ICORBAObject Interface, and Section 13.7.6, IORBObject Interface, respectively.
- Interworking solutions that expose Automation Views of CORBA objects are required to expose the CORBA-specific Automation Dual interfaces DICORBAObject and DIORBObject, defined in Section 13.7.5, ICORBAObject Interface, and Section 13.7.6, IORBObject Interface, respectively.
- OMG IDL interfaces exposed as COM or Automation Views are not required to provide type library and registration information in the COM client environment where the interface is to be used. If such information is provided, however, then it must be provided in the prescribed manner.

- Each COM and Automation View must map onto one and only one CORBA object reference, and must also expose the IForeignObject interface, described in Section 13.7.4, IForeignObject Interface. This constraint guarantees the ability to obtain an unambiguous CORBA object reference from any COM or Automation View via the IForeignObject interface.
- If COM or Automation Views expose the IMonikerProvider interface, they shall do so as specified in Section 13.7.2, IMonikerProvider Interface and Moniker Use.
- All COM interfaces specified in this submission have associated COM Interface IDs. Compliant interworking solutions must use the IIDs specified herein, to allow interoperability between interworking solutions.
- All compliant products that support distributed interworking must support the CORBA 2.0 Internet Inter-ORB Protocol (IIOP), and use the interoperability architecture described in CORBA 2.0 in the manner prescribed in Section 13.8, Distribution. Interworking solutions are free to use any additional proprietary or public protocols desired.
- Interworking solutions that expose COM Views of CORBA objects are required to provide the ICORBAFactory object as defined in Section 13.7.3, ICORBAFactory Interface.
- Interworking solutions that expose Automation Views of CORBA objects are required to provide the DICORBAFactory object as defined in Section 13.7.3, ICORBAFactory Interface.
- Interworking solutions that expose CORBA Views of COM or Automation objects are required to derive the CORBA View interfaces from **`CosLifeCycle::LifeCycleObject`** as described in CORBA View of COM/Automation Life Cycle, under Section 13.6.3.

Mapping: COM and CORBA 13B

This chapter describes the data type and interface mapping between COM and CORBA. The mappings are described in the context of both Win16 and Win32 COM due to the differences between the versions of COM and between the automated tools available to COM developers under these environments. The mapping is designed to be able to be fully implemented by automated interworking tools.

13.1 Data Type Mapping

The data type model used in this mapping for Win32 COM is derived from MIDL (a derivative of DCE IDL). COM interfaces using “custom marshalling” must be hand-coded and require special treatment to interoperate with CORBA using automated tools. This specification does not address interworking between CORBA and custom-marshaled COM interfaces.

The data type model used in this mapping for Win16 COM is derived from ODL since Microsoft RPC and the Microsoft MIDL compiler are not available for Win16. The ODL data type model was chosen since it is the only standard, high-level representation available to COM object developers on Win16.

Note that although the MIDL and ODL data type models are used as the reference for the data model mapping, there is no requirement that either MIDL or ODL be used to implement a COM/CORBA interworking solution.

In many cases, there is a one-to-one mapping between COM and CORBA data types. However, in cases without exact mappings, run-time conversion errors may occur. Conversion errors will be discussed in Mapping for Exception Types under Section 13.2.10.

13.2 CORBA to COM Data Type Mapping

13.2.1 Mapping for Basic Data Types

The basic data types available in OMG IDL map to the corresponding data types available in Microsoft IDL as shown in Table 13-1.

Table 13-1 OMG IDL to MIDL Intrinsic Data Type Mappings

OMG IDL	Microsoft IDL	Microsoft ODL	Description
short	short	short	Signed integer with a range of -215...215 - 1
long	long	long	Signed integer with a range of -231...231 - 1
unsigned short	unsigned short	unsigned short	Unsigned integer with a range of 0...216 - 1
unsigned long	unsigned long	unsigned long	Unsigned integer with a range of 0...232 - 1
float	float	float	IEEE single-precision floating point number
double	double	double	IEEE double-precision floating point number
char	char	char	8-bit quantity limited to the ISO Latin-1 character set
boolean	boolean	boolean	8-bit quantity which is limited to 1 and 0
octet	byte	unsigned char	8-bit opaque data type, guaranteed to not undergo any conversion during transfer between systems.

13.2.2 Mapping for Constants

The mapping of the OMG IDL keyword const to Microsoft IDL and ODL is almost exactly the same. The following OMG IDL definitions for constants:

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

maps to the following the following Microsoft IDL and ODL definitions for constants:

```
// Microsoft IDL and ODL
const short S = ...;
const long L = ...;
const unsigned short US = ...;
const unsigned long UL = ...;
const char C = ...;
const boolean B = ...;
const string STR = “...”;
```

Note that OMG IDL supports the definition of constants for the data types **`float`** and **`double`**, while COM does not. Because of this, any tool that generates Microsoft IDL or ODL from OMG IDL should raise an error when a float or double constant is encountered.

13.2.3 Mapping for Enumerators

CORBA has enumerators that are not explicitly tagged with values. Microsoft IDL and ODL support enumerators that are explicitly tagged with values. The constraint is that any language mapping that permits two enumerators to be compared or defines successor or predecessor functions on enumerators must conform to the ordering of the enumerators as specified in the OMG IDL.

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

CORBA enumerators are mapped to COM enumerations directly as per the CORBA C language binding. The Microsoft IDL keyword **`v1_enum`** is required in order for an enumeration to be transmitted as 32-bit values. Microsoft recommends that this keyword be used on 32-bit platforms, since it increases the efficiency of marshalling and unmarshalling data when such an enumerator is embedded in a structure or union.

```
// Microsoft IDL and ODL
 typedef [v1_enum] enum tagA_or_B_orC { A = 0, B, C }
A_or_B_or_C;
```

A maximum of 2ˆ32 identifiers may be specified in an enumeration in CORBA. Enumerators in Microsoft IDL and ODL will only support 2ˆ16 identifiers, and therefore, truncation may result.

13.2.4 Mapping for String Types

CORBA currently defines the data type **`string`** to represent strings that consist of 8-bit quantities, which are NULL-terminated.

Microsoft IDL and ODL define a number of different data types which are used to represent both 8-bit character strings and strings containing wide characters based on Unicode.

Table 13-2 illustrates how to map the string data types in OMG IDL to their corresponding data types in both Microsoft IDL and ODL.

Table 13-2 OMG IDL to Microsoft IDL/ODL String Mappings

OMG IDL	**Microsoft IDL**	**Microsoft ODL**	**Description**
string	LPSTR, char *	LPSTR	Null terminated 8-bit character string
	LPTSTR	LPTSTR	Null terminated 8-bit or Unicode string (depends upon compiler flags used)

If a BSTR containing embedded nulls is passed to a CORBA server, the COM client will receive an E_DATA_CONVERSION.

OMG IDL supports two different types of strings: *bounded* and *unbounded*. Bounded strings are defined as strings that have a maximum length specified, whereas unbounded string do not have a maximum length specified.

Mapping for Unbounded String Types

The definition of an unbounded string limited to 8-bit quantities in OMG IDL

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

is mapped to the following syntax in Microsoft IDL and ODL, which denotes the type of a “stringified unique pointer to character.”

```
 // Microsoft IDL and ODL
typedef [string, unique] char * UNBOUNDED_STRING;
```

In other words, a value of type UNBOUNDED_STRING is a non-NULL pointer to a one-dimensional null-terminated character array whose extent and number of valid elements can vary at run-time.

Mapping for Bounded String Types

Bounded strings have a slightly different mapping between OMG IDL and Microsoft IDL and ODL. The following OMG IDL definition for a bounded string:

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

maps to the following syntax in Microsoft IDL and ODL for a “stringified non-conformant array.”

```
// Microsoft IDL and ODL
  const long N = ... ;
  typedef [string, unique] char (* BOUNDED_STRING) [N];
```

In other words, the encoding for a value of type BOUNDED_STRING is that of a null-terminated array of characters whose extent is known at compile time, and the number of valid characters can vary at run-time.

13.2.5 Mapping for Struct Types

OMG IDL uses the keyword struct to define a record type, consisting of an ordered set of name-value pairs representing the member types and names. A structure defined in OMG IDL maps bidirectionally to Microsoft IDL and ODL structures. Each member of the structure is mapped according to the mapping rules for that data type.

An OMG IDL struct type with members of types T0, T1, T2, and so on

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

has an encoding equivalent to a Microsoft IDL and ODL structure definition, as follows.

```
// Microsoft IDL and ODL
typedef ... T0;
typedef ... Tl;
typedef ... T2;
...
typedef ... Tn;
typedef struct
 {
      T0 m0;
      Tl ml;
      T2 m2;
       ...
      TN mN;
     } STRUCTURE;
```

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

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

is mapped as:

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

13.2.6 Mapping for Union Types

OMG IDL defines unions to be encapsulated discriminated unions: the discriminator itself must be encapsulated within the union.

In addition, the OMG IDL union discriminants must be constant expressions. The discriminator tag must be a previously defined **long**, **short**, **unsigned long**, **unsigned short**, **char**, **boolean**, or **enum** constant. The default case can appear at most once in the definition of a discriminated union, and case labels must match or be automatically castable to the defined type of the discriminator.

The following definition for a discriminated union in OMG IDL

```
// OMG IDL
enum UNION_DISCRIMINATOR
   {
    dChar,
    dShort,
    dLong,
    dFloat,
    dDouble
    };

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

is mapped into encapsulated unions in Microsoft IDL as follows:

```
// Microsoft IDL
typedef enum
   {
   dchar,
   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
```

13.2.7 Mapping for Sequence Types

OMG IDL defines the keyword **sequence** to be a one-dimensional array with two characteristics: an optional maximum size which is fixed at compile time, and a length that is determined at run-time. Like the definition of strings, OMG IDL allows sequences to be defined in one of two ways: bounded and unbounded. A sequence is bounded if a maximum size is specified, else it is considered unbounded.

Mapping for Unbounded Sequence Types

The mapping of the following OMG IDL syntax for the unbounded sequence of type T

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

maps to the following Microsoft IDL and ODL syntax:

```
// Microsoft IDL or ODL
typedef ... U;
typedef struct
    {
   unsigned long cbMaxSize;
   unsigned long cbLengthUsed;
   [size_is(cbMaxSize), length_is(cbLengthUsed), unique]

            U * pValue;
   } UNBOUNDED_SEQUENCE;
```

The encoding for an unbounded OMG IDL sequence of type T is that of a Microsoft IDL or ODL struct containing a unique pointer to a conformant array of type U, where U is the Microsoft IDL or ODL mapping of T. The enclosing struct in the Microsoft IDL/ODL mapping is necessary to provide a scope in which extent and data bounds can be defined.

Mapping for Bounded Sequence Types

The mapping for the following OMG IDL syntax for the bounded sequence of type T which can grow to be N size

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

maps to the following Microsoft IDL or ODL syntax:

```
// Microsoft IDL or ODL
const long N = ...;
typedef ...U;
typedef struct
   {
   unsigned long cbMaxSize;
   unsigned long cbLengthUsed;
   [length_is(cbLengthUsed)] U Value[N];
   } BOUNDED_SEQUENCE_OF_N;
```

13.2.8 Mapping for Array Types

OMG IDL arrays are fixed length multidimensional arrays. Both Microsoft IDL and ODL also support fixed length multidimensional arrays. Arrays defined in OMG IDL map bidirectionally to COM fixed length arrays. The type of the array elements is mapped according to the data type mapping rules.

The mapping for an OMG IDL array of some type T is that of an array of the type U as defined in Microsoft IDL and ODL, where U is the result of mapping the OMG IDL T into Microsoft IDL or ODL.

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

 // Microsoft IDL or ODL for T
const long N = ...;
typedef ... U;
typedef U ARRAY_OF_U[N];
```

In Microsoft IDL and ODL, the name ARRAY_OF_U denotes the type of a “one-dimensional nonconformant and nonvarying array of U.” The value N can be of any integral type, and const means (as in OMG IDL) that the value of N is fixed and known at IDL compilation time. The generalization to multidimensional arrays follows the obvious mapping of syntax.

Note that if the ellipsis were **octet** in the OMG IDL, then the ellipsis would have to be **byte** in Microsoft IDL or ODL. That is why the types of the array elements have different names in the two texts.

13.2.9 Mapping for the **any** Type

The CORBA **any** type permits the specification of values that can express any OMG IDL data type. There is no direct or simple mapping of this type into COM, thus we map it to the following interface definition:

```
// Microsoft IDL
typedef [v1_enum] enum CORBAAnyDataTagEnum {
   anySimpleValTag,
   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 cbLengthUsed;
         [size_is(cbMaxSize), length_is(cbLengthUsed),
            unique]
            union CORBAAnyDataUnion *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(74105F50-3C68-11cf-9588-AA0004004A09) ]
   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 );
      }
```

However, the data types that can be included in a VARIANT are too restrictive to represent the data types that can be included in an **any**, such as structs and unions. In cases where the data types can be represented in a VARIANT, they will be; in other cases, they will optionally be returned as an IStream pointer in the VARIANT. An implementation may choose not to represent these types as an IStream, in which case an SCODE value of E_DATA_CONVERSION is returned when the VARIANT is requested.

13.2.10 Interface Mapping

Mapping for Interface Identifiers

Interface identifiers are used in both CORBA and COM to uniquely identify interfaces. These allow the client code to retrieve information about, or to inquire about other interfaces of an object.

CORBA identifies interfaces using the RepositoryId. The RepositoryId is a unique identifier for, among other things, an interface. COM identifies interfaces using a structure similar to the DCE UUID (in fact, identical to a DCE UUID on Win32) known as an IID. As with CORBA, COM specifies that the textual names of interfaces are only for convenience and need not be globally unique.

The CORBA RepositoryId is mapped, bidirectionally, to the COM IID. The algorithm for creating the mapping is detailed in Section 13.5.4, Mapping Interface Identity.

Mapping for Exception Types

The CORBA object model uses the concept of exceptions to report error information. Additional, exception-specification information may accompany the exception. The exception-specific information is a specialized form of a record. Because it is defined as a record, the additional information may consist of any of the basic data types or a complex data type constructed from one or more basic data types. Exceptions are classified into two types: System (Standard) Exceptions and User Exceptions.

COM provides error information to clients only if an operation uses a return result of type HRESULT. A COM HRESULT with a value of zero indicates success. The HRESULT then can be converted into an SCODE (the SCODE is explicitly specified as being the same as the HRESULT on Win32 platforms). The SCODE can then be examined to determine whether the call succeeded or failed. The error or success code, also contained within the SCODE, is composed of a “facility” major code (13 bits on Win32 and 4 bits on Win16) and a 16-bit minor code.

Unlike CORBA, COM provides no standard way to return user-defined exception data to the client. Also, there is no standard mechanism in COM to specify the completion status of an invocation. In addition, it is not possible to predetermine what set of errors a COM interface might return based on the definition of the interface as specified in Microsoft IDL, ODL, or in a type library. Although the set of status codes that can be returned from a COM operation must be fixed when the operation is defined, there is currently no machine-readable way to discover the set of valid codes.

Since the CORBA exception model is significantly richer than the COM exception model, mapping CORBA exceptions to COM requires an additional protocol to be defined for COM. However, this protocol does not violate backwards compatibility, nor does it require any changes to COM. To return the User Exception data to a COM client, an optional parameter is added to the end of a COM operation signature when mapping CORBA operations, which raise User Exceptions. System exception information is returned in a standard OLE Error Object.

Mapping for System Exceptions

System exceptions are standard exception types, which are defined by the CORBA specification and are used by the Object Request Broker (ORB) and object adapters (OA). Standard exceptions may be returned as a result of any operation invocation, regardless of the interface on which the requested operation was attempted.

There are two aspects to the mapping of System Exceptions. One aspect is generating an appropriate HRESULT for the operation to return. The other aspect is conveying System Exception information via a standard OLE Error Object.

The following table shows the HRESULT, which must be returned by the COM View when a CORBA System Exception is raised. Each of the CORBA System Exceptions is assigned a 16-bit numerical ID starting at 0x200 to be used as the code (lower 16 bits) of the HRESULT. Because these errors are interface-specific, the COM facility code FACILITY_ITF is used as the facility code in the HRESULT.

Bits 12-13 of the HRESULT contain a bit mask, which indicates the completion status of the CORBA request. The bit value 00 indicates that the operation did not complete, a bit value of 01 indicates that the operation did complete, and a bit value of 02 indicates that the operation may have completed. Table 13-3 lists the HRESULT constants and their values.

Table 13-3 Standard Exception to SCODE Mapping

HRESULT Constant	**HRESULT Value**
ITF_E_UNKNOWN_NO	0x40200
ITF_E_UNKNOWN_YES	0x41200
ITF_E_UNKNOWN_MAYBE	0x42200
ITF_E_BAD_PARAM_NO	0x40201
ITF_E_BAD_PARAM_YES	0x41201
ITF_E_BAD_PARAM_MAYBE	0x42201
ITF_E_NO_MEMORY_NO	0x40202
ITF_E_NO_MEMORY_YES	0x41202
ITF_E_NO_MEMORY_MAYBE	0x42202
ITF_E_IMP_LIMIT_NO	0x40203

Table 13-3 Standard Exception to SCODE Mapping

HRESULT Constant	**HRESULT Value**
ITF_E_IMP_LIMIT_YES	0x41203
ITF_E_IMP_LIMIT_MAYBE	0x42203
ITF_E_COMM_FAILURE_NO	0x40204
ITF_E_COMM_FAILURE_YES	0x41204
ITF_E_COMM_FAILURE_MAYBE	0x42204
ITF_E_INV_OBJREF_NO	0x40205
ITF_E_INV_OBJREF_YES	0x41205
ITF_E_INV_OBJREF_MAYBE	0x42205
ITF_E_NO_PERMISSION_NO	0x40206
ITF_E_NO_PERMISSION_YES	0x41206
ITF_E_NO_PERMISSION_MAYBE	0x42206
ITF_E_INTERNAL_NO	0x40207
ITF_E_INTERNAL_YES	0x41207
ITF_E_INTERNAL_MAYBE	0x42207
ITF_E_MARSHAL_NO	0x40208
ITF_E_MARSHAL_YES	0x41208
ITF_E_MARSHAL_MAYBE	0x42208
ITF_E_INITIALIZE_NO	0x40209
ITF_E_INITIALIZE_YES	0x41209
ITF_E_INITIALIZE_MAYBE	0x42209
ITF_E_NO_IMPLEMENT_NO	0x4020A
ITF_E_NO_IMPLEMENT_YES	0x4120A
ITF_E_NO_IMPLEMENT_MAYBE	0x4220A
ITF_E_BAD_TYPECODE_NO	0x4020B
ITF_E_BAD_TYPECODE_YES	0x4120B
ITF_E_BAD_TYPECODE_MAYBE	0x4220B
ITF_E_BAD_OPERATION_NO	0x4020C
ITF_E_BAD_OPERATION_YES	0x4120C
ITF_E_BAD_OPERATION_MAYBE	0x4220C

Table 13-3 Standard Exception to SCODE Mapping

HRESULT Constant	**HRESULT Value**
ITF_E_NO_RESOURCES_NO	0x4020D
ITF_E_NO_RESOURCES_YES	0x4120D
ITF_E_NO_RESOURCES_MAYBE	0x4220D
ITF_E_NO_RESPONSE_NO	0x4020E
ITF_E_NO_RESPONSE_YES	0x4120E
ITF_E_NO_RESPONSE_MAYBE	0x4220E
ITF_E_PERSIST_STORE_NO	0x4020F
ITF_E_PERSIST_STORE_YES	0x4120F
ITF_E_PERSIST_STORE_MAYBE	0x4220F
ITF_E_BAD_INV_ORDER_NO	0x40210
ITF_E_BAD_INV_ORDER_YES	0x41210
ITF_E_BAD_INV_ORDER_MAYBE	0x42210
ITF_E_TRANSIENT_NO	0x40211
ITF_E_TRANSIENT_YES	0x41211
ITF_E_TRANSIENT_MAYBE	0x42211
ITF_E_FREE_MEM_NO	0x40212
ITF_E_FREE_MEM_YES	0x41212
ITF_E_FREE_MEM_MAYBE	0x42212
ITF_E_INV_IDENT_NO	0x40213
ITF_E_INV_IDENT_YES	0x41213
ITF_E_INV_IDENT_MAYBE	0x42213
ITF_E_INV_FLAG_NO	0x40214
ITF_E_INV_FLAG_YES	0x41214
ITF_E_INV_FLAG_MAYBE	0x42214
ITF_E_INTF_REPOS_NO	0x40215
ITF_E_INTF_REPOS_YES	0x41215
ITF_E_INTF_REPOS_MAYBE	0x42215
ITF_E_BAD_CONTEXT_NO	0x40216
ITF_E_BAD_CONTEXT_YES	0x41216

Table 13-3 Standard Exception to SCODE Mapping

HRESULT Constant	**HRESULT Value**
ITF_E_BAD_CONTEXT_MAYBE	0x42216
ITF_E_OBJ_ADAPTER_NO	0x40217
ITF_E_OBJ_ADAPTER_YES	0x41217
ITF_E_OBJ_ADAPTER_MAYBE	0x42217
ITF_E_DATA_CONVERSION_NO	0x40218
ITF_E_DATA_CONVERSION_YES	0x41218
ITF_E_DATA_CONVERSION_MAYBE	0x42218

It is not possible to map a System Exception's minor code and RepositoryId into the HRESULT. Therefore, OLE Error Objects may be used to convey these data. Writing the exception information to an OLE Error Object is optional. However, if the Error Object is used for this purpose, it must be done according to the following specifications.

- The COM View must implement the standard COM interface ISupportErrorInfo such that the View can respond affirmatively to an inquiry from the client as to whether Error Objects are supported by the View Interface.
- The COM View must call SetErrorInfo with a NULL value for the IErrorInfo pointer parameter when the mapped CORBA operation is completed without an exception being raised. Calling SetErrorInfo in this fashion assures that the Error Object on that thread is thoroughly destroyed.

The properties of the OLE Error Object must be set according to Table 13-4.

Table 13-4 Error Object Usage for CORBA System Exceptions

Property	**Description**
bstrSource	<interface name>.<operation name> *where the interface and operation names are those of the CORBA interface that this Automation View is representing.*
bstrDescription	CORBA System Exception: [<exception repository id>] minor code [<minor code>][<completion status>] *where the <exception repository id> and <minor code> are those of the CORBA system exception. <completion status> is "YES," "NO," or "MAYBE" based upon the value of the system exception's CORBA completion status. Spaces and square brackets are literals and must be included in the string.*
bstrHelpFile	Unspecified
dwHelpContext	Unspecified
GUID	The IID of the COM View Interface

A COM View supporting error objects would have code, which approximates the following C++ example.

```
SetErrorInfo(OL,NULL); // Initialize the thread-local error
object
try
{
   // Call the CORBA operation
}
catch(...)
{
      ...

      CreateErrorInfo(&pICreateErrorInfo);
      pICreateErrorInfo->SetSource(...);
      pICreateErrorInfo->SetDescription(...);
      pICreateErrorInfo->SetGUID(...);
      pICreateErrorInfo
      ->QueryInterface(IID_IErrorInfo,&pIErrorInfo);
      pICreateErrorInfo->SetErrorInfo(OL,pIErrorInfo);
      pIErrorInfo->Release();
      pICreateErrorInfo->Release();

      ...
}
```

A client to a COM View would access the OLE Error Object with code approximating the following.

```
// 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(0,&pIErrorInfo);

   // S_FALSE means that error data is not available,
      NO_ERROR
   // means it is
   if (hr == NO_ERROR)
   {
   pIErrorInfo->GetSource(...);

      // Has repository id & minor code. hrOperation (above)
      // has the completion status encoded into it.
      pIErrorInfo->GetDescription(...);
   }
}
```

The COM client program could use C++ exception handling mechanisms to avoid doing this explicit check after every call to an operation on the COM View.

Mapping for User Exception Types

User exceptions are defined by users in OMG IDL and used by the methods in an object server to report operation-specific errors. The definition of a User Exception is identified in an OMG IDL file with the keyword exception. The body of a User Exception is described using the syntax for describing a structure in OMG IDL.

When CORBA User Exceptions are mapped into COM, a structure is used to describe various information about the exception — hereafter called an Exception structure. The structure contains members, which indicate the type of the CORBA exception, the identifier of the exception definition in a CORBA Interface Repository, and interface pointers to User Exceptions. The name of the structure is constructed from the name of the CORBA module in which the exception is defined (if specified), the name of the interface in which the exception is either defined or used, and the word “Exceptions.” A template illustrating this naming convention is as follows.

```
// Microsoft IDL and ODL
typedef enum { NO_EXCEPTION, USER_EXCEPTION}
      ExceptionType;

typedef struct
{
      ExceptionType     type;
      LPTSTR            repositoryId;
<ModuleName><InterfaceName>UserException
   *....piUserException;

} <ModuleName><InterfaceName>Exceptions;
```

The Exceptions structure is specified as an output parameter, which appears as the last parameter of any operation mapped from OMG IDL to Microsoft IDL, which raises a User Exception. The Exceptions structure is always passed by indirect reference. Because of the memory management rules of COM, passing the Exceptions structure as an output parameter by indirect reference allows the parameter to be treated as optional by the callee. The following example illustrates this point.

```
// Microsoft IDL
interface IAccount
   {
    HRESULT Withdraw(      [in] float fAmount,
                           [out] float pfNewBalance,
                           [out] BankExceptions
                              ** ppException);
   };
```

The caller can indicate that no exception information should be returned, if an exception occurs, by specifying NULL as the value for the Exceptions parameter of the operation. If the caller 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. COM’s memory management rules state that it is the responsibility of the caller to release this memory when it is no longer required.

If the caller provides a non-NULL value for the Exceptions parameter and the callee is to return exception information, the callee is responsible for allocating any memory used to hold the exception information being returned. If no exception is to be returned, the callee need do nothing with the parameter value.

If a CORBA exception is not raised, then S_OK must be returned as the value of the HRESULT to the callee, indicating the operation succeeded. The value of the HRESULT returned to the callee when a CORBA exception has been raised depends upon the type of exception being raised and whether an Exception structure was specified by the caller.

The following OMG IDL statements show the definition of the format used to represent User Exceptions

```
// OMG IDL
module BANK
      {
      ...
      exception InsufFunds { float balance };
      exception InvalidAmount { float amount };
.     ..
      interface Account
            {
            exception NotAuthorized { };
            float Deposit( in  float  Amount )
               raises( InvalidAmount );
            float Withdraw( in  float  Amount )
               raises( InvalidAmount, NotAuthorized );
            };
      };
```

and map to the following statements in Microsoft IDL and ODL.

```
//  Microsoft IDL and ODL
struct BankInsufFunds
         {
         float balance;
         };

struct BankInvalidAmount
         {
         float amount;
         };

struct BankAccountNotAuthorized
         {
         };

interface IBankAccountUserExceptions : IUnknown
         {
         HRESULT get_InsufFunds( [out]  BankInsufFunds
                                * exceptionBody );
         HRESULT get_InvalidAmount( [out]  BankInvalidAmount
                                * exceptionBody );
         HRESULT get_NotAuthorized( [out]
                                   BankAccountNotAuthorized
                                * exceptionBody );
         };

typedef struct
         {
         ExceptionType      type;
         LPTSTR          repositoryId;
         IBankAccountUserExceptions * piUserException;
      }  BankAccountExceptions;
```

User exceptions are mapped to a COM interface and a structure which describes the body of information to be returned for the User Exception. A COM interface is defined for each CORBA interface containing an operation that raises a User Exception. The name of the interface defined for accessing User Exception information is constructed from the fully scoped name of the CORBA interface on which the exception is raised. A structure is defined for each User Exception, which contains the body of information to be returned as part of that exception. The name of the structure follows the naming conventions used to map CORBA structure definitions.

Each User Exception that can be raised by an operation defined for a CORBA interface is mapped into an operation on the Exception interface. The name of the operation is constructed by prefixing the name of the exception with the string “get_”. Each accessor operation defined takes one output parameter in which to return the body of information defined for the User Exception. The data type of the output parameter is a structure that is defined for the exception. The operation is defined to return an HRESULT value.

If a CORBA User Exception is to be raised, the value of the HRESULT returned to the caller is E_FAIL.

If the caller specified a non-NULL value for the Exceptions structure parameter, the callee must allocate the memory to hold the exception information and fill in the Exceptions structure as in Table 13-5.

Table 13-5 User Exceptions Structure

Member	**Description**
type	Indicates the type of CORBA exception that is being raised. Must be USER_EXCEPTION.
repositoryId	Indicates the repository identifier for the exception definition.
piUserException	Points to an interface with which to obtain information about the User Exception raised.

When data conversion errors occur while mapping the data types between object models (during a call from a COM client to a CORBA server), an HRESULT with the code E_DATA_CONVERSION and the facility value FACILITY_NULL is returned to the client.

Mapping User Exceptions: A Special Case

If a CORBA operation raises only one User Exception, and it is the COM_ERROR User Exception (defined under Section 13.3.10, Mapping for COM Errors), then the mapped COM operation should not have the additional parameter for exceptions. This proviso enables a CORBA implementation of a preexisting COM interface to be mapped back to COM without altering the COM operation’s original signature.

COM_ERROR is defined as part of the CORBA to COM mapping. However, this special rule in effect means that a COM_ERROR raises clause can be added to an operation specifically to indicate that the operation was originally defined as a COM operation.

Mapping for Operations

Operations defined for an interface are defined in OMG IDL within interface definitions. The definition of an operation constitutes the operations signature. An operation signature consists of the operation’s name, parameters (if any), and return value. Optionally, OMG IDL allows the operation definition to indicate exceptions that can be raised, and the context to be passed to the object as implicit arguments, both of which are considered part of the operation.

OMG IDL parameter directional attributes **`in`**, **`out`**, **`inout`** map directly to Microsoft IDL and ODL parameter direction attributes `[in]`, `[out]`, `[in,out]`. Operation request parameters are represented as the values of **`in`** or **`inout`** parameters in OMG

IDL, and operation response parameters are represented as the values of **`inout`** or **`out`** parameters. An operation return result can be any type that can be defined in OMG IDL, or void if a result is not returned.

The OMG IDL sample (next) shows the definition of two operations on the Bank interface. The names of the operations are bolded to make them stand out. Operations can return various types of data as results, including nothing at all. The operation **`Bank::Transfer`** is an example of an operation that does not return a value. The operation **`Bank::OpenAccount`** returns an object as a result of the operation.

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

The operations defined in the preceding OMG IDL code is mapped to the following lines of Microsoft IDL code

```
// Microsoft IDL
[ object, uuid(682d22fb-78ac-0000-0c03-4d0000000000),
pointer_default(unique) ]
interface IBank : IUnknown
    {
    HRESULT  OpenAccount(  [in]   float    StartingBalance,
                           [in]   AccountTypes AccountType,
                           [out] IAccount   **ppiNewAccount);
    HRESULT  Transfer(     [in]IAccount * Account1,
                           [in] IAccount * Account2,
                           [in] float Amount,
                           [out] IBankUserExceptions
                              ** ppiUserException);
    };
```

and to the following statements in Microsoft ODL.

```
// Microsoft ODL
[ uuid(682d22fb-78ac-0000-0c03-4d0000000000) ]
interface IBank: IUnknown
    {
    HRESULT OpenAccount([in] float      StartingBalance,
                        [in] AccountTypes     AccountType,
                        [out, retval] IAccount
                           ** ppiNewAccount );
    HRESULT Transfer( [in] IAccount * Account1,
                        [in] IAccount  * Account2,
                        [in] float   Amount,
                        [out]IBankUserExceptions
                           ** ppiUserException);
  };
```

The ordering and names of parameters in the Microsoft IDL and ODL mapping is identical to the order in which parameters are specified in the text of the operation definition in OMG IDL. The COM mapping of all CORBA operations must obey the COM memory ownership and allocation rules specified.

It is important to note that the signature of the operation as written in OMG IDL is different from the signature of the same operation in Microsoft IDL or ODL. In particular, the result value returned by an operation defined in OMG IDL will be mapped as an output argument at the end of the signature when specified in Microsoft IDL or ODL. This allows the signature of the operation to be natural to the COM developer. When a result value is mapped as an output argument, the result value becomes an HRESULT. Without an HRESULT return value, there would be no way for COM to signal errors to clients when the client and server are not collocated. The value of the HRESULT is determined based on a mapping of the CORBA exception, if any, that was raised.

It is also important to note that if any user's exception information is defined for the operation, an additional parameter is added as the last argument of the operation signature. The user exception parameter follows the return value parameter, if both exist. See Mapping for Exception Types under Section 13.2.10 for further details.

Mapping for Oneway Operations

OMG IDL allows an operation's definition to indicate the invocation semantics the communication service must provide for an operation. This indication is done through the use of an operation attribute. Currently, the only operation attribute defined by CORBA is the oneway attribute.

The oneway attribute specifies that the invocation semantics are best-effort, which does not guarantee delivery of the request. Best-effort implies that the operation will be invoked, at most, once. Along with the invocation semantics, the use of the oneway operation attribute restricts an operation from having output parameters, must have no result value returned, and cannot raise any user-defined exceptions.

It may seem that the Microsoft IDL maybe operation attribute provides a closer match since the caller of an operation does not expect any response. However, Microsoft RPC maybe does not guarantee at most once semantics, and therefore is not sufficient. Because of this, the mapping of an operation defined in OMG IDL with the oneway operation attribute maps the same as an operation that has no output arguments.

Mapping for Attributes

OMG IDL allows the definition of attributes for an interface. Attributes are essentially a short-hand for a pair of accessor functions to an object's data; one to retrieve the value and possibly one to set the value of the attribute. The definition of an attribute must be contained within an interface definition and can indicate whether the value of the attribute can be modified or just read. In the example OMG IDL next, the attribute Profile is defined for the Customer interface and the read-only attribute is Balance defined for the Account interface. The keyword attribute is used by OMG IDL to indicate that the statement is defining an attribute of an interface.

The definition of attributes in OMG IDL are restricted from raising any user-defined exceptions. Because of this, the implementation of an attribute's accessor function is limited to only raising system exceptions. The value of the HRESULT is determined based on a mapping of the CORBA exception, if any, that was raised.

```
// 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;
     };
```

When mapping attribute statements in OMG IDL to Microsoft IDL or ODL, the name of the get accessor is the same as the name of the attribute prefixed with _get_ in Microsoft IDL and contains the operation attribute [propget] in Microsoft ODL. The name of the put accessor is the same as the name of the attribute prefixed with _put_ in Microsoft IDL and contains the operation attribute [propput] in Microsoft ODL.

Mapping for Read-Write Attributes

In OMG IDL, attributes are defined as supporting a pair of accessor functions: one to retrieve the value and one to set the value of the attribute unless the keyword readonly precedes the attribute keyword. In the preceding example, the attribute Profile is mapped to the following statements in Microsoft IDL.

```
// Microsoft IDL
[ object, uuid(682d22fb-78ac-0000-0c03-4d0000000000),
pointer_default(unique) ]
interface ICustomer : IUnknown
    {
    HRESULT  _get_Profile( [out]  CustomerData  * Profile );
    HRESULT  _put_Profile( [in]  CustomerData  * Profile );
    };
```

`Profile` is mapped to these statements in Microsoft ODL.

```
// Microsoft ODL
[ uuid(682d22fb-78ac-0000-0c03-4d0000000000) ]
interface ICustomer : IUnknown
    {
    [propget]  HRESULT  Profile( [out]  CustomerData
      * Profile );
    [propput]  HRESULT  Profile( [in]  CustomerData
      * Profile );
    };
```

Note that the attribute is actually mapped as two different operations in both Microsoft IDL and ODL. The ICustomer::Get_Profile, in Microsoft IDL operations and the [propget] Profile, in Microsoft ODL operations are used to retrieve the value of the attribute. The ICustomer::Set_Profile operation is used to set the value of the attribute.

Mapping for Read-Only Attributes

In OMG IDL, an attribute preceded by the keyword **`readonly`** is interpreted as only supporting a single accessor function used to retrieve the value of the attribute. In the previous example, the mapping of the attribute **`Balance`** is mapped to the following statements in Microsoft IDL.

```
// Microsoft IDL
[ object, uuid(682d22fb-78ac-0000-0c03-4d0000000000) ]
interface IAccount: IUnknown
    {
    HRESULT _get_Balance([out] float Balance);
   };
```

and the following statements in Microsoft ODL.

```
// Microsoft ODL
[ uuid(682d22fb-78ac-0000-0c03-4d0000000000) ]
interface IAccount: IUnknown
    {
    [propget] HRESULT Balance([out] float Balance);
    };
```

Note that only a single operation was defined since the attribute was defined to be read-only.

13.2.11 Inheritance Mapping

Both CORBA and COM have similar models for individual interfaces. However, the models for inheritance and multiple interfaces are different.

In CORBA, an interface can singly or multiply inherit from other interfaces. In language bindings supporting typed object references, widening and narrowing support convert object references as allowed by the true type of that object.

However, there is no built-in mechanism in CORBA to access interfaces without an inheritance relationship. The run-time interfaces of an object, as defined in *CORBA 2.0* (for example, **`CORBA::Object::is_a`**, **`CORBA::Object::get_interface`**) use a description of the object's principle type, which is defined in OMG IDL. CORBA allows many ways in which implementations of interfaces can be structured, including using implementation inheritance.

In COM V2.0, interfaces can have single inheritance. However, as opposed to CORBA, there is a standard mechanism by which an object can have multiple interfaces (without an inheritance relationship between those interfaces) and by which clients can query for these at run-time. (It defines no common way to determine if two interface references refer to the same object, or to enumerate all the interfaces supported by an entity.)

An observation about COM is that some COM objects have a required minimum set of interfaces, which they must support. This type of statically defined interface relation is conceptually equivalent to multiple inheritance; however, discovering this relationship is only possible if ODL or type libraries are always available for an object.

COM describes two main implementation techniques: aggregation and delegation. C++ style implementation inheritance is not possible.

The mapping for CORBA interfaces into COM is more complicated than COM interfaces into CORBA, since CORBA interfaces might be multiply inherited and COM does not support multiple interface inheritance.

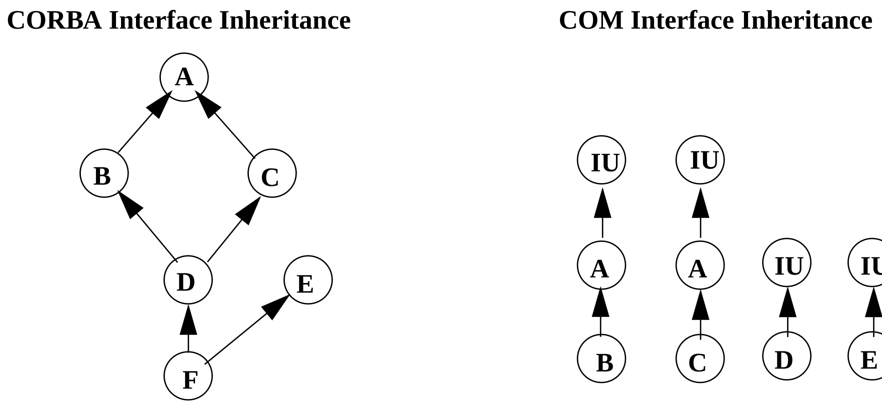
If a CORBA interface is singly inherited, this maps directly to single inheritance of interfaces in COM. The base interface for all CORBA inheritance trees is IUnknown. Note that the Object interface is not surfaced in COM. For single inheritance, although the most derived interface can be queried using **`IUnknown::QueryInterface`**, each individual interface in the inheritance hierarchy can also be queried separately.

The following rules apply to mapping CORBA to COM inheritance.

- Each OMG IDL interface that does not have a parent is mapped to an MIDL interface deriving from IUnknown.
- Each OMG IDL interface that inherits from a single parent interface is mapped to an MIDL interface that derives from the mapping for the parent interface.
- Each OMG IDL interface that inherits from multiple parent interfaces is mapped to an MIDL interface deriving from IUnknown.
- For each CORBA interface, the mapping for operations precede the mapping for attributes.
- The resulting mapping of operations within an interface are ordered based upon the operation name. The ordering is lexicographic by bytes in machine-collating order.
- The resulting mapping of attributes within an interface are ordered based upon the attribute name. The ordering is lexicographic by bytes in machine-collating order. If the attribute is not readonly, the get_<attribute name> method immediately precedes the set_<attribute name> method.

Figure 13-1 and the following OMG IDL and Microsoft MIDL illustrate this mapping.

Figure 13-1 CORBA Interface Inheritance to COM Interface Inheritance Mapping



```
//OMG IDL
//
interface A {
    void opA();
    attribute long val;
};
interface B : A {
    void opB();
};
interface C : A {
    void opC();
};
interface D : B, C {
    void opD();
};
interface E {
    void opE();
};
interface F : D, E {
    void opF();

}//Microsoft MIDL
//
[object, uuid(b97267fa-7855-e044-71fb-12fa8a4c516f)]
interface IA: IUnknown{
   HRESULT opA();
   HRESULT get_val([out] long * val);
   HRESULT set_val([in] long val);
};
[object, uuid(fa2452c3-88ed-1c0d-f4d2-fcf91ac4c8c6)]
interface IB: IA {
   HRESULT opB();
};
[object,uuid(dc3a6c32-f5a8-d1f8-f8e2-64566f815ed7)]
interface IC: IA {
   HRESULT opC();
};
[object, uuid(b718adec-73e0-4ce3-fc72-0dd11a06a308)]
interface ID: IUnknown {
   HRESULT opD();
};
[object, uuid(d2cb7bbc-0d23-f34c-7255-d924076e902f)]
interface IE: IUnknown{
   HRESULT opE();
};
[object, uuid(de6ee2b5-d856-295a-fd4d-5e3631fbfb93)]
interface IF: IUnknown {
   HRESULT opF();
};
```

Note that the co-class statement in Microsoft ODL allows the definition of an object class that allows QueryInterface between a set of interfaces.

Also note that when the interface defined in OMG IDL is mapped to its corresponding statements in Microsoft IDL, the name of the interface is proceeded by the letter I to indicate that the name represents the name of an interface. This also makes the mapping more natural to the COM programmer, since the naming conventions used follow those suggested by Microsoft.

13.2.12 Mapping for Pseudo-Objects

CORBA defines a number of different kinds of pseudo-objects. Pseudo-objects differ from other objects in that they cannot be invoked with the Dynamic Invocation Interface (DII) and do not have object references. Most pseudo-objects cannot be used as general arguments. Currently, only the TypeCode and Principal pseudo-objects can be used as general arguments to a request in CORBA.

The CORBA NamedValue and NVList are not mapped into COM as arguments to COM operation signatures.

Mapping for TypeCode Pseudo-Object

CORBA TypeCodes represent the types of arguments or attributes and are typically retrieved from the interface repository. The mapping of the CORBA TypeCode interface follows the same rules as mapping any other CORBA interface to COM. The result of this mapping is as follows.

```
// Microsoft IDL or ODL
typedef struct { } TypeCodeBounds;
typedef struct { } TypeCodeBadKind;
[uuid(9556EA20-3889-11cf-9586-AA0004004A09), object,
pointer_default(unique)]

interface ICORBA_TypeCodeUserExceptions : IUnknown
{
HRESULT get_Bounds( [out] TypeCodeBounds *ExceptionBody);
HRESULT get_BadKind( [out] TypeCodeBadKind  * pExceptionBody
);
};

typedef struct
{
    ExceptionType       type;
    LPTSTR              repositoryId;
    long                minorCode;
    CompletionStatus    completionStatus;
    ICORBA_SystemException  * pSystemException;
    ICORBA_TypeCodeExceptions   * pUserException;
} CORBATypeCodeExceptions;

typedef LPTSTR      RepositoryId;
typedef LPTSTR      Identifier;

typedef [v1_enum]
enum tagTCKind { tk_null = 0, 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
} TCKind;

[uuid(9556EA21-3889-11cf-9586-AA0004004A09), object,
pointer_default(unique)]

interface ICORBA_TypeCode : IUnknown
{
    HRESULT equal(
[in]  ICORBA_TypeCode * piTc,
[out] boolean      * pbRetVal,
[out] CORBATypeCodeExceptions** ppUserExceptions );
HRESULT kind(
[out] TCKind       * pRetVal,
[out] CORBATypeCodeExceptions ** ppUserExceptions );
    HRESULT id(
[out] RepositoryId  * pszRetVal,
[out] CORBATypeCodeExceptions ** ppUserExceptions );
    HRESULT name(
[out] Identifier   * pszRetVal,
[out] CORBATypeCodeExceptions ** ppUserExceptions );
    HRESULT member_count(
[out] unsigned long  * pulRetVal,
[out] CORBATypeCodeExceptions ** ppUserExceptions );
    HRESULT member_name(
[in]  unsigned long    ulIndex,
[out] Identifier   * pszRetVal,
[out] CORBATypeCodeExceptions ** ppUserExceptions );
    HRESULT member_type(
[in]  unsigned long    ulIndex,
[out] ICORBA_TypeCode ** ppRetVal,
[out] CORBATypeCodeExceptions ** ppUserExceptions );
    HRESULT member_label(
[in]  unsigned long    ulIndex,
[out] ICORBA_Any   ** ppRetVal,
[out] CORBATypeCodeExceptions ** ppUserExceptions );
    HRESULT discriminator_type(
[out] ICORBA_TypeCode** ppRetVal,
[out] CORBATypeCodeExceptions ** ppUserExceptions );
    HRESULT default_index(
[out] long         * plRetVal,
[out] CORBATypeCodeExceptions ** ppUserExceptions);
    HRESULT length(
[out] unsigned long        * pulRetVal,
[out] CORBATypeCodeExceptions   ** ppUserExceptions );
    HRESULT content_type(
[out] ICORBA_TypeCode         ** ppRetVal,
[out] CORBATypeCodeExceptions ** ppUserExceptions );
    HRESULT param_count(
[out] long                        * plRetVal,
[out] CORBATypeCodeExceptions ** ppUserExceptions );
    HRESULT parameter(
[in]  long           lIndex,
[out] ICORBA_Any   ** ppRetVal,
[out] CORBATypeCodeExceptions    ** ppUserExceptions );
}
```

Mapping for Context Pseudo-Object

This specification provides no mapping for CORBA's Context pseudo-object into COM. Implementations that choose to provide support for Context could do so in the following way. Context pseudo-objects should be accessed through the ICORBA Context interface. This would allow clients (if they are aware that the object they are dealing with is a CORBA object) to set a single Context pseudo-object to be used for all subsequent invocations on the CORBA object from the client process space until such time as the ICORBA_Context interface is released.

The ICORBA_Context interface has the following definition in Microsoft IDL and ODL:

```
// Microsoft IDL and ODL
typedef struct
    {
    unsigned long cbMaxSize;
    unsigned long cbLengthUsed;
    [size_is(cbMaxSize), length_is(cbLengthUsed), unique]
         LPTSTR * pszValue;
    } ContextPropertyValue;

[ object, uuid(74105F51-3C68-11cf-9588-AA0004004A09),
pointer_default(unique) ]
interface ICORBA_Context: IUnknown
    {
    HRESULT GetProperty([in]LPTSTR Name,
                        [out] ContextPropertyValue
                           ** ppValues );
    HRESULT SetProperty([in] LPTSTR,
                        [in] ContextPropertyValue
                           * pValues);
    };
```

If a COM client application knows it is using a CORBA object, the client application can use QueryInterface to obtain an interface pointer to the ICORBA_Context interface. Obtaining the interface pointer results in a CORBA context pseudo-object being created in the View, which is used with any CORBA request operation that requires a reference to a CORBA context object. The context pseudo-object should be destroyed when the reference count on the ICORBA_Context interface reaches zero.

This interface should only be generated for CORBA interfaces that have operations defined with the context clause.

Mapping for Principal Pseudo-Object

The CORBA Principal is not currently mapped into COM. As both the COM and CORBA security mechanisms solidify, security interworking will need to be defined between the two object models.

13.2.13 Interface Repository Mapping

Name spaces within the CORBA interface repository are conceptually similar to COM type libraries. However, the CORBA interface repository looks, to the client, to be one unified service. Type libraries, on the other hand, are each stored in a separate file. Clients do not have a unified, hierarchical interface to type libraries.

Table 13-6 defines the mapping between equivalent CORBA and COM interface description concepts. Where there is no equivalent, the field is left blank.

Table 13-6 CORBA Interface Repository to OLE Type Library Mappings

TypeCode	**TYPEDESC**
Repository	
ModuleDef	ITypeLib
InterfaceDef	ITypeInfo
AttributeDef	VARDESC
OperationDef	FUNCDESC
ParameterDef	ELEMDESC
TypeDef	ITypeInfo
ConstantDef	VARDESC
ExceptionDef	

Using this mapping, implementations must provide the ability to call **`Object::get_interface`** on CORBA object references to COM objects to retrieve an InterfaceDef. When CORBA objects are accessed from COM, implementations may provide the ability to retrieve the ITypeInfo for a CORBA object interface using the IProvideClassInfo COM interface.

13.3 COM to CORBA Data Type Mapping

13.3.1 Mapping for Basic Data Types

The basic data types available in Microsoft IDL and ODL map to the corresponding data types available in OMG IDL as shown in Table 13-7.

Table 13-7 Microsoft IDL and ODL to OMG IDL Intrinsic Data Type Mappings

Microsoft IDL	**Microsoft ODL**	**OMG IDL**	**Description**
short	short	short	Signed integer with a range of $-2^{15}...2^{15}$ -1
long	long	long	Signed integer with a range of $-2^{31}...2^{31}$ -1
unsigned short	unsigned short	unsigned short	Unsigned integer with a range of $0...2^{16}$ -1
unsigned long	unsigned long	unsigned long	Unsigned integer with a range of $0...2^{32}$ -1
float	float	float	IEEE single -precision floating point number
double	double	double	IEEE double-precision floating point number
char	char	char	8-bit quantity limited to the ISO Latin-1 character set
boolean	boolean	boolean	8-bit quantity, which is limited to 1 and 0
byte	unsigned char	octet	8-bit opaque data type, guaranteed to not undergo any conversion during transfer between systems

13.3.2 Mapping for Constants

The mapping of the Microsoft IDL keyword const to OMG IDL const is almost exactly the same. The following Microsoft IDL definitions for constants

```
// Microsoft IDL
const short S = ...;
const long L = ...;
const unsigned short US = ...;
const unsigned long UL = ...;
const char C = ...;
const boolean B = ...;
const string STR = “...”;
```

map to the following OMG IDL definitions for constants.

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

13.3.3 Mapping for Enumerators

COM enumerations can have enumerators explicitly tagged with values. When COM enumerations are mapped into CORBA, the enumerators are presented in CORBA, ordered according to their tagged values. This Microsoft IDL or ODL

```
// Microsoft IDL or ODL
  typedef [v1_enum] enum tagA_or_B_orC  {  A = 0, B, C }
A_or_B_or_C;
```

would be represented as the following statements in OMG IDL:

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

Because COM allows enumerators to be defined with explicit tagged values, the enumerators are mapped to OMG IDL in the same order they appear in Microsoft IDL or ODL and it is the COM View's responsibility to maintain the mapping based on names.

13.3.4 Mapping for String Types

COM support for strings includes the concepts of bounded and unbounded strings. Bounded strings are defined as strings that have a maximum length specified, whereas unbounded strings do not have a maximum length specified. COM also supports Unicode strings where the characters are wider than 8 bits. As in OMG IDL, non-Unicode strings in COM are NULL-terminated. The mapping of COM definitions for bounded and unbounded strings differs from that specified in OMG IDL.

Table 13-8 illustrates how to map the string data types in OMG IDL to their corresponding data types in both Microsoft IDL and ODL.

Table 13-8 Microsoft IDL/ODL to OMG IDL String Mappings

Microsoft IDL	**Microsoft ODL**	**OMG IDL**	**Description**
LPSTR, char *	LPSTR,	string	Null terminated 8-bit character string
LPTSTR	LPTSTR	string	Null terminated 8-bit character string
	BSTR on Win16	string	Null-terminated 8-bit character string

If a COM Server returns a BSTR containing embedded nulls to a CORBA client, a E_DATA_CONVERSION exception will be raised.

Mapping for Unbounded String Types

The definition of an unbounded string in Microsoft IDL and ODL denotes the unbounded string as a stringified unique pointer to a character. The following Microsoft IDL statement

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

is mapped to the following syntax in OMG IDL.

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

In other words, a value of type UNBOUNDED_STRING is a non-NULL pointer to a one-dimensional null-terminated character array whose extent and number of valid elements can vary at run-time.

Mapping for Bounded String Types

Bounded strings have a slightly different mapping between OMG IDL and Microsoft IDL. Bounded strings are expressed in Microsoft IDL as a “stringified nonconformant array.” The following Microsoft IDL and ODL definition for a bounded string

```
// Microsoft IDL and ODL
 const long N = ...;
 typedef [string, unique] char (* BOUNDED_STRING) [N];
```

maps to the following syntax in OMG IDL.

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

In other words, the encoding for a value of type BOUNDED_STRING is that of a null-terminated array of characters whose extent is known at compile time, and the number of valid characters can vary at run-time.

Mapping for Unicode Unbounded String Types

The mapping for a Unicode unbounded string type in Microsoft IDL or ODL is no different from that used for ANSI string types. The following Microsoft IDL and ODL statement

```
// Microsoft IDL and ODL
 typedef [string, unique] LPTSTR UNBOUNDED_UNICODE_STRING;
```

is mapped to the following syntax in OMG IDL.

```
// OMG IDL
 typedef wstring UNBOUNDED_UNICODE_STRING;
```

It is the responsibility of the mapping implementation to perform the conversions between ANSI and Unicode formats when dealing with strings.

Mapping for Unicode Bound String Types

The mapping for a Unicode bounded string type in Microsoft IDL or ODL is no different from that used for ANSI string types. The following Microsoft IDL and ODL statements

```
// Microsoft IDL and ODL
 const long N = ...;
 typedef [string, unique] TCHAR (* BOUNDED_UNICODE_STRING)
[N];
```

map to the following syntax in OMG IDL.

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

It is the responsibility of the mapping implementation to perform the conversions between ANSI and Unicode formats when dealing with strings.

13.3.5 Mapping for Structure Types

Support for structures in Microsoft IDL and ODL maps bidirectionally to OMG IDL. Each structure members is mapped according to the mapping rules for that data type. The structure definition in Microsoft IDL or ODL is as follows.

```
// Microsoft IDL and ODL
 typedef ... T0;
 typedef ... Tl;
 ...
 typedef ...TN;
 typedef struct
   {
   T0 m0;
   Tl ml;
   ...
   TN mN;
   } STRUCTURE;
```

The structure has an equivalent mapping in OMG IDL, as follows.

```
// OMG IDL
 typedef ... T0
 typedef ... T1;
 ...
 typedef ... TN;
 struct STRUCTURE
    {
    T0 m0;
    T1 ml;
    ...
    Tn mm;
    };
```

13.3.6 Mapping for Union Types

ODL unions are not discriminated unions and must be custom marshaled in any interfaces that use them. For this reason, this specification does not provide any mapping for ODL unions to CORBA unions.

MIDL unions, while always discriminated, are not required to be encapsulated. The discriminator for a nonencapsulated MIDL union could, for example, be another argument to the operation. The discriminants for MIDL unions are not required to be constant expressions.

Mapping for Encapsulated Unions

When mapping from Microsoft IDL to OMG IDL, Microsoft IDL encapsulated unions having constant discriminators are mapped to OMG IDL unions as shown next.

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

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

The OMG IDL definition is as follows.

```
// OMG IDL
 enum UNION_DISCRIMINATOR
    {
    dChar,
    dShort,
    dLong,
    dFloat,
    dDouble
    };

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

Mapping for Nonencapsulated Unions

Microsoft IDL nonencapsulated unions and Microsoft IDL encapsulated unions with nonconstant discriminators are mapped to an **any** in OMG IDL. The type of the **any** is determined at run-time during conversion of the Microsoft IDL union.

```
// Microsoft IDL
typedef [switch_type( short )] union
tagUNION_OF_CHAR_AND_ARITHMETIC
      {
      [case(0)] char c;
      [case(1)] short s;
      [case(2)] long l;
      [case(3)] float f;
      [case(4)] double d;
      [default] byte v[8];
      } UNION_OF_CHAR_AND_ARITHMETIC;
```

The corresponding OMG IDL syntax is as follows.

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

13.3.7 Mapping for Array Types

COM supports fixed-length arrays, just as in CORBA. As in the mapping from OMG IDL to Microsoft IDL, the arrays can be mapped bidirectionally. The type of the array elements is mapped according to the data type mapping rules. The following statements in Microsoft IDL and ODL describe a nonconformant and nonvarying array of U.

```
// Microsoft IDL for T
const long N = ...;
typedef ... U;
typedef U ARRAY_OF_N[N];
typedef float DTYPE[0..10];   // Equivalent to [11]
```

The value N can be of any integral type, and const means (as in OMG IDL) that the value of N is fixed and known at compilation time. The generalization to multidimensional arrays follows the obvious trivial mapping of syntax.

The corresponding OMG IDL syntax is as follows.

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

Mapping for Nonfixed Arrays

In addition to fixed length arrays, as well as conformant arrays, COM supports varying arrays, and conformant varying arrays. These are arrays whose bounds and size can be determined at run-time. Nonfixed length arrays in Microsoft IDL and ODL are mapped to sequence in OMG IDL, as shown in the following statements.

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

The corresponding OMG IDL syntax is as follows.

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

Mapping for SAFEARRAY

Microsoft ODL also defines SAFEARRAY as a variable length, variable dimension array. Both the number of dimensions and the bounds of the dimensions are determined at run-time. Only the element type is predefined. A SAFEARRAY in Microsoft ODL is mapped to a CORBA sequence, as shown in the following statements.

```
// Microsoft ODL
SAFEARRAY(element-type) * ArrayName;
```

```
// OMG IDL
typedef sequence<element-type> SequenceName;
```

If a COM server returns a multidimensional SAFEARRAY to a CORBA client, an E_DATA_CONVERSION exception will be raised.

13.3.8 Mapping for VARIANT

The COM VARIANT provides semantically similar functionality to the CORBA **any**. However, its allowable set of data types are currently limited to the data types supported by OLE Automation. VARTYPE is an enumeration type used in the VARIANT structure. The structure member *vt* is defined using the data type VARTYPE. Its value acts as the discriminator for the embedded union and governs the interpretation of the union. The list of valid values for the data type VARTYPE are listed in Table 13-9, along with a description of how to use them to represent the OMG IDL **any** data type.

Table 13-9 Valid OLE VARIANT Data Types

Value	Description
VT_EMPTY	No value was specified. If an argument is left blank, you should **not** return VT_EMPTY for the argument. Instead, you should return the VT_ERROR value: DISP_E_MEMBERNOTFOUND.
VT_EMPTY | VT_BYREF	Illegal.
VT_UI1	An unsigned 1-byte character is stored in *bVal*.
VT_UI1 | VT_BYREF	A reference to an unsigned 1-byte character was passed; a pointer to the value is in *pbVal*.
VT_I2	A 2-byte integer value is stored in *iVal*.
VT_I2 | VT_BYREF	A reference to a 2-byte integer was passed; a pointer to the value is in *piVal*.
VT_I4	A 4-byte integer value is stored in *lVa*l.
VT_I4 | VT_BYREF	A reference to a 4-byte integer was passed; a pointer to the value is in *plVa*l.
VT_R4	An IEEE 4-byte real value is stored in *fltVal*.
VT_R4 | VT_BYREF	A reference to an IEEE 4-byte real was passed; a pointer to the value is in *pfltVal*.
VT_R8	An 8-byte IEEE real value is stored in *dblVal*.
VT_R8 | VT_BYREF	A reference to an 8-byte IEEE real was passed; a pointer to its value is in *pdblVal*.
VT_CY	A currency value was specified. A currency number is stored as an 8-byte, two's complement integer, scaled by 10,000 to give a fixed-point number with 15 digits to the left of the decimal point and 4 digits to the right. The value is in *cyVal*.
VT_CY | VT_BYREF	A reference to a currency value was passed; a pointer to the value is in *pcyVal*.
VT_BSTR	A string was passed; it is stored in *bstrVa*l. This pointer must be obtained and freed via the BSTR functions.
VT_BSTR | VT_BYREF	A reference to a string was passed. A BSTR*, which points to a BSTR, is in *pbstrVal*. The referenced pointer must be obtained or freed via the BSTR functions.
VT_NULL	A propagating NULL value was specified. This should not be confused with the NULL pointer. The NULL value is used for tri-state logic as with SQL.
VT_NULL | VT_BYREF	Illegal.

Table 13-9 Valid OLE VARIANT Data Types

Value	Description
VT_ERROR	An SCODE was specified. The type of the error is specified in *code*. Generally, operations on error values should raise an exception or propagate the error to the return value, as appropriate.
VT_ERROR | VT_BYREF	A reference to an SCODE was passed. A pointer to the value is in *pscode*.
VT_BOOL	A Boolean (True/False) value was specified. A value of 0xFFFF (all bits one) indicates True; a value of 0 (all bits zero) indicates False. No other values are legal.
VT_BOOL | VT_BYREF	A reference to a Boolean value. A pointer to the Boolean value is in *pbool*.
VT_DATE	A value denoting a date and time was specified. Dates are represented as double-precision numbers, where midnight, January 1, 1900 is 2.0, January 2, 1900 is 3.0, and so on. The value is passed in *date*. This is the same numbering system used by most spreadsheet programs, although some incorrectly believe that February 29, 1900 existed, and thus set January 1, 1900 to 1.0. The date can be converted to and from an MS-DOS representation using VariantTimeToDosDateTime.
VT_DATE | VT_BYREF	A reference to a date was passed. A pointer to the value is in *pdate*.
VT_DISPATCH	A pointer to an object was specified. The pointer is in *pdispVal*. This object is only known to implement IDispatch; the object can be queried as to whether it supports any other desired interface by calling QueryInterface on the object. Objects that do not implement IDispatch should be passed using VT_UNKNOWN.
VT_DISPATCH | VT_BYREF	A pointer to a pointer to an object was specified. The pointer to the object is stored in the location referred to by *ppdispVal*.
VT_VARIANT	Illegal. VARIANTARGs must be passed by reference.
VT_VARIANT | VT_BYREF	A pointer to another VARIANTARG is passed in *pvarVal*. This referenced VARIANTARG will never have the VT_BYREF bit set in *vt*, so only one level of indirection can ever be present. This value can be used to support languages that allow functions to change the types of variables passed by reference.
VT_UNKNOWN	A pointer to an object that implements the IUnknown interface is passed in *punkVal*.
VT_UNKNOWN | VT_BYREF	A pointer to a pointer to the IUnknown interface is passed in *ppunkVal*. The pointer to the interface is stored in the location referred to by *ppunkVal*.

Table 13-9 Valid OLE VARIANT Data Types

Value	Description
VT_ARRAY | <anything>	An array of data type <anything> was passed. (VT_EMPTY and VT_NULL are illegal types to combine with VT_ARRAY.) The pointer in *pByrefVal* points to an array descriptor, which describes the dimensions, size, and in-memory location of the array. The array descriptor is never accessed directly, but instead is read and modified using functions.

A COM VARIANT is mapped to the CORBA **any** without loss. If at run-time a CORBA client passes an inconvertible **any** to a COM server, a DATA_CONVERSION exception is raised.

13.3.9 Mapping for Pointers

MIDL supports three types of pointers:

- Reference pointer; a non-null pointer to a single item. The pointer cannot represent a data structure with cycles or aliasing (two pointers to the same address).
- Unique pointer; a (possibly null) pointer to a single item. The pointer cannot represent a data structure with cycles or aliasing.
- Full pointer; a (possibly null) pointer to a single item. Full pointers can be used for data structures, which form cycles or have aliases.

A reference pointer is mapped to a CORBA sequence containing one element. Unique pointers and full pointers with no aliases or cycles are mapped to a CORBA sequence containing zero or one elements. If at run-time a COM client passes a full pointer containing aliases or cycles to a CORBA server, E_DATA_CONVERSION is returned to the COM client. If a COM server attempts to return a full pointer containing aliases or cycles to a CORBA client, a DATA_CONVERSION exception is raised.

13.3.10 Interface Mapping

COM is a binary standard based upon standard machine calling conventions. Although interfaces can be expressed in Microsoft IDL, Microsoft ODL, or C++, the following interface mappings between COM and CORBA will use Microsoft ODL as the language of expression for COM constructs.

COM interface pointers bidirectionally map to CORBA Object references with the appropriate mapping of Microsoft IDL and ODL interfaces to OMG IDL interfaces.

Mapping for Interface Identifiers

Interface identifiers are used in both CORBA and COM to uniquely identify interfaces. These allow the client code to retrieve information about, or to inquire about other interfaces of an object.

COM identifies interfaces using a structure similar to the DCE UUID (in fact, identical to a DCE UUID on Win32) known as an IID. As with CORBA, COM specifies that the textual names of interfaces are only for convenience and need not be globally unique.

The COM interface identifier (IID and CLSID) are bidirectionally mapped to the CORBA RepositoryId.

Mapping for COM Errors

COM will provide error information to clients only if an operation uses a return result of type HRESULT. The COM HRESULT, if zero, indicates success. The HRESULT, if nonzero, can be converted into an SCODE (the SCODE is explicitly specified as being the same as the HRESULT on Win32). The SCODE can then be examined to determine whether the call succeeded or failed. The error or success code, also contained within the SCODE, is composed of a "facility" major code (13 bits on Win32 and 4 bits on Win16) and a 16-bit minor code.

COM object developers are expected to use one of the predefined SCODE values, or use the facility FACILITY_ITF and an interface specific minor code. SCODE values can indicate either success codes or error codes. A typical use is to overload the SCODE with a boolean value, using S_OK and S_FALSE success codes to indicate a true or false return. If the COM server returns S_OK or S_FALSE, a CORBA exception will not be raised and the value of the SCODE will be mapped as the return value. This is because COM operations, which are defined to return an HRESULT, are mapped to CORBA as returning an HRESULT.

Unlike CORBA, COM provides no standard way to return user-defined exception data to the client. Also, there is no standard mechanism in COM to specify the completion status of an invocation. In addition, it is not possible to predetermine what set of errors a COM interface might return. Although the set of success codes that can be returned from a COM operation must be fixed when the operation is defined, there is currently no machine-readable way to discover what the set of valid success codes are.

COM exceptions have a straightforward mapping into CORBA. COM system error codes are mapped to the CORBA standard exceptions. COM user-defined error codes are mapped to CORBA user exceptions.

COM system error codes are defined with the FACILITY_NULL and FACILITY_RPC facility codes. All FACILITY_NULL and FACILITY_RPC COM errors are mapped to CORBA standard exceptions. Table 13-10 lists the mapping from COM FACILITY_NULL exceptions to CORBA standard exceptions.

Table 13-10 Mapping from COM FACILITY_NULL Error Codes to CORBA Standard (System) Exceptions

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

Table 13-11 lists the mapping from COM FACILITY_RPC exceptions to CORBA standard exceptions. All FACILITY_RPC exceptions not listed in this table are mapped to the new CORBA standard exception COM.

Table 13-11 Mapping from COM FACILITY_RPC Error Codes to CORBA Standard (System) Exceptions

COM	**CORBA**
RPC_E_CALL_CANCELED	TRANSIENT
RPC_E_CANTPOST_INSENDCALL	COMM_FAILURE
RPC_E_CANTCALLOUT_INEXTERNALCALL	COMM_FAILURE
RPC_E_CONNECTION_TERMINATED	NV_OBJREF
RPC_E_SERVER_DIED	INV_OBJREF
RPC_E_SERVER_DIED_DNE	INV_OBJREF
RPC_E_INVALID_DATAPACKET	COMM_FAILURE
RPC_E_CANTTRANSMIT_CALL	TRANSIENT
RPC_E_CLIENT_CANTMARSHAL_DATA	MARSHAL
RPC_E_CLIENT_CANTUNMARSHAL_DATA	MARSHAL
RPC_E_SERVER_CANTMARSHAL_DATA	MARSHAL
RPC_E_SERVER_CANTUNMARSHAL_DATA	MARSHAL
RPC_E_INVALID_DATA	COMM_FAILURE
RPC_E_INVALID_PARAMETER	BAD_PARAM
RPC_E_CANTCALLOUT_AGAIN	COMM_FAILURE
RPC_E_SYS_CALL_FAILED	NO_RESOURCES
RPC_E_OUT_OF_RESOURCES	NO_RESOURCES
RPC_E_NOT_REGISTERED	NO_IMPLEMENT
RPC_E_DISCONNECTED	INV_OBJREF
RPC_E_RETRY	TRANSIENT
RPC_E_SERVERCALL_REJECTED	TRANSIENT
RPC_E_NOT_REGISTERED	NO_IMPLEMENT

COM SCODEs, other than those previously listed, are mapped into CORBA user exceptions and will require the use of the **`raises`** clause in OMG IDL. Since the OMG IDL mapping from the Microsoft IDL and ODL is likely to be generated, this is not a burden to the average programmer. The following OMG IDL illustrates such a user exception.

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

When data conversion errors occur while mapping the data types between object models (during a call from a CORBA client to a COM server), the system exception DATA_CONVERSION will be raised.

Mapping for Operations

Operations defined for an interface are defined in Microsoft IDL and ODL within interface definitions. The definition of an operation constitutes the operations signature. An operation signature consists of the operation's name, parameters (if any), and return value. Unlike OMG IDL, Microsoft IDL and ODL does not allow the operation definition to indicate the error information that can be returned.

Microsoft IDL and ODL parameter directional attributes (**[in]**, **[out]**, **[in, out]**) map directly to OMG IDL (**in**, **out**, **inout**). Operation request parameters are represented as the values of **[in]** or **[inout]** parameters in Microsoft IDL, and operation response parameters are represented as the values of **[inout]** or **[out]** parameters. An operation return result can be any type that can be defined in Microsoft IDL/ODL, or void if a result is not returned. By convention, most operations are defined to return an HRESULT. This provides a consistent way to return operation status information.

When Microsoft ODL methods are mapped to OMG IDL, they undergo the following transformations. First, if the last parameter is tagged with the Microsoft ODL keyword retval, that argument will be used as the return type of the operation. If the last parameter is not tagged with retval, then the signature is mapped directly to OMG IDL following the mapping rules for the data types of the arguments. Some example mappings from COM methods to OMG IDL operations are shown in the following code.

```
// Microsoft ODL
interface IFoo: IUnknown
   {
   HRESULT stringify ([in] VARIANT value,
                      [out, retval] LPSTR * pszValue);

   HRESULT permute(  [inout] short   * value);

   HRESULT tryPermute([inout] short * value,
                      [out] long newValue);
   };
```

In OMG IDL this becomes:

```
typedef long HRESULT;
interface IFoo: CORBA::Composite, CosLifeCycle::LifeCycleObject
    {
    string stringify(in any value) raises (COM_ERROR);

    HRESULT permute(inout short value);

    HRESULT tryPermute(inout short value, out long newValue)
    };
```

Mapping for Properties

In COM, only Microsoft ODL and OLE Type Libraries provide support for describing properties. Microsoft IDL does not support this capability. Any operations that can be determined to be either a put/set or get accessor are mapped to an attribute in OMG IDL. Because Microsoft IDL does not provide a means to indicate that something is a property, a mapping from Microsoft IDL to OMG IDL will not contain mappings to the attribute statement in OMG IDL.

When mapping between Microsoft ODL or OLE Type Libraries, properties in COM are mapped in a similar fashion to that used to map attributes in OMG IDL to COM. For example, the following Microsoft ODL statements define the attribute Profile for the ICustomer interface and the read-only attribute Balance for the IAccount interface. The keywords [propput] and [propget] are used by Microsoft ODL to indicate that the statement is defining a property of an interface.

```
// Microsoft ODL
interface IAccount
    {
    [propget] HRESULT Balance([out, retval] float
      * pfBalance );
    ...
    };

interface ICustomer
    {
    [propget] HRESULT Profile([out] CustomerData  * Profile);
    [propput] HRESULT Profile([in] CustomerData  * Profile);
    };
```

The definition of attributes in OMG IDL are restricted from raising any user-defined exceptions. Because of this, the implementation of an attribute’s accessor function is limited to raising system exceptions. The value of the HRESULT is determined by a mapping of the CORBA exception, if any, that was raised.

13.3.11 Mapping for Read-Only Attributes

In Microsoft ODL, an attribute preceded by the keyword [propget] is interpreted as only supporting an accessor function, which is used to retrieve the value of the attribute. In the example above, the mapping of the attribute Balance is mapped to the following statements in OMG IDL.

```
// OMG IDL
interface Account
    {
    readonly attribute float Balance;
    ...
    };
```

13.3.12 Mapping for Read-Write Attributes

In Microsoft ODL, an attribute preceded by the keyword [propput] is interpreted as only supporting an accessor function which is used to set the value of the attribute. In the previous example, the attribute Profile is mapped to the following statements in OMG IDL.

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

interface Customer
    {
    attribute CustomerData  Profile;
    ...
    };
```

Since CORBA does not have the concept of write-only attributes, the mapping must assume that a property that has the keyword [propput] is mapped to a single read-write attribute, even if there is no associated [propget] method defined.

Inheritance Mapping

Both CORBA and COM have similar models for individual interfaces. However, the models for inheritance and multiple interfaces are different.

In CORBA, an interface can singly or multiply inherit from other interfaces, and in language bindings supporting typed object references, widening and narrowing support convert object references as allowed by the true type of that object.

However, there is no built-in mechanism in CORBA to access interfaces without an inheritance relationship. The run-time interfaces of an object (for example, **`CORBA::Object::is_a`**, **`CORBA::Object::get_interface`**) use a description of the object's principle type, which is defined in OMG IDL. In terms of implementation, CORBA allows many ways in which implementations of interfaces can be structured, including using implementation inheritance.

In COM V2.0, interfaces can have single inheritance. However, as opposed to CORBA, there is a standard mechanism by which an object can have multiple interfaces (without an inheritance relationship between those interfaces) and by which clients can query for these at run-time. (It defines no common way to determine if two interface references refer to the same object, or to enumerate all the interfaces supported by an entity.)

An observation about COM is that some COM objects have a required minimum set of interfaces that they must support. This type of statically-defined interface relation is conceptually equivalent to multiple inheritance; however, discovering this relationship is only possible if ODL or type libraries are always available for an object.

COM describes two main implementation techniques: aggregation and delegation. C++ style implementation inheritance is not possible.

When COM interfaces are mapped into CORBA, their inheritance hierarchy (which can only consist of single inheritance) is directly mapped into the equivalent OMG IDL inheritance hierarchy.[1]

Note that although it is possible, using Microsoft ODL to map multiple COM interfaces in a class to OMG IDL multiple inheritance, the necessary information is not available for interfaces defined in Microsoft IDL. As such, this specification does not define a multiple COM interface to OMG IDL multiple inheritance mapping. It is assumed that future versions of COM will merge Microsoft ODL and Microsoft IDL, at which time the mapping can be extended to allow for multiple COM interfaces to be mapped to OMG IDL multiple inheritance.

`CORBA::Composite` is a general-purpose interface used to provide a standard mechanism for accessing multiple interfaces from a client, even though those interfaces are not related by inheritance. Any existing ORB can support this interface, although in some cases a specialized implementation framework may be desired to take advantage of this interface.

1.This mapping fails in some cases, for example, if operation names are the same.

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

The root of a COM interface inheritance tree, when mapped to CORBA, is multiply inherited from **`CORBA::Composable`** and **`CosLifeCycle::LifeCycleObject`**. Note that the IUnknown interface is not surfaced in OMG IDL. Any COM method parameters that require IUnknown interfaces as arguments are mapped, in OMG IDL, to object references of type **`CORBA::Object`**.

```
// Microsoft IDL or ODL
interface IFoo: IUnknown
   {
   HRESULT inquire([in] IUnknown *obj);
   };
```

In OMG IDL, this becomes:

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

Type Library Mapping

Name spaces within the OLE Type Library are conceptually similar to CORBA interface repositories. However, the CORBA interface repository looks, to the client, to be one unified service. Type libraries, on the other hand, are each stored in a separate file. Clients do not have a unified, hierarchical interface to type libraries.

The following table defines the mapping between equivalent CORBA and COM interface description concepts. Where there is no equivalent, the field is left blank.

Table 13-12 CORBA Interface Repository to OLE Type Library Mappings

CORBA	**COM**
TypeCode	TYPEDESC
Repository	
ModuleDef	ITypeLib
InterfaceDef	ITypeInfo
AttributeDef	VARDESC
OperationDef	FUNCDESC
ParameterDef	ELEMDESC
TypeDef	ITypeInfo
ConstantDef	VARDESC
ExceptionDef	

Using this mapping, implementations must provide the ability to call **`Object::get_interface`** on CORBA object references to COM objects to retrieve an InterfaceDef. When CORBA objects are accessed from COM, implementations may provide the ability to retrieve the ITypeInfo for CORBA object interface using the IProvideClassInfo COM interface.

Mapping: OLE Automation and CORBA 13C

This chapter describes the bidirectional data type and interface mapping between OLE Automation and CORBA.

Microsoft's Object Description Language (ODL) is used to describe Automation object model constructs. However, many constructs supported by ODL are not supported by Automation. Therefore, this specification is confined to the Automation-compatible ODL constructs.

As described in Chapter 13A, Interworking Architecture, many implementation choices are open to the vendor in building these mappings. One valid approach is to generate and compile mapping code, an essentially static approach. Another is to map objects dynamically.

Although some features of the CORBA-Automation mappings address the issue of inverting a mapping back to its original platform, this specification does not assume the requirement for a totally invertible mapping between Automation and CORBA.

13.1 Mapping CORBA Objects to OLE Automation

13.1.1 Architectural Overview

There are seven main pieces involved in the invocation of a method on a remote CORBA object: the OLE Automation Controller; the COM Communication Infrastructure; the OLE system registry; the client-side Automation View; the operation's type information; the Object Request Broker; and the CORBA object's implementation. These are illustrated in Figure 13-1 (the call to the Automation View could be a call in the same process).

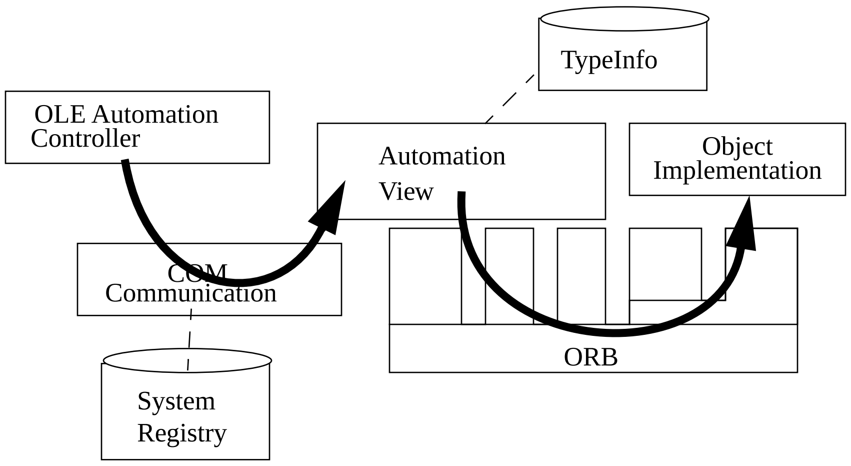


Figure 13-1 CORBA Object Architectural Overview

The Automation View is an OLE Automation server with a dispatch interface that is isomorphic to the mapped OMG IDL interface. We call this dispatch interface an Automation View Interface. The Automation server encapsulates a CORBA object reference and maps incoming OLE Automation invocations into CORBA invocations on the encapsulated reference. The creation and storage of the type information is not specified.

There is a one-to-one correspondence between the methods of the Automation View Interface and operations in the CORBA interface. The Automation View Interface's methods translate parameters bidirectionally between a CORBA reference and an OLE reference.

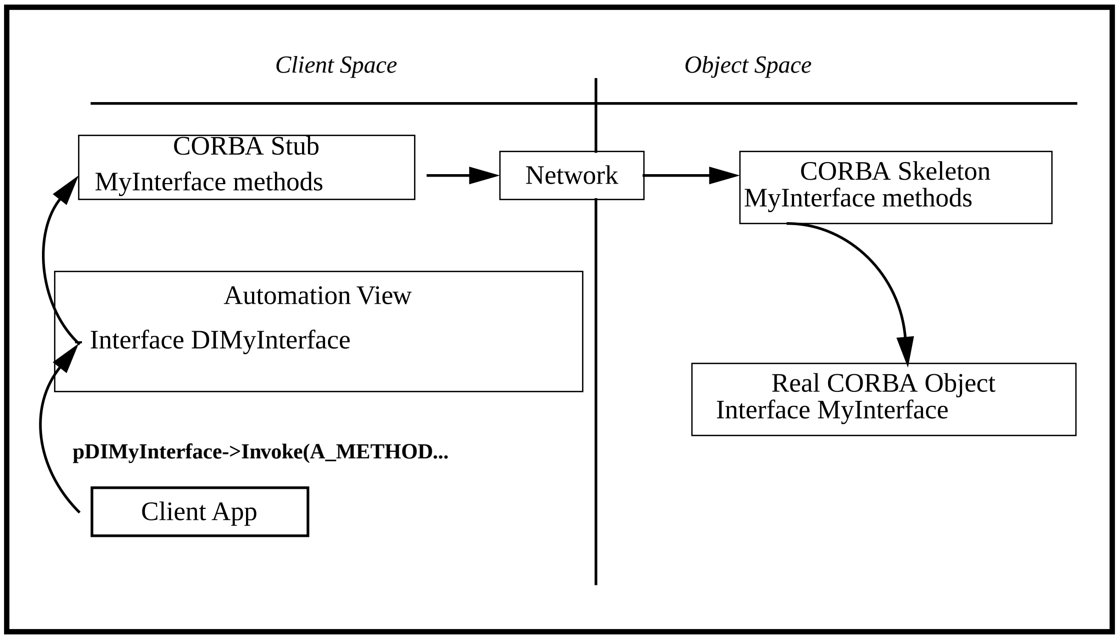


Figure 13-2 Methods of the Automation View Interface delegate to the CORBA Stub

13.1.2 Main Features of the Mapping

- OMG IDL attributes and operations map to Automation properties and methods respectively.
- OMG IDL interfaces map to Automation interfaces.
- The OMG IDL basic types map to corresponding basic types in Automation where possible. Since Automation supports a limited set of data types, some OMG IDL types cannot be mapped directly. Specifically:
  - OMG IDL constructed types such as structs and unions map to Automation interfaces with appropriate attributes and operations. User exceptions are mapped in the same way.
  - OMG IDL unsigned types map as closely as possible to Automation types, and overflow conditions are identified.
- OMG IDL sequences and arrays map to Automation Safearrays.

13.1.3 Mapping for Interfaces

A CORBA interface maps in a straightforward fashion to an Automation View Interface. For example, the following CORBA interface

```
module MyModule // OMG IDL
{
    interface MyInterface
    {
        // Attributes and operations;
        ...
    };
};
```

maps to the following Automation View Interface:

```
[odl, dual, uuid(...)]
   interface DIMyModule_MyInterface: IDispatch
   {
      // Properties and methods;
      ...
   };
```

The interface **`IMyModule_account`** is an OLE Automation Dual Interface. A Dual Interface is a COM vtable-based interface which derives from IDispatch, meaning that its methods can be late-bound via **`IDispatch::Invoke`** or early-bound through the vtable portion of the interface. Thus, **`IMyModule_account`** contains the methods of IDispatch as well as separate vtable-entries for its operations and property get/set methods.

Mapping for Attributes and Operations

An OMG IDL operation maps to an isomorphic Automation operation. An OMG IDL attribute maps to an ODL property, which has one method to *get* and one to *set* the value of the property. An OMG IDL readonly attribute maps to an OLE property, which has a single method to get the value of the property.

The order of the property and method declarations in the mapped Automation interface follows the rules described in “Ordering Rules for the CORBA->OLE Automation Transformation” part of Section 13.5.2, Detailed Mapping Rules, in Chapter 13A, Interworking Architecture.

For example, given the following CORBA interface,

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

the corresponding Automation View Interface is:

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

OMG IDL **`in`**, **`out`**, and **`inout`** parameters map to ODL **`[in]`**, **`[out]`**, and **`[in,out]`** parameters, respectively. Section 13.14, Mapping for Basic Data Types, explains the mapping for basic data types. The mapping for CORBA oneway operations is the same as for normal operations.

An operation of a Dual Interface always returns HRESULT, but the last argument in the operation's signature may be tagged **`[retval,out]`**. An argument tagged in this fashion is considered syntactically to be a return value. Automation controller macro languages map this special argument to a return value in their language syntax. Thus, a CORBA operation's return value is mapped to the last argument in the corresponding operation of the Automation View Interface.

Additional, Optional Parameter

All operations on the Automation View Interface have an optional **`out`** parameter of type VARIANT. The optional parameter returns explicit exception information in the context of each property set/get or method invocation. See Section 13.1.18, Mapping CORBA Exceptions to Automation Exceptions, for a detailed discussion of how this mechanism works.

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

Mapping for OMG IDL Single Inheritance

A hierarchy of singly-inherited OMG IDL interfaces maps to an identical hierarchy of Automation View Interfaces.

For example, given the interface account and its derived interface **`checkingAccount`** defined as follows,

```
module MyModule {          // OMG IDL
    interface account {
        attribute          float balance;
        readonly attributestring 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 ();
    };
};
```

the corresponding Automation View Interfaces are as follows.

```
// ODL
[odl, dual, uuid(20c31e22-dcb2-aa79-1dc4-34a4ad297579)]
interface DIMyModule_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 *
                  [IT_retval];
   [propput] HRESULT balance ([in] float balance);
   [propget] HRESULT owner ([retval,out] BSTR * IT_retval);
};

[odl, dual, uuid(ffe752b2-a73f-2a28-1de4-21754778ab4b)]
interface DIMyModule_checkingAccount: IMyModule_account {
      HRESULT orderChequeBook(
               [optional, out] VARIANT * excep_OBJ,
               [retval,out] short * IT_retval);
      [propget] HRESULT overdraftLimit (
               [retval,out] short * IT_retval);
};
```

Mapping of OMG IDL Multiple Inheritance

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

The mechanism for determining which interfaces appear on which strands is based on a left branch traversal of the inheritance tree. At points of multiple inheritance, the interface that is first in an ordering of the parent interfaces is included in what we call the main strand, and other interfaces are assigned to other, secondary strands. (The ordering of parent interfaces is explained later in this section.) For example, consider the CORBA interface hierarchy, shown in Figure 13-3.

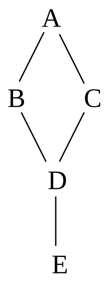


Figure 13-3 A CORBA Interface Hierarchy Using Multiple Inheritance

We read this hierarchy as follows:

- B and C derive from A
- D derives from B and C
- E derives from D

This CORBA hierarchy maps to the following two Automation single inheritance hierarchies, shown in Figure 13-4.

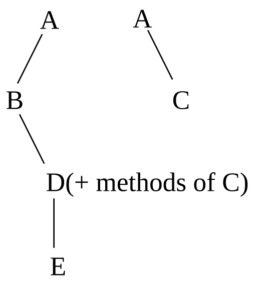


Figure 13-4 The Mapped Automation Hierarchy Splits at the Point of Multiple Inheritance

Consider the multiple inheritance point D, which inherits from B and C. Following the left strand B at this point, our main strand is A-B-D and our secondary strand is A-C. However, to access all of the object's methods, a controller would have to navigate among these disjoint strands via QueryInterface. While such navigation is expected of COM clients and might be an acceptable requirement of C++ automation controllers, many Automation controller environments do not support such navigation.

To accommodate such controllers, at points of multiple inheritance we aggregate the operations of the secondary strands into the interface of the main strand. In our example, we add the operations of C to D (A's operations are not added because they already exist in the main strand). Thus, D has all the methods of the hierarchy and, more important, an Automation controller holding a reference to D can access all of the methods of the hierarchy without calling QueryInterface.

In order to have a reliable, deterministic, portable way to determine the inheritance chain at points of multiple inheritance, an explicit ordering model must be used. Furthermore, to achieve interoperability of virtual function tables for dual interfaces, a precise model for ordering operations and attributes within an interface must be specified.

Within an interface, attributes should appear before operations and both should be ordered lexicographically by bytes in machine-collating sequence. For non-readonly attributes, the `[propget]` method immediately precedes the `[propput]` method. This ordering determines the position of the vtable portion of a Dual Interface. At points of multiple inheritance, the base interfaces should be ordered from left to right lexicographically by bytes in machine-collating order. (In all cases, the ordering is based on ISO Latin-1.) Thus, the leftmost branch at a point of multiple inheritance is the one ordered first among the base classes, not necessarily the one listed first in the inheritance declaration.

Continuing with the example, the following OMG IDL code expresses a hierarchy conforming to Figure 13-3.

```
// OMG IDL
module MyModule {
    interface A {
        void    aOp1();
        void    zOp1();
    };
    interface B: A{
        void    aOp2();
        void    zOp2();
    };
    interface C: A {
        void    aOp3();
        void    zOp3();
    };
    interface D: C, B{
        void    aOp4();
        void    zOp4();
    };
};
```

The OMG IDL maps to the following two Automation View hierarchies. Note that the ordering of the base interfaces for D has been changed based on our ISO Latin-1 alphabetic ordering model and that operations from C are added to interface D.

```
// ODL
// strand 1: A-B-D
[odl, dual, uuid(8db15b54-c647-553b-1dc9-6d098ec49328)]
interface DIMyModule_A: IDispatch {
   HRESULT  aOp1([optional,out] VARIANT * excep_OBJ);
   HRESULT  zOp1([optional,out] VARIANT * excep_OBJ);
}
[odl, dual, uuid(ef8943b0-cef8-21a5-1dc0-37261e082e51)]
interface DIMyModule_B: DIMyModule_A {
   HRESULT  aOp2([optional,out] VARIANT * excep_OBJ);
   HRESULT  zOp2([optional,out] VARIANT * excep_OBJ);
}
[odl, dual, uuid(67528a67-2cfd-e5e3-1de2-d59a444fe593)]
interface DIMyModule_D: DIMyModule_B {
   // C’s aggregated operations
   HRESULT  aOp3([optional,out] VARIANT * excep_OBJ);
   HRESULT  zOp3([optional,out] VARIANT * excep_OBJ);
   // D’s normal operations
   HRESULT  aOp4([optional,out] VARIANT * excep_OBJ);
   HRESULT  zOp4([optional,out] VARIANT * excep_OBJ);
}

// strand 2: A-C
[odl, dual, uuid(327885f8-ae9e-19c0-1dd5-d1ea05bcaae5)]
interface DIMyModule_C: DIMyModule_A {
   HRESULT  aOp3([optional,out] VARIANT * excep_OBJ);
   HRESULT  zOp3([optional,out] VARIANT * excep_OBJ);
}
```

Also note that the repeated operations of the aggregated strands are listed before D’s operations. The ordering of these operations obeys the rules for operations within C and is independent of the ordering within D.

13.1.4 Mapping for Basic Data Types

Basic Automation Types

Table 13-1 lists the basic data types supported by OLE Automation. The table contains fewer data types than those allowed by ODL because not all types recognized by ODL can be handled by the marshaling of IDispatch interfaces and by the implementation of **`ITypeInfo::Invoke`**. Arguments and return values of operations and properties are restricted to these basic types.

Table 13-1 OLE Automation Basic Types

Type	**Description**
boolean	True = -1, False = 0.
double	64-bit IEEE floating-point number.
float	32-bit IEEE floating-point number.
long	32-bit signed integer.
short	16-bit signed integer.
void	Allowed only as return type for a function, or in a function parameter list to indicate no parameters.
BSTR	Length-prefixed string. Prefix is an integer.
CURRENCY	8-byte fixed-point number.
DATE	64-bit floating-point fractional number of days since December 30, 1899.
SCODE	Built-in error type. In Win16, does not include additional data contained in an HRESULT. In Win32, identical to HRESULT.
IDispatch *	Pointer to IDispatch interface. From the viewpoint of the mapping, an IDispatch pointer parameter is an object reference.
IUnknown *	Pointer to IUnknown interface. (Any OLE interface can be represented by its IUnknown interface.)

The formal mapping of CORBA types to Automation types is shown in Table 13-2.

Table 13-2 OMG CORBA to OLE Automation Data Type Mappings

CORBA Type	**OLE Automation Type**
boolean	boolean
char	short
double	double
float	float
long	long
octet	short
short	short
unsigned long	long
unsigned short	long

13.1.5 *Special Cases of Basic Data Type Mapping*

An operation of an Automation View Interface must perform bidirectional translation of the Automation and CORBA parameters and return types. It must map from Automation to CORBA for **`in`** parameters and from CORBA to Automation for **`out`** parameters. The translation logic must handle the special conditions described in the following sections.

Translating Automation long to CORBA unsigned long

If the Automation long parameter is a negative number, then the View operation should return the HRESULT DISP_E_OVERFLOW.

Translating CORBA unsigned long to Automation long

If the **`CORBA::ULong`** parameter is greater than the maximum value of an Automation long, then the View operation should return the HRESULT DISP_E_OVERFLOW.

Translating Automation long to CORBA unsigned short

If the Automation long parameter is negative or is greater than the maximum value of a **`CORBA::UShort`**, then the View operation should return the HRESULT DISP_E_OVERFLOW.

Translating Automation boolean to CORBA boolean and CORBA boolean to Automation boolean

True and false values for CORBA boolean are, respectively, one (1) and zero (0). True and false values for Automation boolean are, respectively, negative one (-1) and zero (0). Therefore, true values need to be adjusted accordingly.

13.1.6 Mapping for Strings

An OMG IDL bounded or unbounded string maps to an OLE BSTR. For example, given the OMG IDL definitions,

```
// OMG IDL
string   sortCode<20>;
string   name;
```

the corresponding ODL code is:

```
// ODL
   BSTR  sortCode;
   BSTR  name;
```

On Win32 platforms, a BSTR maps to a Unicode string. The use of BSTR is the only support for internationalization of strings defined at this time.

13.1.7 A Complete IDL to ODL Mapping for the Basic Data Types

There is no requirement that the OMG IDL code expressing the mapped CORBA interface actually exists. Other equivalent expressions of CORBA interfaces, such as the contents of an Interface Repository, may be used. Moreover, there is no requirement that ODL code corresponding to the CORBA interface be generated.

However, OMG IDL is the appropriate medium for describing a CORBA interface and ODL is the appropriate medium for describing an Automation View Interface. Therefore, the following OMG IDL code describes a CORBA interface that exercises all of the CORBA base data types in the roles of attribute, operation **`in`** parameter, operation **`out`** parameter, operation **`inout`** parameter, and return value. The OMG IDL code is followed by ODL code describing the Automation View Interface that would result from a conformant mapping.

```
module MyModule // OMG IDL
{
    interface TypesTest
    {
        attribute boolean   boolTest;
        attribute char      charTest;
        attribute double    doubleTest;
        attribute float     floatTest;
        attribute long      longTest;
        attribute octet     octetTest;
        attribute short     shortTest;
        attribute string    stringTest;
        attribute string<10>stringnTest;
        attribute unsigned long ulongTest;
        attribute unsigned short ushortTest;

        readonly attribute short readonlyShortTest;

        // Sets all the attributes
        boolean setAll (
                    in boolean          boolTest,
                    in char             charTest,
                    in double           doubleTest,
                    in float            floatTest,
                    in long             longTest,
                    in octet            octetTest,
                    in short            shortTest,
                    in string           stringTest,
                    in string<10>       stringnTest,
                    in unsigned long    ulongTest,
                    in unsigned short   ushortTest);

        // Gets all the attributes
        boolean getAll (
                    out boolean         boolTest,
                    out char            charTest,
                    out double          doubleTest,
                    out float           floatTest,
                    out long            longTest,
                    out octet           octetTest,
                    out short           shortTest,
                    out string          stringTest,
                    out string<10>      stringnTest,
                    out unsigned long   ulongTest,
                    out unsigned short ushortTest);

        boolean setAndIncrement (
                    inout boolean       boolTest,
                    inout char          charTest,
                    inout double        doubleTest,
                    inout float         floatTest,
                    inout long          longTest,
                    inout octet         octetTest,
                    inout short         shortTest,
                    inout string        stringTest,
                    inout string<10>    stringnTest,
                    inout unsigned longulongTest,
                    inout unsigned shortushortTest);

        boolean         boolReturn ();
        char            charReturn ();
        double          doubleReturn();
        float           floatReturn();
        long            longReturn ();
        octet           octetReturn();
        short           shortReturn ();
        string          stringReturn();
        string<10>      stringnReturn();
        unsigned long ulongReturn ();
        unsigned shortushortReturn();

    }; // End of Interface TypesTest

}; // End of Module MyModule
```

The corresponding ODL code is as follows.

```
[odl, dual, uuid(180d4c5a-17d2-a1a8-1de1-82e7a9a4f93b)]
interface DIMyModule_TypesTest: IDispatch {
   HRESULT boolReturn ([optional,out] VARIANT * excep_OBJ,
               [retval,out] short *IT_retval);
   HRESULT charReturn ([optional,out] VARIANT * excep_OBJ,
               [retval,out] short *IT_retval);
   HRESULT doubleReturn ([optional,out] VARIANT * excep_OBJ,
               [retval,out] double *IT_retval);
   HRESULT floatReturn ([optional,out] VARIANT * excep_OBJ,
               [retval,out] float *IT_retval);
   HRESULT getAll ([out] short *boolTest,
               [out] short *charTest,
               [out] double *doubleTest,
               [out] float *floatTest,
               [out] long *longTest,
               [out] short *octetTest,
               [out] short *shortTest,
               [out] BSTR stringTest,
               [out] BSTR *stringnTest,
               [out] long *ulongTest,
               [out] long *ushortTest,
               [optional,out] VARIANT * excep_OBJ,
               [retval,out] short * IT_retval);
   HRESULT longReturn ([optional,out] VARIANT * excep_OBJ,
               [retval,out] long *IT_retval);
   HRESULT octetReturn ([optional,out] VARIANT * excep_OBJ,
               [retval,out] short *IT_retval);
   HRESULT setAll ([in] short boolTest,
               [in] short charTest,
               [in] double doubleTest,
               [in] float floatTest,
               [in] long longTest,
               [in] short octetTest,
               [in] short shortTest,
               [in] BSTR stringTest,
               [in] BSTR stringnTest,
               [in] long ulongTest,
               [in] long ushortTest,
               [optional,out] VARIANT * excep_OBJ,
               [retval,out] short * IT_retval);
   HRESULT setAndIncrement ([in,out] short *boolTest,
               [in,out] short *charTest,
               [in,out] double *doubleTest,
               [in,out] float *floatTest,
               [in,out] long *longTest,
               [in,out] short *octetTest,
               [in,out] short *shortTest,
               [in,out] BSTR *stringTest,
               [in,out] BSTR *stringnTest,
               [in,out] long *ulongTest,
               [in,out] long *ushortTest,
               [optional,out] VARIANT * excep_OBJ,
               [retval,out] short *IT_retval);
   HRESULT shortReturn ([optional,out] VARIANT * excep_OBJ,
               [retval,out] short *IT_retval);
   HRESULT stringReturn ([optional,out] VARIANT * excep_OBJ,
               [retval,out] BSTR *IT_retval);
   HRESULT stringnReturn ([optional,out] VARIANT *
                                          exep_OBJ,
               [retval,out] BSTR *IT_retval);
   HRESULT ulongReturn ([optional,out] VARIANT * excep_OBJ,
               [retval,out] long *IT_retval);
   HRESULT ushortReturn ([optional,out] VARIANT * excep_OBJ,
               [retval,out] long *IT_retval);
   [propget] HRESULT boolTest([retval,out] short *IT_retval);
   [propput] HRESULT boolTest([in] short boolTest);
   [propget] HRESULT charTest([retval,out] short *IT_retval);
   [propput] HRESULT charTest([in] short charTest);
   [propget] HRESULT doubleTest([retval,out] double
                     *IT_retval);
   [propput] HRESULT doubleTest([in] double doubleTest);
   [propget] HRESULT floatTest([retval,out] float
                     *IT_retval);
   [propput] HRESULT floatTest([in] float floatTest);
   [propget] HRESULT longTest([retval,out] long *IT_retval);
   [propput] HRESULT longTest([in] long longTest);
   [propget] HRESULT octetTest([retval,out] short
                     *IT_retval);
   [propput] HRESULT octetTest([in] short octetTest);
   [propget] HRESULT readonlyShortTest([retval,out] short
                     *IT_retval);
   [propget] HRESULT shortTest([retval,out] short
                     *IT_retval);
   [propput] HRESULT shortTest([in] short shortTest);
   [propget] HRESULT stringTest([retval,out] BSTR
                     *IT_retval);
   [propput] HRESULT stringTest([in] BSTR stringTest);
   [propget] HRESULT stringnTest([retval,out] BSTR
                     *IT_retval);
   [propput] HRESULT stringnTest([in] BSTR stringnTest);
   [propget] HRESULT ulongTest([retval,out] long *IT_retval);
   [propput] HRESULT ulongTest([in] long ulongTest);
   [propget] HRESULT ushortTest([retval,out] long
                     *IT_retval);
   [propput] HRESULT ushortTest([in] long ushortTest);
}
```

13.1.8 Mapping for Object References

Type Mapping

The mapping of an object reference as a parameter or return value can be fully expressed by the following OMG IDL and ODL code. The OMG IDL code defines an interface Simple and another interface that references Simple as an **`in`** parameter, as an **`out`** parameter, as an **`inout`** parameter, and as a return value. The ODL code describes the Automation View Interface that results from an accurate mapping.

```
module MyModule // OMG IDL
{
    // A simple object we can use for testing object references
    interface Simple
    {
        attribute short shortTest;
    };

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

}; // End of Module MyModule
```

The ODL code for the Automation View Dispatch Interface follows.

```
[odl, dual, uuid(c166a426-89d4-f515-1dfe-87b88727b4ea)]
interface DIMyModule_Simple: IDispatch
{
   [propget] HRESULT shortTest([retval, out] short *
                              IT_retval);
   [propput] HRESULT shortTest([in] short shortTest);
}

[odl, dual, uuid(04843769-120e-e003-1dfd-6b75107d01dd)]
interface DIMyModule_ObjRefTest: IDispatch
{
   HRESULT simpleOp([in]DIMyModule_Simple *inTest,
            [out] DIMyModule_Simple **outTest,
            [in,out] DIMyModule_Simple **inoutTest,
            [optional, out] VARIANT * excep_OBJ,
            [retval, out] DIMyModule_Simple ** IT_retval);

   [propget] HRESULT simpleTest([retval, out]
                              DIMyModule_Simple **
                              IT_retval);
   [propput] HRESULT simpleTest([in] DIMyModule_Simple
                                    *simpleTest);

}
```

Object Reference Parameters and IForeignObject

As described in Chapter 13A, Interworking Architecture, Automation and COM Views 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, which is passed as an **in** parameter to an operation M in View A. Operation M must somehow convert View B to a valid CORBA object reference.

In Figure 13-5, Automation Views expose IForeignObject, as required of all Views.

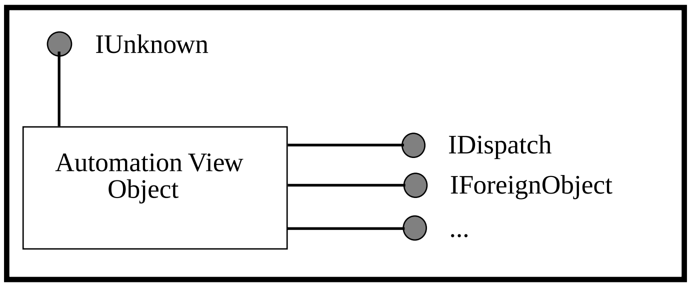


Figure 13-5 Partial Picture of the Automation View

The sequence of events involving **IForeignObject::GetForeignReference** is as follows:

- The client calls **Automation-View-A::M**, passing an IDispatch-derived pointer to Automation-View-B.
- Automation-View-A::M calls **IDispatch::QueryInterface** for IForeignObject.
- Automation-View-A::M calls **IForeignObject::GetForeignReference** to get the reference to the CORBA object of type B.
- Automation-View-A::M calls **CORBA-Stub-A::M** with the reference, narrowed to interface type B, as the object reference **in** parameter.

13.1.9 Mapping for Enumerated Types

CORBA enums map to Automation enums.

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

Consider the following example, which maps to the following ODL:

```
// ODL
typedef enum {red, green, blue} MyModule_color;

[odl,dual,uuid(7d1951f2-b5d3-8b7c-1dc3-aa0d5b3d6a2b)]
interface DIMyModule_foo: IDispatch {
   HRESULT op1([in] MyModule_color col, [optional,out]
                                  VARIANT * excep_OBJ);
}
```

Internally, OLE Automation maps enum parameters to the platform's integer type. (For Win32, the integer type is equivalent to a long.) If the number of elements in the CORBA enum exceeds the maximum value of an integer, the condition should be trapped at some point during static or dynamic construction of the Automation View Interface corresponding to the CORBA interface in which the enum type appears as a parameter. If the overflow is detected at run-time, the Automation View operation should return the HRESULT DISP_E_OVERFLOW.

If an actual parameter applied to the mapped parameter in the Automation View Interface exceeds the maximum value of the enum, the View operation should return the HRESULT DISP_E_OVERFLOW.

Since all Automation controllers do not promote the ODL definition of enums into the controller scripting language context, vendors may wish to generate a header file containing an appropriate enum declaration or a set of constant declarations for the client language. Since the method for doing so is an implementation detail, it is not specified here. However, it should be noted that some languages type enums other than as longs, introducing the possibility of conversion errors or faults. If such problems arise, it is best to use a series of constant declarations rather than an enumerated type declaration in the client header file.

For example, the following **enum** declaration

```
enum color {red, green, blue, yellow, white};// OMG IDL
```

could be translated to the following Visual Basic code:

```
' Visual Basic
Global const color_red = 0
Global const color_green = 1
Global const color_blue = 2
Global const color_yellow = 3
Global const color_white = 4
```

In this case the default naming rules for the enum values should follow those for interfaces. That is, the name should be fully scoped with the names of enclosing modules or interfaces. (See Section 13.7.7, Naming Conventions for View Components in Chapter 13A, Interworking Architecture.)

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

13.1.10 Mapping for Arrays and Sequences

OLE Automation methods may have array parameters called Safearrays. Safearrays are one or multidimensional arrays whose elements are of any of the basic Automation types. The following ODL syntax describes an array parameter:

```
SAFEARRAY (elementtype) arrayname
```

A Safearray may be passed by reference, using the following syntax:

```
SAFEARRAY (elementtype) *arrayname
```

Safearrays have a header which describes certain characteristics of the array including bounding information, and are thus relatively safe for marshaling. Note that the ODL declaration of Safearrays does not include bound specifiers. OLE provides an API for allocating and manipulating Safearrays, which includes a procedure for resizing the array.

IDL arrays and sequences, both bounded and unbounded, are mapped to Safearrays. Bounded sequences are mapped to Safearrays with the same boundaries; they do not grow dynamically up to the bounded size but are statically allocated to the bounded size. Unbounded sequences are mapped to Safearrays with some default bound. Attempts to access past the boundary result in a resizing of the Safearray.

Since ODL Safearray declarations contain no boundary specifiers, the bounding knowledge is contained in the Automation View. A method of the Automation View Interface, which has a Safearray as a parameter, has the intelligence to handle the parameter properly. When Safearrays are submitted as **`in`** parameters, the View method uses the Safearray API to dynamically repackage the Safearray as a CORBA array, bounded sequence, or unbounded sequence. When Safearrays are **`out`** parameters, the View method uses the Safearray API to dynamically repackage the CORBA array or sequence as a Safearray. When an unbounded sequence grows beyond the current boundary of the corresponding Safearray, the View's method uses the Safearray API to increase the size of the array by one allocation unit. The size of an allocation unit is unspecified. If a Safearray is mapped from a bounded sequence and a client of the View attempts to write to the Safearray past the maximum element of the bounded sequence, the View operation considers this a run-time error and returns the HRESULT DISP_E_OVERFLOW.

Multidimensional OMG IDL arrays map to multidimensional Safearrays. The order of dimensions in the OMG IDL array from left to right corresponds to ascending order of dimensions in the Safearray.

13.1.11 Mapping for CORBA Complex Types

CORBA constructed types—Structs, Unions and Exceptions—cannot be mapped directly to ODL constructed types, as Automation does not support them as valid parameter types. Instead, constructed types are mapped to Pseudo-Automation Interfaces. The objects that implement Pseudo-Automation Interfaces are called pseudo-objects. Pseudo-objects do not expose the IForeignObject interface.

Pseudo-Automation Interfaces are Dual Interfaces, but do not derive directly from IDispatch as do Automation View Interfaces. Instead, they derive from DIForeignComplexType:

```
// ODL
[odl, dual, uuid(...)]
interface DIForeignComplexType: IDispatch
{
   [propget] HRESULT INSTANCE_repositoryId([retval,out]
            BSTR *IT_retval);
   HRESULT INSTANCE_clone([in] IDispatch *pDispatch,
            [retval, out] IDispatch **IT_retval);
}
```

The UUID for DIForeignComplexType is:

```
{A8B553C0-3B72-11cf-BBFC-444553540000}
```

This interface can also be implemented as generic (nondual) Automation Interface, in which case it is named DForeignComplexType and its UUID is:

```
{E977F900-3B75-11cf-BBFC-444553540000}
```

The purpose of the **`DIForeignComplexType::INSTANCE_clone`** method is to provide the client programmer a way to duplicate a complex type. **`INSTANCE_clone`** creates a new instance of the type with values identical to the input instance. Therefore, **`INSTANCE_clone`** does not simply duplicate a reference to a complex type.

The purpose of the **`INSTANCE_repositoryId`** readonly property is to support the ability of DICORBAAny (see Section 13.1.13, Mapping for **`any`**s), when it wraps an instance of a complex type, to produce a type code for the instance when asked to do so via DICORBAAny's readonly typeCode property.

Mapping for Structure Types

CORBA structures are mapped to a Pseudo-Struct, which is an Pseudo-Automation Interface containing properties corresponding to the members of the struct. The names of a Pseudo-Struct's properties are identical to the names of the corresponding CORBA struct members.

A Pseudo-Struct derives from DICORBAStruct which, in turn, derives from DIForeignComplexType:

```
// ODL
[odl, dual, uuid(...)]
interface DICORBAStruct: DIForeignComplexType
{
}
```

The GUID for DICORBAStruct is:

`{A8B553C1-3B72-11cf-BBFC-444553540000}`

This interface can also be implemented as generic (nondual) Automation Interface, in which case it is named DCORBAStruct and its UUID is:

`{E977F901-3B75-11cf-BBFC-444553540000}`

The purpose of the methodless DICORBAStruct interface is to mark the interface as having its origin in the mapping of a CORBA struct. This information, which can be stored in a type library, is essential for the task of mapping the type back to CORBA in the event of an inverse mapping.

An example of mapping a CORBA struct to a Pseudo-Struct follows. The struct

```
struct S// IDL
{
    long l;
    double d;
    float f;
};
```

maps to Automation as follows, except that the mapped Automation Dual Interface derives from DICORBAStruct.

```
// IDL
interface S
{
    attribute long l;
    attribute double d;
    attribute float f;
};
```

Mapping for Union Types

CORBA unions are mapped to a Pseudo-Automation Interface called a Pseudo-Union. A Pseudo-Union contains properties that correspond to the members of the union, with the addition of a discriminator property. The discriminator property's name is **UNION_d**, and its type is the Automation type that corresponds to the OMG IDL union discriminant.

If a union element is accessed from the Pseudo-Union, and the current value of the discriminant does not match the property being requested, then the operation of the Pseudo-Union returns **DISP_E_TYPEMISMATCH**. Whenever an element is set, the discriminant's value is set to the value that corresponds to that element.

A Pseudo-Union derives from the methodless interface DICORBAUnion which, in turn, derives from DIForeignComplexType:

```
// ODL
[odl, dual, uuid(...)]
interface DICORBAUnion: DIForeignComplexType // ODL
{
}
```

The UUID for DICORBAUnion is:

```
{A8B553C2-3B72-11cf-BBFC-444553540000}
```

This interface can also be implemented as generic (nondual) Automation Interface, in which case it is named DCORBAUnion and its UUID is:

```
{E977F902-3B75-11cf-BBFC-444553540000}
```

An example of mapping a CORBA union to a Pseudo-Union follows. The union

```
interface A;                                    // IDL

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

maps to Automation as if it were defined as follows, except that the mapped Automation Dual Interface derives from DICORBAUnion.

```
interface A;                                    // IDL

interface U
{
    // Switch discriminant
    readonly attribute long UNION_d;

    attribute long l;
    attribute float f;
    attribute A obj;
};
```

13.1.12 Mapping for TypeCodes

The OMG IDL TypeCode data type maps to the DICORBATypeCode interface. The DICORBATypeCode interface is defined as follows.

```
// ODL
typedef enum {
   tk_null = 0, 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
   } CORBATCKind;

[odl, dual, uuid(...)]
interface DICORBATypeCode: DIForeignComplexType {
   [propget] HRESULT kind([retval,out] TCKind * IT_retval);

   // for 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] IDispatch ** IT_retval),

// tk_union
   HRESULT member_label([in] long index,[retval,out]
         VARIANT * IT_retval);
   [propget] HRESULT discriminator_type([retval,out]
         IDispatch ** 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_sequence, tk_array, tk_alias
   [propget] HRESULT content_type([retval,out]
         IDispatch ** IT_retval);
}
```

The UUID for DICORBATypeCode is:

{A8B553C3-3B72-11cf-BBFC-444553540000}

This interface can also be implemented as generic (nondual) Automation Interface, in which case it is named DCORBATypeCode and its UUID is:

{E977F903-3B75-11cf-BBFC-444553540000}

When generating Visual Basic constants corresponding to the values of the CORBATCKind enumeration, the constants should be declared as follows.

```
Global const CORBATCKind_tk_null =0
Global const CORBATCKind_tk_void = 1
. . .
```

Since DICORBATypeCode derives from DIForeignComplexType, objects which implement it are, in effect, pseudo-objects. See Section 13.1.11, Mapping for CORBA Complex Types, for a description of the DIForeignComplexType interface.

13.1.13 Mapping for ***any***s

The OMG IDL **any** data type maps to the DICORBAAny interface, which is declared as:

```
//ODL
[odl, dual, uuid(...)]
interface DICORBAAny: DIForeignComplexType
{
   [propget] HRESULT value([retval,out]
      VARIANT * IT_retval);
   [propput] HRESULT value([in] VARIANT val);
   [propget] HRESULT typeCode([retval,out]
      DICORBATypeCode ** IT_retval);
}
```

The UUID for DICORBAAny is:

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

This interface can also be implemented as generic (nondual) Automation Interface, in which case it is named DCORBAAny and its UUID is:

```
{E977F904-3B75-11cf-BBFC-444553540000}
```

Since DICORBAAny derives from DIForeignComplexType, objects that implement it are, in effect, pseudo-objects. See Section 13.1.11, Mapping for CORBA Complex Types, for a description of the DIForeignComplexType interface.

Note that the VARIANT value property of DICORBAAny can represent a Safearray or can represent a pointer to a DICORBAStruct or DICORBAUnion interface. Therefore, the mapping for **any** is valid for an **any** that represents a CORBA array, sequence, structure, or union.

13.1.14 Mapping for Typedefs

The mapping of OMG IDL **typedef** definitions to OLE depends on the OMG IDL type for which the **typedef** is defined. No mapping is provided for **typedef** definitions for the basic types: **float**, **double**, **long, short**, **unsigned long**, **unsigned short**, **char**, **boolean**, and **octet**. Hence, a Visual Basic programmer cannot make use of these **typedef** definitions.

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

For complex types, the mapping creates an alias for the pseudo-object. For interfaces, the mapping creates an alias for the Automation View object. A conforming implementation may register these aliases in the Windows System Registry.

Creating a View for this interface would require something like the following:

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

13.1.15 Mapping for Constants

The notion of a constant does not exist in OLE Automation. Therefore, no mapping is prescribed for a CORBA constant.

As with the mapping for enums, some vendors may wish to generate a header file containing an appropriate constant declaration for the client language. For example, the following OMG IDL declaration

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

could be translated to the following in Visual Basic:

```
' Visual Basic
   Global Const Max = 1000
```

The naming rules for these constants should follow that of enums.

13.1.16 Getting Initial CORBA Object References

The DICORBAFactory interface, described in Section 13.7.3, ICORBAFactory Interface in Chapter 13A, Interworking Architecture, provides a mechanism that is more suitable for the typical programmer in an Automation controller environment such as Visual Basic.

The implementation of the DICORBAFactory interface is not prescribed, but possible options include delegating to the OMG Naming Service and using the Windows System Registry[1].

The use of this interface from Visual Basic would appear as:

```
Dim theORBfactory as Object
Dim Target as Object
Set theORBfactory=CreateObject(“CORBA.Factory”)
Set Target=theORBfactory.GetObject
   (“software.sales.accounts”)
```

In Visual Basic 4.0 projects that have preloaded the standard CORBA Type Library, the code could appear as follows:

```
Dim Target as Object
Set Target=theORBfactory.GetObject(“soft-
ware.sales.accounts”)
```

The stringified name used to identify the desired target object should follow the rules for arguments to **`DICORBAFactory::GetObject`** described in Section 13.7.3, ICORBAFactory Interface in Chapter 13A, Interworking Architecture.

A special name space for names with a period in the first position can be used to resolve an initial reference to the OMG Object Services (for example, the Naming Service, the Life Cycle Service, and so forth). For example, a reference for the Naming Service can be found using:

```
Dim NameContext as Object
Set NameContext=theORBfactory.GetObject(“.NameService”)
```

Generally the GetObject method will be used to retrieve object references from the Registry/Naming Service. The CreateObject **`method`** is really just a shorthand notation for GetObject(“someName”).create. It is intended to be used for object references to objects supporting a CORBAServices Factory interface.

1. It is always permissible to directly register a CORBA/OLE Automation bridging object directly with the Windows Registry. The administration and assignment of ProgIds for direct registration should follow the naming rules described in Chapter 13A, Interworking Architecture.

13.1.17 Creating Initial in Parameters for Complex Types

Although CORBA complex types are represented by Automation Dual Interfaces, creating an instance of a mapped CORBA complex type is not the same as creating an instance of a mapped CORBA interface. The main difference lies in the fact that the name space for CORBA complex types differs fundamentally from the CORBA object and factory name spaces.

To support creation of instances of Automation objects exposing Pseudo-Automation Interfaces, we define a new interface, derived from DICORBAFactory (see Section 13.7.3, ICORBAFactory Interface in Chapter 13A, Interworking Architecture, for a description of DICORBAFactory).

```
// ODL
[odl, dual, uuid(...)]
interface DICORBAFactoryEx: DICORBAFactory
{
   HRESULT CreateType([in] IDispatch *scopingObject,
      [in] BSTR typeName,
                [retval,out] VARIANT *val);
   HRESULT CreateTypeById([in] IDispatch *scopingObject,
      [in] BSTR repositoryId,
               [retval,out] VARIANT *val);
}
```

The UUID for DICORBAFactoryEx is:

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

This interface can also be implemented as generic (nondual) Automation Interface, in which case it is named DCORBAFactoryEx and its UUID is:

```
{E977F905-3B75-11cf-BBFC-444553540000}
```

The Automation object having the ProgId “CORBA.Factory” shown next actually exposes DICORBAFactoryEx.

The CreateType method creates an Automation object that has been mapped from a CORBA complex type. The parameters are used to determine the specific type of object returned.

The first parameter, scopingObject, is a pointer to an Automation View Interface. The most derived interface type of the CORBA object bound to the View identifies the scope within which the second parameter, typeName, is interpreted. For example, assume the following CORBA interface exists:

```
// OMG IDL
module A {
    module B {
        interface C {
            struct S {
                // ...
            }
            void op(in S s);
        // ....
        }
    }
}
```

The following Visual Basic example illustrates the primary use of CreateType.

```
‘ Visual Basic
Dim myC as Object
Dim myS as Object
Dim myCORBAFactory as Object
Set myCORBAFactory = CreateObject(“CORBA.Factory”)
Set myC = myCORBAFactory.CreateObject( “...” )

‘ creates Automation View of the CORBA object
        supporting interface ‘ A::B::C
Set myS = myCORBAFactory.CreateType(myC, “S”)
myC.op(myS)
```

The following rules apply to CreateType.

- The typeName parameter can contain a fully-scoped name (i.e., the name begins with a double colon “::”). If so, then the first parameter defines the type name space within which the fully scoped name will be resolved.
- If the scopingObject parameter does not point to a valid Automation View Interface, then CreateObject returns the HRESULT DISP_E_UNKNOWNINTERFACE.
- If the typeName parameter does not identify a valid type in the name space associated with the scopingObject parameter, then CreateObject returns the HRESULT TYPE_E_UNDEFINEDTYPE.

The CreateTypeByID method accomplishes the same general goal of CreateType, the creation of Automation objects that are mapped from CORBA constructed types. The second parameter, repositoryID, is a string containing the CORBA Interface Repository ID of the CORBA type whose mapped Automation Object is to be created. The Interface Repository associated with the CORBA object identified by the scopingObject parameter defines the repository within which the ID will be resolved.

The following rules apply to CreateTypeById.

- If the scopingObject parameter does not point to a valid Automation View Interface, then CreateObject returns the HRESULT DISP_E_UNKNOWNINTERFACE.

- If the repositoryID parameter does not identify a valid type in the Interface Repository associated with the scopingObject parameter, then CreateObject returns the HRESULT TYPE_E_UNDEFINEDTYPE.

ITypeFactory Interface

The DICORBAFactory interface delegates its CreateType and CreateTypeByID methods to an ITypeFactory interface on the scoping object. ITypeFactory is defined as a COM interface because it is not intended to be exposed to Automation controllers. Every Automation View object must support the ITypeFactory interface:

```
//MIDL
interface ITypeFactory: IUnknown
{
   HRESULT CreateType([in] LPSTR typeName, [out] VARIANT
            *IT_retval);
   HRESULT CreateTypeById([in] RepositoryId repositoryID,
            [out] VARIANT *IT_retval);
}
```

The UUID for ITypeFactory is:

```
{A8B553C6-3B72-11cf-BBFC-444553540000}
```

The methods on ITypeFactory provide the behaviors previously described for the corresponding DICORBAFactory methods.

13.1.18 Mapping CORBA Exceptions to Automation Exceptions

Overview of Automation Exception Handling

Automation's notion of exceptions does not resemble true exception handling as defined in C++ and CORBA. Automation methods are invoked with a call to **`IDispatch::Invoke`** or to a vtable method on a Dual Interface. These methods return a 32-bit HRESULT, as do almost all COM methods. HRESULT values, which have the *severity* bit (bit 31 being the high bit) set, indicate that an error occurred during the call, and thus are considered to be error codes. (In Win16, an SCODE was defined as the lower 31 bits of an HRESULT, whereas in Win32 and for our purposes HRESULT and SCODE are identical.) HRESULTs also have a multibit field called the facility. One of the predefined values for this field is FACILITY_DISPATCH. Visual Basic 4.0 examines the return HRESULT. If the severity bit is set and the facility field has the value FACILITY_DISPATCH, then Visual Basic executes a built-in error handling routine, which pops up a message box and describes the error.

Invoke has among its parameters one of type EXCEPINFO*. The caller can choose to pass a pointer to an EXCEPINFO structure in this parameter or to pass NULL. If a non-NULL pointer is passed, the callee can choose to handle an error condition by returning the HRESULT DISP_E_EXCEPTION and by filling in the EXCEPINFO structure.

OLE also provides Error Objects, which are task local objects containing similar information to that contained in the EXCEPINFO structure. Error objects provide a way for Dual Interfaces to set detailed exception information.

Visual Basic allows the programmer to set up error traps, which are automatically fired when an invocation returns an HRESULT with the severity bit set. If the HRESULT is DISP_E_EXCEPTION, or if a Dual Interface has filled an Error Object, the data in the EXCEPINFO structure or in the Error Object can be extracted in the error handling routine.

CORBA Exceptions

CORBA exceptions provide data not directly supported by the Automation error handling model. Therefore, all methods of Automation View Interfaces have an additional, optional **out** parameter of type VARIANT which is filled in by the View when a CORBA exception is detected.

Both CORBA System exceptions and User exceptions map to Pseudo-Automation Interfaces called pseudo-exceptions. Pseudo-exceptions derive from IForeignException which, in turn, derives from IForeignComplexType:

```
//ODL
[odl, dual, uuid(...)]
interface DIForeignException: DIForeignComplexType
{
   [propget] HRESULT EX_majorCode([retval,out] long
            *IT_retval);
   [propget] HRESULT EX_repositoryID([retval,out] BSTR
            *IT_retval);
};
```

The UUID for DIForeignException is:

```
{A8B553C7-3B72-11cf-BBFC-444553540000}
```

This interface can also be implemented as generic (nondual) Automation Interface, in which case it is named **DForeignException** and its UUID is:

```
{E977F907-3B75-11cf-BBFC-444553540000}
```

The attribute EX_majorCode defines the broad category of exception raised, and has one of the following numeric values:

```
NO_EXCEPTION = 0
SYSTEM_EXCEPTION = 1
USER_EXCEPTION = 2
```

These values may be specified as an enum in the typelibrary information:

```
typedef enum {NO_EXCEPTION,
         SYSTEM_EXCEPTION,
         USER_EXCEPTION } CORBA_ExceptionType;
```

The attribute **EX_repositoryID** is a unique string that identifies the exception. It is the exception type’s repository ID from the CORBA Interface Repository.

CORBA User Exceptions

A CORBA user exception is mapped to a properties-only pseudo-exception whose properties correspond one-to-one with the attributes of the CORBA user exception, and which derives from the methodless interface DICORBAUserException:

```
//ODL
[odl, dual, uuid(...)]
interface DICORBAUserException: DIForeignException
{
}
```

The UUID for DICORBAUserException is:

```
{A8B553C8-3B72-11cf-BBFC-444553540000}
```

This interface can also be implemented as generic (nondual) Automation Interface, in which case it is named DCORBAUserException and its UUID is:

```
{E977F908-3B75-11cf-BBFC-444553540000}
```

Thus, an OMG IDL exception declaration is mapped to an OLE definition as though it were defined as an interface. The declaration

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

maps to the following ODL:

```
//ODL
[odl, dual, uuid(6bfaf02d-9f3b-1658-1dfb-7f056665a6bd)]
interface DIreject: DICORBAUserException
{
   [propget] HRESULT reason([retval,out] BSTR reason);
}
```

Operations that Raise User Exceptions

If the optional exception parameter is supplied by the caller and a User Exception occurs, the parameter is filled in with an IDispatch pointer to an exception Pseudo-Automation Interface, and the operation on the Pseudo-Interface returns the HRESULT S_FALSE. S_FALSE does not have the severity bit set, so that returning it from the operation prevents an active Visual Basic Error Trap from being fired, allowing the caller to retrieve the exception parameter in the context of the invoked method. The View fills in the VARIANT by setting its *vt* field to VT_DISPATCH and setting the **`pdispval`** field to point to the pseudo-exception. If no exception occurs, the optional parameter is filled with an IForeignException pointer on a pseudo-exception object whose **`EX_majorCode`** property is set to NO_EXCEPTION.

If the optional parameter is not supplied and an exception occurs, and

- If the operation was invoked via **`IDispatch::Invoke`**, then
  - The operation returns DISP_E_EXCEPTION.
  - If the caller provided an EXCEPINFO, then it is filled by the View.
- If the method was called via the vtable portion of a Dual Interface, then the OLE Error Object is filled by the View.

Note that in order to support Error Objects, Automation Views must implement the standard OLE interface ISupportErrorInfo.

Table 13-3 EXCEPINFO Usage for CORBA User Exceptions

Field	**Description**
wCode	Must be zero.
bstrSource	<interface name>.<operation name> *where the interface and operation names are those of the CORBA interface, which this Automation View is representing.*
bstrDescription	CORBA User Exception [<exception repository id>] *where the repository id is that of the CORBA user exception.*
bstrHelpFile	Unspecified
dwHelpContext	Unspecified
pfnDeferredFillIn	NULL
scode	DISP_E_EXCEPTION

Table 13-4 ErrorObject Usage for CORBA User Exceptions

Property	**Description**
bstrSource	<interface name>.<operation name> *where the interface and operation names are those of the CORBA interface, which this Automation View is representing.*
bstrDescription	CORBA User Exception: [<exception repository id>] *where the repository id is that of the CORBA user exception.*
bstrHelpFile	Unspecified
dwHelpContext	Unspecified
GUID	The IID of the Automation View Interface.

CORBA System Exceptions

A CORBA System Exception is mapped to the Pseudo-Exception DICORBASystemException, which derives from DIForeignException:

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

The UUID for DICORBASystemException is:

```
{1E5FFCA0-563B-11cf-B8FD-444553540000}
```

This interface can also be implemented as generic (nondual) Automation Interface, in which case it is named DCORBASystemException and its UUID is:

```
{1E5FFCA1-563B-11cf-B8FD-444553540000}
```

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

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

These values may be specified as an enum in the typelibrary information:

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

Operations that Raise System Exceptions

As is the case for UserExceptions, system exceptions can be returned to the caller using the optional last parameter, which is present on all mapped methods.

If the optional parameter is supplied and a system exception occurs, the optional parameter is filled in with an IForeignException pointer to the pseudo-exception, and the automation return value is S_FALSE. If no exception occurs, the optional parameter is filled with an IForeignException pointer whose **`EX_majorCode`** property is set to NO_EXCEPTION.

If the optional parameter is not supplied and a system exception occurs, the exception is looked up in Table 3-5. This table maps a subset of the CORBA system exceptions to semantically equivalent FACILITY_DISPATCH HRESULT values. If the exception is on the table, the equivalent HRESULT is returned. If the exception is not on the table, that is, if there is no semantically equivalent FACILITY_DISPATCH HRESULT, then the exception is mapped to an HRESULT according to Table 13-3 in Chapter 13B, Mapping: COM and CORBA. This new HRESULT is used as follows.

- If the operation was invoked via **`IDispatch::Invoke`**:
    - The operation returns DISP_E_EXCEPTION.
    - If the caller provided an EXCEPINFO, then it is filled with the scode field set to the new HRESULT value.
- If the method was called via the vtable portion of a Dual Interface:
    - The OLE Error Object is filled.
    - The method returns the new HRESULT.

Table 13-5 CORBA Exception to COM Error Codes

CORBA Exception	COM Error Codes
BAD_OPERATION	DISP_E_MEMBERNOTFOUND
NO_RESPONSE	DISP_E_PARAMNOTFOUND
BAD_INV_ORDER	DISP_E_BADINDEX
INV_IDENT	DISP_E_UNKNOWNNAME
INV_FLAG	DISP_E_PARAMNOTFOUND
DATA_CONVERSION	DISP_E_OVERFLOW

Table 13-6 EXCEPINFO Usage for CORBA System Exceptions

Field	**Description**
wCode	Must be zero.
bstrSource	<interface name>.<operation name> *where the interface and operation names are those of the CORBA interface, which this Automation View is representing.*
bstrDescription	CORBA System Exception: [<exception repository id>] minor code [<minor code>][<completion status>] *where the <exception repository id> and <minor code> are those of the CORBA system exception. <completion status> is "YES," "NO," or "MAYBE" based upon the value of the system exceptions's CORBA completion status. Spaces and square brackets are literals and must be included in the string.*
bstrHelpFile	Unspecified
dwHelpContext	Unspecified
pfnDeferredFillIn	NULL
scode	Mapped COM error code from Table 13-3 in Chapter 13B.

Table 13-7 ErrorObject Usage for CORBA System Exceptions

Property	**Description**
bstrSource	<interface name>.<operation name> *where the interface and operation names are those of the CORBA interface, which this Automation View is representing.*
bstrDescription	CORBA System Exception: [<exception repository id>] minor code [<minor code>][<completion status>] *where the <exception repository id> and <minor code> are those of the CORBA system exception. <completion status> is "YES," "NO," or "MAYBE" based upon the value of the system exceptions's CORBA completion status. Spaces and square brackets are literals and must be included in the string.*
bstrHelpFile	Unspecified
dwHelpContext	Unspecified
GUID	The IID of the Automation View Interface.

13.1.19 Conventions for Naming Components of the Automation View

The conventions for naming components of the Automation View are detailed in Section 13.7.7, Naming Conventions for View Components in Chapter 13A, Interworking Architecture.

13.1.20 Naming Conventions for Pseudo-Structs, Pseudo-Unions, and Pseudo-Exceptions

The formulas used to name components of the Automation View (see Section 13.7.7, Naming Conventions for View Components in Chapter 13A, Interworking Architecture) are also used to name components Pseudo-Structs, Pseudo-Unions, and Pseudo-Exceptions. The CORBA type name is used as input to the formulas, just as the CORBA interface name is used as input to the formulas when mapping interfaces.

These formulas apply to the name and IID of the Pseudo-Automation Interface, and to the Program Id and Class Id of an object implementing the Pseudo-Automation Interface if it is registered in the Windows System Registry.

13.1.21 Automation View Interface as a Dispatch Interface (Nondual)

In addition to implementing the Automation View Interface as an OLE Automation Dual Interface, it is also acceptable to map it as a generic Dispatch Interface.

In this case, the normal methods and attribute accessor/assign methods are not required to have HRESULT return values. Instead, an additional “dispinterface” is defined, which can use the standard OLE dispatcher to dispatch invocations.

For example, a method declared in a dual interface in ODL as follows:

```
HRESULT aMethod([in] <type> arg1, [out] <type> arg2,
   [retval, out] <return type> IT_retval)
```

would be declared in ODL in a dispatch interface in the following form:

```
<return type> aMethod([in] <type> arg1, [out] <type> arg2)
```

Using the example from Section 13.1.3, Mapping for Interfaces:

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

the corresponding Iaccount interfaces are defined as follows.

```
[odl, uuid(e268443e-43d9-3dab-1d7e-f303bbe9642f)]
interface Iaccount: IUnknown {   // ODL
      void makeLodgement ([in] float amount,
         [out] float balance,[out,optional]
            VARIANT *excep_OBJ);
      void makeWithdrawal([in] float amount,
         [out] float balance,[out,optional]
         VARIANT *excep_OBJ);
      [propget] float balance ([retval,out] *IT_retval);
      [propput] void balance ([in] float balance)
      [propget] BSTR owner ([retval,out] *IT_retval);
}
[uuid(e268443e-43d9-3dab-1dbe-f303bbe9642f)]
dispinterface Daccount {
   interface Iaccount;
};
```

A separate “dispinterface” declaration is required because Iaccount derives from IUnknown. The dispatch interface is DIaccount. Thus, in the example used for mapping object references in Section 13.1.8, Mapping for Object References, the reference to the Simple interface in the OMG IDL would map to a reference to **`IMyModule_Simple`** rather than **`DIMyModule_Simple`**. The naming conventions for Dispatch Interfaces (and for their IIDs) exposed by the View are slightly different from Dual Interfaces. See Section 13.7.7, Naming Conventions for View Components in Chapter 13A, Interworking Architecture, for details.

The Automation View Interface must correctly respond to a QueryInterface for the specific Dispatch Interface Id (DIID) for that View. By conforming to this requirement, the Automation View can be strongly type-checked. For example, **`ITypeInfo::Invoke`**, when handling a parameter that is typed as a pointer to a specific DIID, calls QueryInterface on the object for that DIID to make sure the object is of the required type.

Pseudo-Automation Interfaces representing CORBA complex types such as structs, unions, exceptions and the other noninterface constructs mapped to dispatch interfaces can also be exposed as nondual dispatch interfaces.

13.1.22 *Aggregation of Automation Views*

COM’s implementation reuse mechanism is aggregation. Automation View objects must either be capable of being aggregated in the standard COM fashion or must follow COM rules to indicate their inability or unwillingness to be aggregated.

The same rule applies to pseudo-objects.

13.1.23 *DII, DSI, and BOA*

OLE Automation interfaces are inherently self-describing and may be invoked dynamically. There is no utility in providing a mapping of the DII interfaces and related pseudo-objects into OLE Automation interfaces.

13.2 Mapping OLE Automation Objects as CORBA Objects

This problem is the reverse of exposing CORBA objects as Automation objects. It is best to solve this problem in a manner similar to the approach for exposing CORBA objects as Automation objects.

13.2.1 Architectural Overview

We begin with ODL or type information for an Automation object, which implements one or more dispatch interfaces and whose server application exposes a class factory for its COM class.

We then create a CORBA View object, which provides skeletal implementations of the operations of each of those interfaces. The CORBA View object is in every way a legal CORBA object. It is not an Automation object. The skeleton is placed on the machine where the real Automation object lives.

The CORBA View is not fully analogous to the Automation View which, as previously explained, is used to represent a CORBA object as an Automation object. The Automation View has to reside on the client side because COM is not distributable. A copy of the Automation View needs to be available on every client machine.

The CORBA View, however, can live in the real CORBA object's space and can be represented on the client side by the CORBA system's stub because CORBA is distributable. Thus, only one copy of this View is required.

Note – Throughout this section, the term *CORBA View* is distinct from CORBA stubs and skeletons, from COM proxies and stubs, and from Automation Views.

The CORBA View is an Automation client. Its implementations of the CORBA operations translate parameter types and delegate to the corresponding methods of the real Automation object. When a CORBA client wishes to instantiate the real Automation object, it instantiates the CORBA View.

Thus, from the point of view of the client, it is interacting with a CORBA object which may be a remote object. CORBA handles all of the interprocess communication and marshaling. No COM proxies or stubs are created.

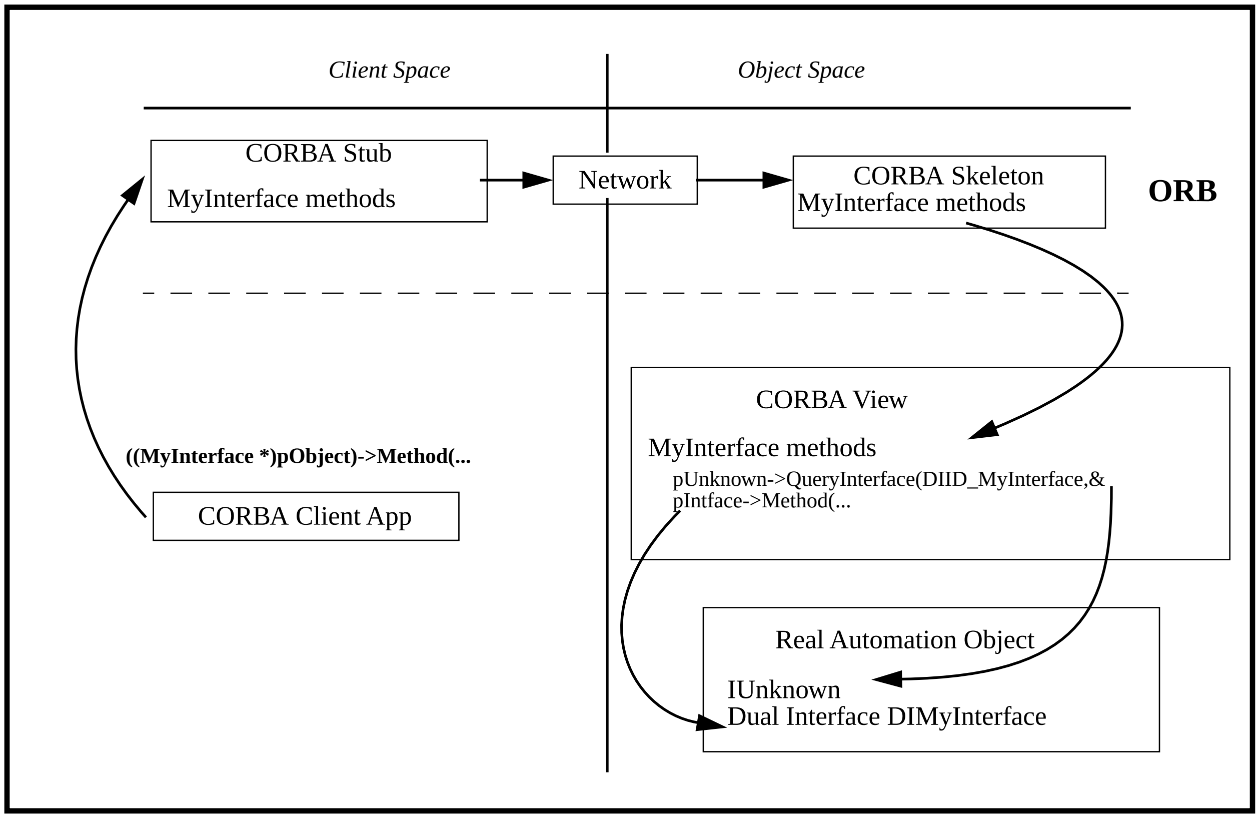


Figure 13-6 The CORBA View: a CORBA Object, which is a Client of a COM Object

13.2.2 Main Features of the Mapping

- ODL or type library information can form the input for the mapping.
- Automation properties and methods map to OMG IDL attributes and operations, respectively.
- Automation interfaces map to OMG IDL interfaces.
- Automation basic types map to corresponding OMG IDL basic types where possible.
- Automation errors are mapped similarly to COM errors.

13.2.3 Getting Initial Object References

The OMG Naming Service can be used to get initial references to the CORBA View Interfaces. These interfaces may be registered as normal CORBA objects on the remote machine.

13.2.4 Mapping for Interfaces

The mapping for an ODL interface to a CORBA View interface is straightforward. Each interface maps to an OMG IDL interface. In general, we map all methods and properties with the exception of the IUnknown and IDispatch methods.

For example, given the ODL interface **`IMyModule_account`**,

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

the following is the OMG IDL equivalent:

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

If the ODL interface does not have a parameter with the **`[retval,out]`** attributes, its return type is mapped to long. This allows COM SCODE values to be passed through to the CORBA client.

13.2.5 Mapping for Inheritance

A hierarchy of Automation interfaces is mapped to an identical hierarchy of CORBA View Interfaces.

For example, given the interface “account” and its derived interface “checkingAccount” defined next,

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

the corresponding CORBA View Interfaces are:

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

13.2.6 Mapping for ODL Properties and Methods

An ODL property which has either a get/set pair or just a set method is mapped to an OMG IDL attribute. An ODL property with just a get accessor is mapped to an OMG IDL **`readonly`** attribute.

Given the ODL interface definition

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

the corresponding OMG IDL interface is:

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

ODL **`[in]`**, **`[out]`**, and **`[in,out]`** parameters map to OMG IDL **`in`**, **`out`**, and **`inout`** parameters, respectively. Section 13.1.4, Mapping for Basic Data Types, explains the mapping for basic types.

13.2.7 *Mapping for Automation Basic Data Types*

Basic Automation Types

The basic data types allowed by OLE Automation as parameters and return values are detailed in Section 13.1.4, Mapping for Basic Data Types.

The formal mapping of CORBA types to Automation types is shown in Table 13-8.

Table 13-8 Mapping of Automation Types to OMG IDL Types

OLE Automation Type	**OMG IDL Type**
boolean	boolean
short	short
double	double
float	float
long	long
BSTR	string
CURRENCY	COM::Currency
DATE	double
SCODE	long

The Automation CURRENCY type is a 64-bit integer scaled by 10,000, giving a fixed point number with 15 digits left of the decimal point and 4 digits to the right. The **`COM::Currency`** type is thus defined as follows:

```
module COM
{
   struct Currency
   {
      unsigned long lower;
      long upper;
   }
}
```

This mapping of the CURRENCY type is transitional and should be revised when the extended data types revisions to OMG IDL are adopted. These revisions are slated to include a 64-bit integer.

The Automation DATE type is an IEEE 64-bit floating-point number representing the number of days since December 30, 1899.

13.2.8 Conversion Errors

An operation of a CORBA View Interface must perform bidirectional translation of the Automation and CORBA parameters and return types. It must map from CORBA to Automation for **`in`** parameters and from Automation to CORBA for **`out`** parameters.

When the CORBA View encounters an error condition while translating between CORBA and Automation data types, it raises the CORBA system exception DATA_CONVERSION.

13.2.9 Special Cases of Data Type Conversion

Translating COM::Currency to Automation CURRENCY

If the supplied **`COM::Currency`** value does not translate to a meaningful Automation CURRENCY value, then the CORBA View should raise the CORBA System Exception DATA_CONVERSION.

Translating CORBA double to Automation DATE

If the CORBA double value is negative or converts to an impossible date, then the CORBA View should raise the CORBA System Exception DATA_CONVERSION.

Translating CORBA boolean to Automation boolean and Automation boolean to CORBA boolean

True and false values for CORBA boolean are, respectively, one and zero. True and false values for Automation boolean are, respectively, negative one (-1) and zero. Therefore, true values need to be adjusted accordingly.

13.2.10 A Complete OMG IDL to ODL Mapping for the Basic Data Types

As previously stated, there is no requirement that the ODL code expressing the mapped Automation interface actually exist. Other equivalent expressions of Automation interfaces, such as the contents of a Type Library, may be used. Moreover, there is no requirement that OMG IDL code corresponding to the CORBA View Interface be generated.

However, ODL is the appropriate medium for describing an Automation interface, and OMG IDL is the appropriate medium for describing a CORBA View Interface. Therefore, we provide the following ODL code to describe an Automation interface, which exercises all of the Automation base data types in the roles of properties, method **`[in]`** parameter, method **`[out]`** parameter, method **`[inout]`** parameter, and return value. The ODL code is followed by OMG IDL code describing the CORBA View Interface, which would result from a conformant mapping.

```
// ODL
[odl, dual, uuid(...)]
interface DIMyModule_TypesTest: IForeignObject {
   [propput] HRESULT boolTest([in] short boolTest);
   [propget] HRESULT boolTest([retval,out] short *IT_retval);
   [propput] HRESULT doubleTest([in] double doubleTest);
   [propget] HRESULT doubleTest([retval,out] double
      *IT_retval);
   [propput] HRESULT floatTest([in] float floatTest);
   [propget] HRESULT floatTest([retval,out] float
      *IT_retval);
   [propput] HRESULT longTest([in] long longTest);
   [propget] HRESULT longTest([retval,out] long *IT_retval);
   [propput] HRESULT shortTest([in] short shortTest);
   [propget] HRESULT shortTest([retval,out] short
      *IT_retval);
   [propput] HRESULT stringTest([in] BSTR stringTest);
   [propget] HRESULT stringTest([retval,out] BSTR
      *IT_retval);
   [propput] HRESULT dateTest([in] DATE stringTest);
   [propget] HRESULT dateTest([retval,out] DATE *IT_retval);
   [propput] HRESULT currencyTest([in] CURRENCY stringTest);
   [propget] HRESULT currencyTest([retval,out] CURRENCY
      *IT_retval);
   [propget] HRESULT readonlyShortTest([retval,out] short
      *IT_retval);
   HRESULT setAll ([in] short boolTest,
            [in] double doubleTest,
            [in] float floatTest,
            [in] long longTest,
            [in] short shortTest,
            [in] BSTR stringTest,
            [in] DATE dateTest,
            [in] CURRENCY currencyTest,
            [retval,out] short * IT_retval);
   HRESULT getAll ([out] short *boolTest,
            [out] double *doubleTest,
            [out] float *floatTest,
            [out] long *longTest,
            [out] short *shortTest,
            [out] BSTR stringTest,
            [out] DATE * dateTest,
            [out] CURRENCY *currencyTest,
            [retval,out] short * IT_retval);
  HRESULT setAndIncrement ([in,out] short *boolTest,
            [in,out] double *doubleTest,
            [in,out] float *floatTest,
            [in,out] long *longTest,
            [in,out] short *shortTest,
            [in,out] BSTR *stringTest,
            [in,out] DATE * dateTest,
         [in,out] CURRENCY * currencyTest,
         [retval,out] short *IT_retval);
HRESULT boolReturn ([retval,out] short *IT_retval);
HRESULT doubleReturn ([retval,out] double *IT_retval);
HRESULT floatReturn ([retval,out] float *IT_retval);
HRESULT longReturn ([retval,out] long *IT_retval);
HRESULT shortReturn ([retval,out] short *IT_retval);
HRESULT stringReturn ([retval,out] BSTR *IT_retval);
HRESULT octetReturn ([retval,out] DATE *IT_retval);
HRESULT currencyReturn ([retval,out] CURRENCY
   *IT_retval);
}
```

The corresponding OMG IDL is as follows.

```
// OMG IDL
interface MyModule_TypesTest
{
    attribute boolean   boolTest;
    attribute double    doubleTest;
    attribute float     floatTest;
    attribute long      longTest;
    attribute short     shortTest;
    attribute string    stringTest;
    attribute double    dateTest;
    attribute COM::Currency currencyTest;

    readonly attribute short readonlyShortTest;

    // Sets all the attributes
    boolean setAll (in boolean      boolTest,
                    in double       doubleTest,
                    in float        floatTest,
                    in long         longTest,
                    in short        shortTest,
                    in string       stringTest,
                    in double       dateTest,
                    in COM::Currency currencyTest);

    // Gets all the attributes
    boolean getAll (out boolean     boolTest,
                    out double      doubleTest,
                    out float       floatTest,
                    out long        longTest,
                    out short       shortTest,
                    out string      stringTest,
                    out double      dateTest,
                    out COM::Currency currencyTest);

    boolean setAndIncrement (
                    inout boolean   boolTest,
                    inout double    doubleTest,
                    inout float     floatTest,
                    inout long      longTest,
                    inout short     shortTest,
                    inout string    stringTest,
                    inout double    dateTest,
                    inout COM::Currency currencyTest);

    boolean         boolReturn ();
    double          doubleReturn();
    float           floatReturn();
    long            longReturn ();
    short           shortReturn ();
    string          stringReturn();
    double          dateReturn ();
```

```
   COM::CurrencycurrencyReturn();

}; // End of Interface TypesTest
```

13.2.11 Mapping for Object References

The mapping of an object reference as a parameter or return value can be fully expressed by the following OMG IDL and ODL code. The ODL code defines an interface “Simple” and another interface that references Simple as an **in** parameter, an **out** parameter, an **inout** parameter, and as a return value. The OMG IDL code describes the CORBA View Interface that results from a proper mapping.

```
// ODL
[odl, dual, uuid(...)]
interface DIMyModule_Simple: IDispatch
{
   [propget] HRESULT shortTest([retval, out]
      short * IT_retval);
   [propput] HRESULT shortTest([in] short sshortTest);
}

[odl, dual, uuid(...)]
interface DIMyModule_ObjRefTest: IDispatch
{
   [propget] HRESULT simpleTest([retval, out]
      DIMyModule_Simple ** IT_retval);
   [propput] HRESULT simpleTest([in] DIMyModule_Simple
      *pSimpleTest);

   HRESULT simpleOp([in] DIMyModule_Simple *inTest,
               [out] DIMyModule_Simple **outTest,
               [in,out]DIMyModule_Simple **inoutTest,
               [retval, out] DIMyModule_Simple **IT_retval);
}
```

The OMG IDL code for the CORBA View Dispatch Interface is as follows.

```
// OMG IDL
// A simple object we can use for testing object references
interface MyModule_Simple
{
    attribute short shortTest;
};

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

13.2.12 Mapping for Enumerated Types

ODL enumerated types are mapped to OMG IDL enums; for example:

```
// ODL
typedef enum MyModule_color {red, green, blue};

[odl,dual,uuid(...)]
interface DIMyModule_foo: IDispatch {
   HRESULT op1([in] MyModule_color col);
}

// OMG IDL
module COM {
    enum MyModule_color {red, green, blue};
    interfacefoo: COM::CORBA_View {
        long op1(in MyModule_color col);
    };
};
```

13.2.13 Mapping for SafeArrays

Automation SafeArrays should be mapped to CORBA unbounded sequences.

A method of the CORBA View Interface, which has a SafeArray as a parameter, will have the knowledge to handle the parameter properly.

When SafeArrays are **`in`** parameters, the View method uses the Safearray API to dynamically repackage the SafeArray as a CORBA sequence. When arrays are **`out`** parameters, the View method uses the Safearray API to dynamically repackage the CORBA sequence as a SafeArray.

Multidimensional SafeArrays

SafeArrays are allowed to have more than one dimension. However, the bounding information for each dimension, and indeed the number of dimensions, is not available in the static typelibrary information or ODL definition. It is only available at run-time.

For this reason, SafeArrays, which have more than one dimension, are mapped to an identical linear format and then to a sequence in the normal way.

This linearization of the multidimensional SafeArray should be carried out as follows:

- The number of elements in the linear sequence is the product of the dimensions.
- The position of each element is deterministic; for a SafeArray with dimensions d0, d1, d2, the location of an element [p0][p1][p2] is defined as:

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

Consider the following example: SafeArray with dimensions 5, 8, 9.

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

13.2.14 Mapping for Typedefs

ODL typedefs map directly to OMG IDL typedefs. The only exception to this is the case of an ODL enum, which is required to be a typedef. In this case the mapping is as per Section 13.1.9, Mapping for Enumerated Types.

13.2.15 Mapping for VARIANTs

The VARIANT data type maps to a CORBA **any**. If the VARIANT contains a DATE or CURRENCY element, these are mapped as per Section 13.2.7, Mapping for Automation Basic Data Types.

13.2.16 Mapping Automation Exceptions to CORBA

There are several ways in which an HRESULT (or SCODE) can be obtained by an Automation client such as the CORBA View. These ways differ based on the signature of the method and the behavior of the server. For example, for vtable invocations on dual interfaces, the HRESULT is the return value of the method. For **`IDispatch::Invoke`** invocations, the significant HRESULT may be the return value from Invoke, or may be in the EXCEPINFO parameter's SCODE field.

Regardless of how the HRESULT is obtained by the CORBA View, the mapping of the HRESULT is the exactly the same as for COM to CORBA (see Mapping for COM Errors under Section 13.3.10 in Chapter 13B, Mapping: COM and CORBA). The View raises either a standard CORBA system exception or the COM_HRESULT user exception.

CORBA Views must supply an EXCEPINFO parameter when making **IDispatch::Invoke** invocations to take advantage of servers using EXCEPINFO. Servers do not use the EXCEPINFO parameter if it is passed to Invoke as NULL.

An Automation method with an HRESULT return value and an argument marked as a **[retval]** maps to an IDL method whose return value is mapped from the **[retval]** argument. This situation is common in dual interfaces and means that there is no HRESULT available to the CORBA client. It would seem on the face of it that there is a problem mapping S_FALSE scodes in this case because the fact that no system exception was generated means that the HRESULT on the vtable method could have been either S_OK or S_FALSE. However, this should not truly be a problem. A method in a dual interface should never attach semantic meaning to the distinction between S_OK and S_FALSE because a Visual Basic program acting as a client would never be able to determine whether the return value from the actual method was S_OK or S_FALSE.

An Automation method with an HRESULT return value and no argument marked as **[retval]** maps to a CORBA interface with a long return value. The long HRESULT returned by the original Automation operation is passed back as the long return value from the CORBA operation.

OMG IDL Tags B

This appendix lists the standardized profile, service, and component tags described in the Interoperability chapters. Implementor-defined tags can also be registered in this manual. Requests to register tags with the OMG should be sent to **tag_request@omg.org.**

TBL. 16Standard Profile Tags

Tag Name	Tag Value	Described in
ProfileId	TAG_INTERNET_IOP = 0	Section 10.6.2, “Interoperable Object References: IORs,” on page 10-14
ProfileId	TAG_MULTIPLE_COMPONENTS = 1	Section 10.6.2, “Interoperable Object References: IORs,” on page 10-14

TBL. 17Standard Service Tags

Tag Name	Tag Value	Described in
ServiceId	TransactionService = 0	Section 10.6.6, “Object Service Context,” on page 10-18

TBL. 18Standard Component Tags

Tag Name	Tag Value	Described in
ComponentId	TAG_DCE_STRING_BINDING = 100	Section 13.5.1, “DCE-CIOP String Binding Component,” on page 13-16
ComponentId	TAG_DCE_BINDING = 101	Section 13.5.2, “DCE-CIOP Binding Name Component,” on page 13-17
ComponentId	TAG_DCE_NO_PIPES = 102	Section 13.5.3, “DCE-CIOP No Pipes Component,” on page 13-18
ComponentId	TAG_OBJECT_KEY = 10	Section 13.5.4, “Object Key Component,” on page 13-19
ComponentId	TAG_ENDPOINT_ID = 11	Section 13.5.5, “Endpoint ID Component,” on page 13-19
ComponentId	TAG_LOCATION_POLICY = 12	Section 13.5.6, “Location Policy Component,” on page 13-20 and Section 13.6.3, “Basic Location Algorithm,” on page 13-22

Sample Solutions for Older OLE Automation Controllers C

This appendix provides some solutions that vendors might implement to support existing and older OLE Automation controllers. These solutions are suggestions; they are strictly optional.

C.1 Mapping for OMG IDL Arrays and Sequences to Collections

Some OLE Automation controllers do not support the use of SAFEARRAYs. For this reason, arrays and sequences can also be mapped to OLE collection objects.

A collection object allows generic iteration over its elements. While there is no explicit ICollection type interface, OLE does specify guidelines on the properties and methods a collection interface should export.

```
// ODL
[odl, dual, uuid(...)]
interface DICollection: IDispatch {
   [propget] HRESULT Count([retval,out] long * count);
   [propget, id(DISPID_VALUE)] HRESULT Item([in] long index,
               [retval,out] VARIANT * retval);
   [propput, id(DISPID_VALUE)] HRESULT Item([in] long index,
    [in] VARIANT val);
   [propget, id(NEW_ENUM)] HRESULT _NewEnum(
            [retval, out] IEnumVARIANT * newEnum);
}
```

The UUID for DICollection is:

```
{A8B553C9-3B72-11cf-BBFC-444553540000}
```

This interface can also be implemented as generic (nondual) Automation Interface, in which case it is named DCollection and its UUID is:

```
{E977F909-3B75-11cf-BBFC-444553540000}
```

In controller scripting languages such as VBA in MS-Excel, the FOR...EACH language construct can automatically iterate over a collection object such as that previously described.

```
' Visual Basic:
Dim doc as Object
For Each doc in DocumentCollection
doc.Visible = False
Next doc
```

The specification of DISPID_VALUE as the id() for the Item property means that access code like the following is possible.

```
' Visual Basic:
Dim docs as Object
Set docs = SomeCollection

docs(4).Visible = False
```

Multidimensional arrays can be mapped to collections of collections with access code similar to the following.

```
' Visual Basic
Set docs = SomeCollection

docs.Item(4).Item(5).Visible = False
```

If the Collection mapping for OMG IDL Arrays and Sequences is chosen, then the signatures for operations accepting SAFEARRAYs should be modified to accept a VARIANT instead. In addition, the implementation code for the View wrapper method should detect the kind of object being passed.

Example Mappings D

D.1 Mapping the OMG Naming Service to OLE Automation

This section provides an example of how a standard OMG Object Service, the Naming Service, would be mapped according to the Interworking specification.

The Naming Service provides a standard service for CORBA applications to obtain object references. The reference for the Naming Service is found by using the **`resolve_initial_references()`** method provided on the ORB pseudo-interface:

```
CORBA::ORB_ptr theORB = CORBA::ORB_init(argc, argv,
CORBA::ORBid, ev)
CORBA::Object_var obj =
    theORB->resolve_initial_references(“NameService”, ev);
CosNaming::NamingContext_var inital_nc_ref =
    CosNaming::NamingContext::_narrow(obj,ev);
CosNaming::Name factory_name;
factory_name.length(1);
factory_name[0].id = “myFactory”;
factory_name[0].kind = ““;
CORBA::Object_var objref = initial_nc_ref->resolve(factory_name, ev);
```

The Naming Service interface can be directly mapped to an equivalent OLE Automation interface using the mapping rules contained in the rest of this section. A direct mapping would result in code from VisualBasic that appears as follows.

```
Dim CORBA as Object
Dim ORB as Object
Dim NamingContext as Object
Dim NameSequence as Object
Dim Target as Object

Set CORBA=GetObject(“CORBA.ORB”)
Set ORB=CORBA.init(“default”)
Set NamingContext = ORB.resolve_initial_reference(“Naming-
Service”)
Set NameSequence=NamingContext.create_type(“Name”)
ReDim NameSequence as Object(1)
NameSequence[0].name = “myFactory”
NameSequence[0].kind = ““
Set Target=NamingContext.resolve(NameSequence)
```

D.2 Mapping a COM Service to OMG IDL

This section provides an example of mapping a Microsoft IDL-described set of interfaces to an equivalent set of OMG IDL-described interfaces. The interface is mapped according to the rules provided in Section 13.3, COM to CORBA Data Type Mapping in Chapter 13B. The example chosen is the COM ConnectionPoint set of interfaces. The ConnectionPoint service is commonly used for supporting event notification in OLE custom controls (OCXs). The service is a more general version of the IDataObject/IAdviseSink interfaces.

The ConnectionPoint service is defined by four interfaces, described in Table D-1.

Table D-1 Interfaces of the ConnectionPoint Service

IConnectionPointContainer	Used by a client to acquire a reference to one or more of an object’s notification interfaces
IConnectionPoint	Used to establish and maintain notification connections
IEnumConnectionPoints	An iterator over a set of IConnectionPoint references
IEnumConnections	Used to iterate over the connections currently associated with a ConnectionPoint

For purposes of this example, we describe these interfaces in Microsoft IDL. The IConnectionPointContainer interface is shown next.

```
// Microsoft IDL
interface IConnectionPoint;
interface IEnumConnectionPoints;
typedef struct {
 unsigned long Data1;
 unsigned short Data2;
 unsigned short Data3;
 unsigned char Data4[8];
} REFIID;
[object, uuid(B196B284-BAB4-101A-B69C-00AA00241D07),
      pointer_default(unique)]
interface IConnectionPointContainer: IUnknown
{
HRESULT EnumConnectionPoints ([out] IEnumConnectionPoints
      **pEnum);
HRESULT FindConnectionPoint([in] REFIID iid, [out]
      IConnectionPoint **cp);
};
MIDL definition for IConnectionPointContainer
```

This IConnectionPointContainer interface would correspond to the OMG IDL interface shown next.

```
// OMG IDL
    interface IConnectionPoint;
    interface IEnumConnectionPoints;
    struct REFIID {
    unsigned long Data1;
    unsigned short Data2;
    unsigned short Data3;
    unsigned char Data4[8];
    };
    interface IConnectionPointContainer: CORBA::Composite,
    CosLifeCycle::LifeCycleObject
{
    HRESULT EnumConnectionPoints (out IEnumConnectionPoints
                pEnum) raises (COM_HRESULT);
    HRESULT FindConnectionPoint(in REFIID iid, out
                IConnectionPoint cp) raises (COM_HRESULT);
#pragma ID IConnectionPointContainer =“DCE:B196B284-BAB4-
    101A-B69C-00AA00241D07”;
};
```

Similarly, the forward declared ConnectionPoint interface shown next is remapped to the OMG IDL definition shown in the second following example.

```
// Microsoft IDL
interface IEnumConnections;
[object, uuid(B196B286-BAB4-101A-B69C-00AA00241D07),
pointer_default(unique)]
interface IConnectionPoint: IUnknown
{
   HRESULT GetConnectionInterface([out] IID *pIID);
   HRESULT GetConnectionPointContainer([out]
      IConnectionPointContainer **ppCPC);
   HRESULT Advise([in] IUnknown *pUnkSink, [out] DWORD
      *pdwCookie);
   HRESULT Unadvise(in DWORD dwCookie);
   HRESULT EnumConnections([out] IEnumConnections **ppEnum);
};
```

```
// OMG IDL
interface IEnumConnections;
interface IConnectionPoint:: CORBA::Composite,
    CosLifeCycle::LifeCycleObject
{
    HRESULT GetConnectionInterface(out IID pIID)
            raises (COM_HRESULT);
    HRESULT GetConnectionPointContainer
            (out IConnectionPointContainer pCPC)
            raises (COM_HRESULT);
    HRESULT Advise(in IUnknown pUnkSink, out DWORD pdwCookie)
            raises (COM_HRESULT);
    HRESULT Unadvise(in DWORD dwCookie)
            raises (COM_HRESULT);
    HRESULT EnumConnections(out IEnumConnections ppEnum)
            raises (COM_HRESULT);
#pragma ID IConnectionPoint = “DCE:B196B286-BAB4-101A-B69C-
00AA00241D07”;
};
```

Finally, the MIDL definition for IEnumConnectionPoints and IEnum Connections interfaces are shown next.

```
typedef struct tagCONNECTDATA {
   IUnknown * pUnk;
   DWORD dwCookie;
} CONNECTDATA;

[object, uuid(B196B285-BAB4-101A-B69C-00AA00241D07),
   pointer_default(unique)]
interface IEnumConnectionPoints: IUnknown
{
      HRESULT Next([in] unsigned long cConnections,
         [out] IConnectionPoint **rcpcn,
         [out] unsigned long *lpcFetched);
      HRESULT Skip([in] unsigned long cConnections);
      HRESULT Reset();
      HRESULT Clone([out] IEnumConnectionPoints **pEnumval);
};
[object, uuid(B196B287-BAB4-101A-B69C-00AA00241D07),
   pointer_default(unique)]
interface IEnumConnections: IUnknown
{
      HRESULT Next([in] unsigned long cConnections,
         [out] IConnectionData **rcpcn,
         [out] unsigned long *lpcFetched);
      HRESULT Skip([in] unsigned long cConnections);
      HRESULT Reset();
      HRESULT Clone([out] IEnumConnections **pEnumval);
};
```

The corresponding OMG IDL definition for EnumConnectionPoints and EnumConnections is shown next.

```
struct CONNECTDATA {
    IUnknown * pUnk;
    DWORD dwCookie;
};
interface IEnumConnectionPoints: CORBA::Composite,
CosLifeCycle::LifeCycleObject
{
    HRESULT Next(in unsigned long cConnections,
        out IConnectionPoint rcpcn,
        out unsigned long lpcFetched) raises (COM_HRESULT);
    HRESULT Skip(in unsigned long cConnections) raises
        (COM_HRESULT);
    HRESULT Reset() raises (COM_HRESULT);
    HRESULT Clone(out IEnumConnectionPoints pEnumval)
        raises(COM_HRESULT)
#pragma ID IEnumConnectionPoints =
    “DCE:B196B285-BAB4-101A-B69C-00AA00241D07”;

};

interface IEnumConnections: CORBA::Composite,
    CosLifeCycle::LifeCycleObject
{
    HRESULT Next(in unsigned long cConnections,
        out IConnectData rgcd,
        out unsigned long lpcFetched) raises (COM_HRESULT);
    HRESULT Skip(in unsigned long cConnections) raises
        (COM_HRESULT);
    HRESULT Reset() raises (COM_HRESULT);
    HRESULT Clone(out IEnumConnectionPoints pEnumVal) raises
        (COM_HRESULT);
#pragma ID IEnumConnections =
    “DCE:B196B287-BAB4-101A-B69C-00AA00241D07”;
};
```

D.3 Mapping an OMG Object Service to OLE Automation

This section provides an example of mapping an OMG-defined interface to an equivalent OLE Automation interface. This example is based on the OMG Naming Service and follows the mapping rules from Chapter 13C, Mapping: OLE Automation and CORBA. The Naming Service is defined by two interfaces and some associated types, which are scoped in the OMG IDL CosNaming module.

Table D-2 Interfaces of the OMG Naming Service

Interface	Description
CosNaming::NamingContext	Used by a client to establish the name space in which new associations between names and object references can be created, and to retrieve an object reference that has been associated with a given name.
CosNaming::BindingIterator	Used by a client to walk a list of registered names that exist within a naming context.

Microsoft ODL does not explicitly support the notions of modules or scoping domains. To avoid name conflicts, all types defined in the scoping space of *CosNaming* are expanded to global names.

The data type portion (interfaces excluded) of the *CosNaming* interface is shown next.

```
// OMG IDL
module CosNaming{
    typedef string Istring;
    struct NameComponent {
    Istring id;
    Istring kind;
};
typedef sequence <NameComponent> Name;
enum BindingType { nobject, ncontext };
struct Binding {
    Name binding_name;
    BindingType binding_type;
};
typedef sequence <Binding> BindingList;
interface BindingIterator;
interface NamingContext;
// ...
}
```

The corresponding portion (interfaces excluded) of the Microsoft ODL interface is shown next.

```
[uuid(a1789c86-1b2c-11cf-9884-08000970dac7)] // from COMID
association
 library CosNaming
 {
importlib(“stdole32.tlb”);
importlib(“corba.tlb”); / for standard CORBA types
typedef CORBA_string CosNaming_Istring;
[uuid((04b8a791-338c-afcf-1dec-cf2733995279), help-
string(“struct NameComponent”),
oleautomation, dual]
interface CosNaming_NameComponent: ICORBAStruct {
[propget] HRESULT id([out, retval]CosNaming_Istring **val);
[propput] HRESULT id([in]CosNaming_IString* val);
[propget] HRESULT kind([out, retval]CosNaming_Istring
   ** val);
[propget] HRESULT kind([in]CosNaming_Istring *val);
};
# define Name SAFEARRAY(CosNaming_NameComponent *)
   // typedef doesn’t work
typedef enum { [helpstring(“nobject”)]nobject,
      [helpstring(“ncontext”)]ncontext
} CosNaming_BindingType;
#define CosNaming_BindingList SAFEARRAY(CosNaming_Binding *)
[uuid(58fbe618-2d20-d19f-1dc2-560cc6195add),
   helpstring(“struct Binding”),
oleautomation, dual]
interface DICosNaming_Binding: ICORBAStruct {
[propget] HRESULT binding_name([retval, out]
   CosNaming_IString ** val);
 [propput] HRESULT binding_name([in]
   CosNaming_IString * vall);
[propget] HRESULT binding_type([retval, out]
   CosNaming_BindingType *val);
[propset] HRESULT binding_type([in]
   CosNaming_BindingType val);
};
#define CosNaming_BindingList SAFEARRAY(CosNaming_Binding)
interface DICosNaming_BindingIterator;
interface DICosNaming_NamingContext;
// ...
};
```

The types scoped in an OMG IDL interface are also expanded using the notation [<modulename>_]*[<interfacename>_]typename. Thus the types defined within the *CosNaming::NamingContext* interface (shown next) are expanded in Microsoft ODL as shown in the second following example.

```
module CosNaming{
// ...
    interface NamingContext
    {
        enum NotFoundReason { missing_node, not_context,
        not_object };
        exception NotFound {
            NotFoundReason why;
            Name rest_of_name;
    };
    exception CannotProceed {
        NamingContext cxt;
        Name rest_of_name;
    };
    exception InvalidName {};
    exception AlreadyBound {};
    exception NotEmpty {};
    void bind(in Name n, in Object obj)
        raises( NotFound, CannotProceed, InvalidName,
        AlreadyBound );
    void rebind(in Name n, in Object obj)
        raises( NotFound, CannotProceed, InvalidName );
    void bind_context(in Name n, in NamingContext nc)
        raises( NotFound, CannotProceed, InvalidName,
        AlreadyBound );
    void rebind_context(in Name n, in NamingContext nc)
        raises( NotFound, CannotProceed, InvalidName );
    Object resolve(in Name n)
        raises( NotFound, CannotProceed, InvalidName );
    void unbind(in Name n)
        raises( NotFound, CannotProceed, InvalidName );
    NamingContext new_context();
    NamingContext bind_new_context(in Name n)
        raises( NotFound, AlreadyBound, CannotProceed, InvalidName );
    void destroy()
        raises( NotEmpty );
    void list(in unsigned long how_many,
        out BindingList bl, out BindingIterator bi );
    };
// ...
};

[uuid(d5991293-3e9f-0e16-1d72-7858c85798d1)]
library CosNaming
 {// ...
interface DICosNaming_NamingContext;
[uuid(311089b4-8f88-30f6-1dfb-9ae72ca5b337),
   helpstring(“exception NotFound”),
oleautomation, dual]
 interface DICosNaming_NamingContext_NotFound:
   ICORBAException {
```

```
   [propget] HRESULT why([out, retval] long* _val);
   [propput] HRESULT why([in] long _val);
   [propget] HRESULT rest_of_name([out, retval]
                  CosNaming_Name ** _val);
   [propput] HRESULT rest_of_name([in] CosNaming_Name
      * _val);
};
[uuid(d2fc8748-3650-cedd-1df6-026237b92940),
   helpstring(“exception CannotProceed”),
oleautomation, dual]
interface DICosNaming_NamingContext_CannotProceed:
      DICORBAException{
   [propget] HRESULT cxt([out, retval]
      DICosNaming_NamingContext ** _val);
[propput] HRESULT cxt([in] DICosNaming_NamingContext
      * _val);
[propget] HRESULT rest_of_name([out, retval]
      CosNaming_Name ** _val);
[propput] HRESULT rest_of_name([in] CosNaming_Name * _val);
};
[uuid(7edaca7a-c123-42a1-1dca-a7e317aafe69),
   helpstring(“exception InvalidName”),
oleautomation, dual]
interface DICosNaming_NamingContext_InvalidName:
   DICORBAException {};
[uuid(fee85a90-1f6b-c47a-1dd0-f1a2fc1ab67f),
   helpstring(“exception AlreadyBound”),
oleautomation, dual]
interface DICosNaming_NamingContext_AlreadyBound:
   DICORBAException {};
[uuid(8129b3e1-16cf-86fc-1de4-b3080e6184c3),
   helpstring(“exception NotEmpty”),
oleautomation, dual]
interface CosNaming_NamingContext_NotEmpty:
   DICORBAException {};
typedef enum {[helpstring(“missing_node”)]
   NamingContext_missing_node,
      [helpstring(“not_context”) NamingContext_not_context,
      [helpstring(“not_object”) NamingContext_not_object
} CosNaming_NamingContext_NotFoundReason;
[uuid(4bc122ed-f9a8-60d4-1dfb-0ff1dc65b39a),
   helpstring(“NamingContext”),
oleautomation,dual]
interface DICosNaming_NamingContext {
HRESULT bind([in] CosNaming_Name * n, [in] IDispatch * obj,
   [out, optional] VARIANT * _user_exception);
HRESULT rebind([in] CosNaming_Name * n, in] IDispatch * obj,
   [out, optional] VARIANT * _user_exception);
HRESULT bind_context([in] CosNaming_Name * n,
[in] DICosNaming_NamingContext * nc,
[out, optional] VARIANT * _user_exception);
HRESULT rebind_context([in] CosNaming_Name * n,
[in] DICosNaming_NamingContext * nc,
[out, optional ] VARIANT * _user_exception);
HRESULT resolve([in] CosNaming_Name * n,
[out, retval] IDispatch** pResult,
[out, optional] VARIANT * _user_exception)
HRESULT unbind([in] CosNaming_Name * n,
[out, optional] VARIANT * _user_exception);
HRESULT new_context([out, retval] DICosNaming_NamingContext
** pResult);
HRESULT bind_new_context([in] CosNaming_Name * n,
   [out, retval] DICosNaming_NamingContext ** pResult,
[out, optional] VARIANT * _user_exception);
HRESULT destroy([out, optional] VARIANT* _user_exception);
HRESULT list([in] unsigned long how_many, [out]
CosNaming_BindingList ** bl,
   [out] DICosNaming_BindingIterator ** bi);
};
};
```

The *BindingIterator* interface is mapped in a similar manner, as shown in the next two examples.

```
module CosNaming {
   //...
   interface BindingIterator {
   boolean next_one(out Binding b);
   boolean next_n(in unsigned long how_many,
   out BindingList bl);
   void destroy();
   };
};
```

```
[uuid(a1789c86-1b2c-11cf-9884-08000970dac7)]
library CosNaming
 {// ...
   [uuid(5fb41e3b-652b-0b24-1dcc-a05c95edf9d3),
   help  string(“BindingIterator”),
   helpcontext(1), oleautomation, dual]
   interface DICosNaming_IBindingIterator: IDispatch {
      HRESULT next_one([out] DICosNaming_Binding ** b,
         [out, retval] boolean* pResult);
      HRESULT next_n([in] unsigned long how_many,
         [out] CosNaming_BindingList ** bl,
         [out, retval] boolean* pResult);
      HRESULT destroy();
   };
}
```

C Language Mapping 14

CORBA is independent of the programming language used to construct clients or implementations. In order to use the ORB, it is necessary for programmers to know how to access ORB functionality from their programming languages. This chapter defines the mapping of OMG IDL constructs to the C programming language.

14.1 Requirements for a Language Mapping

All language mappings have approximately the same structure. They must define the means of expressing in the language:

- All OMG IDL basic data types
- All OMG IDL constructed data types
- References to constants defined in OMG IDL
- References to objects defined in OMG IDL
- Invocations of operations, including passing parameters and receiving results
- Exceptions, including what happens when an operation raises an exception and how the exception parameters are accessed
- Access to attributes
- Signatures for the operations defined by the ORB, such as the dynamic invocation interface, the object adapters, and so forth.

A complete language mapping will allow a programmer to have access to all ORB functionality in a way that is convenient for the particular programming language. To support source portability, all ORB implementations must support the same mapping for a particular language.

14.1.1 Basic Data Types

A language mapping must define the means of expressing all of the data types defined in Section 3.8.1, Basic Types. The ORB defines the range of values supported, but the language mapping defines how a programmer sees those values. For example, the C mapping might define TRUE as one, and FALSE as zero, whereas the LISP mapping might define TRUE as T and FALSE as NIL. The mapping must specify the means to construct and operate on these data types in the programming language.

14.1.2 Constructed Data Types

A language mapping must define the means of expressing the constructed data types defined in Section 3.8.2, Constructed Types. The ORB defines aggregates of basic data types that are supported, but the language mapping defines how a programmer sees those aggregates. For example, the C mapping might define an OMG IDL struct as a C struct, whereas the LISP mapping might define an OMG IDL struct as a list. The mapping must specify the means to construct and operate on these data types in the programming language.

14.1.3 Constants

OMG IDL definitions may contain named constant values that are useful as parameters for certain operations. The language mapping should provide the means to access these constants by name.

14.1.4 Objects

There are two parts of defining the mapping of ORB objects to a particular language. The first specifies how an object is represented in the program and passed as a parameter to operations. The second is how an object is invoked. The representation of an object reference in a particular language is generally opaque, that is, some language-specific data type is used to represent the object reference, but the program does not interpret the values of that type. The language-specific representation is independent of the ORB representation of an object reference, so that programs are not ORB-dependent. In an object-oriented programming language, it may be convenient to represent an ORB object as a programming language object. Any correspondence between the programming language object types and the OMG IDL types including inheritance, operation names, etc., is up to the language mapping.

There are only three uses that a program can make of an object reference: it may specify it as a parameter to an operation (including receiving it as an output parameter), it can invoke an operation on it, or it can perform an ORB operation (including object adapter operations) on it.

14.1.5 Invocation of Operations

An operation invocation requires the specification of the object to be invoked, the operation to be performed, and the parameters to be supplied. There are a variety of possible mappings, depending to a large extent on the procedure mechanism in the particular language. Some possible choices for language mapping of invocation include: interface-specific stub routines, a single general-purpose routine, a set of calls to construct a parameter list and initiate the operation, or mapping ORB operations to operations on objects defined in an object-oriented programming language.

The mapping must define how parameters are associated with the call, and how the operation name is specified. It is also necessary to specify the effect of the call on the flow of control in the program, including when an operation completes normally and when an exception is raised.

The most natural mapping would be to model a call on an ORB object as the corresponding call in the particular language. However, this may not always be possible for languages where the type system or call mechanism is not powerful enough to handle ORB objects. In this case, multiple calls may be required. For example, in C, it is necessary to have a separate interface for dynamic construction of calls, since C does not permit discovery of new types at run-time. In LISP, however, it may be possible to make a language mapping that is the same for objects whether or not they were known at compile time.

In addition to defining how an operation is expressed, it is necessary to specify the storage allocation policy for parameters, for example, what happens to storage of input parameters, and how and where output parameters are allocated. It is also necessary to describe how a return value is handled, for operations that have one.

14.1.6 Exceptions

There are two aspects to the mapping of exceptions into a particular language. First is the means for handling an exception when it occurs, including deciding which exception occurred. If the programming language has a model of exceptions that can accommodate ORB exceptions, that would likely be the most convenient choice; if it does not, some other means must be used, for example, passing additional parameters to the operations that receive the exception status.

It is commonly the case that the programmer associates specific code to handle each kind of exception. It is desirable to make that association as convenient as possible.

Second, when an exception has been raised, it must be possible to access the parameters of the exception. If the language exception mechanism allows for parameters, that mechanism could be used. Otherwise, some other means of obtaining the exception values must be provided.

14.1.7 Attributes

The ORB models attributes as a pair of operations, one to set and one to get the attribute value. The language mapping defines the means of expressing these operations. One reason for distinguishing attributes from pairs of operations is to allow the language mapping to define the most natural way for accessing them. Some possible choices include defining two operations for each attribute; defining two operations that can set or get, respectively, any attribute; defining operations that can set or get groups of attributes, and so forth.

14.1.8 ORB Interfaces

Most of a language mapping is concerned with how the programmer-defined objects and data are accessed. Programmers who use the ORB must also access some interfaces implemented directly by the ORB, for example, to convert an object reference to a string. A language mapping must also specify how these interfaces appear in the particular programming language.

Various approaches may be taken, including defining a set of library routines, allowing additional ORB-related operations on objects, or defining interfaces that are similar to the language mapping for ordinary objects.

The last approach is called defining pseudo-objects. A pseudo-object has an interface that can (with a few exceptions) be defined in OMG IDL, but is not necessarily implemented as an ORB object. Using stubs, a client of a pseudo-object writes calls to it in the same way as if it were an ordinary object. Pseudo-object operations cannot be invoked with the Dynamic Invocation Interface. However, the ORB may recognize such calls as special and handle them directly. One advantage of pseudo-objects is that the interface can be expressed in OMG IDL independent of the particular language mapping, and the programmer can understand how to write calls by knowing the language mapping for the invocations of ordinary objects.

It is not necessary for a language mapping to use the pseudo-object approach. However, this document defines interfaces in subsequent chapters using OMG IDL wherever possible. A language mapping must define how these interfaces are accessed, either by defining them as pseudo-objects and supporting a mapping similar to ordinary objects, by defining language-specific interfaces for them, or in some other way.

14.2 Scoped Names

The C programmer must always use the global name for a type, constant, exception, or operation. The C global name corresponding to an OMG IDL global name is derived by converting occurrences of “**::**” to “_” (an underscore) and eliminating the leading underscore.

Consider the following example:

```
typedef string<256> filename_t;
interface example0 {
    enum color {red, green, blue};
    union bar switch (enum foo {room, bell}) { ... };
    • • •
};
```

Code to use this interface would appear as follows:

```
#include "example0.h"                          /* C */

filename_t FN;
example0_color C = example0_red;
example0_bar myUnion;

switch (myUnion._d) {
case example0_bar_room: • • •
case example0_bar_bell: • • •
};
```

Note that the use of underscores to replace the "**::**" separators can lead to ambiguity if the OMG IDL specification contains identifiers with underscores in them. Consider the following example:

```
typedef long foo_bar;
interface foo {
    typedef short bar; /* A legal OMG IDL statement, but
    ambigous in C */
    • • •
};
```

Due to such ambiguities, it is advisable to avoid the indiscriminate use of underscores in identifiers.

14.3 Mapping for Interfaces

All interfaces must be defined at global scope (*no* nested interfaces). The mapping for an interface declaration is as follows:

```
interface example1 {
    long op1(in long arg1);
};
```

The preceding example generates the following C declarations[1].

1. Section 14.15, Implicit Arguments to Operations, describes the additional arguments added to an operation in the C mapping.

```
typedef CORBA_Object example1 ;                          /* C */
extern CORBA_long example1_op1(
    example1 o,
    CORBA_long arg1,
    CORBA_Environment *ev
);
```

All object references (typed interface references to an object) are of the well-known, opaque type **CORBA_Object**. The representation of **CORBA_Object** is a pointer. To permit the programmer to decorate a program with typed references, a type with the name of the interface is defined to be a **CORBA_Object**. The literal **CORBA_OBJECT_NIL** is legal wherever a **CORBA_Object** may be used; it is guaranteed to pass the **is_nil** operation defined in Section 7.2.3, Nil Object References.

OMG IDL permits specifications in which arguments, return results, or components of constructed types may be interface references. Consider the following example:

```
#include "example1.idl"

interface example2 {
    example1 op2();
};
```

This is equivalent to the following C declaration.

```
#include "example1.h"                            /* C */

typedef CORBA_Object example2;
extern example1 example2_op2(example2 o, CORBA_Environment
*ev);
```

A C fragment for invoking such an operation is as follows.

```
#include "example2.h"                            /* C */

example1 ex1;
example2 ex2;
CORBA_Environment ev;

/* code for binding ex2 */

ex1 = example2_op2(ex2, &ev);
```

14.4 Inheritance and Operation Names

OMG IDL permits the specification of interfaces that inherit operations from other interfaces. Consider the following example.

```
interface example3 : example1 {
    void op3(in long arg3, out long arg4);
};
```

This is equivalent to the following C declarations.

```
typedef CORBA_Object example3;                  /* C */
extern CORBA_long example3_op1(
    example3 o,
    CORBA_long arg1,
    CORBA_Environment *ev
);
extern void example3_op3(
    example3 o,
    CORBA_long arg3,
    CORBA_long *arg4,
    CORBA_Environment *ev
);
```

As a result, an object written in C can access **`op1`** as if it was directly declared in **`example3`**. Of course, the programmer could also invoke **`example1_op1`** on an **`Object`** of type **`example3`**; the virtual nature of operations in interface definitions will cause invocations of either function to cause the same method to be invoked.

14.5 Mapping for Attributes

The mapping for attributes is best explained through example. Consider the following specification:

```
interface foo {
    struct position_t {
    float x, y;
};

    attribute float radius;
    readonly attribute position_t position;
};
```

This is exactly equivalent to the following illegal OMG IDL specification:

```
interface foo {
    struct position_t {
    float x, y;
};

    float        _get_radius();
    void         _set_radius(in float r);
    position_t _get_position();
};
```

This latter specification is illegal, since OMG IDL identifiers are not permitted to start with the underscore (_) character.

The language mapping for attributes then becomes the language mapping for these equivalent operations. More specifically, the function signatures generated for the above operations are as follows.

```
typedef struct foo_position_t {              /* C */
   CORBA_float x, y;
} foo_position_t;

extern CORBA_float foo__get_radius(foo o, CORBA_Environment
*ev);
extern void foo__set_radius(
   foo o,
   CORBA_float r,
   CORBA_Environment *ev
);
extern foo_position_t foo__get_position(foo o,
CORBA_Environment *ev);
```

Note that two underscore characters (__) separate the name of the interface from the words "**get**" or "**set**" in the names of the functions.

If the "**set**" accessor function fails to set the attribute value, the method should return one of the standard exceptions defined in Section 3.15, Standard Exceptions.

14.6 Mapping for Constants

Constant identifiers can be referenced at any point in the user's code where a literal of that type is legal. In C, these constants are **#define**d.

The fact that constants are **#define**d may lead to ambiguities in code. All names mandated by the mappings for any of the structured types below start with an underscore.

14.7 Mapping for Basic Data Types

The basic data types have the mappings shown in Table 14-1. Implementations are responsible for providing typedefs for **CORBA_short**, **CORBA_long**, and so forth, consistent with OMG IDL requirements for the corresponding data types.

Table 14-1 Data Type Mappings

OMG IDL	C
short	CORBA_short
long	CORBA_long
unsigned short	CORBA_unsigned_short
unsigned long	CORBA_unsigned_long
float	CORBA_float
double	CORBA_double
char	CORBA_char

Table 14-1 Data Type Mappings *(Continued)*

OMG IDL	C
boolean	CORBA_boolean
octet	CORBA_octet
enum	CORBA_enum
any	typedef struct CORBA_any { CORBA_TypeCode _type; void *_value; } CORBA_any;

The C mapping of the OMG IDL **boolean** types is **unsigned char** with only the values 1 (**TRUE**) and 0 (**FALSE**) defined; other values produce undefined behavior. **CORBA_boolean** is provided for symmetry with the other basic data type mappings.

The C mapping of OMG IDL **enum** types is an unsigned integer type capable of representing 2^{32} enumerations. Each enumerator in an **enum** is **#define**d with an appropriate unsigned integer value conforming to the ordering constraints described in Section 3.8.2, Enumerations.

TypeCodes are described in Section 6.7, TypeCodes. The **_value** member for an **any** is a pointer to the actual value of the datum.

The **any** type supports the notion of ownership of its **_value** member. By setting a release flag in the **any** when a value is installed, programmers can control ownership of the memory pointed to by **_value**. The location of this release flag is implementation-dependent, so the following two ORB-supplied functions allow for the setting and checking of the **any** release flag.

```
void CORBA_any_set_release(CORBA_any*, CORBA_boolean);/* C
*/
CORBA_boolean CORBA_any_get_release(CORBA_any*);
```

CORBA_any_set_release can be used to set the state of the release flag. If the flag is set to **TRUE**, the **any** effectively "owns" the storage pointed to by **_value**; if **FALSE**, the programmer is responsible for the storage. If, for example, an **any** is returned from an operation with its release flag set to **FALSE**, calling **CORBA_free()** on the returned **any*** will not deallocate the memory pointed to by **_value**. Before calling **CORBA_free()** on the **_value** member of an **any** directly, the programmer should check the release flag using **CORBA_any_get_release**. If it returns **FALSE**, the programmer should not invoke **CORBA_free()** on the **_value** member; doing so produces undefined behavior. Also, passing a null pointer to either **CORBA_any_set_release** or **CORBA_any_get_release** produces undefined behavior.

If **CORBA_any_set_release** is never called for a given instance of **any**, the default value of the release flag for that instance is **FALSE**.

14.8 Mapping Considerations for Constructed Types

The mapping for OMG IDL structured types (structs, unions, arrays, and sequences) can vary slightly depending on whether the data structure is *fixed-length* or *variable-length*. A type is *variable-length* if it is one of the following types:

- The type **any**
- A bounded or unbounded string
- A bounded or unbounded sequence
- An object reference or reference to a transmissible pseudo-object[2]
- A struct or union that contains a member whose type is variable-length
- An array with a variable-length element type
- A typedef to a variable-length type

The reason for treating fixed- and variable-length data structures differently is to allow more flexibility in the allocation of **out** parameters and return values from an operation. This flexibility allows a client-side stub for an operation that returns a sequence of strings, for example, to allocate all the string storage in one area that is deallocated in a single call.

The mapping of a variable-length type as an **out** parameter or operation return value is a pointer to the associated class or array, as shown in Table 14-2.

For types whose parameter passing modes require heap allocation, an ORB implementation will provide allocation functions. These types include variable-length **struct**, variable-length **union**, **sequence**, **any**, **string**, and array of a variable-length type. The return value of these allocation functions must be freed using **CORBA_free().** For one of these listed types **T**, the ORB implementation will provide the following type-specific allocation function:

```
T *T__alloc();                              /* C */
```

The functions are defined at global scope using the fully-scoped name of T converted into a C language name (as described in Section 14.2, Scoped Names) followed by the suffix __**alloc** (note the double underscore). For **any** and **string**, the allocation functions are, respectively:

```
CORBA_any *CORBA_any_alloc();
char *CORBA_string_alloc();
```

14.9 Mapping for Structure Types

OMG IDL structures map directly onto C **struct**s. Note that all OMG IDL types that map to C **struct**s may potentially include padding.

2.Transmissible pseudo-objects are listed as "general arguments" in Table 14 on page A-2.

14.10 Mapping for Union Types

OMG IDL discriminated unions are mapped onto C **struct**s. Consider the following OMG IDL declaration.

```
union Foo switch (long) {
    case 1: long x;
    case 2: float y;
    default: char z;
};
```

This is equivalent to the following **struct** in C:

```
typedef struct {                              /* C */
   CORBA_long _d;
   union {
   CORBA_long x;
   CORBA_float y;
   CORBA_char z;
   } _u;
} Foo;
```

The discriminator in the struct is always referred to as **_d**; the union in the struct is always referred to as **_u.**

Reference to union elements is as in normal C:

```
Foo *v;                                       /* C */

/* make a call that returns a pointer to a Foo in v */

switch(v->_d) {
   case 1:  printf("x = %ld\n", v->_u.x); break;
   case 2:  printf("y = %f\n", v->_u.y); break;
   default: printf("z = %c\n", v->_u.z); break;
}
```

An ORB implementation need not use a C **union** to hold the OMG IDL **union** elements; a C struct may be used instead. In either case, the programmer accesses the union elements via the **_u** member.

14.11 Mapping for Sequence Types

The OMG IDL data type **sequence** permits passing of unbounded arrays between objects. Consider the following OMG IDL declaration:

```
typedef sequence<long,10> vec10;
```

In C, this is converted to:

```
typedef struct {                                /* C */
   CORBA_unsigned_long _maximum;
   CORBA_unsigned_long _length;
   CORBA_long *_buffer;
} vec10;
```

An instance of this type is declared as follows:

```
vec10 x = {10L, 0L, (CORBA_long *)NULL);        /* C */
```

Prior to passing **&x** as an **in** parameter, the programmer must set the **_buffer** member to point to a **CORBA_long** array of 10 elements, and must set the **_length** member to the actual number of elements to transmit.

Prior to passing the address of a **vec10*** as an **out** parameter (or receiving a **vec10*** as the function return), the programmer does nothing. The client stub will allocate storage for the returned sequence; for bounded sequences, it also allocates a buffer of the specified size, while for unbounded sequences, it also allocates a buffer big enough to hold what was returned by the object. Upon successful return from the invocation, the **_maximum** member will contain the size of the allocated array, the **_buffer** member will point at allocated storage, and the **_length** member will contain the number of values that were returned in the **_buffer** member. The client is responsible for freeing the allocated sequence using **CORBA_free()**.

Prior to passing **&x** as an **inout** parameter, the programmer must set the **_buffer** member to point to a **CORBA_long** array of 10 elements. The **_length** member must be set to the actual number of elements to transmit. Upon successful return from the invocation, the **_length** member will contain the number of values that were copied into the buffer pointed to by the **_buffer** member. If more data must be returned than the original buffer can hold, the callee can deallocate the original **_buffer** member using **CORBA_free()** (honoring the release flag) and assign **_buffer** to point to new storage.

For bounded sequences, it is an error to set the **_length** or **_maximum** member to a value larger than the specified bound.

Sequence types support the notion of ownership of their **_buffer** members. By setting a release flag in the sequence when a buffer is installed, programmers can control ownership of the memory pointed to by **_buffer**. The location of this release flag is implementation-dependent, so the following two ORB-supplied functions allow for the setting and checking of the sequence release flag:

```
void CORBA_sequence_set_release(void*, CORBA_boolean);/* C
*/
CORBA_boolean CORBA_sequence_get_release(void*);
```

CORBA_sequence_set_release can be used to set the state of the release flag. If the flag is set to **TRUE**, the sequence effectively “owns” the storage pointed to by **_buffer**; if **FALSE**, the programmer is responsible for the storage. If, for example, a sequence is returned from an operation with its release flag set to **FALSE**, calling

CORBA_free() on the returned sequence pointer will not deallocate the memory pointed to by **_buffer**. Before calling **CORBA_free()** on the **_buffer** member of a sequence directly, the programmer should check the release flag using **CORBA_sequence_get_release**. If it returns **FALSE**, the programmer should not invoke **CORBA_free()** on the **_buffer** member; doing so produces undefined behavior. Also, passing a null pointer or a pointer to something other than a sequence type to either **CORBA_sequence_set_release** or **CORBA_sequence_get_release** produces undefined behavior.

CORBA_sequence_set_release should only be used by the creator of a sequence. If it is not called for a given sequence instance, then the default value of the release flag for that instance is **FALSE**.

Two sequence types are the same type if their sequence element type and size arguments are identical. For example,

```
const long SIZE = 25;
typedef long seqtype;

typedef sequence<long, SIZE> s1;
typedef sequence<long, 25> s2;
typedef sequence<seqtype, SIZE> s3;
typedef sequence<seqtype, 25> s4;
```

declares **s1**, **s2**, **s3**, and **s4** to be of the same type.

The OMG IDL type

```
sequence<type,size>
```

maps to

```
#ifndef _CORBA_sequence_type_defined            /* C */
#define _CORBA_sequence_type_defined
typedef struct {
   CORBA_unsigned_long _maximum;
   CORBA_unsigned_long _length;
   type *_buffer;
} CORBA_sequence_type;
#endif /* _CORBA_sequence_type_defined */
```

The **ifdef**s are needed to prevent duplicate definition where the same type is used more than once. The type name used in the C mapping is the type name of the effective type, e.g. in

```
typedef CORBA_long FRED;                        /* C */
typedef sequence<FRED,10> FredSeq;
```

the sequence is mapped onto

```
struct { ... } CORBA_sequence_long;
```

If the **`type`** in **`sequence<type,size>`** consists of more than one identifier (e.g. unsigned long), then the generated type name consists of the string **`CORBA_sequence_`** concatenated to the string consisting of the concatenation of each identifier separated by underscores (e.g. **`unsigned_long`**).

If the **`type`** is a **`string`**, the string "string" is used to generate the type name. If the **`type`** is a **`sequence`**, the string "sequence" is used to generate the type name, recursively. For example

```
sequence<sequence<long> >
```

generates a type of

```
CORBA_sequence_sequence_long
```

These generated type names may be used to declare instances of a sequence type.

In addition to providing a type-specific allocation function for each sequence, an ORB implementation must provide a buffer allocation function for each sequence type. These functions allocate vectors of type **`T`** for **`sequence<T>`**. They are defined at global scope and are named similarly to sequences:

```
T *CORBA_sequence_T_allocbuf(CORBA_unsigned_long len);/* C
*/
```

Here, **`T`** refers to the type name. For the type

```
sequence<sequence<long> >
```

for example, the sequence buffer allocation function is named

```
T *CORBA_sequence_sequence_long_allocbuf(CORBA_unsigned_long
len);
```

Buffers allocated using these allocation functions are freed using **`CORBA_free().`**

14.12 Mapping for Strings

OMG IDL strings are mapped to 0-byte terminated character arrays; i.e. the length of the string is encoded in the character array itself through the placement of the 0-byte. Note that the storage for C strings is one byte longer than the stated OMG IDL bound. Consider the following OMG IDL declarations:

```
typedef string<10> sten;
typedef string sinf;
```

In C, this is converted to:

```
typedef CORBA_char *sten;                       /* C */
typedef CORBA_char *sinf;
```

Instances of these types are declared as follows:

```
sten s1 = NULL;                                 /* C */
sinf s2 = NULL;
```

Two string types are the same type if their size arguments are identical. For example,

```
const long SIZE = 25;                           /* C */

typedef string<SIZE> sx;
typedef string<25> sy;
```

declares **sx** and **sy** to be of the same type.

Prior to passing **s1** or **s2** as an **in** parameter, the programmer must assign the address of a character buffer containing a 0-byte terminated string to the variable. The caller cannot pass a null pointer as the string argument.

Prior to passing **&s1** or **&s2** as an **out** parameter (or receiving an **sten** or **sinf** as the return result), the programmer does nothing. The client stub will allocate storage for the returned buffer; for bounded strings, it allocates a buffer of the specified size, while for unbounded strings, it allocates a buffer big enough to hold the returned string. Upon successful return from the invocation, the character pointer will contain the address of the allocated buffer. The client is responsible for freeing the allocated storage using **CORBA_free()**.

Prior to passing **&s1** or **&s2** as an **inout** parameter, the programmer must assign the address of a character buffer containing a 0-byte terminated array to the variable. If the returned string is larger than the original buffer, the client stub will call **CORBA_free()** on the original string and allocate a new buffer for the new string. The client should therefore never pass an **inout** string parameter that was not allocated using **CORBA_string_alloc**. The client is responsible for freeing the allocated storage using **CORBA_free()**, regardless of whether or not a reallocation was necessary.

Strings are dynamically allocated using the following ORB-supplied function:

```
char *CORBA_string_alloc(CORBA_unsigned_long len);
```

This function allocates **len+1** bytes, enough to hold the string and its terminating NULL character.

Strings allocated in this manner are freed using **CORBA_free()**.

14.13 Mapping for Arrays

OMG IDL arrays map directly to C arrays. All array indices run from zero to <**size-**1>.

For each named array type in OMG IDL, the mapping provides a C typedef for pointer to the array's *slice*. A slice of an array is another array with all the dimensions of the original except the first. For example, given the following OMG IDL definition:

```
typedef long LongArray[4][5];
```

The C mapping provides the following definitions:

```
typedef CORBA_long LongArray[4][5];
typedef CORBA_long LongArray_slice[5];
```

The generated name of the slice typedef is created by appending **_slice** to the original array name.

If the return result, or an **out** parameter for an array holding a variable-length type of an operation is an array, the array storage is dynamically allocated by the stub; a pointer to the array slice of the dynamically allocated array is returned as the value of the client stub function. When the data is no longer needed, it is the programmer's responsibility to return the dynamically allocated storage by calling **CORBA_free()**.

For an array, **T** of a variable-length type is dynamically allocated using the following ORB-supplied function:

```
T_slice *T__alloc();                           /* C */
```

This function is identical to the allocation functions described in Section 14.8, Mapping Considerations for Constructed Types, except that the return type is pointer to array slice, not pointer to array.

14.14 Mapping for Exception Types

Each defined exception type is defined as a struct tag and a typedef with the C global name for the exception. An identifier for the exception, in string literal form, is also **#define**d, as is a type-specific allocation function. For example:

```
exception foo {
    long dummy;
};
```

yields the following C declarations:

```
typedef struct foo {                              /* C */
   CORBA_long dummy;
   /* ...may contain additional
    * implementation-specific members...
    */
} foo;
#define ex_foo <unique identifier for exception>
foo *foo__alloc();
```

The identifier for the exception uniquely identifies this exception type. For example, it could be the Interface Repository identifier for the exception (see Section 6.5.19, ExceptionDef).

The allocation function dynamically allocates an instance of the exception and returns a pointer to it. Each exception type has its own dynamic allocation function. Exceptions allocated using a dynamic allocation function are freed using **`CORBA_free()`**.

14.15 Implicit Arguments to Operations

From the point of view of the C programmer, all operations declared in an interface have additional leading parameters preceding the operation-specific parameters:

- The first parameter to each operation is a **`CORBA_Object`** input parameter; this parameter designates the object to process the request.
- The last parameter to each operation is a (**`CORBA_Environment *)`** output parameter; this parameter permits the return of exception information.
- If an operation in an OMG IDL specification has a context specification, then a **`CORBA_Context`** input parameter precedes the (**`CORBA_Environment *`**) parameter and follows any operation-specific arguments.

As described above, the **`CORBA_Object`** type is an opaque type. The **`CORBA_Environment`** type is partially opaque; Section 14.20, Handling Exceptions, provides a description of the nonopaque portion of the exception structure and an example of how to handle exceptions in client code. The **`CORBA_Context`** type is opaque; see Chapter 4, Dynamic Invocation Interface, for more information on how to create and manipulate context objects.

14.16 Interpretation of Functions with Empty Argument Lists

A function declared with an empty argument list is defined to take *no* operation-specific arguments.

14.17 Argument Passing Considerations

For all OMG IDL types (except arrays), if the OMG IDL signature specifies that an argument is an **out** or **inout** parameter, then the caller must always pass the address of a variable of that type (or the value of a pointer to that type); the callee must dereference the parameter to get to the type. For arrays, the caller must pass the address of the first element of the array.

For **in** parameters, the value of the parameter must be passed for all of the basic types, enumeration types, and object references. For all arrays, the address of the first element of the array must be passed. For all other structured types, the address of a variable of that type must be passed, regardless of whether they are fixed- or variable-length. For strings, a **char*** must be passed.

For **inout** parameters, the address of a variable of the correct type must be passed for all of the basic types, enumeration types, object references, and structured types. For strings, the address of a **char*** must be passed. For all arrays, the address of the first element of the array must be passed.

Consider the following OMG IDL specification:

```
interface foo {
typedef long Vector[25];

    void bar(out Vector x, out long y);
};
```

Client code for invoking the **bar** operation would look like:

```
foo object;                                   /* C */
foo_Vector_slice x;
CORBA_long y;
CORBA_Environment ev;

/* code to bind object to instance of foo */

foo_bar(object, &x, &y, &ev);
```

For **out** parameters of type variable-length **struct**, variable-length **union**, **string**, **sequence**, an array holding a variable-length type, or **any**, the ORB will allocate storage for the output value using the appropriate type-specific allocation function. The client may use and retain that storage indefinitely, and must indicate when the value is no longer needed by calling the procedure **CORBA_free**, whose signature is:

```
extern void CORBA_free (void *storage);       /* C */
```

The parameter to **CORBA_free()** is the pointer used to return the **out** parameter. **CORBA_free()** releases the ORB-allocated storage occupied by the **out** parameter, including storage indirectly referenced, such as in the case of a sequence of strings or array of object reference. If a client does not call **CORBA_free()** before reusing the pointers that reference the **out** parameters, that storage might be wasted. Passing a null pointer to **CORBA_free()** is allowed; **CORBA_free()** simply ignores it and returns without error.

14.18 Return Result Passing Considerations

When an operation is defined to return a nonvoid return result, the following rules hold:

- If the return result is one of the types **float**, **double**, **long**, **short**, **unsigned long**, **unsigned short**, **char**, **boolean**, **octet**, **Object**, or an **enumeration**, then the value is returned as the operation result.
- If the return result is one of the fixed-length types **struct** or **union**, then the value of the C struct representing that type is returned as the operation result. If the return result is one of the variable-length types **struct**, **union**, **sequence**, or **any**, then a pointer to a C struct representing that type is returned as the operation result.
- If the return result is of type **string**, then a pointer to the first character of the string is returned as the operation result.
- If the return result is of type **array**, then a pointer to the slice of the array is returned as the operation result.

Consider the following interface:

```
interface X {
    struct y {
        long a;
        float b;
    };

    long op1();
    y op2();
}
```

The following C declarations ensue from processing the specification:

```
typedef CORBA_Object X;                            /* C */
typedef struct X_y {
   CORBA_long a;
   CORBA_float b;
} X_y;

extern CORBA_long X_op1(X object, CORBA_Environment *ev);
extern X_y X_op2(X object, CORBA_Environment *ev);
```

For operation results of type variable-length **struct**, variable-length **union**, **string**, **sequence**, **array**, or **any**, the ORB will allocate storage for the return value using the appropriate type-specific allocation function. The client may use and retain that storage indefinitely, and must indicate when the value is no longer needed by calling the procedure **`CORBA_free()`** described in Section 14.17, Argument Passing Considerations.

14.19 Summary of Argument/Result Passing

Table 14-2 summarizes what a client passes as an argument to a stub and receives as a result. For brevity, the **`CORBA_`** prefix is omitted from type names in the tables.

Table 14-2 Basic Argument and Result Passing

Data Type	In	Inout	Out	Return
short	short	short*	short*	short
long	long	long*	long*	long
unsigned short	unsigned_short	unsigned_short*	unsigned_short*	unsigned_short
unsigned long	unsigned_long	unsigned_long*	unsigned_long*	unsigned_long
float	float	float*	float*	float
double	double	double*	double*	double
boolean	boolean	boolean*	boolean*	boolean
char	char	char*	char*	char
octet	octet	octet*	octet*	octet
enum	enum	enum*	enum*	enum
object reference ptr[1]	objref_ptr	objref_ptr*	objref_ptr*	objref_ptr
struct, fixed	struct*	struct*	struct*	struct
struct, variable	struct*	struct*	struct**	struct*
union, fixed	union*	union*	union*	union
union, variable	union*	union*	union**	union*
string	char*	char**	char**	char*
sequence	sequence*	sequence*	sequence**	sequence*
array, fixed	array	array	array	array slice*[2]
array, variable	array	array	array slice**[2]	array slice*[2]
any	any*	any*	any**	any*

1. Including pseudo-object references.
2. A slice is an array with all the dimensions of the original except the first one.

A client is responsible for providing storage for all arguments passed as **`in`** arguments.

Table 14-3 Client Argument Storage Responsibilities

Type	Inout Param	Out Param	Return Result
short	1	1	1
long	1	1	1
unsigned short	1	1	1
unsigned long	1	1	1
float	1	1	1
double	1	1	1
boolean	1	1	1
char	1	1	1
octet	1	1	1
enum	1	1	1
object reference ptr	2	2	2
struct, fixed	1	1	1
struct, variable	1	3	3
union, fixed	1	1	1
union, variable	1	3	3
string	4	3	3
sequence	5	3	3
array, fixed	1	1	6
array, variable	1	6	6
any	5	3	3

Table 14-4 Argument Passing Cases

Case[1]	
1	Caller allocates all necessary storage, except that which may be encapsulated and managed within the parameter itself. For inout parameters, the caller provides the initial value, and the callee may change that value. For *out* parameters, the caller allocates the storage but need not initialize it, and the callee sets the value. Function returns are by value.
2	Caller allocates storage for the object reference. For *inout* parameters, the caller provides an initial value; if the callee wants to reassign the *inout* parameter, it will first call **`CORBA_Object_release`** on the original input value. To continue to use an object reference passed in as an *inout*, the caller must first duplicate the reference. The client is responsible for the release of all *out* and *return* object references. Release of all object references embedded in other *out* and *return* structures is performed automatically as a result of calling **`CORBA_free`**.
3	For out parameters, the caller allocates a pointer and passes it by reference to the callee. The callee sets the pointer to point to a valid instance of the parameter's type. For returns, the callee returns a similar pointer. The callee is not allowed to return a null pointer in either case. In both cases, the caller is responsible for releasing the returned storage. Following the completion of a request, the caller is not allowed to modify any values in the returned storage—to do so, the caller must first copy the returned instance into a new instance, then modify the new instance.

Table 14-4 Argument Passing Cases *(Continued)*

Case[1]	
4	For *inout* strings, the caller provides storage for both the input string and the **char*** pointing to it. The callee may deallocate the input string and reassign the **char*** to point to new storage to hold the output value. The size of the *out* string is therefore not limited by the size of the *in* string. The caller is responsible for freeing the storage for the *out*. The callee is not allowed to return a null pointer for an *inout*, *out*, or *return* value.
5	For *inout* sequences and **any**s, assignment or modification of the sequence or **any** may cause deallocation of owned storage before any reallocation occurs, depending upon the state of the boolean release in the sequence or **any**.
6	For *out* parameters, the caller allocates a pointer to an array slice, which has all the same dimensions of the original array except the first, and passes the pointer by reference to the callee. The callee sets the pointer to point to a valid instance of the array. For returns, the callee returns a similar pointer. The callee is not allowed to return a null pointer in either case. In both cases, the caller is responsible for releasing the returned storage. Following the completion of a request, the caller is not allowed to modify any values in the returned storage—to do so, the caller must first copy the returned array instance into a new array instance, then modify the new instance.

1. As listed in Table 21.

14.20 Handling Exceptions

The **CORBA_Environment** type is partially opaque; the C declaration contains at least the following:

```
typedef struct CORBA_Environment {           /* C */
   CORBA_exception_type _major;
   ...
} CORBA_Environment;
```

Upon return from an invocation, the **_major** field indicates whether the invocation terminated successfully; **_major** can have one of the values **CORBA_NO_EXCEPTION**, **CORBA_USER_EXCEPTION**, or **CORBA_SYSTEM_EXCEPTION**; if the value is one of the latter two, then any exception parameters signaled by the object can be accessed.

Three functions are defined on an **CORBA_Environment** structure for accessing exception information; their signatures are:

```
extern CORBA_char *CORBA_exception_id(CORBA_Environment
*ev);                                        /* C */
extern void *CORBA_exception_value(CORBA_Environment *ev);
extern void CORBA_exception_free(CORBA_Environment *ev);
```

CORBA_exception_id() returns a pointer to the character string identifying the exception. If invoked on an **CORBA_Environment** which identifies a nonexception (**_major==CORBA_NO_EXCEPTION**), a NULL is returned.

CORBA_exception_value() returns a pointer to the structure corresponding to this exception. If invoked on an **CORBA_Environment** which identifies a nonexception or an exception for which there is no associated information, a NULL is returned.

CORBA_exception_free() returns any storage which was allocated in the construction of the **CORBA_Environment**. It is permissible to invoke **CORBA_exception_free()** regardless of the value of the **_major** field.

Consider the following example:

```
interface exampleX {
    exception BadCall {
    string<80> reason;
    };

    void op() raises(BadCall);
};
```

This interface defines a single operation, which returns no results and can raise a **BadCall** exception. The following user code shows how to invoke the operation and recover from an exception.

```
#include "exampleX.h"                        /* C */

CORBA_Environment ev;
exampleX obj;
exampleX_BadCall *bc;

/*
*some code to initialize obj to a reference to an object
*supporting the exampleX interface
*/

exampleX_op(obj, &ev);
switch(ev._major) {
case CORBA_NO_EXCEPTION:/* successful outcome*/
   /* process out and inout arguments */
   break;
case CORBA_USER_EXCEPTION:/* a user-defined exception */
   if (strcmp(ex_exampleX_BadCall,CORBA_exception_id(&ev))
         == 0) {
      bc = (exampleX_BadCall *)CORBA_exception_value(&ev);
      fprintf(stderr, "exampleX_op() failed - reason: %s\n",
         bc->reason);
   }
   else {   /* should never get here ... */
         fprintf( stderr,
                  "unknown user-defined exception -%s\n",
                  CORBA_exception_id(&ev));
   }
   break;
default:/* standard exception */
   /*
    * CORBA_exception_id() can be used to determine
    * which particular standard exception was
    * raised; the minor member of the struct
    * associated with the exception (as yielded by
    * CORBA_exception_value()) may provide additional
    * system-specific information about the exception
    */
   break;
}
/* free any storage associated with exception */
CORBA_exception_free(&ev);
```

14.21 Method Routine Signatures

The signatures of the methods used to implement an object depend not only on the language binding, but also on the choice of object adapter. Different object adapters may provide additional parameters to access object adapter-specific features.

Most object adapters are likely to provide method signatures similar in most respects to those of the client stubs. In particular, the mapping for the operation parameters expressed in OMG IDL should be the same as for the client side.

See Section 14.25, BOA: Mapping for Object Implementations, for the description of method signatures for implementations using the Basic Object Adapter.

14.22 Include Files

Multiple interfaces may be defined in a single source file. By convention, each interface is stored in a separate source file. All OMG IDL compilers will, by default, generate a header file named **`Foo.h`** from **`Foo.idl.`** This file should be **`#include`**d by clients and implementations of the interfaces defined in **`Foo.idl`**.

Inclusion of **`Foo.h`** is sufficient to define all global names associated with the interfaces in **`Foo.idl`** and any interfaces from which they are derived.

14.23 Pseudo-Objects

In the C language mapping, there are several interfaces defined as pseudo-objects; Table 14 on page A-2 lists the pseudo-objects. A client makes calls on a pseudo-object in the same way as an ordinary ORB object. However, the ORB may implement the pseudo-object directly, and there are restrictions on what a client may do with a pseudo-object.

The ORB itself is a pseudo-object with the following partial definition (see Chapter 7, ORB Interface, for the complete definition):

```
interface ORB {
    string      object_to_string (in Object obj);
    Object      string_to_object (in string str);
};
```

This means that a C programmer may convert an object reference into its string form by calling:

```
CORBA_Environment ev;                        /* C */
CORBA_char *str = CORBA_ORB_object_to_string(orbobj, &ev,
obj);
```

just as if the ORB were an ordinary object. The C library contains the routine **`CORBA_ORB_object_to_string`**, and it does not do a real invocation. The **`orbobj`** is an object reference that specifies which ORB is of interest, since it is possible to choose which ORB should be used to convert an object reference to a string (see Chapter 7, ORB Interface, for details on this specific operation).

Although operations on pseudo-objects are invoked in the usual way defined by the C language mapping, there are restrictions on them. In general, a pseudo-object cannot be specified as a parameter to an operation on an ordinary object. Pseudo-objects are also not accessible using the dynamic invocation interface, and do not have definitions in the interface repository.

Operations on pseudo-objects may take parameters that are not permitted in operations on ordinary objects. For example, the **`set_exception`** operation on the Basic Object Adapter pseudo-object takes a C **`(void *)`** to specify the exception parameters (see Section 14.25.2, Method Signatures, for details). Generally, these parameters will be language-mapping specific.

Because the programmer uses pseudo-objects in the same way as ordinary objects, some ORB implementations may choose to implement some pseudo-objects as ordinary objects. For example, assuming it could be efficient enough, a context object might be implemented as an ordinary object.

14.24 Mapping of the Dynamic Skeleton Interface to C

For general information about mapping of the Dyanmic Skeleton Interface to programming languages, refer to Section 5.3, Dynamic Skeleton Interface: Language Mapping.

This section contains:

- A mapping of the Dynamic Skelton Interface’s ServerRequest to C
- A mapping of the Basic Object Adapter’s Dynamic Implementation Routine to C

14.24.1 Mapping of ServerRequest to C

In the C mapping, a ServerRequest is a pseudo-object in the CORBA module that supports the following operations:

```
CORBA_Identifier        CORBA_ServerRequest_op_name (
                            CORBA_ServerRequest req,
                            CORBA_Environment *env
    );
```

This function returns the name of the operation being performed, as shown in the operation’s OMG IDL specification.

```
CORBA_ContextCORBA_ServerRequest_ctx (
        CORBA_ServerRequest req,
        CORBA_Environment *env
    );
```

This function may be used to determine any context values passed as part of the operation. Context will only be available to the extent defined in the operation’s OMG IDL definition; for example, attribute operations have none.

```
voidCORBA_ServerRequest_params (
        CORBA_ServerRequest req,
        CORBA_NVList parameters,
        CORBA_Environment *env
    );
```

This function is used to retrieve parameters from the ServerRequest, and to find the addresses used to pass pointers to result values to the ORB. It must always be called by each DIR, even when there are no parameters.

The caller passes ownership of the **`parameters`** NVList to the ORB. Before this routine is called, that NVList should be initialized with the TypeCodes for each of the parameters to the operation being implemented: *in*, *out*, and *inout* parameters inclusive. When the call returns, the **`parameters`** NVList is still usable by the DIR, and all *in* and *inout* parameters will have been unmarshaled. Pointers to those parameter values will at that point also be accessible through the **`parameters`** NVList.

The implementation routine will then process the call, producing any result values. If the DIR does not need to report an exception, it will replace pointers to *inout* values in parameters with the values to be returned, and assign pointers to *out* values in that NVList appropriately as well. When the DIR returns, all the parameter memory is freed as appropriate, and the NVList itself is freed by the ORB.

```
void    CORBA_ServerRequest_result (
        CORBA_ServerRequest req,
        CORBA_Any value,
        CORBA_Environment *env
    );
```

This function is used to report any result **`value`** for an operation; if the operation has no result, it must not be called. It also must not be called before the parameters have been retrieved, or if an exception is being reported.

```
void    CORBA_ServerRequest_exception (
        CORBA_ServerRequest req,
        CORBA_exception_type major,
        CORBA_Any value,
        CORBA_Environment *env
    );
```

This function is used to report exceptions, both user and system, to the client who made the original invocation. The parameters are as follows:

- **`major`** indicates whether the exception is a user exception or system exception.
- **`value`** is the value of the exception, including an **exceptionTypeCode**.

14.24.2 Mapping of BOA's Dynamic Implementation Routine to C

In C, a DIR is a function with the following signature.

```
typedef void (*DynamicImplementationRoutine) (/* C */
   CORBA_Object              target,
   CORBA_ServerRequest       request,
   CORBA_Environment         *env
);
```

Such a function will be invoked by the BOA when an invocation is received on an object reference whose implementation has registered a dynamic skeleton.

- **`target`** is the name object reference to which the invocation is directed.
- **`request`** is the ServerRequest used to access explicit parameters and report results (and exceptions).
- **`env`** may be passed to **`CORBA_BOA_get_principal`** if desired.

Unlike other BOA object implementations, the **`CORBA_BOA_set_exception`** API is not used. Instead, **`CORBA_ServerRequest_exception`** is used; this provides the TypeCode for the exception to the ORB, so it does not need to consult the Interface Repository (or rely on compiled stubs) to marshal the exception value.

14.25 BOA: Mapping for Object Implementations

This section describes the details of the OMG IDL-to-C language mapping that apply specifically to the Basic Object Adapter, such as how the implementation methods are connected to the skeleton.

14.25.1 Operation-specific Details

This chapter defines most of the details of naming of parameter types and parameter passing conventions. Generally, for those parameters that are operation-specific, the method implementing the operation appears to receive the same values that would be passed to the stubs.

14.25.2 Method Signatures

With the BOA, implementation methods have signatures that are identical to the stubs. If the following interface is defined in OMG IDL:

```
interface example4 {        // IDL
   long op5(in long arg6);
};
```

a method for the **op5** routine must have the following function signature:

```
CORBA_long example4_op5(                             /* C */
   example4       object,
   CORBA_Environment *ev,
   CORBA_long     arg6
);
```

The **object** parameter is the object reference that was invoked. The method can identify which object was intended by using the **get_id** BOA operation. The **ev** parameter is used for authentication on the **get_principal** BOA operation, and is used for indicating exceptions.

The method terminates successfully by executing a **return** statement returning the declared operation value. Prior to returning the result of a successful invocation, the method code must assign legal values to all **out** and **inout** parameters.

The method terminates with an error by executing the **set_exception** BOA operation prior to executing a **return** statement. The **set_exception** operation has the following C language definition:

```
void CORBA_BOA_set_exception (                       /* C */
   CORBA_Object         boa,
   CORBA_Environment    *ev,
   CORBA_exception_type major,
   CORBA_char           *exceptname,
   void                 *param
);
```

The **ev** parameter is the environment parameter passed into the method. The caller must supply a value for the major parameter. The value of the major parameter constrains the other parameters in the call as follows:

- If the **major** parameter has the value NO_EXCEPTION, then it specifies that this is a normal outcome to the operation. In this case, both **exceptname** and **param** must be NULL. Note that it is *not* necessary to invoke **set_exception()** to indicate a normal outcome; it is the default behavior if the method simply returns.
- For any other value of **major** it specifies either a user-defined or standard exception. The **exceptname** parameter is a string representing the exception type identifier. If the exception is declared to take parameters, the **param** parameter must be the address of a struct containing the parameters according to the C language mapping, coerced to a **void ***; if the exception takes no parameters, **param** must be NULL.

When raising an exception, the method code is *not* required to assign legal values to any **out** or **inout** parameters. Due to restrictions in C, it must return a legal function value.

14.25.3 Binding Methods to Skeletons

It is not specified as part of the language mapping how the skeletons are connected to the methods. Different means will be used in different environments. For example, the skeletons may make references to the methods that are resolved by the linker or there may be a system-dependent call done at program startup to specify the location of the methods.

14.25.4 BOA and ORB Operations

The operations on the BOA defined earlier in this chapter and the operations on the ORB defined in the ORB Interface chapter are used as if they had the OMG IDL definitions described in the document, and then mapped in the usual way with the C language mapping.

For example, the **string_to_object** ORB operation has the following signature:

```
CORBA_Object CORBA_ORB_string_to_object (       /* C */
   CORBA_Object      orb,
   CORBA_Environment *ev,
   CORBA_char        *objectstring
);
```

The **create** BOA operation has the following signature:

```
CORBA_Object CORBA_BOA_create (                 /* C */
   CORBA_Object               boa,
   CORBA_Environment          *ev,
   CORBA_ReferenceData        *id,
   CORBA_InterfaceDef         intf,
   CORBA_ImplementationDef    impl
);
```

Although in each example we are using an “object” that is special (an ORB, an object adapter, or an object reference), the method name is generated as **interface_operation** in the same way as ordinary objects. Also, the signature contains a **CORBA_Environment** parameter for error indications.

In the first two cases, the signature calls for an object reference to represent the particular ORB or object adapter being manipulated. Programs may obtain these objects in a variety of ways, for example, in a global variable before program startup if there is only one ORB or BOA that makes sense, or by obtaining them from a name service if more than one is available. In the third case, the object reference being operated on is specified as the first parameter.

Following the same procedure, the C language binding for the remainder of the ORB, BOA, and object reference operations may be determined.

14.26 ORB and OA/BOA Initialization Operations

14.26.1 ORB Initialization

The following PIDL specifies initialization operations for an ORB; this PIDL is part of the CORBA module (not the ORB interface) and is described in Section 7.4, ORB Initialization.

```
// PIDL
module CORBA {

    typedef string ORBid;
    typedef sequence <string> arg_list;

    ORB ORB_init (inout arg_list argv, in ORBid
    orb_identifier);
};
```

The mapping of the preceding PIDL operations to C is as follows:

```
/* C language mapping */
typedef CORBA_string CORBA_ORBid;
extern CORBA_ORB CORBA_ORB_init (int *argc,
                              char **argv,
                              CORBA_ORBid orb_identifier,
                              CORBA_Environment *env);
```

The C mapping for **ORB_init** deviates from the PIDL in its handling of the **arg_list** parameter. This is intended to provide a meaningful PIDL definition of the initialization interface, which has a natural C (and C++) binding. To this end, the **arg_list** structure is replaced with **argv** and **argc** parameters.

The **argv** parameter is defined as an unbound array of strings (**char ****) and the number of strings in the array is passed in the **int*** parameter.

If a NULL ORBid is used, then **argv** arguments can be used to determine which ORB should be returned. This is achieved by searching the **argv** parameters for one tagged ORBid, e.g. -ORBid “ORBid_example.”

For C, the order of consumption of **argv** parameters may be significant to an application. In order to ensure that applications are not required to handle **argv** parameters, they do not recognize that the ORB initialization function must be called before the remainder of the parameters are consumed. Therefore, after the **ORB_init** call, the **argv** and **argc** parameters will have been modified to remove the ORB understood arguments. It is important to note that the **ORB_init** call can only reorder or remove references to parameters from the **argv** list, this restriction is made in order to avoid potential memory management problems caused by trying to free parts of the **argv** list or extending the **argv** list of parameters. This is why **argv** is passed as a **char**** and not a **char*****.

14.26.2 OA/BOA Initialization

The following PIDL specifies the operations (in the ORB interface) that allow applications to get pseudo object references; it is described in detail in Section 7.5, OA and BOA Initialization.

```
// PIDL

module CORBA {

    interface ORB
    {

        typedef sequence <string> arg_list;
        typedef string OAid;

        // Template for OA initialization operations
        // <OA> <OA>_init (inout arg_list argv,
        //                    in OAid oa_identifier);

        BOA BOA_init       (inout arg_list argv,
                           in OAid boa_identifier);

    };

 }
```

The mapping of the **OAinit** (**BOA_init**) operation (in the ORB interface) to the C programming language is as follows.

```
/* C language mapping */

typedef CORBA_string CORBA_OAid;

/* Template C binding for <OA>_init */
/*
CORBA_<OA> CORBA_ORB_<OA>_init (CORBA_ORB orb,
      int *argc,
      char **argv,
      CORBA_ORB_OAid boa_identifier,
      CORBA_Environment *env);
 */

CORBA_BOA CORBA_ORB_BOA_init (CORBA_ORB orb,
      int *argc,
      char **argv,
      CORBA_ORB_OAid boa_identifier,
      CORBA_Environment *env);
```

The **arglist** structure from the PIDL definition is replaced in the C mapping with **argv** and **argc** parameters. The **argv** parameters is an unbound array of strings (**char****) and the number of strings in the array is passed in the **argc** (**int***).

If a NULL OAid is used, then **argv** arguments can be used to determine which OA should be returned. This is achieved by searching the **argv** parameters for one tagged OAid, e.g. -OAid "OAid_example."

For C, the order of consumption of **argv** parameters may be significant to an application. In order to ensure that applications are not required to handle **argv** parameters, they do not recognize the OA initialization function must be called before the remainder of the parameters are consumed by the application. Therefore, after the **<OA>_init** call, the **argv** and **argc** parameters will have been modified to remove the OA understood arguments. It is important to note that the **OA_init** call can only reorder or remove references to parameters from the **argv** list; this restriction is made in order to avoid potential memory management problems caused by trying to free parts of the **argv** list or extending the **argv** list of parameters. This is why **argv** is passed as a **char**** and not a **char*****.

14.27 Operations for Obtaining Initial Object References

The following PIDL specifies the operations (in the ORB interface) that allow applications to get pseudo-object references for the Interface Repository and Object Services. It is described in detail in Section 7.6, Obtaining Initial Object References.

```
// PIDL interface for getting initial object references

module CORBA {
      interface ORB {
      typedef string ObjectId;
      typedef sequence <ObjectId> ObjectIdList;

      exception InvalidName {};

ObjectIdList list_initial_services ();

Object resolve_initial_references (in ObjectId identifier)
raises (InvalidName);
  }

}
```

The mapping of the preceding PIDL to C is as follows.

```
/* C Mapping */
typedef CORBA_string CORBA_ORB_ObjectId;
typedef CORBA_sequence_CORBA_ORB_ObjectId

                     CORBA_ORB_ObjectIdList;
typedef struct CORBA_ORB_InvalidName CORBA_ORB_InvalidName;

CORBA_ORB_ObjectIdList CORBA_ORB_list_initial_services (

                     CORBA_ORB orb,
                     CORBA_Environment *env);

CORBA_Object CORBA_ORB_resolve_initial_references (
                     CORBA_ORB orb,
                     CORBA_ORB_ObjectId identifier,
                     CORBA_Environment *env);
```

C++ Mapping Overview 15

This chapter explains how the C++ mapping was designed, and how it is organized in this manual.

15.1 Key Design Decisions

The design of the C++ mapping was driven by a number of considerations, including a design that achieves reasonable performance, portability, efficiency, and usability for OMG IDL-to-C++ implementations. Several other considerations are outlined in this section.

For more information about the general requirements of a mapping from OMG IDL to any programming language, refer to Section 14.1, Requirements for a Language Mapping.

15.1.1 Compliance

The C++ mapping tries to avoid limiting the implementation freedoms of ORB developers. For each OMG IDL and CORBA construct, the C++ mapping explains the syntax and semantics of using the construct from C++. A client or server program conforms to this mapping (is CORBA-C++ compliant) if it uses the constructs as described in the C++ mapping chapters. An implementation conforms to this mapping if it correctly executes any conforming client or server program. A conforming client or server program is therefore portable across all conforming implementations. For more information about CORBA compliance, refer to Section 0.6, Definition of CORBA Compliance.

15.1.2 C++ Implementation Requirements

The mapping proposed here assumes that the target C++ environment supports all the features described in *The Annotated C++ Reference Manual* (ARM) by Ellis and Stroustrup as adopted by the ANSI/ISO C++ standardization committees, including exception handling. In addition, it assumes that the C++ environment supports the **namespace** construct recently adopted into the language. Because C++ implementations vary widely in the quality of their support for templates, this mapping does not explicitly require their use, nor does it disallow their use as part of a CORBA-compliant implementation.

15.1.3 C Data Layout Compatibility

Some ORB vendors feel strongly that the C++ mapping should be able to work directly with the CORBA C mapping. This mapping makes every attempt to ensure compatibility between the C and C++ mappings, but it does not mandate such compatibility. In addition to providing better interoperability and portability, the C++ call style solves the memory management problems seen by C programmers who use the C call style. Therefore, the OMG has adopted the C++ call style for OMG IDL. However, to provide continuity for earlier applications, an implementation might choose to support the C call style as an option. If an implementation supports both call styles, it is recommended that the C call style be phased out.

Note that the mapping in Chapter 14, C Language Mapping, has been modified from *CORBA V1.2* to achieve compatibility between the C and C++ mappings.

15.1.4 No Implementation Descriptions

This mapping does not contain implementation descriptions. It avoids details that would constrain implementations, but still allows clients to be fully source compatible with any compliant implementation. Some examples show possible implementations, but these are not required implementations.

15.2 Organization of the C++ Mapping

In addition to this overview, the mapping of OMG IDL to the C++ programming language is divided into the following chapters:

- Mapping of all OMG IDL constructs (as defined in Chapter 3, OMG IDL Syntax and Semantics) to C++ constructs
- Mapping of OMG IDL pseudo-objects to C++
- Server-side mapping, which refers to the portability constraints for an object implementation written in C++

Three appendices are also included at the end of the C++ chapters. One appendix contains C++ definitions for the CORBA module; another contains C++ keywords for the CORBA module; and another contains workarounds for C++ dialects that do not match the assumptions specified in Section 15.1.2, C++ Implementation Requirements.

Mapping of OMG IDL to C++ 16

This chapter explains how OMG IDL constructs are mapped to the constructs of the C++ programming language. It provides mapping information for:

- Interfaces
- Constants
- Basic data types
- Enums
- Types (string, structure, struct, union, sequence, array, typedefs, any, exception)
- Operations and attributes
- Arguments

16.1 Preliminary Information

16.1.1 Scoped Names

Scoped names in OMG IDL are specified by C++ scopes:

- OMG IDL modules are mapped to C++ name spaces.
- OMG IDL interfaces are mapped to C++ classes (as described in Section 16.3, Mapping for Interfaces).
- All OMG IDL constructs scoped to an interface are accessed via C++ scoped names. For example, if a type **`mode`** were defined in interface **`printer,`** then the type would be referred to as **`printer::mode`**.

These mappings allow the corresponding mechanisms in OMG IDL and C++ to be used to build scoped names. For instance:

```
// IDL
module M
{
    struct E {
        long L;
    };
};
```

is mapped into:

```
// C++
namespace M
{
    struct E {
        Long L;
    };
}
```

and **E** can be referred outside of **M** as **M::E**. Alternatively, a C++ **using** statement for name space **M** can be used so that **E** can be referred to simply as **E**:

```
// C++
using namespace M;
E e;
e.L = 3;
```

Another alternative is to employ a **using** statement only for **M::E**:

```
// C++
using M::E;
E e;
e.L = 3;
```

To avoid C++ compilation problems, every use in OMG IDL of a C++ key word as an identifier is mapped into the same name preceded by an underscore. The list of C++ key words from the 05/27/94 working draft of the ANSI/ISO C++ standardization committees (X3J16, WG21) can be found in Appendix C, C++ Definitions for CORBA.

16.1.2 C++ Type Size Requirements

The sizes of the C++ types used to represent OMG IDL types are implementation-dependent. That is, this mapping makes no requirements as to the **sizeof(T)** for anything except basic types (see Section 16.5, Mapping for Basic Data Types) and string (see Section 16.7, Mapping for String Types).

16.1.3 CORBA Module

The mapping relies on some predefined types, classes, and functions that are logically defined in a module named **`CORBA`.** The module is automatically accessible from a C++ compilation unit that includes a header file generated from an OMG IDL specification. In the examples presented in this document, CORBA definitions are referenced without explicit qualification for simplicity. In practice, fully scoped names or C++ **`using`** statements for the **`CORBA`** name space would be required in the application source. See Appendix A, Standard OMG IDL Types.

16.2 Mapping for Modules

A shown in Section 16.1.1, Scoped Names, a module defines a scope, and as such is mapped to a C++ **`namespace`** with the same name:

```
// IDL
module M
{
    // definitions
};

// C++
namespace M
{
    // definitions
}
```

Because name spaces were only recently added to the C++ language, few C++ compilers currently support them. Alternative mappings for OMG IDL modules that do not require C++ name spaces are in Appendix D, Alternative Mappings for C++ Dialects.

16.3 Mapping for Interfaces

An interface is mapped to a C++ class that contains public definitions of the types, constants, operations, and exceptions defined in the interface.

A CORBA-C++-compliant program cannot

- Create or hold an instance of an interface class.
- Use a pointer (**`A*`**) or a reference (**`A&`**) to an interface class.

The reason for these restrictions is to allow a wide variety of implementations. For example, interface classes could not be implemented as abstract base classes if programs were allowed to create or hold instances of them. In a sense, the generated class is like a name space that one cannot enter via a **`using`** statement. This example shows the behavior of the mapping of an interface.

```
// IDL
interface A
{
    struct S { short field; };
};

// C++
// Conformant uses
A::S s; // declare a struct variable
s.field = 3; // field access

// Non-conformant uses:
// one cannot declare an instance of an interface class...
A a;
// ...nor declare a pointer to an interface class...
A *p;
// ...nor declare a reference to an interface class.
void f(A &r);
```

16.3.1 Object Reference Types

The use of an interface type in OMG IDL denotes an object reference. Because of the different ways an object reference can be used and the different possible implementations in C++, an object reference maps to two C++ types. For an interface **A**, these types are named **A_var** and **A_ptr**. For historical reasons, the type **ARef** is defined as a synonym for **A_ptr**, but usage of the **Ref** names is deprecated. These types need not be distinct—**A_var** may be identical to **A_ptr**, for example—so a compliant program cannot overload operations using these types solely.

An operation can be performed on an object by using an arrow (“**->**”) on a reference to the object. For example, if an interface defines an operation **op** with no parameters and **obj** is a reference to the interface type, then a call would be written **obj->op()**. The arrow operator is used to invoke operations on both the **_ptr** and **_var** object reference types.

Client code frequently will use the object reference variable type (**A_var**) because a variable will automatically release its object reference when it is deallocated or when assigned a new object reference. The pointer type (**A_ptr**) provides a more primitive object reference, which has similar semantics to a C++ pointer. Indeed, an implementation may choose to define **A_ptr** as **A***, but is not required to. Unlike C++ pointers, however, conversion to **void***, arithmetic operations, and relational operations, including test for equality, are all noncompliant. A compliant implementation need not detect these incorrect uses because requiring detection is not practical.

For many operations, mixing data of type **A_var** and **A_ptr** is possible without any explicit operations or casts. However, one needs to be careful in doing so because of the implicit release performed when the variable is deallocated. For example, the assignment statement in the following code will result in the object reference held by **p** to be released at the end of the block containing the declaration of **a**.

```
// C++
A_var a;
A_ptr p = // ...somehow obtain an objref...
a = p;
```

16.3.2 Widening Object References

OMG IDL interface inheritance does not require that the corresponding C++ classes are related, though that is certainly one possible implementation. However, if interface B inherits from interface A, the following implicit widening operations for B must be supported by a compliant implementation:

- **B_ptr** to **A_ptr**
- **B_ptr** to **Object_ptr**
- **B_var** to **A_ptr**
- **B_var** to **Object_ptr**

Implicit widening from a **B_var** to **A_var** or **Object_var** need not be supported; instead, widening between **_var** types for object references requires a call to **_duplicate** (described in Section 16.3.3, Object Reference Operations).[1] An attempt to implicitly widen from one **_var** type to another must cause a compile-time error.[2] Assignment between two **_var** objects of the same type is supported, but widening assignments are not and must cause a compile-time error. Widening assignments may be done using **_duplicate**.

```
// C++
B_ptr bp = ...
A_ptr ap = bp;          // implicit widening
Object_ptr objp = bp;   // implicit widening
objp = ap;              // implicit widening

B_var bv = bp;          // bv assumes ownership of bp
ap = bv;                // implicit widening, bv retains
ownership               // of bp
obp = bv;               // implicit widening, bv retains
ownership               // of bp

A_var av = bv;          // illegal, compile-time error
A_var av = B::_duplicate(bv);// av and bv both refer to bp
```

1. When **T_ptr** is mapped to **T***, it is impossible in C++ to provide implicit widening between **T_var** types while also providing the necessary duplication semantics for **T_ptr** types.

2. This can be achieved by deriving all **T_var** types for object references from a base **_var** class, then making the assignment operator for **_var** private within each **T_var** type.

```
B_var bv2 = bv;  // implicit _duplicate
A_var av2;
av2 = av;        // implicit _duplicate
```

16.3.3 Object Reference Operations

Conceptually, the **Object** class in the **CORBA** module is the base interface type for all CORBA objects. Any object reference can therefore be widened to the type **Object_ptr**. As with other interfaces, the CORBA name space also defines the type **Object_var**.

CORBA defines three operations on any object reference: **duplicate**, **release**, and **is_nil**. Note that these are operations on the object reference, not the object implementation. Because the mapping does not require object references to be C++ objects themselves, the "**->**" syntax cannot be employed to express the usage of these operations. Also, for convenience these operations are allowed to be performed on a nil object reference.

The **release** and **is_nil** operations depend only on type **Object**, so they can be expressed as regular functions within the CORBA name space as follows:

```
// C++
void release(Object_ptr obj);
Boolean is_nil(Object_ptr obj);
```

The **release** operation indicates that the caller will no longer access the reference so that associated resources may be deallocated. If the given object reference is nil, **release** does nothing. The **is_nil** operation returns **TRUE** if the object reference contains the special value for a nil object reference as defined by the ORB. Neither the **release** operation nor the **is_nil** operation may throw CORBA exceptions.

The **duplicate** operation returns a new object reference with the same static type as the given reference. The mapping for an interface therefore includes a static member function name **_duplicate** in the generated class. For example:

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

```
// C++
class A
{
  public:
     static A_ptr _duplicate(A_ptr obj);
};
```

If the given object reference is nil, **_duplicate** will return a nil object reference. The **_duplicate** operation can throw CORBA system exceptions.

16.3.4 Narrowing Object References

The mapping for an interface defines a static member function named **_narrow** that returns a new object reference given an existing reference. Like **_duplicate**, the **_narrow** function returns a nil object reference if the given reference is nil. Unlike **_duplicate**, the parameter to **_narrow** is a reference of an object of any interface type (**Object_ptr**). If the actual (run-time) type of the parameter object can be widened to the requested interface's type, then **_narrow** will return a valid object reference. Otherwise, **_narrow** will return a nil object reference. For example, suppose A, B, C, and D are interface types, and D inherits from C, which inherits from B, which in turn inherits from A. If an object reference to a C object is widened to an **A_ptr** variable called **ap**, the

- **A::_narrow(ap)** returns a valid object reference;
- **B::_narrow(ap)** returns a valid object reference;
- **C::_narrow(ap)** returns a valid object reference;
- **D::_narrow(ap)** returns a nil object reference.

Narrowing to A, B, and C all succeed because the object supports all those interfaces. The **D::_narrow** returns a nil object reference because the object does not support the D interface.

If successful, the **_narrow** function creates a new object reference and does not consume the given object reference, so the caller is responsible for releasing both the original and new references.

For example, suppose A, B, C, and D are interface types. C inherits from B, and both B and D inherit from A. Now suppose that an object of type C is passed to a function as an A. If the function calls **B::_narrow** or **C::_narrow**, a new object reference will be returned. A call to **D::_narrow** will fail and return nil.

The **_narrow** operation can throw CORBA system exceptions.

16.3.5 Nil Object Reference

The mapping for an interface defines a static member function named **_nil** that returns a nil object reference of that interface type. For each interface A, the following call is guaranteed to return **TRUE**:

```
// C++
Boolean true_result = is_nil(A::_nil());
```

A compliant application need not call **release** on the object reference returned from the **_nil** function.

As described in Section 16.3.1, Object Reference Types, object references may not be compared using **operator==**, so **is_nil** is the only compliant way an object reference can be checked to see if it is nil.

The **_nil** function may not throw any CORBA exceptions.

A compliant program cannot attempt to invoke an operation through a nil object reference, since a valid C++ implementation of a nil object reference is a null pointer.

16.3.6 Interface Mapping Example

The following example shows one possible mapping for an interface. Other mappings are also possible, but they must provide the same semantics and usage as this example.

```
// IDL
interface A
{
  A op(in A param);
};

// C++
class A;
typedef A *A_ptr;
typedef A_ptr ARef;
class A : public virtual Object
{
  public:
     static A_ptr _duplicate(A_ptr obj);
     static A_ptr _narrow(Object_ptr obj);
     static A_ptr _nil();

     virtual A_ptr op(A_ptr param) = 0;

  protected:
     A();
     virtual ~A();

  private:
     A(const A&);
     void operator=(const A&);
};

class A_var : public _var
{
  public:
     A_var() : ptr_(A::_nil()) {}
     A_var(A_ptr p) : ptr_(p) {}
     A_var(const A_var &a) : ptr_(A::_duplicate(A_ptr(a))) {}
     ~A_var() { free(); }

     A_var &operator=(A_ptr p) {
          reset(p); return *this;
     }

     operator const A_ptr&() const { return ptr_; }
     operator A_ptr&() { return ptr_; }
```

```
    A_ptr operator->() const { return ptr_; }

  protected:
    A_ptr ptr_;
    void free() { release(ptr_); }
    void reset(A_ptr p) { free(); ptr_ = p; }

  private:
    // hidden assignment operators for var types to
    // fulfill the rules specified in Section 16.3.2
    void operator=(const A_var &);
    void operator=(const _var &);
};
```

16.4 Mapping for Constants

OMG IDL constants are mapped directly to a C++ constant definition that may or may not define storage depending on the scope of the declaration. In the following example, a top-level OMG IDL constant maps to a file-scope C++ constant whereas a nested constant maps to a class-scope C++ constant. This inconsistency occurs because C++ file-scope constants may not require storage (or the storage may be replicated in each compilation unit), while class-scope constants always take storage. As a side effect, this difference means that the generated C++ header file might not contain values for constants defined in the OMG IDL file.

```
// IDL
const string name = "testing";

interface A
{
    const float pi = 3.14159;
};
```

```
// C++
static const char *const name = "testing";

class A
{
  public:
    static const Float pi;
};
```

In certain situations, use of a constant in OMG IDL must generate the constant's value instead of the constant's name.[3] For example,

3. A recent change made to the C++ language by the ANSI/ISO C++ standardization committees allows static integer constants to be initialized within the class declaration, so for some C++ compilers, the code generation issues described here may not be a problem.

```
// IDL
interface A
{
    const long n = 10;
    typedef long V[n];
};

// C++
class A
{
  public:
    static const long n;
    typedef long V[10];
};
```

16.5 Mapping for Basic Data Types

The basic data types have the mappings shown in Table 16-1. Note that the mapping of the OMG IDL **`boolean`** type defines only the values 1 (**`TRUE`**) and 0 (**`FALSE`**); other values produce undefined behavior.

Table 16-1 Basic Data Type Mappings

OMG IDL	C++
short	CORBA::Short
long	CORBA::Long
unsigned short	CORBA::UShort
unsigned long	CORBA::ULong
float	CORBA::Float
double	CORBA::Double
char	CORBA::Char
boolean	CORBA::Boolean
octet	CORBA::Octet

Each OMG IDL basic type is mapped to a typedef in the CORBA module. This is because some types, such as **`short`** and **`long`**, may have different representations on different platforms, and the CORBA definitions will reflect the appropriate representation. For example, on a 64-bit machine where a long integer is 64 bits, the definition of **`CORBA::Long`** would still refer to a 32-bit integer. Requirements for the sizes of basic types are shown in Section 3.8.1, Basic Types.

Except for **`boolean`**, **`char`**, and **`octet`**, the mappings for basic types must be distinguishable from each other for the purposes of overloading. That is, one can safely write overloaded C++ functions on **`Short`**, **`UShort`**, **`Long`**, **`ULong`**, **`Float`**, and **`Double`**.

Programmers concerned with portability should use the CORBA types. However, some may feel that using these types with the CORBA qualification impairs readability. If the **`CORBA`** module is mapped to a name space, a C++ **`using`** statement may help this problem. On platforms where the C++ data type is guaranteed to be identical to the OMG IDL data type, a compliant implementation therefore may generate the native C++ type.

For the **Boolean** type, only the values 1 (representing **TRUE**) and 0 (representing **FALSE**) are defined; other values produce undefined behavior. Since many existing C++ software packages and libraries already provide their own preprocessor macro definitions of **TRUE** and **FALSE**, this mapping does not require that such definitions be provided by a compliant implementation. Requiring definitions for **TRUE** and **FALSE** could cause compilation problems for CORBA applications that make use of such packages and libraries. Instead, we recommend that compliant applications simply use the values 1 and 0 directly.[4] Alternatively, for those C++ compilers that support the new **bool** type, the key words **TRUE** and **FALSE** may be used.

16.6 Mapping for Enums

An OMG IDL **enum** maps directly to the corresponding C++ type definition. The only difference is that the generated C++ type may need an additional constant that is large enough to force the C++ compiler to use exactly 32 bits for values declared to be of the enumerated type.

```
// IDL
enum Color { red, green, blue };

// C++
enum Color { red, green, blue };
```

16.7 Mapping for String Types

As in the C mapping, the OMG IDL string type, whether bounded or unbounded, is mapped to **char*** in C++. String data is null-terminated. In addition, the **CORBA** module defines a class **String_var** that contains a **char*** value and automatically frees the pointer when a **String_var** object is deallocated. When a **String_var** is constructed or assigned from a **char***, the **char*** is consumed and thus the string data may no longer be accessed through it by the caller. Assignment or construction from a **const char*** or from another **String_var** causes a copy. The **String_var** class also provides operations to convert to and from **char*** values, as well as subscripting operations to access characters within the string. The full definition of the **String_var** interface is given in Section C.2, **String_var** Class.

Because its mapping is **char***, the OMG IDL string type is the only nonbasic type for which this mapping makes size requirements.

4.Examples and descriptions in this document still use **TRUE** and **FALSE** for purposes of clarity.

For dynamic allocation of strings, compliant programs must use the following functions from the **CORBA** name space:

```
// C++
namespace CORBA {
    char *string_alloc(ULong len);
    char *string_dup(const char*);
    void string_free(char *);
    ...
}
```

The **string_alloc** function dynamically allocates a string, or returns a null pointer if it cannot perform the allocation. It allocates **len+1** characters so that the resulting string has enough space to hold a trailing NULL character. The **string_dup** function dynamically allocates enough space to hold a copy of its string argument, including the NULL character, copies its string argument into that memory, and returns a pointer to the new string. If allocation fails, a null pointer is returned. The **string_free** function deallocates a string that was allocated with **string_alloc** or **string_dup**. Passing a null pointer to **string_free** is acceptable and results in no action being performed. These functions allow ORB implementations to use special memory management mechanisms for strings if necessary, without forcing them to replace global **operator new** and **operator new[]**.

The **string_alloc**, **string_dup**, and **string_free** functions may not throw CORBA exceptions.

Note that a static array of char in C++ decays to a **char***, so care must be taken when assigning one to a **String_var**, since the **String_var** will assume the pointer points to data allocated via **string_alloc,** and thus will eventually attempt to **string_free** it:

```
// C++
// The following is an error, since the char* should point
// to data allocated via string_alloc so it can be consumed
String_var s = “static string”;// error

// The following are OK, since const char* are copied,
// not consumed
const char* sp = “static string”;
s = sp;
s = (const char*)“static string too”;
```

16.8 Mapping for Structured Types

The mapping for **struct**, **union**, and **sequence** (but not **array**) is a C++ struct or class with a default constructor, a copy constructor, an assignment operator, and a destructor. The default constructor initializes object reference members to appropriately-typed nil object references and string members to NULL; all other members are initialized via their default constructors. The copy constructor performs a deep-copy from the existing structure to create a new structure, including calling

_duplicate on all object reference members and performing the necessary heap allocations for all string members. The assignment operator first releases all object reference members and frees all string members, and then performs a deep-copy to create a new structure. The destructor releases all object reference members and frees all string members.

The mapping for OMG IDL structured types (structs, unions, arrays, and sequences) can vary slightly depending on whether the data structure is *fixed-length* or *variable-length*. A type is *variable-length* if it is one of the following types:

- The type **any**
- A bounded or unbounded string
- A bounded or unbounded sequence
- An object reference or reference to a transmissible pseudo-object[5]
- A struct or union that contains a member whose type is variable-length
- An array with a variable-length element type
- A typedef to a variable-length type

The reason for treating fixed- and variable-length data structures differently is to allow more flexibility in the allocation of **out** parameters and return values from an operation. This flexibility allows a client-side stub for an operation that returns a sequence of strings, for example, to allocate all the string storage in one area that is deallocated in a single call.

The mapping of a variable-length type as an **out** parameter or operation return value is a pointer to the associated class or array. As a convenience for managing this pointer, the mapping also provides another class for each variable-length type. This type, which is named by adding the suffix **_var** to the original type's name, automatically deletes the pointer when an instance is destroyed. An object of type **T_var** behaves similarly to the structured type **T**, except that members must be accessed indirectly. For a struct, this means using an arrow ("**->**") instead of a dot ("**.**").

```
// IDL
struct S { string name; float age; };
void f(out S p);

// C++
S a;
S_var b;
f(b);
a = b; // deep-copy
cout << "names " << a.name << ", " << b->name << endl;
```

5. Transmissible pseudo-objects are listed as "general arguments" in Table 14, Pseudo-Objects, in Appendix A.

16.8.1 *T_var Types*

The general form of the **T_var** types is shown next.

```
// C++
class T_var
{
  public:
     T_var();
     T_var(T *);
     T_var(const T_var &);
     ~T_var();

     T_var &operator=(T *);
     T_var &operator=(const T_var &);

     T *operator-> const ();
     // other conversion operators to support
     // parameter passing
};
```

The default constructor creates a **T_var** containing a null **T***. Compliant applications may not attempt to convert a **T_var** created with the default constructor into a **T*** nor use its overloaded **operator->** without first assigning to it a valid **T*** or another valid **T_var**. Due to the difficulty of doing so, compliant implementations are not required to detect this error. Conversion of a null **T_var** to a **T*&** is allowed, however, so that a **T_var** can legally be passed as an **out** parameter.

The **T*** constructor creates a **T_var** that, when destroyed, will **delete** the storage pointed to by the **T*** parameter. The parameter to this constructor should never be a null pointer. Compliant implementations are not required to detect null pointers passed to this constructor.

The copy constructor deep-copies any data pointed to by the **T_var** constructor parameter. This copy will be destroyed when the **T_var** is destroyed or when a new value is assigned to it. Compliant implementations may, but are not required to, utilize some form of reference counting to avoid such copies.

The destructor uses **delete** to deallocate any data pointed to by the **T_var**, except for strings and array types, which are deallocated using the **string_free** and **T_free** (for array type **T**) deallocation functions, respectively.

The **T*** assignment operator results in the deallocation of any old data pointed to by the **T_var** before assuming ownership of the **T*** parameter.

The normal assignment operator deep-copies any data pointed to by the **T_var** assignment parameter. This copy will be destroyed when the **T_var** is destroyed or when a new value is assigned to it.

The overloaded **operator->** returns the **T*** held by the **T_var**, but retains ownership of it. Compliant applications may not call this function unless the **T_var** has been initialized with a valid **T*** or **T_var**.

In addition to the member functions described above, the **T_var** types must support conversion functions that allow them to fully support the parameter passing modes shown in Table 16-2. The form of these conversion functions is not specified so as to allow different implementations, but the conversions must be automatic (i.e., they must require no explicit application code to invoke them).

The **T_var** types are also produced for fixed-length structured types for reasons of consistency. These types have the same semantics as **T_var** types for variable-length types. This allows applications to be coded in terms of **T_var** types regardless of whether the underlying types are fixed- or variable-length.

Each **T_var** type must be defined at the same level of nesting as its **T** type.

T_var types do not work with a pointer to constant **T**, since they provide no constructor nor **operator=** taking a **const T*** parameter. Since C++ does not allow **delete** to be called on a **const T***, the **T_var** object would normally have to copy the const object; instead, the absence of the **const T*** constructor and assignment operators will result in a compile-time error if such an initialization or assignment is attempted. This allows the application developer to decide if a copy is really wanted or not. Explicit copying of **const T*** objects into **T_var** types can be achieved via the copy constructor for **T**:

```
// C++
const T *t = ...;
T_var tv = new T(*t);
```

16.9 Mapping for Struct Types

An OMG IDL struct maps to C++ struct, with each OMG IDL struct member mapped to a corresponding member of the C++ struct. This mapping allows simple field access as well as aggregate initialization of most fixed-length structs. To facilitate such initialization, C++ structs must not have user-defined constructors, assignment operators, or destructors, and each struct member must be of self-managed type. With the exception of strings and object references, the type of a C++ struct member is the normal mapping of the OMG IDL member's type.

For a string or object reference member, the name of the C++ member's type is not specified by the mapping. Therefore, a compliant program cannot create an object of that type. The behavior[6] of the type is the same as the normal mapping (**char*** for string, **A_ptr** for an interface A) except the type's copy constructor copies the member's storage, and its assignment operator releases the member's old storage.

6. Those implementations concerned with data layout compatibility with the C mapping in this manual will also want to ensure that the sizes of these members match those of their C mapping counterparts.

Assignment between a string or object reference member and a corresponding **T_var** type (**String_var** or **A_var**) always results in copying the data, while assignment with a pointer does not. The one exception to the rule for assignment is when a **const char*** is assigned to a member, in which case the storage is copied.

When the old storage must not be freed (for example, it is part of the function's activation record), one can access the member directly as a pointer using the **_ptr** field accessor. This usage is dangerous and generally should be avoided.

```
// IDL
struct Fixed{ float x, y, z; };

// C++
Fixed x1 = {1.2, 2.4, 3.6};
Fixed_var x2 = new Fixed;
x2->y = x1.z;
```

The previous example shows usage of the **T** and **T_var** types for a fixed-length struct. When it goes out of scope, **x2** will automatically free the heap-allocated **Fixed** object using **delete**.

The following examples illustrate mixed usage of **T** and **T_var** types for variable-length types, using the following OMG IDL definition.

```
// IDL
interface A;
struct Variable { string name; };

// C++
Variable str1;          // str1.name is initially NULL
Variable_var str2 = new Variable;// str2->name is initially
NULL
char *non_const;
const char *const2;
String_var string_var;
const char *const3 = "string 1";
const char *const4 = "string 2";

str1.name = const3;     // 1: free old storage, copy
str2->name = const4;    // 2: free old storage, copy
```

In the previous example, the **name** components of variables **str1** and **str2** both start out as null. On the line marked 1, **const3** is assigned to the **name** component of **str1**; this results in the previous **str1.name** being freed, and since **const3** points to const data, the contents of **const3** being copied. In this case, **str1.name** started out as null, so no previous data needs to be freed before the copying of **const3** takes place. Line 2 is similar to line 1, except that **str2** is a **T_var** type.

Continuing with the example:

```
// C++
non_const = str1.name;  // 3: no free, no copy
const2 = str2->name;    // 4: no free, no copy
```

On the line marked 3, **str1.name** is assigned to **non_const**. Since **non_const** is a pointer type (**char***), **str1.name** is not freed, nor are the data it points to copied. After the assignment, **str1.name** and **non_const** effectively point to the same storage, with **str1.name** retaining ownership of that storage. Line 4 is identical to line 3, even though **const2** is a pointer to const char; **str2->name** is neither freed nor copied because **const2** is a pointer type.

```
// C++
str1.name = non_const;  // 5: free, no copy
str1.name = const2;     // 6: free old storage, copy
```

Line 5 involves assignment of a **char*** to **str1.name**, which results in the old **str1.name** being freed and the value of the **non_const** pointer, but not the data it points to, being copied. In other words, after the assignment, **str1.name** points to the same storage as **non_const** points to. Line 6 is the same as line 5 except that because **const2** is a **const char***, the data it points to are copied.

```
// C++
str2->name = str1.name; // 7: free old storage, copy
str1.name = string_var; // 8: free old storage, copy
string_var = str2->name;// 9: free old storage, copy
```

On line 7, assignment is performed to a member from another member, so the original value of the left-hand member is freed and the new value is copied. Similarly, lines 8 and 9 involve assignment to or from a **String_var**, so in both cases the original value of the left-hand side is freed and the new value is copied.

```
// C++
str1.name._ptr = str2.name;// 10: no free, no copy
```

Finally, line 10 uses the **_ptr** field accessor, so no freeing or copying takes place. Such usage is dangerous and generally should be avoided.

ORB implementations concerned with single-process interoperability with the C mapping may overload **operator new()** and **operator delete()** for structs so that dynamic allocation uses the same mechanism as the C language dynamic allocation functions. Whether these operators are overloaded by the implementation or not, compliant programs use **new** to dynamically allocate structs and **delete** to free them.

16.10 Mapping for Union Types

Unions map to C++ classes with access functions for the union members and discriminant. The default union constructor performs no application-visible initialization of the union. It does not initialize the discriminator, nor does it initialize any union members to a state useful to an application. (The implementation of the default constructor can do whatever type of initialization it wants to, but such initialization is implementation-dependent. No compliant application can count on a union ever being properly initialized by the default constructor alone.)

It is therefore an error for an application to access the union before setting it, but ORB implementations are not required to detect this error due to the difficulty of doing so. The copy constructor and assignment operator both perform a deep-copy of their parameters, with the assignment operator releasing old storage if necessary. The destructor releases all storage owned by the union.

The union discriminant access functions have the name **_d** to both be brief and avoid name conflicts with the members. The **_d** discriminator modifier function can only be used to set the discriminant to a value within the same union member. In addition to the **_d** accessors, a union with an implicit default member provides a **`_default()`** member function that sets the discriminant to a legal default value. A union has an implicit default member if it does not have a default case and not all permissible values of the union discriminant are listed.

Setting the union value through an access function automatically sets the discriminant and may release the storage associated with the previous value. Attempting to get a value through an access function that does not match the current discriminant results in undefined behavior. If an access function for a union member with multiple legal discriminant values is used to set the value of the discriminant, the union implementation is free to set the discriminant to any one of the legal values for that member. The actual discriminant value chosen under these circumstances is implementation dependent.

The following example helps illustrate the mapping for union types:

```
// IDL
typedef octet Bytes[64];
struct S { long len; };
interface A;
union U switch (long) {
    case 1: long x;
    case 2: Bytes y;
    case 3: string z;
    case 4:
    case 5: S w;
    default: A obj;
};
```

```
// C++
typedef Octet Bytes[64];
typedef Octet Bytes_slice;
class Bytes_forany { ... };
struct S { Long len; };
typedef ... A_ptr;
class U
{
  public:
    U();
    U(const U&);
    ~U();
    U &operator=(const U&);

    void _d(Long);
    Long _d() const;

    void x(Long);
    Long x() const;

    void y(Bytes);
    Bytes_slice *y() const;

    void z(char*);      // free old storage, no copy
    void z(const char*);// free old storage, copy
    void z(const String_var &);// free old storage, copy
    const char *z() const;

    void w(const S &);  // deep copy
    const S &w() const; // read-only access
    S &w();             // read-write access

    void obj(A_ptr);    // release old objref, duplicate
    A_ptr obj() const;  // no duplicate
};
```

Accessor and modifier functions for union members provide semantics similar to that of struct data members. Modifier functions perform the equivalent of a deep-copy of their parameters, and their parameters should be passed by value (for small types) or by reference to const (for larger types). Accessors that return a reference to a non-const object can be used for read-write access, but such accessors are only provided for the following types: **struct**, **union**, **sequence**, and **any**.

For an array union member, the accessor returns a pointer to the array slice, where the slice is an array with all dimensions of the original except the first (array slices are described in detail in Section 16.12, Mapping for Array Types). The array slice return type allows for read-write access for array members via regular subscript operators. For members of an anonymous array type, supporting typedefs for the array must be generated directly into the union. For example:

```
// IDL
union U switch (long) {
  default: long array[20][20];
};

// C++
class U
{
    public:
       // ...
       void array(long arg[20][20]);
       typedef long _array_slice[20];
       _array_slice * array();
       // ...
};
```

The name of the supporting array slice typedef is created by prepending an underscore and appending **_slice** to the union member name. In the previous example, the array member named “array” results in an array slice typedef called **_array_slice** nested in the union class.

For string union members, the **char*** modifier results in the freeing of old storage before ownership of the pointer parameter is assumed, while the **const char*** modifier and the **String_var** modifier[7] both result in the freeing of old storage before the parameter’s storage is copied. The accessor for a string member returns a **const char*** to allow examination, but not modification, of the string storage.[8]

For object reference union members, object reference parameters to modifier functions are duplicated after the old object reference is released. An object reference return value from an accessor function is not duplicated because the union retains ownership of the object reference.

The restrictions for using the **_d** discriminator modifier function are shown by the following examples, based on the definition of the union **U**, previously shown.

7. A separate modifier for **String_var** is needed because it can automatically convert to both a **char*** and a **const char***; since unions provide modifiers for both of these types, an attempt to set a string member of a union from a **String_var** would otherwise result in an ambiguity error at compile time.

8. A return type of **char*** allowing read-write access could mistakenly be assigned to a **String_var**, resulting in the **String_var** and the union both assuming ownership for the string’s storage.

```
// C++
S s = {10};
U u;
u.w(s);            // member w selected
u._d(4);           // OK, member w selected
u._d(5);           // OK, member w selected
u._d(1);           // error, different member selected
A_ptr a = ...;
u.obj(a);          // member obj selected
u._d(7);           // OK, member obj selected
u._d(1);           // error, different member selected
```

As shown here, the **_d** modifier function cannot be used to implicitly switch between different union members. The following shows an example of how the **_default()** member function is used.

```
// IDL
union Z switch(boolean) {
   case TRUE: short s;
};
```

```
// C++
Z z;
z._default();              // implicit default member selected
Boolean disc = z._d();     // disc == FALSE
U u;                       // union U from previous example
u._default();              // error, no _default() provided
```

For union **Z**, calling the **_default()** member function causes the union's value to be composed solely of the discriminator value of **FALSE**, since there is no explicit default member. For union **U**, calling **_default()** causes a compilation error because **U** has an explicitly declared default case and thus no **_default()** member function. A **_default()** member function is only generated for unions with implicit default members.

ORB implementations concerned with single-process interoperability with the C mapping may overload **operator new()** and **operator delete()** for unions so that dynamic allocation uses the same mechanism as the C language dynamic allocation functions. Whether these operators are overloaded by the implementation or not, compliant programs use **new** to dynamically allocate unions and **delete** to free them.

16.11 Mapping for Sequence Types

A sequence is mapped to a C++ class that behaves like an array with a current length and a maximum length. For a bounded sequence, the maximum length is implicit in the sequence's type and cannot be explicitly controlled by the programmer. For an unbounded sequence, the initial value of the maximum length can be specified in the sequence constructor to allow control over the size of the initial buffer allocation. The programmer may always explicitly modify the current length of any sequence.

For an unbounded sequence, setting the length to a larger value than the current length may reallocate the sequence data. Reallocation is conceptually equivalent to creating a new sequence of the desired new length, copying the old sequence elements *zero through length-1* into the new sequence, and then assigning the old sequence to be the same as the new sequence. Setting the length to a smaller value than the current length does not affect how the storage associated with the sequence is manipulated. Note, however, that the elements orphaned by this reduction are no longer accessible and that their values cannot be recovered by increasing the sequence length to its original value.

For a bounded sequence, attempting to set the current length to a value larger than the maximum length given in the OMG IDL specification produces undefined behavior.

For each different named OMG IDL sequence type, a compliant implementation provides a separate C++ sequence type. For example:

```
// IDL
typedef sequence<long> LongSeq;
typedef sequence<LongSeq, 3> LongSeqSeq;

// C++
class LongSeq           // unbounded sequence
{
  public:
     LongSeq();          // default constructor
     LongSeq(ULong max); // maximum constructor
     LongSeq(            // T *data constructor
          ULong max,
          ULong length,
          Long *value,
          Boolean release = FALSE
     );
     LongSeq(const LongSeq&);
     ~LongSeq();
     ...
};

class LongSeqSeq        // bounded sequence
{
  public:
     LongSeqSeq();       // default constructor
     LongSeqSeq(         // T *data constructor
          ULong length,
          LongSeq *value,
          Boolean release = FALSE
     );
     LongSeqSeq(const LongSeqSeq&);
     ~LongSeqSeq();
     ...
};
```

For both bounded and unbounded sequences, the default constructor (as shown in the previous example) sets the sequence length equal to zero. For bounded sequences, the maximum length is part of the type and cannot be set or modified, while for unbounded sequences, the default constructor also sets the maximum length to zero. The default constructor for a bounded sequence always allocates a contents vector, so it always sets the **`release`** flag to **`TRUE`**.

Unbounded sequences provide a constructor that allows only the initial value of the maximum length to be set (the "maximum constructor" shown in the previous example). This allows applications to control how much buffer space is initially allocated by the sequence. This constructor also sets the length to zero and the **`release`** flag to **`TRUE`**.

The "**`T *data`**" constructor (as shown in the previous example) allows the length and contents of a bounded or unbounded sequence to be set. For unbounded sequences, it also allows the initial value of the maximum length to be set. For this constructor, ownership of the contents vector is determined by the **`release`** parameter—**`FALSE`** means the caller owns the storage, while **`TRUE`** means that the sequence assumes ownership of the storage. If **`release`** is **`TRUE`**, the contents vector must have been allocated using the sequence **`allocbuf`** function, and the sequence will pass it to **`freebuf`** when finished with it. The **`allocbuf`** and **`freebuf`** functions are described in Section 16.11.3, Additional Memory Management Functions.

The copy constructor creates a new sequence with the same maximum and length as the given sequence, copies each of its current elements (items *zero* through *length–1*), and sets the **`release`** flag to **`TRUE`**.

The assignment operator deep-copies its parameter, releasing old storage if necessary. It behaves as if the original sequence is destroyed via its destructor and then the source sequence copied using the copy constructor.

If **`release=TRUE`**, the destructor destroys each of the current elements (items *zero* through *length–1*).

For an unbounded sequence, if a reallocation is necessary due to a change in the length and the sequence was created using the **`release=TRUE`** parameter in its constructor, the sequence will deallocate the old storage. If **`release`** is **`FALSE`** under these circumstances, old storage will not be freed before the reallocation is performed. After reallocation, the **`release`** flag is always set to **`TRUE`**.

For an unbounded sequence, the **`maximum()`** accessor function returns the total amount of buffer space currently available. This allows applications to know how many items they can insert into an unbounded sequence without causing a reallocation to occur. For a bounded sequence, **`maximum()`** always returns the bound of the sequence as given in its OMG IDL type declaration.

The overloaded subscript operators (**`operator[]`**) return the item at the given index. The non-const version must return something that can serve as an lvalue (i.e., something that allows assignment into the item at the given index), while the const version must allow read-only access to the item at the given index.

The overloaded subscript operators may not be used to access or modify any element beyond the current sequence length. Before either form of **`operator[]`** is used on a sequence, the length of the sequence must first be set using the **`length(ULong)`** modifier function, unless the sequence was constructed using the **`T *data`** constructor.

For strings and object references, **`operator[]`** for a sequence must return a type with the same semantics as the types used for string and object reference members of structs and arrays, so that assignment to the string or object reference sequence member via **`operator=()`** will release old storage when appropriate. Note that whatever these special return types are, they must honor the setting of the **`release`** parameter in the **`T *data`** constructor with respect to releasing old storage.

For the **`T *data`** sequence constructor, the type of **`T`** for strings and object references is **`char*`** and **`T_ptr`**, respectively. In other words, string buffers are passed as **`char**`** and object reference buffers are passed as **`T_ptr*`**.

16.11.1 Sequence Example

The next example shows full declarations for both a bounded and an unbounded sequence.

```
// IDL
typedef sequence<T> V1;       // unbounded sequence
typedef sequence<T, 2> V2;    // bounded sequence

// C++
class V1                      // unbounded sequence
{
  public:
     V1();
     V1(ULong max);
     V1(ULong max, ULong length, T *data,
          Boolean release =FALSE);
     V1(const V1&);
     ~V1();
     V1 &operator=(const V1&);

     ULong maximum() const;

     void length(ULong);
     ULong length() const;

     T &operator[](ULong index);
     const T &operator[](ULong index) const;
};

class V2                      // bounded sequence
{
  public:
     V2();
     V2(ULong length, T *data, Boolean release = FALSE);
     V2(const V2&);
     ~V2();
     V2 &operator=(const V2&);

     ULong maximum() const;

     void length(ULong);
     ULong length() const;

     T &operator[](ULong index);
     const T &operator[](ULong index) const;
};
```

16.11.2 Using the "release" Constructor Parameter

Consider the following example:

```
// IDL
typedef sequence<string, 3> StringSeq;

// C++
char *static_arr[] = {"one", "two", "three"};
char **dyn_arr = StringSeq::allocbuf(3);
dyn_arr[0] = string_dup("one");
dyn_arr[1] = string_dup("two");
dyn_arr[2] = string_dup("three");

StringSeq seq1(3, static_arr);
StringSeq seq2(3, dyn_arr, TRUE);

seq1[1] = "2";          // no free, no copy
char *str = string_dup("2");
seq2[1] = str;          // free old storage, no copy
```

In this example, both **seq1** and **seq2** are constructed using user-specified data, but only **seq2** is told to assume management of the user memory (because of the **release=TRUE** parameter in its constructor). When assignment occurs into **seq1[1]**, the right-hand side is not copied, nor is anything freed because the sequence does not manage the user memory. When assignment occurs into **seq2[1]**, however, the old user data must be freed before ownership of the right-hand side can be assumed, since **seq2** manages the user memory. When **seq2** goes out of scope, it will call **string_free** for each of its elements and **freebuf** on the buffer given to it in its constructor.

When the **release** flag is set to **TRUE** and the sequence element type is either a string or an object reference type, the sequence will individually release each element before releasing the contents buffer. It will release strings using **string_free**, and it will release object references using the **release** function from the **CORBA** name space.

In general, assignment should never take place in a sequence element via **operator[]** unless **release=TRUE** due to the possibility of memory management errors. In particular, a sequence constructed with **release=FALSE** should never be passed as an **inout** parameter because the callee has no way to determine the setting of the **release** flag, and thus must always assume that **release** is set to **TRUE**. Code that creates a sequence with **release=FALSE** and then knowingly and correctly manipulates it in that state as shown with **seq1** in the previous example is compliant, but care should always be taken to avoid memory leaks under these circumstances.

As with other **out** and return values, **out** and return sequences must not be assigned to by the caller without first copying them. This is more fully explained in Section 16.18, Argument Passing Considerations.

When a sequence is constructed with **release=TRUE**, a compliant application should make no assumptions about the continued lifetime of the data buffer passed to the constructor, since a compliant sequence implementation is free to copy the buffer and immediately free the original pointer.

16.11.3 Additional Memory Management Functions

ORB implementations concerned with single-process interoperability with the C mapping may overload **operator new()** and **operator delete()** for sequences so that dynamic allocation uses the same mechanism as the C language dynamic allocation functions. Whether these operators are overloaded by the implementation or not, compliant programs use **new** to dynamically allocate sequences, and **delete** to free them.

Sequences also provide additional memory management functions for their buffers. For a sequence of type **T**, the following static member functions are provided in the sequence class public interface.

```
// C++
static T *allocbuf(ULong nelems);
static void freebuf(T *);
```

The **allocbuf** function allocates a vector of **T** elements that can be passed to the **T *data** constructor. The length of the vector is given by the **nelems** function argument. The **allocbuf** function initializes each element using its default constructor except for strings, which are initialized to null pointers, and object references, which are initialized to suitably typed nil object references. A null pointer is returned if **allocbuf** for some reason cannot allocate the requested vector. Vectors allocated by **allocbuf** should be freed using the **freebuf** function. The **freebuf** function ensures that the destructor for each element is called before the buffer is destroyed except for string elements, which are freed using **`string_free()`**, and object reference elements, which are freed using **`release()`**. The **`freebuf`** function will ignore null pointers passed to it. Neither **`allocbuf`** nor **`freebuf`** may throw CORBA exceptions.

16.11.4 *Sequence* ***T_var*** *Type*

In addition to the regular operations defined for **`T_var`** types, the **`T_var`** for a sequence type also supports an overloaded **`operator[]`** that forwards requests to the **`operator[]`** of the underlying sequence.[9] This subscript operator should have the same return type as that of the corresponding operator on the underlying sequence type.

16.12 *Mapping for Array Types*

Arrays are mapped to the corresponding C++ array definition, which allows the definition of statically initialized data using the array. If the array element is a string or an object reference, then the mapping uses the same type as for structure members. That is, assignment to an array element will release the storage associated with the old value.

```
// IDL
typedef float F[10];
typedef string V[10];
typedef string M[1][2][3];
void op(out F p1, out V p2, out M p3);

// C++
F f1; F_var f2;
V v1; V_var v2;
M m1; M_var m2;

f(f2, v2, m2);
f1[0] = f2[1];
v1[1] = v2[1];          // free old storage, copy
m1[0][1][2] = m2[0][1][2];// free old storage, copy
```

In the previous example, the last two assignments result in the storage associated with the old value of the left-hand side being automatically released before the value from the right-hand side is copied.

9.Note that since **`T_var`** types do not handle **`const T*`**, there is no need to provide the const version of **`operator[]`** for **`Sequence_var`** types.

As shown in Table 16-2, **out** and return arrays are handled via pointer to array *slice*, where a slice is an array with all the dimensions of the original specified, except the first one. As a convenience for application declaration of slice types, the mapping also provides a typedef for each array slice type. The name of the slice typedef consists of the name of the array type followed by the suffix **_slice**. For example:

```
// IDL
typedef long LongArray[4][5];

// C++
typedef Long LongArray[4][5];
typedef Long LongArray_slice[5];
```

A **T_var** type for an array should overload **operator[]** instead of **operator->**. The use of array slices also means that a **T_var** type for an array should have a constructor and assignment operator that each take a pointer to array slice as a parameter, rather than **T***. The **T_var** for the previous example would be:

```
// C++
class LongArray_var
{
  public:
    LongArray_var();
    LongArray_var(LongArray_slice*);
    LongArray_var(const LongArray_var &);
    ~LongArray_var();
    LongArray_var &operator=(LongArray_slice*);
    LongArray_var &operator=(const LongArray_var &);

    LongArray_slice &operator[](ULong index);
    const LongArray_slice &operator[](Ulong index) const;
    // other conversion operators to support
    // parameter passing
};
```

Because arrays are mapped into regular C++ arrays, they present special problems for the type-safe **any** mapping described in Section 16.14, Mapping for the **any** Type. To facilitate their use with the **any** mapping, a compliant implementation must also provide for each array type a distinct C++ type whose name consists of the array name followed by the suffix **_forany**. These types must be distinct so as to allow functions to be overloaded on them. Like **Array_var** types, **Array_forany** types allow access to the underlying array type, but unlike **Array_var**, the **Array_forany** type does not **delete** the storage of the underlying array upon its own destruction. This is because the **any** mapping retains storage ownership, as described in Section 16.14.3, Extraction from **any**.

The interface of the **Array_forany** type is identical to that of the **Array_var** type, but it may not be implemented as a typedef to the **Array_var** type by a compliant implementation since it must be distinguishable from other types for purposes of function overloading. Also, the **Array_forany** constructor taking an **Array_slice*** parameter also takes a **Boolean** *nocopy* parameter, which defaults to **FALSE**.

```
// C++
class Array_forany
{
  public:
    Array_forany(Array_slice*, Boolean nocopy = FALSE);
    ...
};
```

The *nocopy* flag allows for a noncopying insertion of an **Array_slice*** into an **any**.

Each **Array_forany** type must be defined at the same level of nesting as its **Array** type.

For dynamic allocation of arrays, compliant programs must use special functions defined at the same scope as the array type. For array **T**, the following functions will be available to a compliant program.

```
// C++
T_slice *T_alloc();
T_slice *T_dup(const T_slice*);
void T_free(T_slice *);
```

The **T_alloc** function dynamically allocates an array, or returns a null pointer if it cannot perform the allocation. The **T_dup** function dynamically allocates a new array with the same size as its array argument, copies each element of the argument array into the new array, and returns a pointer to the new array. If allocation fails, a null pointer is returned. The **T_free** function deallocates an array that was allocated with **T_alloc** or **T_dup**. Passing a null pointer to **T_free** is acceptable and results in no action being performed. These functions allow ORB implementations to utilize special memory management mechanisms for array types if necessary, without forcing them to replace global **operator new** and **operator new[]**.

The **T_alloc**, **T_dup**, and **T_free** functions may not throw CORBA exceptions.

16.13 Mapping for Typedefs

A typedef creates an alias for a type. If the original type maps to several types in C++, then the typedef creates the corresponding alias for each type. The following example illustrates the mapping.

```
// IDL
typedef long T;
interface A1;
typedef A1 A2;
typedef sequence<long> S1;
typedef S1 S2;
```

```
// C++
typedef Long T;

// ...definitions for A1...

typedef A1 A2;
typedef A1_ptr A2_ptr;
typedef A1Ref A2Ref;
typedef A1_var A2_var;

// ...definitions for S1...

typedef S1 S2;
typedef S1_var S2_var;
```

For a typedef of an OMG IDL type that maps to multiple C++ types such as arrays, the typedef maps to all of the same C++ types and functions that its base type requires. For example:

```
// IDL
typedef long array[10];
typedef array another_array;

// C++
// ...C++ code for array not shown...
typedef array another_array;
typedef array_var another_array_var;
typedef array_slice another_array_slice;
typedef array_forany another_array_forany;

inline another_array_slice *another_array_alloc() {
    return array_alloc();
}

inline another_array_slice*
another_array_dup(another_array_slice *a) {
    return array_dup(a);
}

inline void another_array_free(another_array_slice *a) {
    array_free(a);
}
```

16.14 *Mapping for the* **any** *Type*

A C++ mapping for the OMG IDL type **any** must fulfill two different requirements:

- Handling C++ types in a type-safe manner.
- Handling values whose types are not known at implementation compile time.

The first item covers most normal usage of the **any** type—the conversion of typed values into and out of an **any**. The second item covers situations such as those involving the reception of a request or response containing an **any** that holds data of a type unknown to the receiver when it was created with a C++ compiler.

16.14.1 Handling Typed Values

To decrease the chances of creating an **any** with a mismatched **TypeCode** and value, the C++ function overloading facility is utilized. Specifically, for each distinct type in an OMG IDL specification, overloaded functions to insert and extract values of that type are provided by each ORB implementation. Overloaded operators are used for these functions so as to completely avoid any name space pollution. The nature of these functions, which are described in detail, is that the appropriate **TypeCode** is implied by the C++ type of the value being inserted into or extracted from the **any**.

Since the type-safe **any** interface described next is based upon C++ function overloading, it requires C++ types generated from OMG IDL specifications to be distinct. However, there are special cases in which this requirement is not met:

- As noted in Section 16.5, Mapping for Basic Data Types, the **boolean**, **octet**, and **char** OMG IDL types are not required to map to distinct C++ types, which means that a separate means of distinguishing them from each other for the purpose of function overloading is necessary. The means of distinguishing these types from each other is described in Section 16.14.4, Distinguising boolean, octet, char, and Bounded String.
- Since all strings are mapped to **char*** regardless of whether they are bounded or unbounded, another means of creating or setting an **any** with a bounded string value is necessary. This is described in Section 16.14.4, Distinguishing boolean, octet, char, and Bounded String.
- In C++, arrays within a function argument list decay into pointers to their first elements. This means that function overloading cannot be used to distinguish between arrays of different sizes. The means for creating or setting an **any** when dealing with arrays is described next and in Section 16.12, Mapping for Array Types.

*16.14.2 Insertion into **any***

To allow a value to be set in an **any** in a type-safe fashion, an ORB implementation must provide the following overloaded operator function for each separate OMG IDL type **T**.

```
// C++
void operator<<=(Any&, T);
```

This function signature suffices for types that are normally passed by value:

- **Short**, **UShort**, **Long**, **ULong**, **Float**, **Double**
- Enumerations

- Unbounded strings (**char*** passed by value)
- Object references (**T_ptr**)

For values of type **T** that are too large to be passed by value efficiently, two forms of the insertion function are provided.

```
// C++
void operator<<=(Any&, const T&);// copying form
void operator<<=(Any&, T*);      // non-copying form
```

Note that the copying form is largely equivalent to the first form shown, as far as the caller is concerned.

These “left-shift-assign” operators are used to insert a typed value into an **any** as follows.

```
// C++
Long value = 42;
Any a;
a <<= value;
```

In this case, the version of **operator<<=** overloaded for type **Long** must be able to set both the value and the **TypeCode** properly for the **any** variable.

Setting a value in an **any** using **operator<<=** means that:

- For the copying version of **operator<<=**, the lifetime of the value in the **any** is independent of the lifetime of the value passed to **operator<<=**. The implementation of the **any** may not store its value as a reference or pointer to the value passed to **operator<<=**.
- For the noncopying version of **operator<<=**, the inserted **T*** is consumed by the **any**. The caller may not use the **T*** to access the pointed-to data after insertion, since the **any** assumes ownership of it, and it may immediately copy the pointed-to data and destroy the original.
- With both the copying and noncopying versions of **operator<<=**, any previous value held by the **any** is properly deallocated. For example, if the **Any(TypeCode_ptr,void*,TRUE)** constructor (described in Section 16.14.6, Handling Untyped Values) was called to create the **any**, the **any** is responsible for deallocating the memory pointed to by the **void*** before copying the new value.

Copying insertion of a string type causes the following function to be invoked:

```
// C++
void operator<<=(Any&, const char*);
```

Since all string types are mapped to **char***, this insertion function assumes that the value being inserted is an unbounded string. Section 16.14.4, Distinguishing boolean, octet, char, and Bounded String, describes how bounded strings may be correctly inserted into an **any**. Noncopying insertion of both bounded and unbounded strings can be achieved using the **Any::from_string** helper type described in Section 16.14.4, Distinguishing boolean, octet, char, and Bounded String.

Type-safe insertion of arrays uses the **Array_forany** types described in Section 16.12, Mapping for Array Types. Compliant implementations must provide a version of **operator<<=** overloaded for each **Array_forany** type. For example:

```
// IDL
typedef long LongArray[4][5];

// C++
typedef Long LongArray[4][5];
typedef Long LongArray_slice[5];
class LongArray_forany { ... };

void operator<<=(Any &, const LongArray_forany &);
```

The **Array_forany** types are always passed to **operator<<=** by reference to const. The *nocopy* flag in the **Array_forany** constructor is used to control whether the inserted value is copied (*nocopy* == **FALSE**) or consumed (*nocopy* == **TRUE**). Because the *nocopy* flag defaults to **FALSE**, copying insertion is the default.

Because of the type ambiguity between an array of **T** and a **T***, it is highly recommended that portable code explicitly[10] use the appropriate **Array_forany** type when inserting an array into an **any**:

```
// IDL
struct S {... };
typedef S SA[5];

// C++
struct S { ... };
typedef S SA[5];
typedef S SA_slice;
class SA_forany { ... };

SA s;
// ...initialize s...
Any a;
a <<= s;                // line 1
a <<= SA_forany(s);     // line 2
```

10.A mapping implementor may use the new C++ key word “explicit” to prevent implicit conversions through the **Array_forany** constructor, but this feature is not yet widely available in current C++ compilers.

Line 1 results in the invocation of the noncopying **`operator<<=(Any&, S*)`** due to the decay of the **`SA`** array type into a pointer to its first element, rather than the invocation of the copying **`SA_forany`** insertion operator. Line 2 explicitly constructs the **`SA_forany`** type and thus results in the desired insertion operator being invoked.

The noncopying version of **`operator<<=`** for object references takes the address of the **`T_ptr`** type.

```
// IDL
interface T { ... };
```

```
// C++
void operator<<=(Any&, T_ptr);    // copying
void operator<<=(Any&, T_ptr*);   // non-copying
```

The noncopying object reference insertion consumes the object reference pointed to by **`T_ptr*`**; therefore after insertion the caller may not access the object referred to by **`T_ptr`** since the **`any`** may have duplicated and then immediately released the original object reference. The caller maintains ownership of the storage for the **`T_ptr`** itself.

The copying version of **`operator<<=`** is also supported on the **`Any_var`** type. Note that due to the conversion operators that convert **`Any_var`** to **`Any&`** for parameter passing, only those **`operator<<=`** functions defined as member functions of **`any`** need to be explicitly defined for **`Any_var`**.

16.14.3 Extraction from **any**

To allow type-safe retrieval of a value from an **`any`**, the mapping provides the following operators for each OMG IDL type **`T`**:

```
// C++
Boolean operator>>=(const Any&, T&);
```

This function signature suffices for primitive types that are normally passed by value. For values of type **`T`** that are too large to be passed by value efficiently, this function may be prototyped as follows:

```
// C++
Boolean operator>>=(const Any&, T*&);
```

The first form of this function is used only for the following types:

- **`Boolean`**, **`Char`**, **`Octet`**, **`Short`**, **`UShort`**, **`Long`**, **`ULong`**, **`Float`**, **`Double`**
- Enumerations
- Unbounded strings (**`char*`** passed by reference, i.e., **`char*&`**)
- Object references (**`T_ptr`**)

For all other types, the second form of the function is used.

All versions of **operator>>=** implemented as member functions of class **any**, such as those for primitive types, should be marked as **const**.

This "right-shift-assign" operator is used to extract a typed value from an **any** as follows:

```
// C++
Long value;
Any a;
a <<= Long(42);
if (a >>= value) {
    // ... use the value ...
}
```

In this case, the version of **operator>>=** for type **Long** must be able to determine whether the **any** truly does contain a value of type **Long** and, if so, copy its value into the reference variable provided by the caller and return **TRUE**. If the **any** does not contain a value of type **Long**, the value of the caller's reference variable is not changed, and **operator>>=** returns **FALSE**.

For nonprimitive types, extraction is done by pointer. For example, consider the following IDL struct:

```
// IDL
struct MyStruct {
    long lmem;
    short smem;
};
```

Such a struct could be extracted from an **any** as follows:

```
// C++
Any a;
// ... a is somehow given a value of type MyStruct ...
MyStruct *struct_ptr;
if (a >>= struct_ptr) {
    // ... use the value ...
}
```

If the extraction is successful, the caller's pointer will point to storage managed by the **any**, and **operator>>=** will return **TRUE**. The caller must not try to **delete** or otherwise release this storage. The caller also should not use the storage after the contents of the **any** variable are replaced via assignment, insertion, or the **replace** function, or after the **any** variable is destroyed. Care must be taken to avoid using **T_var** types with these extraction operators, since they will try to assume responsibility for deleting the storage owned by the **any**.

If the extraction is not successful, the value of the caller's pointer is set equal to the null pointer, and **operator>>=** returns **FALSE**.

Correct extraction of array types relies on the **`Array_forany`** types described in Section 16.12, Mapping for Array Types.

```
// IDL
typedef long A[20];
typedef A B[30][40][50];

// C++
typedef Long A[20];
typedef Long A_slice;
class A_forany { ... };
typedef A B[30][40][50];
typedef A B_slice[40][50];
class B_forany { ... };

Boolean operator>>=(const Any &, A_forany&);// for type A
Boolean operator>>=(const Any &, B_forany&);// for type B
```

The **`Array_forany`** types are always passed to **`operator>>=`** by reference.

For strings and arrays, applications are responsible for checking the **`TypeCode`** of the **`any`** to be sure that they do not overstep the bounds of the array or string object when using the extracted value.

The **`operator>>=`** is also supported on the **`Any_var`** type. Note that due to the conversion operators that convert **`Any_var`** to **`const Any&`** for parameter passing, only those **`operator>>=`** functions defined as member functions of **`any`** need to be explicitly defined for **`Any_var`**.

16.14.4 Distinguishing boolean, octet, char, and Bounded String

Since the **`boolean`**, **`octet`**, and **`char`** OMG IDL types are not required to map to distinct C++ types, another means of distinguishing them from each other is necessary so that they can be used with the type-safe **`any`** interface. Similarly, since both bounded and unbounded strings map to **`char*`**, another means of distinguishing them must be provided. This is done by introducing several new helper types nested in the **`any`** class interface. For example, this can be accomplished as shown next.

```
// C++
class Any
{
  public:
     // special helper types needed for boolean, octet, char,
     // and bounded string insertion
     struct from_boolean {
          from_boolean(Boolean b) : val(b) {}
          Boolean val;
     };
     struct from_octet {
          from_octet(Octet o) : val(o) {}
```

```
    Octet val;
};
struct from_char {
    from_char(Char c) : val(c) {}
    Char val;
};
struct from_string {
    from_string(char* s, ULong b,
                    Boolean nocopy = FALSE) :
            val(s), bound(b) {}
    char *val;
    ULong bound;
};

void operator<<=(from_boolean);
void operator<<=(from_char);
void operator<<=(from_octet);
void operator<<=(from_string);

// special helper types needed for boolean, octet,
// char, and bounded string extraction
struct to_boolean {
    to_boolean(Boolean &b) : ref(b) {}
    Boolean &ref;
};
struct to_char {
    to_char(Char &c) : ref(c) {}
    Char &ref;
};
struct to_octet {
    to_octet(Octet &o) : ref(o) {}
    Octet &ref;
};
struct to_string {
    to_string(char *&s, ULong b) : val(s), bound(b) {}
    char *&val;
    ULong bound;
};

Boolean operator>>=(to_boolean) const;
Boolean operator>>=(to_char) const;
Boolean operator>>=(to_octet) const;
Boolean operator>>=(to_string) const;

// other public Any details omitted
```

```
private:
     // these functions are private and not implemented
     // hiding these causes compile-time errors for
     // unsigned char
     void operator<<=(unsigned char);
     Boolean operator>>=(unsigned char &) const;
};
```

An ORB implementation provides the overloaded **`operator<<=`** and **`operator>>=`** functions for these special helper types. These helper types are used as shown next.

```
// C++
Boolean b = TRUE;
Any any;
any <<= Any::from_boolean(b);
// ...
if (any >>= Any::to_boolean(b)) {
     // ...any contained a Boolean...
}

char* p = "bounded";
any <<= Any::from_string(p, 8);
// ...
if (any >>= Any::to_string(p, 8)) {
     // ...any contained a string<8>...
}
```

A bound value of zero indicates an unbounded string.

For noncopying insertion of a bounded or unbounded string into an **`any`**, the **`nocopy`** flag on the **`from_string`** constructor should be set to **`TRUE`**.

```
// C++
char* p = string_alloc(8);
// ...initialize string p...
any <<= Any::from_string(p, 8, 1);// any consumes p
```

Assuming that **`boolean`**, **`char`**, and **`octet`** all map the C++ type **`unsigned char`**, the private and unimplemented **`operator<<=`** and **`operator>>=`** functions for **`unsigned char`** will cause a compile-time error if straight insertion or extraction of any of the **`boolean`**, **`char`**, or **`octet`** types is attempted.

```
// C++
Octet oct = 040;
Any any;
any <<= oct;            // this line will not compile
any <<= Any::from_octet(oct);// but this one will
```

It is important to note that the previous example is only one possible implementation for these helpers, not a mandated one. Other compliant implementations are possible, such as providing them via in-lined static **any** member functions if **boolean**, **char**, and **octet** are in fact mapped to distinct C++ types. All compliant C++ mapping implementations must provide these helpers, however, for purposes of portability.

16.14.5 Widening to Object

Sometimes it is desirable to extract an object reference from an **any** as the base **Object** type. This can be accomplished using a helper type similar to those required for extracting **boolean**, **char**, and **octet**.

```
// C++
class Any
{
  public:
     ...
     struct to_object {
          to_object(Object_ptr &obj) : ref(obj) {}
          Object_ptr &ref;
     };
     Boolean operator>>=(to_object) const;
     ...
};
```

The **to_object** helper type is used to extract an object reference from an **any** as the base **Object** type. If the **any** contains a value of an object reference type as indicated by its **TypeCode**, the extraction function **operator>>=(to_object)** explicitly widens its contained object reference to **Object** and returns **TRUE**, otherwise it returns **FALSE**. This is the only object reference extraction function that performs widening on the extracted object reference. As with regular object reference extraction, no duplication of the object reference is performed by the **to_object** extraction operator.

16.14.6 Handling Untyped Values

Under some circumstances the type-safe interface to **any** is not sufficient. An example is a situation in which data types are read from a file in binary form and used to create values of type **any**. For these cases, the **any** class provides a constructor with an explicit **TypeCode** and generic pointer:

```
// C++
Any(TypeCode_ptr tc, void *value, Boolean release = FALSE);
```

The constructor is responsible for duplicating the given **TypeCode** pseudo object reference. If the **release** parameter is **TRUE**, then the **any** object assumes ownership of the storage pointed to by the **value** parameter. A compliant application should make no assumptions about the continued lifetime of the **value** parameter once it has been handed to an **any** with **release=TRUE**, since a compliant **any** implementation is allowed to copy the **value** parameter and immediately free the original pointer. If the **release** parameter is **FALSE** (the default case), then the **any** object assumes the caller will manage the memory pointed to by **value**. The **value** parameter can be a null pointer.

The **any** class also defines three unsafe operations:

```
// C++
void replace(
     TypeCode_ptr,
     void *value,
     Boolean release = FALSE
);
TypeCode_ptr type() const;
const void *value() const;
```

The **replace** function is intended to be used with types that cannot be used with the type-safe insertion interface, and so is similar to the constructor previously described. The existing **TypeCode** is released and value storage deallocated, if necessary. The **TypeCode** function parameter is duplicated. If the **release** parameter is **TRUE**, then the **any** object assumes ownership for the storage pointed to by the **value** parameter. A compliant application should make no assumptions about the continued lifetime of the **value** parameter once it has been handed to the **Any::replace** function with **release=TRUE**, since a compliant **any** implementation is allowed to copy the **value** parameter and immediately free the original pointer. If the **release** parameter is **FALSE** (the default case), then the **any** object assumes the caller will manage the memory occupied by the value. The **value** parameter of the **replace** function can be a null pointer.

For C++ mapping implementations that use **Environment** parameters to pass exception information, the default **release** argument can be simulated by providing two overloaded **replace** functions, one that takes a nondefaulted **release** parameter and one that takes no **release** parameter. The second function simply invokes the first with the **release** parameter set to **FALSE**.

Note that neither the constructor shown above nor the **replace** function is type-safe. In particular, no guarantees are made by the compiler or run-time as to the consistency between the **TypeCode** and the actual type of the **void*** argument. The behavior of an ORB implementation when presented with an **any** that is constructed with a mismatched **TypeCode** and value is not defined.

The **type** function returns a **TypeCode_ptr** pseudo-object reference to the **TypeCode** associated with the **any**. Like all object reference return values, the caller must release the reference when it is no longer needed, or assign it to a **TypeCode_var** variable for automatic management.

The **value** function returns a pointer to the data stored in the **any**. If the **any** has no associated value, the **value** function returns a null pointer. The type to which the **void*** returned by the **value** function may be cast depends on the ORB implementation; thus, use of the **value** function is not portable across ORB implementations and its usage is therefore deprecated. Note that ORB implementations are allowed to make stronger guarantees about the **void*** returned from the **value** function, if so desired.

*16.14.7 **any** Constructors, Destructor, Assignment Operator*

The default constructor creates an **any** with a **TypeCode** of type **tk_null**, and no value. The copy constructor calls **_duplicate** on the **TypeCode_ptr** of its **any** parameter and deep-copies the parameter's value. The assignment operator releases its own **TypeCode_ptr** and deallocates storage for the current value if necessary, then duplicates the **TypeCode_ptr** of its **any** parameter and deep-copies the parameter's value. The destructor calls **release** on the **TypeCode_ptr** and deallocates storage for the value, if necessary.

Other constructors are described in Section 16.14.6, Handling Untyped Values.

ORB implementations concerned with single-process interoperability with the C mapping may overload **operator new()** and **operator delete()** for **any**s so that dynamic allocation uses the same mechanism as the C language dynamic allocation functions. Whether these operators are overloaded by the implementation or not, compliant programs use **new** to dynamically allocate **any**s and **delete** to free them.

*16.14.8 **any** Class*

The full definition of the **any** class can be found in Section C.3, **any** Class.

*16.14.9 **Any_var** Class*

Since **any**s are returned via pointer as **out** and return parameters, there exists an **Any_var** class similar to the **T_var** classes for object references. **Any_var** obeys the rules for **T_var** classes described in Section 16.8, Mapping for Structured Types, calling **delete** on its **Any*** when it goes out of scope or is otherwise destroyed. The full interface of the **Any_var** class is shown in Section C.4, **Any_var** Class.

16.15 Mapping for Exception Types

An OMG IDL exception is mapped to a C++ class that derives from the standard **UserException** class defined in the **CORBA** module (see Section 16.1.3, CORBA Module). The generated class is like a variable-length struct, regardless of whether or not the exception holds any variable-length members. Just as for variable-length structs, each exception member must be self-managing with respect to its storage.

The copy constructor, assignment operator, and destructor automatically copy or free the storage associated with the exception. For convenience, the mapping also defines a constructor with one parameter for each exception member—this constructor initializes the exception members to the given values. For exception types that have a string member, this constructor should take a **const char*** parameter, since the constructor must copy the string argument. Similarly, constructors for exception types that have an object reference member must call **_duplicate** on the corresponding object reference constructor parameter. The default constructor performs no explicit member initialization.

The **UserException** class is derived from a base **Exception** class, which is also defined in the **CORBA** module.

All standard exceptions are derived from a **SystemException** class, also defined in the **CORBA** module. Like **UserException**, **SystemException** is derived from the base **Exception** class. The **SystemException** class interface is shown next.

```
// C++
enum CompletionStatus {
     COMPLETED_YES,
     COMPLETED_NO,
     COMPLETED_MAYBE
};
class SystemException : public Exception
{
  public:
     SystemException();
     SystemException(const SystemException &);
     SystemException(ULong minor, CompletionStatus status);
     ~SystemException();
     SystemException &operator=(const SystemException &);

     ULong minor() const;
     void minor(ULong);

     CompletionStatus completed() const;
     void completed(CompletionStatus);
};
```

The default constructor for **SystemException** causes **minor()** to return zero and **completed()** to return **COMPLETED_NO.**

Each specific system exception (described in Section 14.1.6, Exceptions) is derived from **SystemException**.

```
// C++
class UNKNOWN : public SystemException { ... };
class BAD_PARAM : public SystemException { ... };
// etc.
```

All specific system exceptions are defined within the **CORBA** module.

This exception hierarchy allows any exception to be caught by simply catching the **Exception** type.

```
// C++
try {
    ...
} catch (const Exception &exc) {
    ...
}
```

Alternatively, all user exceptions can be caught by catching the **UserException** type, and all system exceptions can be caught by catching the **SystemException** type.

```
// C++
try {
    ...
} catch (const UserException &ue) {
    ...
} catch (const SystemException &se) {
    ...
}
```

Naturally, more specific types can also appear in **catch** clauses.

Exceptions are normally thrown by value and caught by reference. This approach lets the exception destructor release storage automatically.

C++ compilers that support official C++ Run-Time Type Information (RTTI) need not support narrowing for the **Exception** hierarchy. RTTI supports, among other things, determination of the run-time type of a C++ object. In particular, the **dynamic_cast<T*>** operator[11] allows for narrowing from a base pointer to a more derived pointer if the object pointed to really is of the more derived type. This operator is not useful for narrowing object references, since it cannot determine the actual type of remote objects, but it can be used to narrow within the exception hierarchy. Since catch clauses can catch by type, this feature is mainly used for narrowing exceptions received via **Environment**s from the DII.

For those C++ environments that do not support **dynamic_cast<T*>**, the exception hierarchy provides a narrowing mechanism. This is described in Section D.4, Without Run-Time Type Information (RTTI).

Request invocations made through the DII may result in user-defined exceptions that cannot be fully represented in the calling program because the specific exception type was not known at compile time. The mapping provides the **UnknownUserException** so that such exceptions can be represented in the calling process.

11.It is unlikely that a compiler would support RTTI without supporting exceptions, since much of a C++ exception handling implementation is based on RTTI mechanisms.

```
// C++
class UnknownUserException : public UserException
{
  public:
     Any &exception();
};
```

As shown here, **UnknownUserException** is derived from **UserException**. It provides the **exception()** accessor that returns an **any** holding the actual exception. Ownership of the returned **any** is maintained by the **UnknownUserException**—the **any** merely allows access to the exception data. Conforming applications should never explicitly throw exceptions of type **UnknownUserException**—it is intended for use with the DII.

16.16 Mapping for Operations and Attributes

An operation maps to a C++ function with the same name as the operation. Each read-write attribute maps to a pair of overloaded C++ functions (both with the same name), one to set the attribute's value and one to get the attribute's value. The *set* function takes an **in** parameter with the same type as the attribute, while the *get* function takes no parameters and returns the same type as the attribute. An attribute marked **readonly** maps to only one C++ function, to get the attribute's value. Parameters and return types for attribute functions obey the same parameter passing rules as for regular operations.

OMG IDL **oneway** operations are mapped the same as other operations; that is, there is no way to know by looking at the C++ whether an operation is **oneway** or not.

The mapping does not define whether exceptions specified for an OMG IDL operation are part of the generated operation's type signature or not.

```
// IDL
interface A
{
    void f();
    oneway void g();
    attribute long x;
};

// C++
A_var a;
a->f();
a->g();
Long n = a->x();
a->x(n + 1);
```

Unlike the C mapping, C++ operations do not require an additional **Environment** parameter for passing exception information—real C++ exceptions are used for this purpose. See Section 16.15, Mapping for Exception Types, and Section D.3, Without Exception Handling, for more details.

16.17 Implicit Arguments to Operations

If an operation in an OMG IDL specification has a context specification, then a **`Context_ptr`** input parameter (see Section 17.8.1, Context Interface) follows all operation-specific arguments. In an implementation that does not support real C++ exceptions, an output **`Environment`** parameter is the last argument following all operation-specific arguments, and following the context argument if present. The parameter passing mode for **`Environment`** is described in Section D.3, Without Exception Handling.

16.18 Argument Passing Considerations

The mapping of parameter passing modes attempts to balance the need for both efficiency and simplicity. For primitive types, enumerations, and object references, the modes are straightforward, passing the type *P* for primitives and enumerations and the type **`A_ptr`** for an interface type *A*.

Aggregate types are complicated by the question of when and how parameter memory is allocated and deallocated. Mapping **`in`** parameters is straightforward because the parameter storage is caller-allocated and read-only. The mapping for **`out`** and **`inout`** parameters is more problematic. For variable-length types, the callee must allocate some if not all of the storage. For fixed-length types, such as a *Point* type represented as a struct containing three floating point members, caller allocation is preferable (to allow stack allocation).

To accommodate both kinds of allocation, avoid the potential confusion of split allocation, and eliminate confusion with respect to when copying occurs, the mapping is **`T&`** for a fixed-length aggregate **`T`** and **`T*&`** for a variable-length **`T`**. This approach has the unfortunate consequence that usage for structs depends on whether the struct is fixed- or variable-length; however, the mapping is consistently **`T_var&`** if the caller uses the managed type **`T_var`**.

The mapping for **`out`** and **`inout`** parameters additionally requires support for deallocating any previous variable-length data in the parameter when a **`T_var`** is passed. Even though their initial values are not sent to the operation, we include **`out`** parameters because the parameter could contain the result from a previous call. There are many ways to implement this support. The mapping does not require a specific implementation, but a compliant implementation must free the inaccessible storage associated with a parameter passed as a **`T_var`** managed type. The following examples demonstrate the compliant behavior.

```
// IDL
struct S { string name; float age; };
void f(out S p);
```

```
// C++
S_var s;
f(s);
// use s
f(s); // first result will be freed

S *sp; // need not initialize before passing to out
f(sp);
// use sp
delete sp; // cannot assume next call will free old value
f(sp);
```

Note that implicit deallocation of previous values for **`out`** and **`inout`** parameters works only with **`T_var`** types, not with other types.

```
// IDL
void q(out string s);

// C++
char *s;
for (int i = 0; i < 10; i++)
     q(s);          // memory leak!
```

Each call to the **`q`** function in the loop results in a memory leak because the caller is not invoking **`string_free`** on the **`out`** result. There are two ways to fix this, as shown next.

```
// C++
char *s;
String_var svar;
for (int i = 0 ; i < 10; i++) {
     q(s);
     string_free(s);     // explicit deallocation
     // OR:
     q(svar);            // implicit deallocation
}
```

Using a plain **`char*`** for the **out** parameter means that the caller must explicitly deallocate its memory before each reuse of the variable as an **out** parameter, while using a **`String_var`** means that any deallocation is performed implicitly upon each use of the variable as an **out** parameter.

Variable-length data must be explicitly released before being overwritten. For example, before assigning to an **`inout`** string parameter, the implementor of an operation may first delete the old character data. Similarly, an **`inout`** interface parameter should be released before being reassigned. One way to ensure that the parameter storage is released is to assign it to a local **`T_var`** variable with an automatic release, as in the following example.

```
// IDL
interface A;
void f(inout string s, inout A obj);

// C++
void Aimpl::f(char *&s, A_ptr &obj) {
     String_var s_tmp = s;
     s = /* new data */;
     A_var obj_tmp = obj;
     obj = /* new reference */
}
```

To allow the callee the freedom to allocate a single contiguous area of storage for all the data associated with a parameter, we adopt the policy that the callee-allocated storage is not modifiable by the caller. However, trying to enforce this policy by returning a **`const`** type in C++ is problematic, since the caller is required to release the storage, and calling **`delete`** on a **`const`** object is an error[12]. A compliant mapping therefore is not required to detect this error.

For parameters that are passed or returned as a pointer (**`T*`**) or reference to pointer (**`T*&`**), a compliant program is not allowed to pass or return a null pointer; the result of doing so is undefined. In particular, a caller may not pass a null pointer under any of the following circumstances:

- **`in`** and **`inout`** string
- **`in`** and **`inout`** array (pointer to first element)

A caller may pass a reference to a pointer with a null value for **`out`** parameters, however, since the callee does not examine the value but rather just overwrites it. A callee may not return a null pointer under any of the following circumstances:

- **`out`** and return variable-length struct
- **`out`** and return variable-length union
- **`out`** and return string
- **`out`** and return sequence
- **`out`** and return variable-length array, return fixed-length array
- **`out`** and return any

12.It is very likely that the upcoming ANSI/ISO C++ standard will allow **`delete`** on a **`const`** object, but many C++ compilers do not yet support this feature.

Since OMG IDL has no concept of pointers in general or null pointers in particular, allowing the passage of null pointers to or from an operation would project C++ semantics onto OMG IDL operations.[13] A compliant implementation is allowed but not required to raise a **BAD_PARAM** exception if it detects such an error.

Table 16-2 displays the mapping for the basic OMG IDL parameter passing modes and return type according to the type being passed or returned, while Table 16-3 displays the same information for **T_var** types. Table 16-2 is merely for informational purposes; it is expected that operation signatures will be written in terms of the parameter passing modes shown in Table 16-2, and that **T_var** types will support the necessary conversion operators to allow them to be passed directly.

In Table 16-2, fixed-length arrays are the only case where the type of an **out** parameter differs from a return value, which is necessary because C++ does not allow a function to return an array. The mapping returns a pointer to a *slice* of the array, where a slice is an array with all the dimensions of the original specified except the first one.

Table 16-2 Basic Argument and Result Passing

Data Type	In	Inout	Out	Return
short	Short	Short&	Short&	Short
long	Long	Long&	Long&	Long
unsigned short	UShort	UShort&	UShort&	UShort
unsigned long	ULong	ULong&	ULong&	ULong
float	Float	Float&	Float&	Float
double	Double	Double&	Double&	Double
boolean	Boolean	Boolean&	Boolean&	Boolean
char	Char	Char&	Char&	Char
octet	Octet	Octet&	Octet&	Octet
enum	enum	enum&	enum&	enum
object reference ptr[1]	objref_ptr	objref_ptr&	objref_ptr&	objref_ptr
struct, fixed	const struct&	struct&	struct&	struct
struct, variable	const struct&	struct&	struct*&	struct*
union, fixed	const union&	union&	union&	union
union, variable	const union&	union&	union*&	union*
string	const char*	char*&	char*&	char*
sequence	const sequence&	sequence&	sequence*&	sequence*
array, fixed	const array	array	array	array slice*[2]
array, variable	const array	array	array slice*&[2]	array slice*[2]
any	const any&	any&	any*&	any*

13.When real C++ exceptions are not available, however, it is important that null pointers are returned whenever an **Environment** containing an exception is returned; see Section D.3, Without Exception Handling, for more details.

1. Including pseudo-object references.
2. A slice is an array with all the dimensions of the original except the first one.

A caller is responsible for providing storage for all arguments passed as **`in`** arguments.

Table 16-3 T_var Argument and Result Passing

Data Type	In	Inout	Out	Return
object reference var[1]	const objref_var&	objref_var&	objref_var&	objref_var
struct_var	const struct_var&	struct_var&	struct_var&	struct_var
union_var	const union_var&	union_var&	union_var&	union_var
string_var	const string_var&	string_var&	string_var&	string_var
sequence_var	const sequence_var&	sequence_var&	sequence_var&	sequence_var
array_var	const array_var&	array_var&	array_var&	array_var
any_var	const any_var&	any_var&	any_var&	any_var

1. Including pseudo-object references.

Table 16-4 and Table 16-5 describe the caller's responsibility for storage associated with **`inout`** and **`out`** parameters and for return results.

Table 16-4 Caller Argument Storage Responsibilities

Type	Inout Param	Out Param	Return Result
short	1	1	1
long	1	1	1
unsigned short	1	1	1
unsigned long	1	1	1
float	1	1	1
double	1	1	1
boolean	1	1	1
char	1	1	1
octet	1	1	1
enum	1	1	1
object reference ptr	2	2	2
struct, fixed	1	1	1
struct, variable	1	3	3
union, fixed	1	1	1
union, variable	1	3	3
string	4	3	3
sequence	5	3	3
array, fixed	1	1	6

Table 16-4 Caller Argument Storage Responsibilities *(Continued)*

Type	Inout Param	Out Param	Return Result
array, variable	1	6	6
any	5	3	3

Table 16-5 Argument Passing Cases

Case[1]	
1	Caller allocates all necessary storage, except that which may be encapsulated and managed within the parameter itself. For inout parameters, the caller provides the initial value, and the callee may change that value. For out parameters, the caller allocates the storage but need not initialize it, and the callee sets the value. Function returns are by value.
2	Caller allocates storage for the object reference. For inout parameters, the caller provides an initial value; if the callee wants to reassign the inout parameter, it will first call CORBA::release on the original input value. To continue to use an object reference passed in as an inout, the caller must first duplicate the reference. The caller is responsible for the release of all out and return object references. Release of all object references embedded in other structures is performed automatically by the structures themselves.
3	For out parameters, the caller allocates a pointer and passes it by reference to the callee. The callee sets the pointer to point to a valid instance of the parameter's type. For returns, the callee returns a similar pointer. The callee is not allowed to return a null pointer in either case. In both cases, the caller is responsible for releasing the returned storage. To maintain local/remote transparency, the caller must always release the returned storage, regardless of whether the callee is located in the same address space as the caller or is located in a different address space. Following the completion of a request, the caller is not allowed to modify any values in the returned storage—to do so, the caller must first copy the returned instance into a new instance, then modify the new instance.
4	For inout strings, the caller provides storage for both the input string and the **`char*`** pointing to it. Since the callee may deallocate the input string and reassign the **`char*`** to point to new storage to hold the output value, the caller should allocate the input string using **`string_alloc()`**. The size of the out string is therefore not limited by the size of the in string. The caller is responsible for deleting the storage for the out **`using string_free()`**. The callee is not allowed to return a null pointer for an inout, out, or return value.
5	For inout sequences and **any**s, assignment or modification of the sequence or **any** may cause deallocation of owned storage before any reallocation occurs, depending upon the state of the boolean release parameter with which the sequence or **any** was constructed.
6	For out parameters, the caller allocates a pointer to an array slice, which has all the same dimensions of the original array except the first, and passes the pointer by reference to the callee. The callee sets the pointer to point to a valid instance of the array. For returns, the callee returns a similar pointer. The callee is not allowed to return a null pointer in either case. In both cases, the caller is responsible for releasing the returned storage. To maintain local/remote transparency, the caller must always release the returned storage, regardless of whether the callee is located in the same address space as the caller or is located in a different address space. Following completion of a request, the caller is not allowed to modify any values in the returned storage—to do so, the caller must first copy the returned array instance into a new array instance, then modify the new instance.

1. As listed in Table 16-4.

Mapping of Pseudo-Objects to C++ 17

CORBA pseudo-objects may be implemented either as normal CORBA objects or as *serverless objects*. In the CORBA specification, the fundamental differences between these strategies are:

- Serverless object types do not inherit from **`CORBA::Object.`**
- Individual serverless objects are not registered with any ORB.
- Serverless objects do not necessarily follow the same memory management rules as for regular OMG IDL types.

References to serverless objects are not necessarily valid across computational contexts; for example, address spaces. Instead, references to serverless objects passed as parameters may result in the construction of independent, functionally identical copies of objects used by receivers of these references. To support this, the otherwise hidden representational properties (such as data layout) of serverless objects are made known to the ORB. Specifications for achieving this are not contained in this chapter: making serverless objects known to the ORB is an implementation detail.

This chapter provides a standard mapping algorithm for all pseudo-object types. This avoids the need for piecemeal mappings for each of the nine CORBA pseudo-object types, and accommodates any pseudo-object types that may be proposed in future revisions of CORBA. It also avoids representation dependence in the C mapping while still allowing implementations that rely on C-compatible representations.

17.1 Usage

Rather than C-PIDL, this mapping uses an augmented form of full OMG IDL to describe serverless object types. Interfaces for pseudo-object types follow the exact same rules as normal OMG IDL interfaces, with the following exceptions:

- They are prefaced by the keyword **`pseudo.`**

- Their declarations may refer to other[1] serverless object types not otherwise necessarily allowed in OMG IDL.

As explained in Section 14.23, Pseudo-Objects, the **pseudo** prefix means that the interface may be implemented in either a normal or serverless fashion. That is, apply either the rules described in the following sections or the normal mapping rules described in Chapter 16, Mapping of OMG IDL to C++.

17.2 Mapping Rules

Serverless objects are mapped in the same way as normal interfaces, except for the differences outlined in this section.

Classes representing serverless object types are *not* subclasses of **CORBA::Object**, and are not necessarily subclasses of any other C++ class. Thus, they do not necessarily support, for example, the **Object::create_request** operation.

For each class representing a serverless object type **T**, overloaded versions of the following functions are provided in the **CORBA** name space.

```
// C++
void release(T_ptr);
Boolean is_nil(T_ptr p);
```

The mapped C++ classes are not guaranteed to be usefully subclassable by users, although subclasses can be provided by implementations. Implementations are allowed to make assumptions about internal representations and transport formats that may not apply to subclasses.

The member functions of classes representing serverless object types do not necessarily obey the normal memory management rules. This is due to the fact that some serverless objects, such as **CORBA::NVList**, are essentially just containers for several levels of other serverless objects. Requiring callers to explicitly free the values returned from accessor functions for the contained serverless objects would be counter to their intended usage.

All other elements of the mapping are the same. In particular:

- The types of references to serverless objects, **T_ptr**, may or may not simply be a typedef of **T***.
- Each mapped class supports the following static member functions.

```
// C++
static T_ptr _duplicate(T_ptr p);
static T_ptr _nil();
```

1.In particular, **exception** used as a data type and a function name.

Legal implementations of **`_duplicate`** include simply returning the argument or constructing references to a new instance. Individual implementations may provide stronger guarantees about behavior.

- The corresponding C++ classes may or may not be directly instantiable or have other instantiation constraints. For portability, users should invoke the appropriate constructive operations. When none are listed, users cannot depend on any portable means for constructing such objects, and should consult documentation for their implementations.
- As with normal interfaces, assignment operators are not supported.
- Although they can transparently employ “copy-style” rather than “reference-style” mechanics, parameter passing signatures and rules as well as memory management rules are identical to those for normal objects, unless otherwise noted.

17.3 Relation to the C PIDL Mapping

All serverless object interfaces and declarations that rely on them have direct analogs in the C mapping. The mapped C++ classes can, but need not be, implemented using representations compatible to those chosen for the C mapping. Differences between the pseudo-object specifications for C-PIDL and C++ PIDL are as follows:

- C++-PIDL calls for removal of representation dependencies through the use of interfaces rather than structs and typedefs.
- C++-PIDL calls for placement of operations on pseudo-objects in their interfaces, including a few cases of redesignated functionality as noted.
- In C++-PIDL, the **`release`** performs the role of the associated **`free`** and **`delete`** operations in the C mapping, unless otherwise noted.

Brief descriptions and listings of each pseudo-interface and its C++ mapping are provided in the following sections. Further details, including definitions of types referenced but not defined next, may be found in the relevant sections of this document.

17.4 Environment

`Environment` provides a vehicle for dealing with exceptions in those cases where true exception mechanics are unavailable or undesirable (for example in the DII). They may be set and inspected using the **`exception`** attribute.

As with normal OMG IDL attributes, the **`exception`** attribute is mapped into a pair of C++ functions used to set and get the exception. The semantics of the **`set`** and **`get`** functions, however, are somewhat different than those for normal OMG IDL attributes. The **`set`** C++ function assumes ownership of the **`Exception`** pointer passed to it. The **`Environment`** will eventually call **`delete`** on this pointer, so the **`Exception`** it points to must be dynamically allocated by the caller. The **`get`** function returns a pointer to the **`Exception`**, just as an attribute for a variable-length struct would, but the pointer refers to memory owned by the **`Environment`**. Once the **`Environment`** is destroyed, the pointer is no longer valid. The caller must not call **`delete`** on the

Exception pointer returned by the **get** function. The **Environment** is responsible for deallocating any **Exception** it holds when it is itself destroyed. If the **Environment** holds no exception, the **get** function returns a null pointer.

The **clear()** function causes the **Environment** to **delete** any **Exception** it is holding. It is not an error to call **clear()** on an **Environment** holding no exception. Passing a null pointer to the **set** exception function is equivalent to calling **clear()**. If an **Environment** contains exception information, the caller is responsible for calling **clear()** on it before passing it to an operation.

17.4.1 Environment Interface

```
// IDL
pseudo interface Environment
{
    attribute exception exception;
    void clear();
};
```

17.4.2 Environment C++ Class

```
// C++
class Environment
{
  public:
     void exception(Exception*);
     Exception *exception() const;
     void clear();
};
```

17.4.3 Differences from C-PIDL

The C++-PIDL specification differs from the C-PIDL specification as follows:

- Defines an interface rather than a struct.
- Supports an attribute allowing operations on exception values as a whole rather than on major numbers and/or identification strings.
- Supports a **clear()** function that is used to destroy any **Exception** the **Environment** may be holding.
- Supports a default constructor that initializes it to hold no exception information.

17.4.4 Memory Management

Environment has the following special memory management rules:

- The **void exception(Exception*)** member function adopts the **Exception*** given to it.

- Ownership of the return value of the **Exception *exception()** member function is maintained by the **Environment**; this return value must not be freed by the caller.

17.5 NamedValue

NamedValue is used only as an element of **NVList**, especially in the DII. **NamedValue** maintains an optional name, an **any** value, and labeling flags. Legal flag values are **ARG_IN**, **ARG_OUT**, and **ARG_INOUT**.

The value in a **NamedValue** may be manipulated via standard operations on **any**.

17.5.1 NamedValue Interface

```
// IDL
pseudo interface NamedValue
{
    readonly attribute Identifier name;
    readonly attribute any value;
    readonly attribute Flags flags;
};
```

17.5.2 NamedValue C++ Class

```
// C++
class NamedValue
{
  public:
     const char *name() const;
     Any *value() const;
     Flags flags() const;
};
```

17.5.3 Differences from C-PIDL

The C++-PIDL specification differs from the C-PIDL specification as follows:

- Defines an interface rather than a struct.
- Provides no analog of the **len** field.

17.5.4 Memory Management

NamedValue has the following special memory management rule:

- Ownership of the return values of the **name()** and **value()** functions is maintained by the **NamedValue**; these return values must not be freed by the caller.

17.6 NVList

NVList is a list of **NamedValue**s. A new **NVList** is constructed using the **ORB::create_list** operation (see Section 17.12, ORB). New **NamedValue**s may be constructed as part of an **NVList**, in any of three ways:

- **add**—creates an unnamed value, initializing only the flags.
- **add_item**—initializes name and flags.
- **add_value**—initializes name, value, and flags.
- **add_item_consume**—initializes name and flags, taking over memory management responsibilities for the **char*** name parameter.
- **add_value_consume**—initializes name, value, and flags, taking over memory management responsibilities for both the **char*** name parameter and the **Any*** value parameter.

Each of these operations returns the new item.

Elements may be accessed and deleted via zero-based indexing. The **add**, **add_item**, **add_value**, **add_item_consume**, and **add_value_consume** functions lengthen the **NVList** to hold the new element each time they are called. The **item** function can be used to access existing elements.

17.6.1 NVList Interface

```
// IDL
pseudo interface NVList
{
    readonly attribute unsigned long count;
    NamedValue add(in Flags flags);
    NamedValue add_item(in Identifier item_name,
    in Flags flags);
    NamedValue add_value(
                    in Identifier item_name,
                    in any val,
                    in Flags flags
    );
    NamedValue item(in unsigned long index) raises(Bounds);
    Status remove(in unsigned long index) raises(Bounds);
};
```

17.6.2 NVList C++ Class

```
// C++
class NVList
{
  public:
     ULong count() const;
     NamedValue_ptr add(Flags);
     NamedValue_ptr add_item(const char*, Flags);
     NamedValue_ptr add_value(
                    const char*,
                    const Any&,
                    Flags
     );
     NamedValue_ptr add_item_consume(
                    char*,
                    Flags
     );
     NamedValue_ptr add_value_consume(
                    char*,
                    Any *,
                    Flags
     );
     NamedValue_ptr item(ULong);
     Status remove(ULong);
};
```

17.6.3 Differences from C-PIDL

The C++-PIDL specification differs from the C-PIDL specification as follows:

- Defines an interface rather than a typedef.
- Provides different signatures for operations that add items in order to avoid representation dependencies.
- Provides indexed access methods.

17.6.4 Memory Management

NVList has the following special memory management rules:

- Ownership of the return values of the **add**, **add_item**, **add_value**, **add_item_consume**, and **add_value_consume** functions is maintained by the **NVList**; these return values must not be freed by the caller.
- The **char*** parameters to the **add_item_consume** and **add_value_consume** functions and the **Any*** parameter to the **add_value_consume** function are consumed by the **NVList**. The caller may not access these data after they have been passed to these functions, because the **NVList** may copy them and destroy the

originals immediately. The caller should use the **NamedValue::value()** operation in order to modify the **value** attribute of the underlying **NamedValue**, if desired.

- The **remove** function also calls **CORBA::release** on the removed **NamedValue**.

17.7 Request

Request provides the primary support for DII. A new request on a particular target object may be constructed using the short version of the request creation operation shown in Section 17.13, Object.

```
// C++
Request_ptr Object::_request(Identifier operation);
```

Arguments and contexts may be added after construction via the corresponding attributes in the **Request** interface. Results, output arguments, and exceptions are similarly obtained after invocation. The following C++ code illustrates usage.

```
// C++
Request_ptr req = anObj->_request("anOp");
*(req->arguments()->add(ARG_IN)->value()) <<= anArg;
// ...
req->invoke();
if (req->env()->exception() == NULL) {
     *(req->result()->value()) >>= aResult;
}
```

While this example shows the semantics of the attribute-based accessor functions, the following example shows that it is much easier and preferable to use the equivalent argument manipulation helper functions.

```
// C++
Request_ptr req = anObj->_request("anOp");
req->add_in_arg() <<= anArg;
// ...
req->invoke();
if (req->env()->exception() == NULL) {
     req->return_value() >>= aResult;
}
```

Alternatively, requests can be constructed using one of the long forms of the creation operation shown in the Object interface in Section 17.13, Object.

```
// C++
Status Object::_create_request(
                  Context_ptr ctx,
                  const char *operation,
                  NVList_ptr arg_list,
                  NamedValue_ptr result,
                  Request_ptr &request,
                  Flags req_flags
);
Status Object::_create_request(
                  Context_ptr ctx,
                  const char *operation,
                  NVList_ptr arg_list,
                  NamedValue_ptr result,
                  ExceptionList_ptr,
                  ContextList_ptr,
                  Request_ptr &request,
                  Flags req_flags
);
```

Usage is the same as for the short form except that all invocation parameters are established on construction. Note that the **OUT_LIST_MEMORY** and **IN_COPY_VALUE** flags can be set as flags in the **req_flags** parameter, but they are meaningless and thus ignored because argument insertion and extraction are done via the **any** type.

Request also allows the application to supply all information necessary for it to be invoked without requiring the ORB to utilize the Interface Repository. In order to deliver a request and return the response, the ORB requires:

- A target object reference
- An operation name
- A list of arguments (optional)
- A place to put the result (optional)
- A place to put any returned exceptions
- A **Context** (optional)
- A list of the user-defined exceptions that can be thrown (optional)
- A list of **Context** strings that must be sent with the operation (optional)

Since the **Object::create_request** operation allows all of these except the last two to be specified, an ORB may have to utilize the Interface Repository in order to discover them. Some applications, however, may not want the ORB performing potentially expensive Interface Repository look-ups during a request invocation, so two new serverless objects have been added to allow the application to specify this information instead:

- **ExceptionList**: allows an application to provide a list of **TypeCode**s for all user-defined exceptions that may result when the **Request** is invoked.

- **`ContextList`**: allows an application to provide a list of **`Context`** strings that must be supplied with the **`Request`** invocation.

The **`ContextList`** differs from the **`Context`** in that the former supplies only the context strings whose values are to be looked up and sent with the request invocation (if applicable), while the latter is where those values are obtained.

The IDL descriptions for **`ExceptionList`**, **`ContextList`**, and **`Request`** are shown next.

17.7.1 Request Interface

```
// IDL
pseudo interface ExceptionList
{
        readonly attribute unsigned long count;
        void add(in TypeCode exc);
        TypeCode item(in unsigned long index) raises(Bounds);
        Status remove(in unsigned long index) raises(Bounds);
};

pseudo interface ContextList
{
        readonly attribute unsigned long count;
        void add(in string ctxt);
        string item(in unsigned long index) raises(Bounds);
        Status remove(in unsigned long index) raises(Bounds);
};

pseudo interface Request
{
        readonly attribute Object target;
        readonly attribute Identifier operation;
        readonly attribute NVList arguments;
        readonly attribute NamedValue result;
        readonly attribute Environment env;
        readonly attribute ExceptionList exceptions;
        readonly attribute ContextList contexts;

        attribute context ctx;

        Status invoke();
        Status send_oneway();
        Status send_deferred();
        Status get_response();
        boolean poll_response();
};
```

17.7.2 Request C++ Class

```
// C++
class ExceptionList
{
  public:
     ULong count();
     void add(TypeCode_ptr tc);
     void add_consume(TypeCode_ptr tc);
     TypeCode_ptr item(ULong index);
     Status remove(ULong index);
};

class ContextList
{
  public:
     ULong count();
     void add(const char* ctxt);
     void add_consume(char* ctxt);
     const char* item(ULong index);
     Status remove(ULong index);
};

class Request
{
  public:
     Object_ptr target() const;
     const char *operation() const;
     NVList_ptr arguments();
     NamedValue_ptr result();
     Environment_ptr env();
     ExceptionList_ptr exceptions();
     ContextList_ptr contexts();

     void ctx(Context_ptr);
     Context_ptr ctx() const;

     // argument manipulation helper functions
     Any &add_in_arg();
     Any &add_in_arg(const char* name);
     Any &add_inout_arg();
     Any &add_inout_arg(const char* name);
     Any &add_out_arg();
     Any &add_out_arg(const char* name);
     void set_return_type(TypeCode_ptr tc);
     Any &return_value();
```

```
        Status invoke();
        Status send_oneway();
        Status send_deferred();
        Status get_response();
        Boolean poll_response();
};
```

17.7.3 Differences from C-PIDL

The C++-PIDL specification differs from the C-PIDL specification as follows:

- Replacement of **add_argument**, and so forth, with attribute-based accessors.
- Use of **env** attribute to access exceptions raised in DII calls.
- The **invoke** operation does not take a flag argument, since there are no flag values that are listed as legal in *CORBA V2.0*.
- The **send_oneway** and **send_deferred** operations replace the single **send** operation with flag values, in order to clarify usage.
- The **get_response** operation does not take a flag argument, and an operation **poll_response** is defined to immediately return with an indication of whether the operation has completed. This was done because in *CORBA V2.0*, if the type **Status** is **void**, the version with **RESP_NO_WAIT** does not enable the caller to determine if the operation has completed.
- The **add_*_arg**, **set_return_type**, and **return_value** member functions are added as shortcuts for using the attribute-based accessors.

17.7.4 Memory Management

Request has the following special memory management rule.

- Ownership of the return values of the **target**, **operation**, **arguments**, **result**, **env**, **exceptions**, **contexts**, and **ctx** functions is maintained by the **Request**; these return values must not be freed by the caller.

ExceptionList has the following special memory management rules.

- The **add_consume** function consumes its **TypeCode_ptr** argument. The caller may not access the object referred to by the **TypeCode_ptr** after it has been passed in because the **add_consume** function may copy it and release the original immediately.
- Ownership of the return value of the **item** function is maintained by the **ExceptionList**; this return value must not be released by the caller.

ContextList has the following special memory management rules.

- The **add_consume** function consumes its **char*** argument. The caller may not access the memory referred to by the **char*** after it has been passed in because the **add_consume** function may copy it and free the original immediately.

- Ownership of the return value of the **`item`** function is maintained by the **`ContextList`**; this return value must not be released by the caller.

17.8 Context

A **`Context`** supplies optional context information associated with a method invocation.

17.8.1 Context Interface

```
// IDL
pseudo interface Context
{
    readonly attribute Identifier context_name;
    readonly attribute context parent;

    Status create_child(in Identifier child_ctx_name, out

    Context child_ctx);

    Status set_one_value(in Identifier propname, in any

    propvalue);
    Status set_values(in NVList values);
    Status delete_values(in Identifier propname);
    Status get_values(
                    in Identifier start_scope,
                    in Flags op_flags,
                    in Identifier pattern,
                    out NVList values
    );
};
```

17.8.2 Context C++ Class

```
// C++
class Context
{
  public:
    const char *context_name() const;
    Context_ptr parent() const;

    Status create_child(const char *, Context_ptr&);

    Status set_one_value(const char *, const Any &);
    Status set_values(NVList_ptr);
    Status delete_values(const char *);
    Status get_values(
                const char*,
                Flags,
                const char*,
                NVList_ptr&
    );
};
```

17.8.3 Differences from C-PIDL

The C++-PIDL specification differs from the C-PIDL specification as follows:

- Introduction of attributes for context name and parent.
- The signatures for values are uniformly set to **any**.
- In the C mapping, **set_one_value** used strings, while others used **NamedValues** containing **any**. Even though implementations need only support strings as values, the signatures now uniformly allow alternatives.
- The **release** operation frees child contexts.

17.8.4 Memory Management

Context has the following special memory management rule.

- Ownership of the return values of the **context_name** and **parent** functions is maintained by the **Context**; these return values must not be freed by the caller.

17.9 Principal

A **Principal** represents information about principals requesting operations. There are no defined operations.

There are no differences from the C-PIDL mapping.

17.9.1 Principal Interface

```
// IDL
pseudo interface Principal {};
```

17.9.2 Principal C++ Class

```
// C++
class Principal {};
```

17.10 TypeCode

A **TypeCode** represents OMG IDL type information.

No constructors for **TypeCode**s are defined. However, in addition to the mapped interface, for each basic and defined OMG IDL type, an implementation provides access to a **TypeCode** pseudo-object reference (**TypeCode_ptr**) of the form **_tc_<type>** that may be used to set types in **any**, as arguments for **equal**, and so on. In the names of these **TypeCode** reference constants, **<type>** refers to the local name of the type within its defining scope. Each C++ **_tc_<type>** constant must be defined at the same scoping level as its matching type.

In all C++ **TypeCode** pseudo-object reference constants, the prefix **_tc_** should be used instead of the **TC_** prefix prescribed in Section 6.7, Type Codes. This is to avoid name clashes for CORBA applications that simultaneously use both the C and C++ mappings.

Like all other serverless objects, the C++ mapping for **TypeCode** provides a **_nil()** operation that returns a nil object reference for a **TypeCode**. This operation can be used to initialize **TypeCode** references embedded within constructed types. However, a nil **TypeCode** reference may never be passed as an argument to an operation, since **TypeCode**s are effectively passed as values, not as object references.

17.10.1 TypeCode Interface

```
// IDL
pseudo interface TypeCode
{
    exception Bounds {};
    exception BadKind {};

    // for all TypeCode kinds
    boolean equal(in TypeCode tc);
    TCKind kind();

    // for tk_objref, tk_struct, tk_union, tk_enum, tk_alias,

    and tk_except
    RepositoryId id() raises(BadKind);
    Identifier name() raises(BadKind);

    // for tk_struct, tk_union, tk_enum, and tk_except
    unsigned long member_count() raises(BadKind);
    Identifier member_name(in unsigned long index)
                                    raises(BadKind, Bounds);

    // for tk_struct, tk_union, and tk_except
    TypeCode member_type(in unsigned long index)
                                    raises(BadKind, Bounds);
    // for tk_union
    any member_label(in unsigned long index)
                                       raises(BadKind, Bounds);
    TypeCode discriminator_type() raises(BadKind);
    long default_index() raises(BadKind);

    // for tk_string, tk_sequence, and tk_array
    unsigned long length() raises(BadKind);

    // for tk_sequence, tk_array, and tk_alias
    TypeCode content_type() raises(BadKind);

    // deprecated interface
    long param_count();
    any parameter(in long index) raises(bounds);
};
```

17.10.2 TypeCode C++ Class

```
// C++
class TypeCode
{
  public:
    class Bounds { ... };
    class BadKind { ... };

    Boolean equal(TypeCode_ptr) const;
    TCKind kind() const;

    const char* id() const;
    const char* name() const;

    ULong member_count() const;
    const char* member_name(ULong index) const;

    TypeCode_ptr member_type(ULong index) const;

    Any *member_label(ULong index) const;
    TypeCode_ptr discriminator_type() const;
    Long default_index() const;

    ULong length() const;

    TypeCode_ptr content_type() const;

    Long param_count() const;
    Any *parameter(Long) const;
};
```

17.10.3 Differences from C-PIDL

For C++, use prefix **`_tc_`** instead of **`TC_`** for constants.

17.10.4 Memory Management

`TypeCode` has the following special memory management rule.

- Ownership of the return values of the **`id`**, **`name`**, and **`member_name`** functions is maintained by the **`TypeCode`**; these return values must not be freed by the caller.

17.11 BOA

A **`BOA`** mediates between the ORB and object implementations.

17.11.1 BOA Interface

```
// IDL
pseudo interface BOA
{

    Object create(

            in ReferenceData id,
            in InterfaceDef intf,
            in ImplementationDef impl
    );
    void dispose(in Object obj);
    ReferenceData get_id(in Object obj);
    void change_implementation(in Object obj, in

    ImplementationDef impl);
    Principal get_principal(in Object obj, in
    Environmentev);
    void impl_is_ready(in ImplementationDef impl);
    void deactivate_impl(in ImplementationDef impl);
    void obj_is_ready(in Object obj, in Implementation
    Def impl);
    void deactivate_obj(in Object obj);
};
```

17.11.2 BOA C++ Class

```
// C++
class BOA
{
  public:
     Object_ptr create(
                      const ReferenceData &,
                      InterfaceDef_ptr,
                      ImplementationDef_ptr
     );
     void dispose(Object_ptr);
     ReferenceData *get_id(Object_ptr);
     void change_implementation(
                      Object_ptr,
                      ImplementationDef_ptr
     );
     Principal_ptr get_principal(
                      Object_ptr,
                      Environment_ptr
     );
     void impl_is_ready(ImplementationDef_ptr);
     void deactivate_impl(ImplementationDef_ptr);
     void obj_is_ready(Object_ptr, ImplementationDef_ptr);
     void deactivate_obj(Object_ptr);
};
```

17.11.3 Differences from C-PIDL

Means to set exceptions are moved to **Environment**.

17.12 ORB

An **ORB** is the programmer interface to the Object Request Broker.

17.12.1 ORB Interface

```
// IDL
pseudo interface ORB
{
    typedef sequence<Request> RequestSeq;
    string object_to_string(in Object obj);
    Object string_to_object(in string str);
    Status create_list(in long count, out NVList new_list);
    Status create_operation_list(in OperationDef oper, out
    NVList new_list);
    Status create_named_value(out NamedValue nmval);
    Status create_exception_list(out ExceptionList exclist);
    Status create_context_list(out ContextList ctxtlist);

    Status get_default_context(out Context ctx);
    Status create_environment(out Environment new_env);

    Status send_multiple_requests_oneway(in RequestSeq req);
    Status send_multiple_requests_deferred(in RequestSeq
    req);
    boolean poll_next_response();
    Status get_next_response(out Request req);
};
```

17.12.2 ORB C++ Class

```
// C++
class ORB
{
  public:
     class RequestSeq {...};
     char *object_to_string(Object_ptr);
     Object_ptr string_to_object(const char *);
     Status create_list(Long, NVList_ptr&);
     Status create_operation_list(
                    OperationDef_ptr,
                    NVList_ptr&
     );
     Status create_named_value(NamedValue_ptr&);
     Status create_exception_list(ExceptionList_ptr&);
     Status create_context_list(ContextList_ptr&);

     Status get_default_context(Context_ptr&);
     Status create_environment(Environment_ptr&);

     Status send_multiple_requests_oneway(
                    const RequestSeq&
     );
     Status send_multiple_requests_deferred(
                    const RequestSeq &
     );
     Boolean poll_next_response();
     Status get_next_response(Request_ptr&);
};
```

17.12.3 Differences from C-PIDL

- Added **`create_environment`**. Unlike the struct version, **`Environment`** requires a construction operation. (Since this is overly constraining for implementations that do not support real C++ exceptions, these implementations may allow **`Environment`** to be declared on the stack. See Section D.3, Without Exception Handling, for details.)
- Assigned multiple request support to ORB, made usage symmetrical with that in **`Request`**, and used a sequence type rather than otherwise illegal unbounded arrays in signatures.
- Added **`create_named_value`**, which is required for creating **`NamedValue`** objects to be used as return value parameters for the **`Object::create_request`** operation.
- Added **`create_exception_list`** and **`create_context_list`** (see Section 17.7, Request, for more details).

17.12.4 Mapping of ORB and OA/BOA Initialization Operations

ORB Initialization

The following PIDL specifies initialization operations for an ORB; this PIDL is part of the CORBA module (not the ORB interface) and is described in Section 7.4, ORB Initialization.

```
// PIDL
module CORBA {
    typedef string ORBid;
    typedef sequence <string> arg_list;
    ORB ORB_init (inout arg_list argv, in ORBid
    orb_identifier);
};
```

The mapping of the preceding PIDL operations to C++ is as follows:

```
// C++
namespace CORBA {
    typedef char* ORBid;
    static ORB_ptr ORB_init(
                    int& argc,
                    char** argv,
                    const char* orb_identifier
    );
};
```

The C++ mapping for **`ORB_init`** (and **`OA_init`**, described in the next section) deviates from the OMG IDL PIDL in its handling of the **`arg_list`** parameter. This is intended to provide a meaningful PIDL definition of the initialization interface, which has a natural C and C++ binding. To this end, the **`arg_list`** structure is replaced with **`argv`** and **`argc`** parameters.

The **`argv`** parameter is defined as an unbound array of strings (**`char **`**), and the number of strings in the array is passed in the **`argc`** (**`int &`**) parameter.

If a NULL ORBid is used, then **`argc`** arguments can be used to determine which ORB should be returned. This is achieved by searching the **`argc`** parameters for one tagged ORBid, e.g. -ORBid “ORBid_example.”

For C++, the order of consumption of **`argv`** parameters may be significant to an application. In order to ensure that applications are not required to handle **`argv`** parameters, they do not recognize that the ORB initialization function must be called before the remainder of the parameters are consumed. Therefore, after the **`ORB_init`** call, the **`argv`** and **`argc`** parameters will have been modified to remove the ORB understood arguments. It is important to note that the **`ORB_init`** call can only reorder or remove references to parameters from the **`argv`** list; this restriction is made in order to avoid potential memory management problems caused by trying to free parts of the **`argv`** list or extending the **`argv`** list of parameters. This is why **`argv`** is passed as a **`char**`** and not a **`char**&`**.

OA/BOA Initialization

The following PIDL specifies the operations (in the ORB interface) that allow applications to get pseudo-object references; it is described in detail in Section 7.5, OA and BOA Initialization.

```
// PIDL
module CORBA {
    interface ORB {
        typedef sequence <string> arg_list;
        typedef string OAid;

        // Template for OA initialization operations
        // <OA> <OA>_init (inout arg_list argv,
        //                         in OAid oa_identifier);
        BOA BOA_init (inout arg_list argv,
                            in OAid boa_identifier);
    };
};
```

The mapping of the **`OAinit`** (**`BOA_init`**) operation (in the ORB interface) to the C++ programming language is as follows.

```
// C++
namespace CORBA {
     class ORB
     {
       public:
          typedef string OAid;

          // Template C++ binding for OA init op
          // <OA>_ptr <OA>_init  (int * argc,
          //           char **argv,
          //           OAid oa_identifier);
          BOA_ptr BOA_init(
                     int & argc,
                     char ** argv,
                     const char *boa_identifier
          );
     };
}
```

If a NULL **OAid** is used, then **argc** arguments can be used to determine which OA should be returned. This is achieved by searching the **argc** parameters for one tagged OAid, e.g. -OAid "OAid_example."

For C++, the order of consumption of **argv** parameters may be significant to an application. In order to ensure that applications are not required to handle **argv** parameters, they do not recognize that the OA initialization function must be called before the remainder of the parameters are consumed by the application. Therefore, after the **<OA>_init** call, the **argv** and **argc** parameters will have been modified to remove the OA understood arguments. It is important to note that the **OA_init** call can only reorder or remove references to parameters from the **argv** list; this restriction is made in order to avoid potential memory management problems caused by trying to free parts of the **argv** list or extending the **argv** list of parameters. This is why **argv** is passed as a **char**** and not a **char**&**.

17.12.5 Mapping of Operations to Obtain Initial Object References

The following PIDL specifies the operations (in the ORB interface) that allow applications to get pseudo-object references for the Interface Repository and Object Services. It is described in detail in Section 7.6, Obtaining Initial Object References.

```
// PIDL
module CORBA {
    interface ORB {
        typedef string ObjectId;
        typedef sequence <ObjectId> ObjectIdList;

        exception InvalidName {};

        ObjectIdList list_initial_services ();

        Object resolve_initial_references (in ObjectId
                identifier) raises (InvalidName);
    };
};
```

The mapping of the preceding PIDL to the C++ language is as follows.

```
// C++
namespace CORBA {
    class ORB {
      public:
        typedef char* ObjectId;
        class ObjectIdList {...};
        class InvalidName {...};
        ObjectIdList *list_initial_services();
        Object_ptr resolve_initial_references(
                 const char *identifier
        );
    };
}
```

17.13 Object

The rules in this section apply to OMG IDL interface **`Object`**, the base of the OMG IDL interface hierarchy. Interface **`Object`** defines a normal CORBA object, not a pseudo-object. However, it is included here because it references other pseudo-objects.

17.13.1 Object Interface

```
// IDL
interface Object
{
    boolean is_nil();
    Object duplicate();
    void release();
    ImplementationDef get_implementation();
    InterfaceDef get_interface();
    Status create_request(
            in Context ctx,
            in Identifier operation,
            in NVList arg_list,
            in NamedValue result,
            out Request request,
            in Flags req_flags
    );
    Status create_request2(
            in Context ctx,
            in Identifier operation,
            in NVList arg_list,
            in NamedValue result,
            in ExceptionList exclist,
            in ContextList ctxtlist,
            out Request request,
            in Flags req_flags
    );
};
```

17.13.2 Object C++ Class

In addition to other rules, all operation names in interface **Object** have leading underscores in the mapped C++ class. Also, the mapping for **create_request** is split into three forms, corresponding to the usage styles described in Section 4.2.1, **create_request**, and in Section 17.7, Request. The **is_nil** and **release** functions are provided in the **CORBA** name space, as described in Section 16.3.3, Object Reference Operations.

```
// C++
class Object
{
  public:
    static Object_ptr _duplicate(Object_ptr obj);
    static Object_ptr _nil();
    ImplementationDef_ptr _get_implementation();
    InterfaceDef_ptr _get_interface();
    Status _create_request(
         Context_ptr ctx,
         const char *operation,
         NVList_ptr arg_list,
         NamedValue_ptr result,
         Request_ptr &request,
         Flags req_flags
    );
    Status _create_request(
         Context_ptr ctx,
         const char *operation,
         NVList_ptr arg_list,
         NamedValue_ptr result,
         ExceptionList_ptr,
         ContextList_ptr,
         Request_ptr &request,
         Flags req_flags
    );
    Request_ptr _request(const char* operation);
};
```

Server-Side Mapping 18

Server-side mapping refers to the portability constraints for an object implementation written in C++. The term *server* is not meant to restrict implementations to situations in which method invocations cross address space or machine boundaries. This mapping addresses any implementation of an OMG IDL interface.

The required functionality for a server described here is probably a subset of the functionality an implementor will actually need. As a consequence, in practice, few servers will be completely compliant. However, we expect most of the server code to be portable from one ORB implementation to another. In particular, the body of an operation implementation will usually comply with this mapping.

18.1 Implementing Interfaces

To define an implementation in C++, one defines a C++ class with any valid C++ name. For each operation in the interface, the class defines a nonstatic member function with the mapped name of the operation (the mapped name is the same as the OMG IDL identifier except when the identifier is a C++ keyword, in which case an underscore ('_') is prepended to the identifier, as noted in Section 16.1, Preliminary Information). Note that the ORB implementation may allow one implementation class to derive from another, so the statement "the class defines a member function" does not mean the class must explicitly define the member function—it could inherit the function.

The mapping does not specify how the implementation class is related to any other classes, including the generated class for the interface. This approach allows implementations to use either inheritance or delegation and to include other features from the ORB implementation (such as choosing a default transport representation). The examples in this chapter provide sample solutions for defining implementation classes. CORBA-compliant implementations are not required to use these alternatives.

18.2 Implementing Operations

The signature of an implementation member function is the mapped signature of the OMG IDL operation. Unlike the client side, the server-side mapping requires that the function header include the appropriate exception (**`throw`**) specification. This requirement allows the compiler to detect when an invalid exception is raised, which is necessary in the case of a local C++-to-C++ library call (otherwise the call would have to go through a wrapper that checked for a valid exception). For example:

```
// IDL
interface A
{
    exception B {};
    void f() raises(B);
};

// C++
class MyFavoriteImplementationOfA ...
{
  public:
     class B : public UserException {};
     void f() throw(B);
     ...
};
```

The mapping provides two operations that are accessible from within the body of a member function: **`_this()`** and **`_boa()`**. The **`_this()`** function returns an object reference (**`T_ptr`**) for the target object. The **`_boa()`** function returns a **`BOA_ptr`** to the appropriate BOA object. The implementation may not assume where the **`_boa()`** function is defined, only that it is available within the member function. The **`_boa()`** function could be a member function, a static member function, or a static function defined in a name space that is accessible from the member functions of the implementation. The return values of **`_this()`** and **`_boa()`** must be released via **`CORBA::release()`**.

Within a member function, the “this” pointer refers to the implementation object’s data as defined by the class. In addition to accessing the data, a member function may implicitly call another member function defined by the same class. For example:

```
// IDL
interface A
{
    void f();
    void g();
};
```

```
// C++
class MyFavoriteImplementationOfA ...
{
  public:
     void f();
     void g();
  private:
     long x_;
};

void MyFavoriteImplementationOfA::f()
{
     x_ = 3;
     g();
}
```

18.3 Examples

As with other examples shown in this mapping, the following examples are not meant to mandate a particular implementation. Rather, they show some of the implementations that are possible in order to help clarify the descriptions of the mapping.

18.3.1 Using C++ Inheritance for Interface Implementation

Implementation classes can be derived from a generated base class based on the OMG IDL interface definition. The generated base classes are known as *skeleton classes*, and the derived classes are known as *implementation classes*. Each operation of the interface has a corresponding virtual member function declared in the skeleton class. The signature of the member function is identical to that of the generated client stub class. The implementation class provides implementations for these member functions. The BOA invokes the methods via calls to the skeleton class's virtual functions.

The following OMG IDL interface will be used in all the examples in this section.

```
// IDL
interface A
{
    short op1();
    void op2(in long l);
};
```

An IDL compiler generates an interface class **A** for this interface. This class contains the C++ definitions for the typedefs, constants, exceptions, attributes, and operations in the OMG IDL interface. It has a form similar to the following:

```
// C++
class A : public virtual CORBA::Object
{
  public:
     virtual Short op1() = 0;
     virtual void op2(Long l) = 0;
     ...
};
```

Some ORB implementations might not use public virtual inheritance from **CORBA::Object**, and might not make the operations pure virtual, but the signatures of the operations will be the same.

On the server side, a skeleton class can be generated. This class is partially opaque to the programmer, though it will contain a member function corresponding to each operation in the interface.

```
// C++
class _sk_A : public A
{
  public:
     // ...server-side implementation-specific detail
     // goes here...
     virtual Short op1() = 0;

     virtual void op2(Long l) = 0;
     ...
};
```

To implement this interface, a programmer must derive from this skeleton class and implement each of the operations in the OMG IDL interface. An implementation class declaration for interface **A** would take the following form:

```
// C++
class A_impl : public _sk_A
{
  public:
     Short op1();

     void op2(Long l);
     ...
};
```

18.3.2 Using Delegation for Interface Implementation

Inheritance is not always the best solution for implementing interfaces. Using inheritance from the OMG IDL–generated classes forces a C++ inheritance hierarchy on the implementor. Sometimes, the overhead of such inheritance is too high. For example, implementing OMG IDL interfaces with existing legacy code might be impossible if inheritance from some global class was enforced.

In some cases delegation can be used to good effect to solve this problem. Rather than inheriting from some global class, the implementation can be coded in any way at all, and some wrapper classes will delegate up-calls to that implementation. This section describes how this can be achieved in a type-safe manner using C++ templates.

For the examples in this section, the OMG IDL interface from Section 18.3.1, Using C++ Inheritance for Interface Implementation, will again be used.

```
// IDL
interface A
{
    short op1();
    void op2(in long l);
};
```

An OMG IDL compiler will generate a (possibly abstract) class **A** in C++ defining this interface.

Normally, the server implementor will have to derive from this class or some related class to implement a server-side object. However, the OMG IDL compiler could generate another class, called a *tie*. This class is partially opaque to the application programmer, though like the skeleton, it provides a method corresponding to each OMG IDL operation.

```
// C++
template <class T>
class _tie_A : public A
{
  public:
     _tie_A(T &t);
     Short op1();
     void op2(Long l);
     ...
};
```

This class performs the task of delegation. When the template is instantiated with a class that supports the operations of **A**, then the **_tie_A** class will delegate all operations to that implementation class. When an instance of this class is created, then a reference to the actual implementation class is passed to the constructor. Typically the implementation will just call the corresponding method in the implementation class via this reference.

```
// C++
template <class T>
class _tie_A : public A
{
  public:
     _tie_A(T &t) : _ref(t) {}
     Short op1() {return _ref.op1();}
     void op2(Long l) {_ref.op2(l);}

  private:
     T &_ref;
};
```

18.4 Mapping of Dynamic Skeleton Interface to C++

Section 5.3, Dynamic Skeleton Interface: Language Mapping, contains general information about mapping the Dynamic Skeleton Interface to programming languages.

This section contains the following information:

- Mapping of the Dynamic Skeleton Interface's ServerRequest to C++
- Mapping of the Basic Object Adapter's Dynamic Implementation Routine to C++

18.4.1 Mapping of ServerRequest to C++

The **ServerRequest** pseudo-object maps to a C++ class in the CORBA name space, which supports the following operations and signatures.

```
// C++
class ServerRequest
{
  public:
     Identifier op_name() throw(SystemException);
     OperationDef_ptr op_def() throw(SystemException);
     Context_ptr ctx() throw(SystemException);
     void params(NVList_ptr parameters)
                  throw(SystemException);
     void result(Any *value) throw(SystemException);
     void exception(Any *value) throw(SystemException);
};
```

Note that, as with the rest of the C++ mapping, ORB implementations are free to make such operations virtual, and modify the inheritance as needed.

All of these operations follow the normal memory management rules for data passed into skeletons by the ORB. That is, the DIR is not allowed to modify or change the string returned by **op_name()**, **in** parameters in the NVList, or the context returned by **`ctx()`**. Similarly, data allocated by the DIR and handed to the ORB (the NVList parameters, any result value, and exception values) is freed by the ORB rather than by the DIR.

18.4.2 Handling Operation Parameters and Results

The **`ServerRequest`** provides parameter values when the DIR invokes the **`params()`** operation. The NVList provided by the DIR to the ORB includes the TypeCodes (inside a NamedValue) for all parameters, including **`out`** ones (their values are null pointers at first), for the operation. This allows the ORB to verify that the correct parameter types have been provided before filling their values in, but does not require it to do so. It also relieves the ORB of all responsibility to consult the interface repository, promoting high-performance implementations.

The NVList provided to the ORB then becomes owned by the ORB. It will not be deallocated until after the DIR returns. This allows the DIR to pass the **`out`** values, including the return side of **`inout`** values, to the ORB by modifying the NVList after **`params()`** has been called.

In order to guarantee that the ORB could always verify parameter lists, and to detect errors such as omitted parameters, Dynamic Implementation Routines are always required to call **`params()`**, even when the DIR believes that no parameters are used by the operation. When the DIR believes no parameters are used by the operation, it passes an empty NVList.

The **`ServerRequest`** will not send a response to the invocation until the DIR returns. If a return value is required, the **`result()`** operation must be invoked to provide that value to the ORB. Where no return value is required, this need not be invoked.

The **`params()`** and **`result()`** operations may be called only once, and in that exact order.

18.4.3 Sample Usage

In typical use, the DIR receives an up-call. It will determine the operation signature by using **`op_def()`** to consult a private cache of **`OperationDef`** information. This allows it to create an NVList and fill in the TypeCodes for all the operation’s parameters: the **`in`** values, **`out`** values, and **`inout`** values. Then the DIR calls **`params()`** with that NVList. At this point, the value pointers for all **`in`** and **`inout`** (the input side only) parameters in that NVList are valid.

The DIR then performs the work for the request, using the target object reference to determine to which real object the request relates. Next, it stores the value pointers for **`out`** and **`inout`** parameters into the NVList, and reports any **`result()`** data. It then returns from the DIR up-call, signifying to the ORB that it could send any response message. Finally, the ORB frees the data allocated by the DIR (in the NVList and in the result) after it to the client.

18.4.4 Reporting Exceptions

To report an exception, rather than provide return values, the DIR provides the exception value inside an **any**, and passes that to **exception()**. As with result data, the data would be freed by the ORB after the DIR returns. (The DIR cannot in general throw exceptions, since in order to "throw" or "catch," C++ systems require type information that can only be generated at compile time. DSI, like DII, cannot rely on such compile-time support.)

All exceptions are presented as values embedded in an **any**. This is required since the use of C++ catch/throw for user-defined exceptions relies on data generated by a C++ compiler, which will not be available to general bridges (which are constructed without any OMG IDL compiler support).

The **exception()** routine can be called only once, after **params()** is called. It may not be called if **result()** has been called.

18.4.5 Mapping of BOA's Dynamic Implementation Routine

C++ server side mappings, implementation objects are C++ objects. To use the DSI, an object implements a class in the BOA name space that has a single member function with the following signature:

```
// C++
class DynamicImplementation
{
  public:
     virtual void invoke(
                      CORBA::ServerRequestRef request,
                      CORBA::Environment&env
     ) throw (
// NO exceptions... uses ServerRequest::exception()
         ) = 0;
     ...
};
```

The **env** parameter is used in the **BOA::get_principal()** operation. Note that, as with the rest of the C++ mapping, the implementation inherits this interface, and may support other methods as well.

As with other C++ based operation implementations, two functions are accessible within the body of **methods: _this()**, returning an object reference **(Object_ptr)** for the target object, and **_boa()**, returning a **BOA_ptr** to the appropriate BOA. The method code may not assume where these two routines are defined.

C++ Definitions for CORBA E

This appendix provides a complete set of C++ definitions for the **CORBA** module. The definitions appear within the C++ name space named **CORBA**.

```
// C++
namespace CORBA { ... }
```

Any implementations shown here are merely sample implementations; they are not the required definitions for these types.

E.1 Primitive Types

```
typedef unsigned char Boolean;
typedef unsigned char Char;
typedef unsigned char Octet;
typedef short Short;
typedef unsigned short UShort;
typedef long Long;
typedef unsigned long ULong;
typedef float Float;
typedef double Double;
```

E.2 String_var Class

```
class String_var
{
  public:
   String_var();
   String_var(char *p);
   String_var(const char *p);
   String_var(const String_var &s);
   ~String_var();
   String_var &operator=(char *p);
```

```
      String_var &operator=(const char *p);
      String_var &operator=(const String_var &s);
      operator char*();
      operator const char*() const;
      char &operator[](ULong index);
      char operator[](ULong index) const;
   };
```

E.3 Any Class

```
class Any
{
 public:
   Any();
   Any(const Any&);
   Any(TypeCode_ptr tc, void *value, Boolean release = FALSE);
   ~Any();

   Any &operator=(const Any&);

   void operator<<=(Short);
   void operator<<=(UShort);
   void operator<<=(Long);
   void operator<<=(ULong);
   void operator<<=(Float);
   void operator<<=(Double);
   void operator<<=(const Any&);
   void operator<<=(const char*);

   Boolean operator>>=(Short&) const;
   Boolean operator>>=(UShort&) const;
   Boolean operator>>=(Long&) const;
   Boolean operator>>=(ULong&) const;
   Boolean operator>>=(Float&) const;
   Boolean operator>>=(Double&) const;

Boolean operator>>=(Any&) const;
Boolean operator>>=(char*&) const;

// special types needed for boolean, octet, char,
// and bounded string insertion
// these are suggested implementations only
struct from_boolean {
   from_boolean(Boolean b) : val(b) {}
   Boolean val;
};
struct from_octet {
   from_octet(Octet o) : val(o) {}
   Octet val;
};
```

```
struct from_char {
   from_char(Char c) : val(c) {}
   Char val;
};
struct from_string {
   from_string(char* s, ULong b) : val(s), bound(b) {}
   char *val;
   ULong bound;
};

void operator<<=(from_boolean);
void operator<<=(from_char);
void operator<<=(from_octet);
void operator<<=(from_string);

// special types needed for boolean, octet, char extraction
// these are suggested implementations only
struct to_boolean {
   to_boolean(Boolean &b) : ref(b) {}
   Boolean &ref;
};
struct to_char {
   to_char(Char &c) : ref(c) {}
   Char &ref;
};
struct to_octet {
   to_octet(Octet &o) : ref(o) {}
   Octet &ref;
};
struct to_object {
   to_object(Object_ptr &obj) : ref(obj) {}
   Object_ptr &ref;
};
struct to_string {
      to_string(char *&s, ULong b) : val(s), bound(b) {}
      char *&val;
      ULong bound;
   };

   Boolean operator>>=(to_boolean) const;
   Boolean operator>>=(to_char) const;
   Boolean operator>>=(to_octet) const;
   Boolean operator>>=(to_object) const;
   Boolean operator>>=(to_string) const;

   void replace(TypeCode_ptr, void *value, Boolean release =
   FALSE);

   TypeCode_ptr type() const;
   const void *value() const;

 private:
   // these are hidden and should not be implemented
   // so as to catch erroneous attempts to insert or extract
   // multiple IDL types mapped to unsigned char
   void operator<<=(unsigned char);
   Boolean operator>>=(unsigned char&) const;
};
```

E.4 Any_var Class

```
class Any_var
{
  public:
   Any_var();
   Any_var(Any *a);
   Any_var(const Any_var &a);
   ~Any_var();

   Any_var &operator=(Any *a);
   Any_var &operator=(const Any_var &a);

   Any *operator->();
   // other conversion operators for parameter passing
};
```

E.5 Exception Class

```
// C++
class Exception
{
  public:
   Exception(const Exception &);
   ~Exception();
   Exception &operator=(const Exception &);

  protected:
   Exception();
};
```

E.6 SystemException Class

```
// C++
enum CompletionStatus { COMPLETED_YES, COMPLETED_NO,
COMPLETED_MAYBE };
class SystemException : public Exception
{
  public:
   SystemException();
   SystemException(const SystemException &);
   SystemException(ULong minor, CompletionStatus status);
   ~SystemException();
   SystemException &operator=(const SystemException &);

   ULong minor() const;
   void minor(ULong);

   CompletionStatus completed() const;
   void completed(CompletionStatus);
};
```

E.7 UserException Class

```
// C++
class UserException : public Exception
{
  public:
   UserException();
   UserException(const UserException &);
   ~UserException();
   UserException &operator=(const UserException &);
};
```

E.8 UnknownUserException Class

```
// C++
class UnknownUserException : public UserException
{
  public:
   Any &exception();
};
```

E.9 release and is_nil

```
// C++
namespace CORBA {
   void release(Object_ptr);
   void release(Environment_ptr);
   void release(NamedValue_ptr);
   void release(NVList_ptr);
   void release(Request_ptr);
   void release(Context_ptr);
   void release(Principal_ptr);
   void release(TypeCode_ptr);
   void release(BOA_ptr);
   void release(ORB_ptr);

   Boolean is_nil(Object_ptr);
   Boolean is_nil(Environment_ptr);
   Boolean is_nil(NamedValue_ptr);
   Boolean is_nil(NVList_ptr);
   Boolean is_nil(Request_ptr);
   Boolean is_nil(Context_ptr);
   Boolean is_nil(Principal_ptr);
   Boolean is_nil(TypeCode_ptr);
   Boolean is_nil(BOA_ptr);
   Boolean is_nil(ORB_ptr);
   ...
}
```

E.10 Object Class

```
// C++
class Object
{
  public:
   static Object_ptr _duplicate(Object_ptr obj);
   static Object_ptr _nil();
   ImplementationDef_ptr _get_implementation();
   InterfaceDef_ptr _get_interface();
   Status _create_request(
            Context_ptr ctx,
            const char *operation,
            NVList_ptr arg_list,
            NamedValue_ptr result,
            Request_ptr &request,
            Flags req_flags
         );
   Status _create_request(
      Context_ptr ctx,
      const char *operation,
      NVList_ptr arg_list,
      NamedValue_ptr result,
      ExceptionList_ptr,
      ContextList_ptr,
      Request_ptr &request,
      Flags req_flags
   );
   Request_ptr _request(const char* operation);
};
```

E.11 Environment Class

```
// C++
class Environment
{
  public:
   void exception(Exception*);
   Exception *exception() const;
   void clear();

   static Environment_ptr _duplicate();
   static Environment_ptr _nil();
};
```

E.12 NamedValue Class

```
// C++
class NamedValue
{
  public:
   const char *name() const;
   Any *value() const;
   Flags flags() const;

   static NamedValue_ptr _duplicate();
   static NamedValue_ptr _nil();
};
```

E.13 NVList Class

```
// C++
class NVList
{
  public:
   ULong count() const;
   NamedValue_ptr add(Flags);
   NamedValue_ptr add_item(const char*, Flags);
   NamedValue_ptr add_value(const char*, const Any&, Flags);
   NamedValue_ptr add_item_consume(
      char*,
         Flags
   );
   NamedValue_ptr add_value_consume(
         char*,
         Any *,
         Flags
   );
   NamedValue_ptr item(ULong);
   Status remove(ULong);

   static NVList_ptr _duplicate();
   static NVList_ptr _nil();
};
```

E.14 ExceptionList Class

```
// C++
class ExceptionList
{
 public:
   ULong count();
   void add(TypeCode_ptr tc);
   void add_consume(TypeCode_ptr tc);
   TypeCode_ptr item(ULong index);
   Status remove(ULong index);
};
```

E.15 ContextList Class

```
class ContextList
{
  public:
   ULong count();
   void add(const char* ctxt);
   void add_consume(char* ctxt);
   const char* item(ULong index);
   Status remove(ULong index);
};
```

E.16 Request Class

```
// C++
class Request
{
  public:
   Object_ptr target() const;
   const char *operation() const;
   NVList_ptr arguments();
   NamedValue_ptr result();
   Environment_ptr env();
   ExceptionList_ptr exceptions();
   ContextList_ptr contexts();

   void ctx(Context_ptr);
   Context_ptr ctx() const;
   // argument manipulation helper functions
   Any &add_in_arg();
   Any &add_in_arg(const char* name);
   Any &add_inout_arg();
   Any &add_inout_arg(const char* name);
   Any &add_out_arg();
   Any &add_out_arg(const char* name);
   void set_return_type(TypeCode_ptr tc);
   Any &return_value();

   Status invoke();
   Status send_oneway();
   Status send_deferred();
   Status get_response();
   Boolean poll_response();

   static Request_ptr _duplicate();
   static Request_ptr _nil();
};
```

E.17 Context Class

```
// C++
class Context
{
  public:
   const char *context_name() const;
   Context_ptr parent() const;

   Status create_child(const char*, Context_ptr&);

   Status set_one_value(const char*, const Any&);
   Status set_values(NVList_ptr);
   Status delete_values(const char*);
   Status get_values(const char*, Flags, const char*,
   NVList_ptr&);

   static Context_ptr _duplicate();
   static Context_ptr _nil();
};
```

E.18 Principal Class

```
// C++
class Principal
{
  public:
   static Principal_ptr _duplicate();
   static Principal_ptr _nil();
};
```

E.19 TypeCode Class

```
// C++
class TypeCode
{
  public:
   class Bounds { ... };
   class BadKind { ... };

   TCKind kind() const;
   Boolean equal(TypeCode_ptr) const;

   const char* id() const;
   const char* name() const;

   ULong member_count() const;
   const char* member_name(ULong index) const;

   TypeCode_ptr member_type(ULong index) const;

   Any *member_label(ULong index) const;
   TypeCode_ptr discriminator_type() const;
   Long default_index() const;
   ULong length() const;

   TypeCode_ptr content_type() const;

   Long param_count() const;
   Any *parameter(Long) const;

   static TypeCode_ptr _duplicate();
   static TypeCode_ptr _nil();
};
```

E.20 BOA Class

```
// C++
class BOA
{
  public:
   Object_ptr create(
               const ReferenceData&,
               InterfaceDef_ptr,
               ImplementationDef_ptr
            );
   void dispose(Object_ptr);
   ReferenceData *get_id(Object_ptr);
   void change_implementation(Object_ptr, ImplementationDef_ptr);
   Principal_ptr get_principal(Object_ptr, Environment_ptr);
   void impl_is_ready(ImplementationDef_ptr);
   void deactivate_impl(ImplementationDef_ptr);
   void obj_is_ready(Object_ptr, ImplementationDef_ptr);
   void deactivate_obj(Object_ptr);

   static BOA_ptr _duplicate();
   static BOA_ptr _nil();
};
```

E.21 ORB Class

```
// C++
class ORB
{
  public:
   typedef sequence<Request_ptr> RequestSeq;
   char *object_to_string(Object_ptr);
   Object_ptr string_to_object(const char*);
   Status create_list(Long, NVList_ptr&);
   Status create_operation_list(OperationDef_ptr, NVList_ptr&);
   Status create_named_value(NamedValue_ptr&);
   Status create_exception_list(ExceptionList_ptr&);
   Status create_context_list(ContextList_ptr&);

   Status get_default_context(Context_ptr&);
   Status create_environment(Environment_ptr&);

   Status send_multiple_requests_oneway(const RequestSeq&);
   Status send_multiple_requests_deferred(const RequestSeq&);
   Boolean poll_next_response();
   Status get_next_response(Request_ptr&);

   // OA initialization
   typedef string OAid;

   // Template C++ binding for OA init op
   // <OA>_ptr <OA>_init(int * argc,
   //                       char **argv,
   //                       OAid oa_identifier);
   BOA_ptr BOA_init(int & argc, char ** argv, const char
   *boa_identifier);

   // Obtaining initial object references
   typedef char* ObjectId;
   class ObjectIdList {...};
   class InvalidName {...};
   ObjectIdList *list_initial_services();
   Object_ptr resolve_initial_references(const char *identifier);

   static ORB_ptr _duplicate();
   static ORB_ptr _nil();
};
```

E.22 ORB Initialization

```
// C++
typedef char* ORBid;
static ORB_ptr ORB_init(
                        int& argc,
                        char** argv,
                        const char* orb_identifier
               );
```

E.23 ServerRequest Class

```
// C++
class ServerRequest
{
  public:
     Identifier op_name() throw(SystemException);
     OperationDef_ptr op_def() throw(SystemException);
     Context_ptr ctx() throw(SystemException);
     void params(NVList_ptr parameters)
                   throw(SystemException);
     void result(Any *value) throw(SystemException);
     void exception(Any *value) throw(SystemException);
};
```

Alternative Mappings for C++ Dialects F

This appendix describes alternative mappings for C++ dialects that do not match the assumptions specified in Section 15.1.2, C++ Implementation Requirements. Conforming implementations do not have to provide these workarounds if their C++ compiler supports the required features.

F.1 64-bit Integers

IDL translators that support 64-bit integer types should map the signed type to **`LongLong`** and the unsigned type to **`ULongLong`**, where both names are defined in the **`CORBA`** name space.

F.2 Without Name Spaces

If the target environment does not support the **`namespace`** construct but does support nested classes, then a module should be mapped to a C++ class. If the environment does not support nested classes, then the mapping for modules should be the same as for the CORBA C mapping (concatenating identifiers using an underscore ("_") character as the separator).

Note that module constants map to file-scope constants on systems that support name spaces and class-scope constants on systems that map modules to classes.

F.3 Without Exception Handling

For those C++ environments that do not support real C++ exception handling, referred to here as *non-exception handling (non-EH) C++ environments*, an **`Environment`** parameter passed to each operation is used to convey exception information to the caller.

As shown in Section 17.4, Environment, the **`Environment`** class supports the ability to access and modify the **`Exception`** it holds.

As shown in Section 16.15, Mapping for Exception Types, both user-defined and system exceptions form an inheritance hierarchy that normally allow types to be caught either by their actual type or by a more general base type. When used in a non-EH C++ environment, the narrowing functions provided by this hierarchy allow for examination and manipulation of exceptions.

```
// IDL
interface A
{
exception Broken { ... };
void op() raises(Broken);
};

// C++
Environment ev;
A_ptr obj = ...
obj->op(ev);
if (Exception *exc = ev.exception()) {
if (A::Broken *b = A::Broken::_narrow(exc)) {
// deal with user exception
} else {
// must have been a system exception
SystemException *se = SystemException::_narrow(exc);
...
}
}
```

Section 17.12, ORB, specifies that **`Environment`** must be created using **`ORB::create_environment`**, but this is overly constraining for implementations requiring an **`Environment`** to be passed as an argument to each method invocation. For implementations that do not support real C++ exceptions, **`Environment`** may be allocated as a static, automatic, or heap variable. For example, all of the following are legal declarations on a non-EH C++ environment.

```
// C++
Environment global_env;       // global
static Environment static_env;// file static

class MyClass
{
   public:
...
  private:
static Environment class_env; // class static

};

void func()
{
Environment auto_env;         // auto
Environment *new_env = new Environment;// heap
...
}
```

For ease of use, **`Environment`** parameters are passed by reference in non-EH environments.

```
// IDL
interface A
{
exception Broken { ... };
void op() raises(Broken);
};
```

```
// C++
class A ...
{
  public:
void op(Environment &);
...
};
```

For additional ease of use in non-EH environments, **`Environment`** should support copy construction and assignment from other **`Environment`** objects. These additional features are helpful for propagating exceptions from one **`Environment`** to another under non-EH circumstances.

When an exception is "thrown" in a non-EH environment, object implementors and ORB run-times must ensure that all **`out`** and return pointers are returned to the caller as null pointers. If noninitialized or "garbage" pointer values are returned, client application code could experience run-time errors due to the assignment of bad pointers to **`T_var`** types. When a **`T_var`** goes out of scope, it attempts to **`delete`** the **`T*`** given to it; if this pointer value is garbage, a run-time error will almost certainly occur.

F.4 Without Run-Time Type Information (RTTI)

For C++ environments that do not support RTTI, the **`Exception`** class provides for narrowing within the exception hierarchy.

```
// C++
class UserException : public Exception
{
  public:
static UserException *_narrow(Exception *);
};

class SystemException : public Exception
{
  public:
static SystemException *_narrow(Exception *);
};
```

Each exception class supports a static member function named **_narrow**. The parameter to the **_narrow** call is a pointer to the base class **Exception**. If the parameter is a null pointer, the return type of **_narrow** is a null pointer. If the actual (run-time) type of the parameter exception can be widened to the requested exception's type, then **_narrow** will return a valid pointer to the parameter **Exception**. Otherwise, **_narrow** will return a null pointer.

Unlike the **_narrow** operation on object references, the **_narrow** operation on exceptions returns a suitably-typed pointer to the same exception parameter, not a pointer to a new exception. If the original exception goes out of scope or is otherwise destroyed, the pointer returned by **_narrow** is no longer valid.

C++ Keywords G

Table G-1 lists all C++ keywords from the 4/28/95 Committee Draft of the ANSI (X3J16) C++ Language Standardization Committee.

Table G-1 C++ Keywords

and	and_eq	asm	auto	bitand
bitor	bool	break	case	catch
char	class	compl	const	const_cast
continue	default	delete	do	double
dynamic_cast	else	enum	explicit	extern
false	float	for	friend	goto
if	inline	int	long	mutable
namespace	new	not	not_eq	operator
or	or_eq	private	protected	public
register	reinterpret_cast	return	short	signed
sizeof	static	static_cast	struct	switch
template	this	throw	true	try
typedef	typeid	typename	union	unsigned
using	virtual	void	volatile	wchar_t
while	xor	xor_eq		

Smalltalk Mapping Overview 19

This chapter provides the following information:

- A rationale for the design of the Smalltalk mapping
- An overview of how the Smalltalk mapping is organized in this manual
- A mini-glossary of terms used in the Smalltalk chapters
- Requirements for an implementation of an OMG IDL–to–Smalltalk mapping
- Constraints imposed on an implementation of the OMG IDL–to–Smalltalk mapping

19.1 Key Design Decisions

The mapping of OMG IDL to the Smalltalk programming language was designed with the following goals in mind:

- The Smalltalk mapping does not prescribe a specific implementation. Smalltalk class names are specified, as needed, since client code will need the class name when generating instances of datatypes. A minimum set of messages that classes must support is listed for classes that are not documented in the Smalltalk Common Base. The inheritance structure of classes is never specified.
- Whenever possible, OMG IDL types are mapped directly to existing, portable Smalltalk classes.
- The Smalltalk constructs defined in this mapping rely primarily upon classes and methods described in the Smalltalk Common Base document.
- The Smalltalk mapping only describes the public (client) interface to Smalltalk classes and objects supporting IDL. Individual IDL compilers or CORBA implementations might define additional private interfaces.
- The implementation of OMG IDL interfaces is left unspecified. Implementations may choose to map each OMG IDL interface to a separate Smalltalk class; provide one Smalltalk class to map all OMG IDL interfaces; or allow arbitrary Smalltalk classes to map OMG IDL interfaces.

- Because of the dynamic nature of Smalltalk, the mapping of the **any** and **union** types is such that an explicit mapping is unnecessary. Instead, the value of the **any** and **union** types can be passed directly. In the case of the **any** type, the Smalltalk mapping will derive a **TypeCode** which can be used to represent the value. In the case of the **union** type, the Smalltalk mapping will derive a discriminator which can be used to represent the value.
- The explicit passing of environment and context values on operations is not required.
- Except in the case of object references, no memory management is required for data parameters and return results from operations. All such Smalltalk objects reside within Smalltalk memory, so garbage collection will reclaim their storage when they are no longer used.
- The proposed language mapping has been designed with the following vendor's Smalltalk implementations in mind: VisualWorks; Smalltalk/V; and VisualAge.

19.1.1 Consistency of Style, Flexibility and Portability of Implementation

To ensure flexibility and portability of implementations, and to provide a consistent style of language mapping, the Smalltalk chapters use the programming style and naming conventions as described in the following documents:

- Goldberg, Adele and Robson, David. *Smalltalk-80: The Language*. Addison-Wesley Publishing Company, Reading, MA. 1989.
- *Smalltalk Portability: A Common Base*. ITSC Technical Bulletin GG24-3093, IBM, Boca Raton, FL. September 1992.

(Throughout the Smalltalk chapters, *Smalltalk Portability: A Common Base* is referred to as *Smalltalk Common Base*.)

The items listed below are the same for all Smalltalk classes used in the Smalltalk mapping:

- If the class is described in the Smalltalk Common Base document, the class must conform to the behavior specified in the document. If the class is not described in the Smalltalk Common Base document, the minimum set of class and instance methods that must be available is described for the class.
- All data types (except object references) are stored completely within Smalltalk memory, so no explicit memory management is required.
- The mapping is consistent with the common use of Smalltalk. For example, **sequence** is mapped to instances of **`OrderedCollection`**, instead of creating a Smalltalk class for the mapping.

19.2 Organization of the Smalltalk Mapping

In addition to this overview, the mapping of OMG IDL to the Smalltalk programming language is divided into the following chapters:

- Mapping of all OMG IDL constructs (as defined in Chapter 3, OMG IDL Syntax and Semantics) to Smalltalk constructs
- Mapping of OMG IDL pseudo-objects to Smalltalk

19.3 Glossary of Terms

Smalltalk object. An object defined using the Smalltalk language.

Message. Invocation of a Smalltalk method upon a Smalltalk object.

Message Selector. The name of a Smalltalk message. In this document, the message selectors are denoted by just the message name when the class or protocol they are associated with is given in context, otherwise the notation **`class`**>>**`method`** or ***`protocol`***>>**`method`** will be used to explicitly denote the class or protocol the message is associated with.

Method. The Smalltalk code associated with a message.

Class. A Smalltalk class.

Protocol. A set of messages that a Smalltalk object must respond to. Protocols are used to describe the behavior of Smalltalk objects without specifying their class.

CORBA Object. An object defined in OMG IDL, accessed and implemented through an ORB.

Object Reference. A value which uniquely identifies an object.

IDL compiler. Any software that accesses OMG IDL specifications and generates or maps Smalltalk code that can be used to access CORBA objects.

19.4 Implementation Constraints

This sections describes how to avoid potential problems with an OMG IDL–to–Smalltalk implementation.

19.4.1 Avoiding Name Space Collisions

There is one aspect of the language mapping that can cause an OMG IDL compiler to map to incorrect Smalltalk code and cause name space collisions. Because Smalltalk implementations generally only support a global name space, and disallow underscore characters in identifiers, the mapping of identifiers used in OMG IDL to Smalltalk identifiers can result in a name collision. See Section 20.2, “Conversion of Names to Smalltalk Identifiers,” on page 20-2 for a description of the name conversion rules.

As an example of name collision, consider the following OMG IDL declaration:

```
interface Example {
    void sample_op () ;
void sampleOp () ;
};
```

Both of these operations map to the Smalltalk selector **sampleOp**. In order to prevent name collision problems, each implementation must support an explicit naming mechanism, which can be used to map an OMG IDL identifier into an arbitrary Smalltalk identifier. For example, **#pragma** directives could be used as the mechanism.

19.4.2 Limitations on OMG IDL Types

This language mapping places limitations on the use of certain types defined in OMG IDL.

For the **any** and **union** types, specific integral and floating point types may not be able to be specified as values. The implementation will map such values into an appropriate type, but if the value can be represented by multiple types, the one actually used cannot be determined.[1] For example, consider the **union** definition below.

```
union Foo switch (long) {
 case 1: long x;
 case 2: short y;
};
```

When a Smalltalk object corresponding to this union type has a value that fits in both a **long** and a **short**, the Smalltalk mapping can derive a discriminator 1 or 2, and map the integral value into either a **long** or **short** value (corresponding to the value of the discriminator determined).

19.5 Smalltalk Implementation Requirements

This mapping places requirements on the implementation of Smalltalk that is being used to support the mapping. These are:

- An integral class, conforming to the **Integer** class definition in the Smalltalk Common Base.
- A floating point class, conforming to the **Float** class definition in the Smalltalk Common Base.
- A class named **Character** conforming to the **Character** class definition in the Smalltalk Common Base.
- A class named **Array** conforming to the **Array** class definition in the Smalltalk Common Base.
- A class named **OrderedCollection** conforming to the **OrderedCollection** class definition in the Smalltalk Common Base.
- A class named **Dictionary** conforming to the **Dictionary** class definition in the Smalltalk Common Base.

1.To avoid this limitation for union types, the mapping allows programmers to specify an explicit binding to retain the value of the discriminator. See Section 20.12, “Mapping for Union Types,” on page 20-8 for a complete description.

- A class named **`Association`** conforming to the **`Association`** class definition in the Smalltalk Common Base.
- A class named **`String`** conforming to the **`String`** class definition in the Smalltalk Common Base.
- Objects named **`true`**, **`false`** conforming to the methods defined for **`Boolean`** objects, as specified in the Smalltalk Common Base.
- An object named **`nil`**, representing an object without a value.
- A global variable named **`Processor`**, which can be sent the message **`activeProcess`** to return the current Smalltalk process, as defined in the document *Smalltalk-80: The Language*. This Smalltalk process must respond to the messages **`corbaContext:`** and **`corbaContext`**.
- A class which conforms to the ***`CORBAParameter`*** protocol. This protocol defines Smalltalk instance methods used to create and access **inout** and **out** parameters. The protocol must support the following instance messages:

value

Answers the value associated with the instance

value: anObject

Resets the value associated with the instance to **`anObject`**

To create an object that supports the ***`CORBAParameter`*** protocol, the message **`asCORBAParameter`** can be sent to any Smalltalk object. This will return a Smalltalk object conforming to the ***`CORBAParameter`*** protocol, whose value will be the object it was created from. The value of that ***`CORBAParameter`*** object can be subsequently changed with the **`value`**: message.

Mapping of OMG IDL to Smalltalk 20

This chapter describes the mapping of OMG IDL constructs to Smalltalk constructs.

20.1 *Mapping Summary*

TABLE 30 on page 20-1 provides a brief description of the mapping of OMG IDL constructs to the Smalltalk language, and where in this chapter they are discussed.

Table 20-1 Summary of this Chapter

OMG IDL Construct	**Smalltalk Mapping**	**Where Discussed**
Interface	Set of messages that Smalltalk objects which represent object references must respond to. The set of messages corresponds to the attributes and operations defined in the interface and inherited interfaces.	Section 20.3, “Mapping for Interfaces,” on page 20-3.
Object Reference	Smalltalk object that represents a CORBA object. The Smalltalk object must respond to all messages defined by a CORBA object’s interface.	Section 20.5, “Mapping for Objects,” on page 20-3.
Operation	Smalltalk message.	Section 20.1.7, “Mapping for Operations,” on page 20-10.
Attribute	Smalltalk message.	Section 20.7, ”Mapping for Attributes,” on page 20-4.
Constant	Smalltalk objects available in the CORBAConstants dictionary.	Section 20.7.1, “Mapping for Constants,” on page 20-5.
Integral Type	Smalltalk objects that conform to the `Integer` class.	Section 20.8, “Mapping for Basic Data Types,” on page 20-5.

Table 20-1 Summary of this Chapter *(Continued)*

OMG IDL Construct	Smalltalk Mapping	Where Discussed
Floating Point Type	Smalltalk objects which conform to the **`Float`** class.	Described in Section 20.8, "Mapping for Basic Data Types," on page 20-5.
Boolean Type	Smalltalk **`true`** or **`false`** objects.	Described in Section 20.8, "Mapping for Basic Data Types," on page 20-5.
Enumeration Type	Smalltalk objects which conform to the ***`CORBAEnum`*** protocol.	Section 20.10, "Mapping for Enums," on page 20-7.
Any Type	Smalltalk objects that can be mapped into an OMG IDL type.	Section 20.9, "Mapping for the any Type," on page 20-7.
Structure Type	Smalltalk object that conforms to the **`Dictionary`** class.	Section 20.11, "Mapping for Struct Types," on page 20-8.
Union Type	Smalltalk object that maps to the possible value types of the OMG IDL union or that conform to the ***`CORBAUnion`*** protocol.	Section 20.12, "Mapping for Union Types," on page 20-8.
Sequence Type	Smalltalk object that conforms to the **`OrderedCollection`** class.	Section 20.13, "Mapping for Sequence Types," on page 20-10.
String Type	Smalltalk object that conforms to the **`String`** class.	Section 20.14, "Mapping for String Types," on page 20-10.
Array Type	Smalltalk object that conforms to the **`Array`** class.	Section 20.15, "Mapping for Array Types," on page 20-10.
Exception Type	Smalltalk object that conforms to the **`Dictionary`** class.	Section 20.16, "Mapping for Exception Types," on page 20-10.

20.2 Conversion of Names to Smalltalk Identifiers

The use of underscore characters in OMG IDL identifiers is not allowed in all Smalltalk language implementations. Thus, a conversion algorithm is required to convert names used in OMG IDL to valid Smalltalk identifiers.

To convert an OMG IDL identifier to a Smalltalk identifier, remove each underscore and capitalize the following letter (if it exists). In order to eliminate possible ambiguities which may result from these conventions, an explicit naming mechanism must also be provided by the implementation. For example, the **#pragma** directive could be used.

For example, the OMG IDL identifiers:

add_to_copy_map
describe_contents

become Smalltalk identifiers

`addToCopyMap`
`describeContents`

Smalltalk implementations generally require that class names and global variables have an uppercase first letter, while other names have a lowercase first letter.

20.3 Mapping for Interfaces

Each OMG IDL interface defines the operations that object references with that interface must support. In Smalltalk, each OMG IDL interface defines the methods that object references with that interface must respond to.

Implementations are free to map each OMG IDL interface to a separate Smalltalk class, map all OMG IDL interfaces to a single Smalltalk class, or map arbitrary Smalltalk classes to OMG IDL interfaces.

20.4 Memory Usage

One of the design goals is to make every Smalltalk object used in the mapping a pure Smalltalk object: namely datatypes used in mappings do not point to operating system defined memory. This design goal permits the mapping and users of the mapping to ignore memory management issues, since Smalltalk handles this itself (via garbage collection). Smalltalk objects which are used as object references may contain pointers to operating system memory, and so must be freed in an explicit manner.

20.5 Mapping for Objects

A CORBA object is represented in Smalltalk as a Smalltalk object called an *object reference*. The object must respond to all messages defined by that CORBA object's interface.

An object reference can have a value which indicates that it represents no CORBA object. This value is the standard Smalltalk value **`nil`**.

20.6 Invocation of Operations

OMG IDL and Smalltalk message syntaxes both allow zero or more input parameters to be supplied in a request. For return values, Smalltalk methods yield a single result object, whereas OMG IDL allows an optional result and zero or more out or inout parameters to be returned from an invocation. In this binding, the non-void result of an operation is returned as the result of the corresponding Smalltalk method, whereas out and inout parameters are to be communicated back to the caller via instances of a class conforming to the ***`CORBAParameter`*** protocol, passed as explicit parameters.

For example, the following operations in OMG IDL:

```
boolean definesProperty(in string key);
    void defines_property(
    in string key,
    out boolean is_defined);
```

are used as follows in the Smalltalk language:

```
aBool := self definesProperty: aString.

        self
        definesProperty: aString
        isDefined: (aBool := nil asCORBAParameter).
```

As another example, these OMG IDL operations:

```
boolean has_property_protection(in string key,
    out Protection pval);

ORBStatus create_request (in Context ctx,
    in Identifier operation,
    in NVList arg_list,
    inout DynamicInvocation::NamedValue result,
    out Request request,
    in Flags req_flags);
```

would be invoked in the Smalltalk language as:

```
aBool := self
    hasPropertyProtection: aString
    pval: (protection := nil asCORBAParameter).

        aStatus := ORBObject
        createRequest: aContext
        operation: anIdentifier
        argList: anNVList
        result: (result := aNamedValue asCORBAParameter)
        request: (request := nil asCORBAParameter)
        reqFlags: aFlags.
```

The return value of OMG IDL operations that are specified with a **void** return type is undefined.

20.7 Mapping for Attributes

OMG IDL attribute declarations are a shorthand mechanism to define pairs of simple accessing operations; one to get the value of the attribute and one to set it. Such accessing methods are common in Smalltalk programs as well, thus attribute declarations are mapped to standard methods to get and set the named attribute value, respectively.

For example:

```
attribute string title;
readonly attribute string my_name;
```

means that Smalltalk programmers can expect to use **`title`** and **`title:`** methods to get and set the **title** attribute of the CORBA object, and the **`myName`** method to retrieve the **my_name** attribute.

20.7.1 Mapping for Constants

OMG IDL allows constant expressions to be declared globally as well as in interface and module definitions. OMG IDL constant values are stored in a dictionary named **`CORBAConstants`** under the fully qualified name of the constant, not subject to the name conversion algorithm. The constants are accessed by sending the **`at:`** message to the dictionary with an instance of a **`String`** whose value is the fully qualified name.

For example, given the following OMG IDL specification,

```
module ApplicationBasics{
    const CopyDepth shallow_cpy = 4;
    };
```

the **ApplicationBasics::shallow_cpy** constant can be accessed with the following Smalltalk code

```
value := CORBAConstants at:
    '::ApplicationBasics::shallow_cpy'.
```

After this call, the **`value`** variable will contain the integral value 4.

20.8 Mapping for Basic Data Types

The following basic datatypes are mapped into existing Smalltalk classes. In the case of **short**, **unsigned short**, **long**, **unsigned long**, **float**, **double**, and **octet**, the actual class used is left up to the implementation, for the following reasons:

- There is no standard for Smalltalk that specifies integral and floating point classes and the valid ranges of their instances.
- The classes themselves are rarely used in Smalltalk. Instances of the classes are made available as constants included in code, or as the result of computation.

The basic datatypes are mapped as follows:

short

An OMG IDL **short** integer falls in the range [-2^{15},$2^{15}-1$]. In Smalltalk, a short is represented as an instance of an appropriate integral class.

long

An OMG IDL **long** integer falls in the range [-2^{31},2^{31}-1]. In Smalltalk, a long is represented as an instance of an appropriate integral class.

unsigned short

An OMG IDL **unsigned short** integer falls in the range [0,2^{16}-1]. In Smalltalk, an unsigned short is represented as an instance of an appropriate integral class.

unsigned long

An OMG IDL **unsigned long** integer falls in the range [0,2^{32}-1]. In Smalltalk, an unsigned long is represented as an instance of an appropriate integral class.

float

An OMG IDL **float** conforms to the IEEE single-precision (32-bit) floating point standard (ANSI/IEEE Std 754-1985). In Smalltalk, a float is represented as an instance of an appropriate floating point class.

double

An OMG IDL **double** conforms to the IEEE double-precision (64-bit) floating point standard (ANSI/IEEE Std 754-1985). In Smalltalk, a double is represented as an instance of an appropriate floating point class.

char

An OMG IDL **character** holds an 8-bit quantity mapping to the ISO Latin-1 (8859.1) character set. In Smalltalk, a character is represented as an instance of **`Character`**.

boolean

An OMG IDL **boolean** may hold one of two values: TRUE or FALSE. In Smalltalk, a boolean is represented by the values **`true`** or **`false`**, respectively.

octet

An OMG IDL **octet** is an 8-bit quantity that undergoes no conversion during transmission. In Smalltalk, an octet is represented as an instance of an appropriate integral class with a value in the range [0,255].

20.9 Mapping for the Any Type

Due to the dynamic nature of Smalltalk, where the class of objects can be determined at runtime, an explicit mapping of the **any** type to a particular Smalltalk class is not required. Instead, wherever an **any** is required, the user may pass any Smalltalk object which can be mapped into an OMG IDL type. For instance, if an OMG IDL structure type is defined in an interface, a **Dictionary** for that structure type will be mapped. Instances of this class can be used wherever an **any** is expected, since that Smalltalk object can be mapped to the OMG IDL structure.

Likewise, when an **any** is returned as the result of an operation, the actual Smalltalk object which represents the value of the any data structure will be returned.

20.10 Mapping for Enums

OMG IDL enumerators are stored in a dictionary named **CORBAConstants** under the fully qualified name of the enumerator, not subject to the name conversion algorithm. The enumerators are accessed by sending the **at:** message to the dictionary with an instance of a **String** whose value is the fully qualified name.

These enumerator Smalltalk objects must support the ***CORBAEnum*** protocol, to allow enumerators of the same type to be compared. The order in which the enumerators are named in the specification of an enumeration defines the relative order of the enumerators. The protocol must support the following instance methods:

< aCORBAEnum
Answers **true** if the receiver is less than **aCORBAEnum**, otherwise answers **false**.

<= aCORBAEnum
Answers **true** if the receiver is less than or equal to **aCORBAEnum**, otherwise answers **false**.

= aCORBAEnum
Answers **true** if the receiver is equal to **aCORBAEnum**, otherwise answers **false**.

> aCORBAEnum
Answers **true** if the receiver is greater than **aCORBAEnum**, otherwise answers **false**.

>= aCORBAEnum
Answers **true** if the receiver is greater than or equal to **aCORBAEnum**, otherwise answers **false**.

For example, given the following OMG IDL specification,

```
module Graphics{
    enum ChartStyle
        {lineChart, barChart, stackedBarChart, pieChart};
    };
```

the **Graphics::lineChart** enumeration value can be accessed with the following Smalltalk code

```
value := CORBAConstants at: '::Graphics::lineChart'.
```

After this call, the **value** variable is assigned to a Smalltalk object that can be compared with other enumeration values.

20.11 *Mapping for Struct Types*

An OMG IDL struct is mapped to an instance of the **Dictionary** class. The key for each OMG IDL struct member is an instance of **Symbol** whose value is the name of the element converted according to the algorithm in Section 20.2. For example, a structure with a field of **my_field** would be accessed by sending the **at:** message with the key **#myField**.

For example, given the following OMG IDL declaration:

```
struct  Binding {
    Name binding_name;
    BindingType binding_type;
    };
```

the binding_name element can be accessed as follows:

```
    aBindingStruct at: #bindingName
```

and set as follows:

```
    aBindingStruct at: #bindingName put: aName
```

20.12 *Mapping for Union Types*

For OMG IDL union types, two binding mechanisms are provided: an *implicit* binding and an *explicit* binding.[1] The implicit binding takes maximum advantage of the dynamic nature of Smalltalk and is the least intrusive binding for the Smalltalk programmer. The explicit binding retains the value of the discriminator and provides greater control for the programmer.

Although the particular mechanism for choosing implicit vs. explicit binding semantics is implementation specific, all implementations must provide both mechanisms.

1.Although not required, implementations may choose to provide both implicit and explicit mappings for other OMG IDL types, such as structs and sequences. In the explicit mapping, the OMG IDL type is mapped to a user specified Smalltalk class.

Binding semantics is expected to be specifiable on a per-union declaration basis, for example using the **#pragma** directive.

20.12.1 Implicit Binding

Wherever a **union** is required, the user may pass any Smalltalk object that can be mapped to an OMG IDL type, and whose type matches one of the types of the values in the union. Consider the following example:

```
structure S { long x; long y; };

union U switch (short) {
    case 1: S s;
    case 2: long l;
    default: char c;
    };
```

In the example above, a **Dictionary** for structure S will be mapped. Instances of **Dictionary** with runtime elements as defined in structure **S**, integral numbers, or characters can be used wherever a union of type **U** is expected. In this example, instances of these classes can be mapped into one of the **S, long,** or **char** types, and an appropriate discriminator value can be determined at runtime.

Likewise, when an **union** is returned as the result of an operation, the actual Smalltalk object which represents the value of the **union** will be returned.

20.12.2 Explicit Binding

Use of the explicit binding will result in specific Smalltalk classes being accepted and returned by the ORB. Each union object must conform to the ***CORBAUnion*** protocol. This protocol must support the following instance methods:

discriminator
Answers the discriminator associated with the instance.

discriminator: anObject
Sets the discriminator associated with the instance.

value
Answers the value associated with the instance.

value: anObject
Sets the value associated with the instance

To create an object that supports the ***CORBAUnion*** protocol, the instance method **asCORBAUnion: aDiscriminator** can be invoked by any Smalltalk object. This method will return a Smalltalk object conforming to the ***CORBAUnion*** protocol, whose discriminator will be set to **aDiscriminator** and whose value will be set to the receiver of the message.

20.13 Mapping for Sequence Types

Instances of the **OrderedCollection** class are used to represent OMG IDL elements with the **sequence** type.

20.14 Mapping for String Types

Instances of the Smalltalk **String** class are used to represent OMG IDL elements with the **string** type.

20.15 Mapping for Array Types

Instances of the Smalltalk **Array** class are used to represent OMG IDL elements with the **array** type.

20.16 Mapping for Exception Types

Each defined exception type is mapped to an instance of the **Dictionary** class. See Section 6.20.1 for a complete description.

20.17 Mapping for Operations

OMG IDL operations having zero parameters map directly to Smalltalk unary messages, while OMG IDL operations having one or more parameters correspond to Smalltalk keyword messages. To determine the default selector for such an operation, begin with the OMG IDL operation identifier and concatenate the parameter name of each parameter followed by a colon, ignoring the first parameter. The mapped selector is subject to the identifier conversion algorithm.
For example, the following OMG IDL operations:

```
void add_to_copy_map(
    in CORBA::ORBId id,
    in LinkSet link_set);

void connect_push_supplier(
    in EventComm::PushSupplier push_supplier);

void add_to_delete_map(
    in CORBA::ORBId id,
    in LinkSet link_set);
```

become selectors:

```
addToCopyMap:linkSet:
    connectPushSupplier:
    addToDeleteMap:linkSet:
```

20.18 Implicit Arguments to Operations

Unlike the C mapping, where an object reference, environment, and optional context must be passed as parameters to each operation, this Smalltalk mapping does not require these parameters to be passed to each operation.

The object reference is provided in the client code as the receiver of a message. So although it is not a parameter on the operation, it is a required part of the operation invocation.

This mapping defines the ***CORBAExceptionEvent*** protocol to convey exception information in place of the environment used in the C mapping. This protocol can either be mapped into native Smalltalk exceptions or used in cases where native Smalltalk exception handling is unavailable.

A context expression can be associated with the current Smalltalk process by sending the message **corbaContext:** to the current process, along with a valid context parameter. The current context can be retrieved by sending the **corbaContext** message to the current process.

The current process may be obtained by sending the message **activeProcess** to the Smalltalk global variable named **Processor**.

20.19 Argument Passing Considerations

All parameters passed into and returned from the Smalltalk methods used to invoke operations are allocated in memory maintained by the Smalltalk virtual machine. Thus, explicit **free()**ing of the memory is not required. The memory will be garbage collected when it is no longer referenced.

The only exception is object references. Since object references may contain pointers to memory allocated by the operating system, it is necessary for the user to explicitly free them when no longer needed. This is accomplished by using the operation **release** of the **CORBA::Object** interface.

20.20 Handling Exceptions

OMG IDL allows each operation definition to include information about the kinds of run-time errors which may be encountered. These are specified in an exception definition which declares an optional error structure which will be returned by the operation should an error be detected. Since Smalltalk exception handling classes are not yet standardized between existing implementations, a generalized mapping is provided.

In this binding, an IDL compiler creates exception objects and populates the **CORBAConstants** dictionary. These exception objects are accessed from the **CORBAConstants** dictionary by sending the **at:** message with an instance of a **String** whose value is the fully qualified name. Each exception object must conform to the ***CORBAExceptionEvent*** protocol. This protocol must support the following instance methods:

`corbaHandle: aHandlerBlock do: aBlock`

Exceptions may be handled by sending an exception object the message **`corbaHandle:do:`** with appropriate handler and scoping blocks as parameters. The **`aBlock`** parameter is the Smalltalk block to evaluate. It is passed no parameters. The **`aHandlerBlock`** parameter is a block to evaluate when an exception occurs. It has one parameter: a Smalltalk object which conforms to the ***`CORBAExceptionValue`*** protocol.

`corbaRaise`

Exceptions may be raised by sending an exception object the message **`corbaRaise`**.

`corbaRaiseWith: aDictionary`

Exceptions may be raised by sending an exception object the message **`corbaRaiseWith`**:. The parameter is expected to be an instance of the Smalltalk **`Dictionary`** class, as described below.

For example, given the following OMG IDL specification,

```
interface NamingContext {
    ...

    exception NotEmpty {};
    void destroy ()
        raises (NotEmpty);
    ...
};
```

the **NamingContext::NotEmpty** exception can be raised as follows:

```
(CORBAConstants at: '::NamingContext::NotEmpty')
    corbaRaise.
```

The exception can be handled in Smalltalk as follows:

```
(CORBAConstants at: '::NamingContext::NotEmpty')
    corbaHandle: [:ev | "error handling logic here" ]
    do: [aNamingContext destroy].
```

20.20.1 *Exception Values*

OMG IDL allows values to be returned as part of the exception. Exception values are constructed using instances of the Smalltalk **`Dictionary`** class. The keys of the dictionary are the names of the elements of the exception, the names of which are converted using the algorithm in Section 20.2, “Conversion of Names to Smalltalk Identifiers,” on page 20-2. The following example illustrates how exception values are used:

```
interface NamingContext {

  ...

  exception CannotProceed {
            NamingContext cxt;
            Name rest_of_name;
      };

      Object resolve (in Name n)
                  raises (CannotProceed);

  ...

};
```

would be raised in Smalltalk as follows:

```
(CORBAConstants at: '::NamingContext::CannotProceed')
   corbaRaiseWith: (Dictionary
      with: (Association key: #cxt value:
               aNamingContext)
      with: (Association key: #restOfName value:
               aName)).
```

20.20.2 The CORBAExceptionValue Protocol

When an exception is raised, the exception block is evaluated, passing it one argument which conforms to the ***CORBAExceptionValue*** protocol. This protocol must support the following instance messages:

`corbaExceptionValue`

Answers the **`Dictionary`** the exception was raised with.

Given the **NamingContext** interface defined in the previous section, the following code illustrates how exceptions are handled:

```
(CORBAConstants at: '::NamingContext::NotEmpty')
   corbaHandle:[:ev |
      cxt:=ev corbaExceptionValue at: #cxt.
      restOfName :=ev corbaExceptionValue at:
                  #restOfName]
   do:[aNamingContext destroy].
```

In this example, the **`cxt`** and **`restOfName`** variables will be set to the respective values from the exception structure, if the exception is raised.

Mapping of Pseudo-Objects to Smalltalk 21

CORBA defines a small set of standard interfaces which define types and operations for manipulating object references, for accessing the Interface Repository, and for Dynamic Invocation of operations. Other interfaces are defined in pseudo OMG IDL (PIDL) to represent in a more abstract manner programmer access to ORB services which are provided locally. These PIDL interfaces sometimes resort to non-OMG IDL constructs, such as pointers, which have no meaning to the Smalltalk programmer. This chapter specifies the minimal requirements for the Smalltalk mapping for PIDL interfaces. The operations are specified below as protocol descriptions.

Parameters with the name **`aCORBAObject`** are expected to be Smalltalk objects, which can be mapped to an OMG IDL interface or data type.

Unless otherwise specified, all messages are defined to return undefined objects.

21.1 CORBA::Request

The CORBA::Request interface is mapped to the ***`CORBARequest`*** protocol, which must include the following instance methods:

`addArg: aCORBANamedValue`
Corresponds to the **add_arg** operation.

`invoke`
Corresponds to the **invoke** operation with the **invoke_flags** set to 0.

`invokeOneway`
Corresponds to the **invoke** operation with the **invoke_flags** set to **`CORBA::INV_NO_RESPONSE`**.

`send`
Corresponds to the **send** operation with the **invoke_flags** set to 0.

`sendOneway`
Corresponds to the **`send`** operation with the **invoke_flags** set to **`CORBA::INV_NO_RESPONSE`**.

`pollResponse`
Corresponds to the **get_response** operation, with the **response_flags** set to **`CORBA::RESP_NO_WAIT`**. Answers **`true`** if the response is complete, **`false`** otherwise.

`getResponse`
Corresponds to the **get_response** operation, with the **response_flags** set to 0.

21.2 CORBA::Context

The CORBA::Context interface is mapped to the ***`CORBAContext`*** protocol, which must include the following instance methods:

`setOneValue: anAssociation`
Corresponds to the **set_one_value** operation.

`setValues: aCollection`
Corresponds to the **set_values** operation. The parameter passed in should be a collection of **`Association`**s.

`getValues: aString`
Corresponds to the **get_values** operation without a scope name and op_flags = **`CXT_RESTRICT_SCOPE.`** Answers a collection of **`Association`**s.

`getValues: aString propName: aString`
Corresponds to the **`get_values`** operation with op_flags set to **`CXT_RESTRICT_SCOPE`**. Answers a collection of **`Association`**s.

`getValuesInTree: aString propName: aString`
Corresponds to the **get_values** operation with op_flags set to **`0`**. Answers a collection of **`Association`**s.

`deleteValues: aString`
Corresponds to the **delete_values** operation.

`createChild: aString`
Corresponds to the **create_child** operation. Answers a Smalltalk object conforming to the ***`CORBAContext`*** protocol.

`delete`
Corresponds to the **delete** operation with flags set to **`0`**.

`deleteTree`
Corresponds to the **delete** operation with flags set to **`CTX_DELETE_DESCENDENTS`**.

21.3 CORBA::Object

The CORBA::Object interface is mapped to the ***CORBAObject*** protocol, which must include the following instance methods:

getImplementation
Corresponds to the **get_implementation** operation. Answers a Smalltalk object conforming to the ***CORBAImplementationDef*** protocol.

getInterface
Corresponds to the **get_interface** operation. Answers a Smalltalk object conforming to the ***CORBAInterfaceDef*** protocol.

isNil
Corresponds to the **is_nil** operation. Answers **true** or **false** indicating whether or not the object reference represents an object.

```
createRequest: aCORBAContext
    operation: aCORBAIdentifier
    argList: aCORBANVListOrNil
    result: aCORBAParameter
    request: aCORBAParameter
    reqFlags: flags
```

Corresponds to the **create_request** operation.

duplicate
Corresponds to the **duplicate** operation. Answers a Smalltalk object representing an object reference, conforming to the interface of the CORBA object.

release[1]
Corresponds to the **release** operation.

21.4 CORBA::ORB

The CORBA::ORB interface is mapped to the ***CORBAORB*** protocol, which must include the following instance methods:

objectToString: aCORBAObject
Corresponds to the **object_to_string** operation. Answers an instance of the **String** class.

1.The semantics of this operation will have no meaning for those implementations that rely exclusively on the Smalltalk memory manager.

`stringToObject: aString`
Corresponds to the **string_to_object** operation. Answers an object reference, which will be an instance of a class which corresponds to the **InterfaceDef** of the CORBA object.

`createOperationList: aCORBAOperationDef`
Corresponds to the **create_operation_list** operation. Answers an instance of **`OrderedCollection`** of Smalltalk objects conforming to the ***`CORBANamedValue`*** protocol.

`getDefaultContext`
Corresponds to the **get_default_context** operation. Answers a Smalltalk object conforming to the ***`CORBAContext`*** protocol.

`sendMultipleRequests: aCollection`
Corresponds to the **send_multiple_requests** operation with the **invoke_flags** set to **`0`**.The parameter passed in should be a collection of Smalltalk objects conforming to the ***`CORBARequest`*** protocol.

`sendMultipleRequestsOneway: aCollection`
Corresponds to the **send_multiple_requests** operation with the **invoke_flags** set to **`CORBA::INV_NO_RESPONSE`.** The parameter passed in should be a collection of Smalltalk objects conforming to the ***`CORBARequest`*** protocol.

`pollNextResponse`
Corresponds to the **get_next_response** operation, with the **response_flags** set to **`CORBA::RESP_NO_WAIT`**. Answers **`true`** if there are completed requests pending, **`false`** otherwise.

`getNextResponse`
Corresponds to the **get_next_response** operation, with the **response_flags** set to `0`.

21.5 CORBA::NamedValue

PIDL for C defines **CORBA::NamedValue** as a struct while C++-PIDL specifies it as an interface. **CORBA::NamedValue** in this mapping is specified as an interface that conforms to the ***`CORBANamedValue`*** protocol. This protocol must include the following instance methods:

`name`
Answers the name associated with the instance.

`name: aString`
Resets the name associated with instance to **`aString`**.

`value`
Answers the value associated with the instance.

`value: aCORBAObject`
Resets the value associated with instance to **`aCORBAObject`**.

`flags`
Answers the flags associated with the instance.

`flags: argModeFlags`
Resets the flags associated with instance to **`argModeFlags`**.

To create an object that supports the ***`CORBANamedValue`*** protocol, the instance method **`asCORBANamedValue: aName flags: argModeFlags`** can be invoked by any Smalltalk object. This method will return a Smalltalk object conforming to the ***`CORBANamedValue`*** protocol, whose attributes associated with the instance will be set appropriately.

21.6 CORBA::NVList

The **CORBA::NVList** interface is mapped to the equivalent of the OMG IDL definition

typedef sequence<NamedValue> NVList;

Thus, Smalltalk objects representing the **NVList** type should be instances of the **`OrderedCollection`** class, whose elements are Smalltalk objects conforming to the ***`CORBANamedValue`*** protocol.

Glossary

activation	Preparing an object to execute an operation. For example, copying the persistent form of methods and stored data into an executable address space to allow execution of the methods on the stored data.
adapter	Same as object adapter.
attribute	An identifiable association between an object and a value. An attribute **A** is made visible to clients as a pair of operations: **get_A** and **set_A**. Readonly attributes only generate a **get** operation.
basic object adapter	The object adapter described in Chapter 8.
behavior	The observable effects of an object performing the requested operation including its results binding. See language binding, dynamic invocation, static invocation, or method resolution for alternatives.
class	See interface and implementation for alternatives.
client	The code or process that invokes an operation on an object.
context object	A collection of name-value pairs that provides environmental or user-preference information. See Chapter 4.
CORBA	Common Object Request Broker Architecture.
data type	A categorization of values operation arguments, typically covering both behavior and representation (i.e., the traditional non-OO programming language notion of type).
deactivation	The opposite of activation.
deferred synchronous request	A request where the client does not wait for completion of the request, but does intend to accept results later. Contrast with synchronous request and one-way request.

domain	A concept important to interoperability, it is a distinct scope, within which common characteristics are exhibited, common rules observed, and over which a distribution transparency is preserved.
dynamic invocation	Constructing and issuing a request whose signature is possibly not known until run-time.
dynamic skeleton	An interface-independent kind of skeleton, used by servers to handle requests whose signatures are possibly not known until run-time.
externalized object reference	An object reference expressed as an ORB-specific string. Suitable for storage in files or other external media.
implementation	A definition that provides the information needed to create an object and allow the object to participate in providing an appropriate set of services. An implementation typically includes a description of the data structure used to represent the core state associated with an object, as well as definitions of the methods that access that data structure. It will also typically include information about the intended interface of the object.
implementation definition language	A notation for describing implementations. The implementation definition language is currently beyond the scope of the ORB standard. It may contain vendor-specific and adapter-specific notations.
implementation inheritance	The construction of an implementation by incremental modification of other implementations. The ORB does not provide implementation inheritance. Implementation inheritance may be provided by higher level tools.
implementation object	An object that serves as an implementation definition. Implementation objects reside in an implementation repository.
implementation repository	A storage place for object implementation information.
inheritance	The construction of a definition by incremental modification of other definitions. See *interface* and *implementation inheritance*.
instance	An object is an instance of an interface if it provides the operations, signatures and semantics specified by that interface. An object is an instance of an implementation if its behavior is provided by that implementation.
interface	A listing of the operations and attributes that an object provides. This includes the signatures of the operations, and the types of the attributes. An interface definition ideally includes the semantics as well. An object *satisfies* an interface if it can be specified as the target object in each potential request described by the interface.
interface inheritance	The construction of an interface by incremental modification of other interfaces. The IDL language provides interface inheritance.
interface object	An object that serves to describe an interface. Interface objects reside in an interface repository.

interface repository	A storage place for interface information.
interface type	A type satisfied by any object that satisfies a particular interface.
interoperability	The ability for two or more ORBs to cooperate to deliver requests to the proper object. Interoperating ORBs appear to a client to be a single ORB.
language binding or **mapping**	The means and conventions by which a programmer writing in a specific programming language accesses ORB capabilities.
method	An implementation of an operation. Code that may be executed to perform a requested service. Methods associated with an object may be structured into one or more programs.
method resolution	The selection of the method to perform a requested operation.
multiple inheritance	The construction of a definition by incremental modification of more than one other definition.
object	A combination of state and a set of methods that explicitly embodies an abstraction characterized by the behavior of relevant requests. An object is an instance of an implementation and an interface. An object models a real-world entity, and it is implemented as a computational entity that encapsulates state and operations (internally implemented as data and methods) and responds to request or services.
object adapter	The ORB component which provides object reference, activation, and state related services to an object implementation. There may be different adapters provided for different kinds of implementations.
object creation	An event that causes the existence of an object that is distinct from any other object.
object destruction	An event that causes an object to cease to exist.
object implementation	Same as implementation.
object reference	A value that unambiguously identifies an object. Object references are never reused to identify another object.
objref	An abbreviation for object reference.
one-way request	A request where the client does not wait for completion of the request, nor does it intend to accept results. Contrast with deferred synchronous request and synchronous request.
operation	A service that can be requested. An operation has an associated signature, which may restrict which actual parameters are valid.
operation name	A name used in a request to identify an operation.
ORB	Object Request Broker. Provides the means by which clients make and receive requests and responses.

ORB core	The ORB component which moves a request from a client to the appropriate adapter for the target object.
parameter passing mode	Describes the direction of information flow for an operation parameter. The parameter passing modes are **IN**, **OUT**, and **INOUT**.
persistent object	An object that can survive the process or thread that created it. A persistent object exists until it is explicitly deleted.
referential integrity	The property ensuring that an object reference that exists in the state associated with an object reliably identifies a single object.
repository	See interface repository and implementation repository.
request	A client issues a request to cause a service to be performed. A request consists of an operation and zero or more actual parameters.
results	The information returned to the client, which may include values as well as status information indicating that exceptional conditions were raised in attempting to perform the requested service.
server	A process implementing one or more operations on one or more objects.
server object	An object providing response to a request for a service. A given object may be a client for some requests and a server for other requests.
signature	Defines the parameters of a given operation including their number order, data types, and passing mode; the results if any; and the possible outcomes (normal vs. exceptional) that might occur.
single inheritance	The construction of a definition by incremental modification of one definition. Contrast with multiple inheritance.
skeleton	The object-interface-specific ORB component which assists an object adapter in passing requests to particular methods.
state	The time-varying properties of an object that affect that object’s behavior.
static invocation	Constructing a request at compile time. Calling an operation via a stub procedure.
stub	A local procedure corresponding to a single operation that invokes that operation when called.
synchronous request	A request where the client pauses to wait for completion of the request. Contrast with deferred synchronous request and one-way request.
transient object	An object whose existence is limited by the lifetime of the process or thread that created it.
type	See *data type* and *interface*.
value	Any entity that may be a possible actual parameter in a request. Values that serve to identify objects are called object references.