The Common Object Request Broker: Architecture and Specification

Revised Edition: July 1995
Updated: July 1996
Revision 2.1: August 1997
Revision 2.2: February 1998
Revision 2.3: June1999
Minor revision 2.3.1: October 1999

PATENT

The attention of adopters is directed to the possibility that compliance with or adoption of OMG specifications may require use of an invention covered by patent rights. OMG shall not be responsible for identifying patents for which a license may be required by any OMG specification, or for conducting legal inquiries into the legal validity or scope of those patents that are brought to its attention. OMG specifications are prospective and advisory only. Prospective users are responsible for protecting themselves against liability for infringement of patents.

NOTICE

The information contained in this document is subject to change without notice. The material in this document details an Object Management Group specification in accordance with the license and notices set forth on this page. This document does not represent a commitment to implement any portion of this specification in any company's products.

WHILE THE INFORMATION IN THIS PUBLICATION IS BELIEVED TO BE ACCURATE, THE OBJECT MANAGEMENT GROUP AND THE COMPANIES LISTED ABOVE MAKE NO WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, WITH REGARD TO THIS MATERIAL INCLUDING, BUT NOT LIMITED TO ANY WARRANTY OF TITLE OR OWNERSHIP, IMPLIED WARRANTY OF MERCHANTABILITY OR WARRANTY OF

ISSUE REPORTING

All OMG specifications are subject to continuous review and improvement. As part of this process we encourage readers to report any ambiguities, inconsistencies, or inaccuracies they may find by completing the issue reporting form at *http://www.omg.org/library/issuerpt.htm.*

Contents

Contents

Contents

Contents

Contents

Contents

Contents

Contents

Contents

Contents

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 800 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.

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 Definition of CORBA Compliance

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 the *C++ Language Mapping Specification*.

Interoperability and Interworking are separate compliance points. For detailed information about Interworking compliance, refer to "Compliance to COM/CORBA Interworking" on page 17-34.

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* core specifications are categorized as follows:

CORBA Core, as specified in Chapters 1-11

CORBA Interoperability, as specified in Chapters 12-16

CORBA Interworking, as specified in Chapters 17-21

Note – The CORBA Language Mappings have been separated from the CORBA Core and each language mapping is its own separate book. Refer to the CORBA Language Mapping area on the OMG website for this information.

0.6 Structure of This Manual

This manual is divided into the categories of Core, Interoperability, Interworking, and individual Language Mappings (located in a separate binder). These divisions reflect the compliance points of CORBA. In addition to this preface, *CORBA: Common Object Request Broker Architecture and Specification* contains the following chapters:

Core

Chapter 1 -- The Object Model describes the computation model that underlies the CORBA architecture.

Chapter 2 -- CORBA Overview contains the overall structure of the ORB architecture and includes information about CORBA interfaces and implementations.

Chapter 3 -- OMG IDL Syntax and Semantics details the OMG interface definition language (OMG IDL), which is the language used to describe the interfaces that client objects call and object implementations provide.

Chapter 4-- ORB Interface defines the interface to the ORB functions that do not depend on object adapters: these operations are the same for all ORBs and object implementations.

Chapter 5-- Value Type Semantics describes the semantics of passing an object by value, which is similar to that of standard programming languages.

Chapter 6-- Abstract Interface Semantics explains an IDL abstract interface, which provides the capability to defer the determination of whether an object is passed by reference or by value until runtime.

Chapter 7-- The Dynamic Invocation Interface details the DII, the client's side of the interface that allows dynamic creation and invocation of request to objects.

Chapter 8 -- 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).

Chapter 9-- Dynamic Management of Any Values details the interface for the Dynamic Any type. This interface allows statically-typed programming languages such as C and Java to create or receive values of type Any without compile-time knowledge that the typer contained in the Any.

Chapter 10-- Interface Repository explains the component of the ORB that manages and provides access to a collection of object definitions.

Chapter 11-- Portable Object Adapter defines a group of IDL interfaces than an implementation uses to access ORB functions.

Interoperability

Chapter 12-- Interoperability Overview describes 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).

Chapter 13 -- ORB 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.

Chapter 14 -- Building Inter-ORB Bridges explains how to build bridges for an implementation of interoperating ORBs.

Chapter 15 -- General Inter-ORB Protocol 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).

Chapter 16 -- DCE ESIOP - Environment-Specific Inter-ORB Protocol (ESIOP) details a protocol for the OSF DCE environment. The protocol is called the DCE Environment Inter-ORB Protocol (DCE ESIOP).

Interworking

Chapter 17 -- Interworking Architecture describes the architecture for communication between two object management systems: Microsoft's COM (including OLE) and the OMG's CORBA.

Chapter 18 -- Mapping: COM and CORBA explains the data type and interface mapping between COM and CORBA. The mappings are described in the context of both Win16 and Win32 COM.

Chapter 19 -- Mapping: OLE Automation and CORBA details the two-way mapping between OLE Automation (in ODL) and CORBA (in OMG IDL).

Note: Chapter 19 also includes an appendix describing solutions that vendors might implement to support existing and older OLE Automation controllers and an appendix that provides an example of how the Naming Service could be mapped to an OLE Automation interface according to the Interworking specification.

Chapter 20-- Interoperability with non-CORBA Systems describes the effective access to CORBA servers through DCOM and the reverse.

Chapter 21-- Interceptors defines ORB operations that allow services such as security to be inserted in the invocation path.

Appendix A-- contains OMG IDL tags that can identify a profile, service, component, or policy.

0.7 Acknowledgements

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

- BEA Systems, Inc.
- BNR Europe Ltd.
- Defense Information Systems Agency
- Expersoft Corporation
- FUJITSU LIMITED
- Genesis Development Corporation
- Gensym Corporation
- IBM Corporation
- ICL plc
- Inprise Corporation
- IONA Technologies Ltd.
- Digital Equipment Corporation
- Hewlett-Packard Company

- HyperDesk Corporation
- Micro Focus Limited
- MITRE Corporation
- NCR Corporation
- Novell USG
- Object Design, Inc.
- Objective Interface Systems, Inc.
- OC Systems, Inc.
- Open Group - Open Software Foundation
- Siemens Nixdorf Informationssysteme AG
- Sun Microsystems Inc.
- SunSoft, Inc.
- Sybase, Inc.
- Telefónica Investigación y Desarrollo S.A. Unipersonal
- 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.

0.8 References

IDL Type Extensions RFP, March 1995. OMG TC Document 95-1-35.

The Common Object Request Broker: Architecture and Specification, Revision 2.2, February 1998.

CORBAservices: Common Object Services Specification, Revised Edition, OMG TC Document 95-3-31.

COBOL Language Mapping RFP, December 1995. OMG TC document 95-12-10.

COBOL 85 ANSI X3.23-1985 / ISO 1989-1985.

IEEE Standard for Binary Floating-Point Arithmetic, ANIS/IEEE Std 754-1985.

XDR: External Data Representation Standard, RFC1832, R. Srinivasan, Sun Microsystems, August 1995.

OSF Character and Code Set Registry, OSF DCE SIG RFC 40.1 (Public Version), S. (Martin) O'Donnell, June 1994.

RPC Runtime Support For I18N Characters — Functional Specification, OSF DCE SIG RFC 41.2, M. Romagna, R. Mackey, November 1994.

X/Open System Interface Definitions, Issue 4 Version 2, 1995.

The Object Model 1

The Object Model chapter has been updated based on CORE changes from ptc/98-09-04 and the Object by Value specification (orbos/98-01-18).

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.)

Contents

This chapter contains the following sections.

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 are the details of control structure: the object model does not say whether clients and/or servers are single-threaded or multi-threaded, and does not specify how event loops are programmed nor how threads are created, destroyed, or synchronized.

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.

The term *request* is broadly used to refer to the entire sequence of causally related events that transpires between a client initiating it and the last event causally associated with that initiation. For example:

- the client receives the final response associated with that *request* from the server,
- the server carries out the associated operation in case of a oneway request, or
- the sequence of events associated with the *request* terminates in a failure of some sort. The initiation of a Request is an event.

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 Dynamic Invocation Interface chapter for descriptions of these request forms).

A *value* is anything that may be a legitimate (actual) parameter in a request. More particularly, a value is an instance of an OMG IDL data type. There are non-object values, as well as values that reference objects.

An *object reference* is a value 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 context is a mapping from strings to strings.

A request causes a service to be performed on behalf of the client. One possible 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 *return result value*, as well as the results stored into the output and input-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.
- The return result and the values stored into the output and input-output parameters are undefined if an exception is returned.

For descriptions of the values and exceptions that are permitted, see Section 1.2.4, “Types,” on page 1-4 and Section 1.2.8.3, “Exceptions,” on page 1-8.

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 entities. An entity *satisfies* a type if the predicate is true for that entity. An entity 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 entities that satisfy the type at any particular time.

An *object type* is a type whose members are object references. In other words, an object type is satisfied only by object references.

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

1.2.4.1 Basic types

- 16-bit, 32-bit, and 64-bit signed and unsigned 2’s complement integers.
- Single-precision (32-bit), double-precision (64-bit), and double-extended (a mantissa of at least 64 bits, a sign bit and an exponent of at least 15 bits) IEEE floating point numbers.
- Fixed-point decimal numbers of up to 31 significant digits.
- Characters, as defined in ISO Latin-1 (8859.1) and other single- or multi-byte character sets.
- 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 a non-negative integer, and is available at run-time. The length may have a maximum bound defined.
- A wide character string type, which consist of a variable-length array of (fixed width) wide characters; the length of the wide string is a non-negative integer, and is available at run-time. The length may have a maximum bound defined.
- A container type “any,” which can represent any possible basic or constructed type.
- Wide characters that may represent characters from any wide character set.
- Wide character strings, which consist of a length, available at runtime, and a variable-length array of (fixed width) wide characters.

1.2.4.2 Constructed types

- A record type (called struct), which consists of an ordered set of (name,value) pairs.
- A discriminated union type, which consists of a discriminator (whose exact value is always available) 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-shape multidimensional array of a single type.
- An interface type, which specifies the set of operations which an instance of that type must support.
- A value type, which specifies state as well as a set of operations which an instance of that type must support.

Entities in a request are restricted to values that satisfy these type constraints. The legal entities are shown in Figure 1-1. No particular representation for entities is defined.

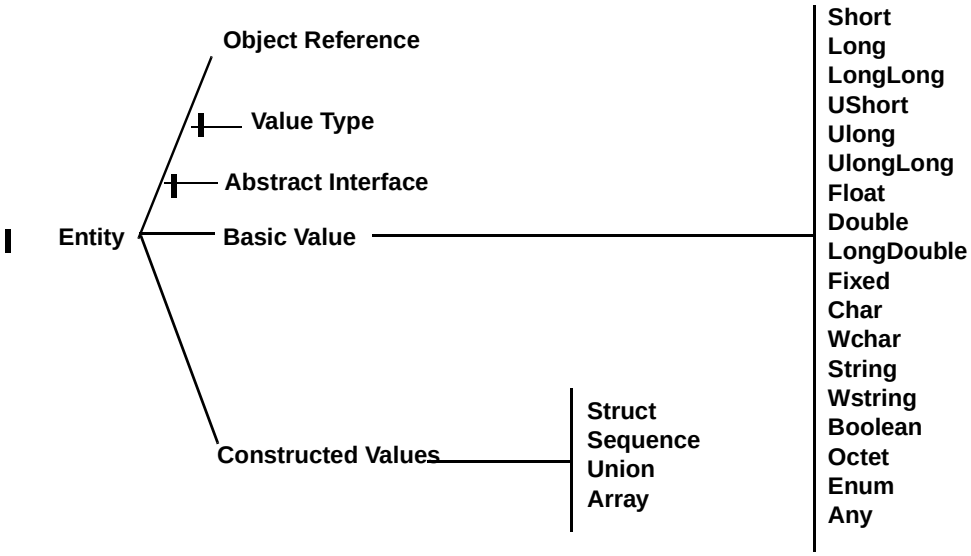


Figure 1-1 Legal Values

1.2.5 Interfaces

An *interface* is a description of a set of possible operations that a client may request of an object, through that interface. It provides a syntactic description of how a service provided by an object supporting this interface, is accessed via this set of operations. An object *satisfies* an interface if it provides its service through the operations of the interface according to the specification of the operations (see Section 1.2.8, “Operations,” on page 1-7).

The *interface type* for a given interface is an object type, such that an object reference will satisfy the type, if and only if the referent object also satisfies the 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 Value Types

A *value type* is an entity which shares many of the characteristics of interfaces and structs. It is a description of both a set of operations that a client may request and of state that is accessible to a client. Instances of a value type are always local concrete implementations in some programming language.

A value type, in addition to the operations and state defined for itself, may also inherit from other value types, and through multiple inheritance support other interfaces.

Value types are specified in OMG IDL.

An *abstract value types* describes a value type that is a “pure” bundle of operations with no state.

1.2.7 Abstract Interfaces

An *abstract interface* is an entity which may at runtime represent either a regular interface (see Section 1.2.5, “Interfaces,” on page 1-6) or a value type (see Section 1.2.6, “Value Types,” on page 1-6). Like an abstract value type, it is a pure bundle of operations with no state. Unlike an abstract value type, it does not imply pass-by-value semantics, and unlike a regular interface type, it does not imply pass-by-reference semantics. Instead, the entity's runtime type determines which of these semantics are used.

1.2.8 Operations

An *operation* is an identifiable entity that denotes the indivisible primitive of service provision that can be requested. The act of requesting an operation is referred to as *invoking the operation*. An operation is identified by an *operation identifier.*

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.
- An identification of the user exceptions that may be raised by an invocation of the operation.
- A specification of additional contextual information that may affect the invocation.
- An indication of the execution semantics the client should expect from an invocation of the operation.

Operations are (potentially) *generic*, meaning that a single operation can be uniformly invoked 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 an invocation of 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.

1.2.8.1 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.

1.2.8.2 Return Result

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

1.2.8.3 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, "Types," on page 1-4.

All signatures implicitly include the system exceptions; the standard system exceptions are described in Section 3.17, "Standard Exceptions," on page 3-51.

1.2.8.4 Contexts

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

1.2.8.5 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.

1.2.9 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 results 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 and input-output parameters and return result 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 types 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.

Note – Changes from the CORBA Core RTF (ptc/98-07-05) have been incorporated into this chapter.

Contents

This chapter contains the following sections.

2.1 Structure of an Object Request Broker

Figure 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.

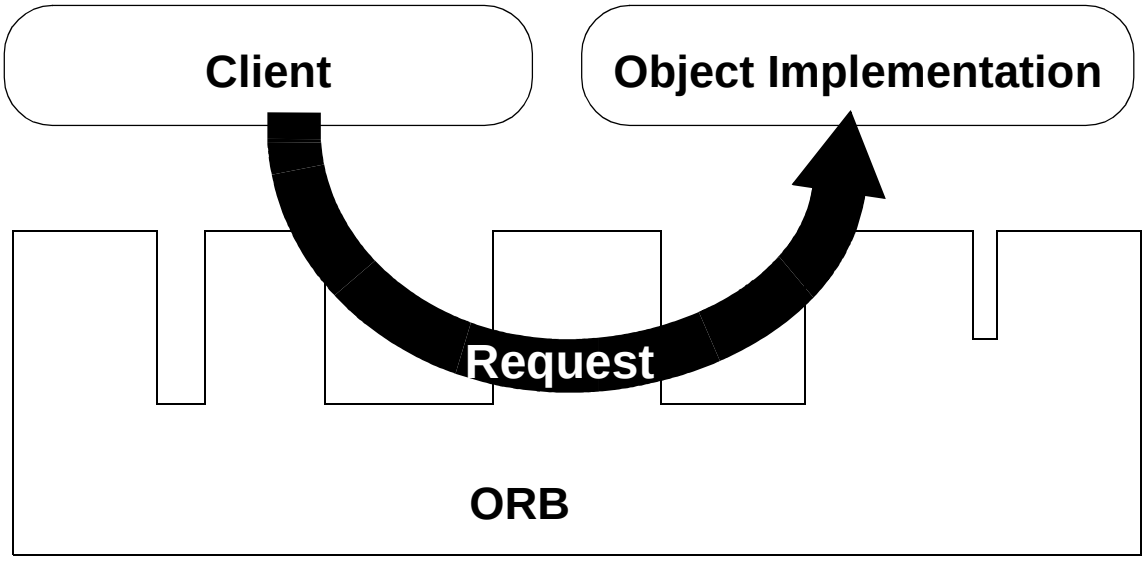


Figure 2-1 A Request Being Sent Through the Object Request Broker

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.

Figure 2-2 on page 2-3 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.

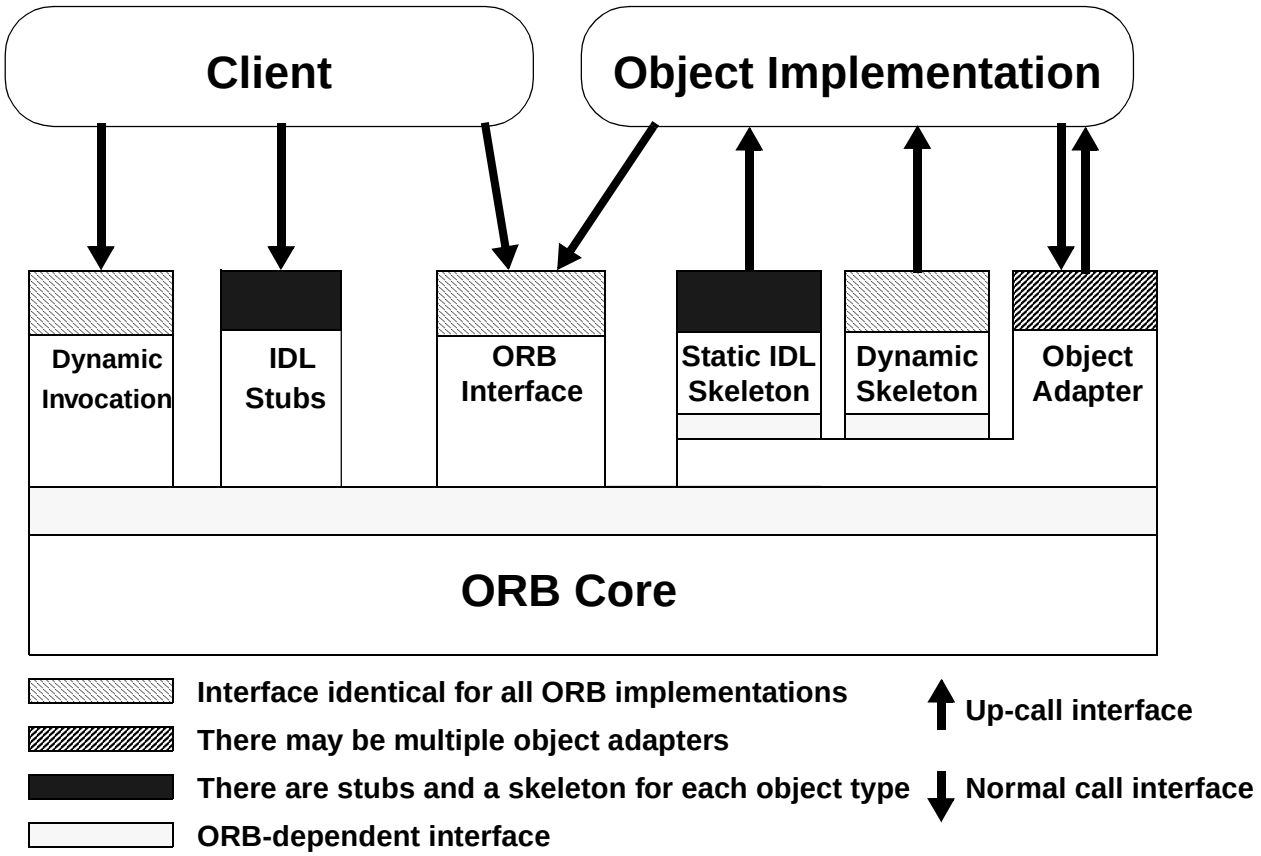


Figure 2-2 The Structure of Object Request Interfaces

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.

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 Figure 2-3).

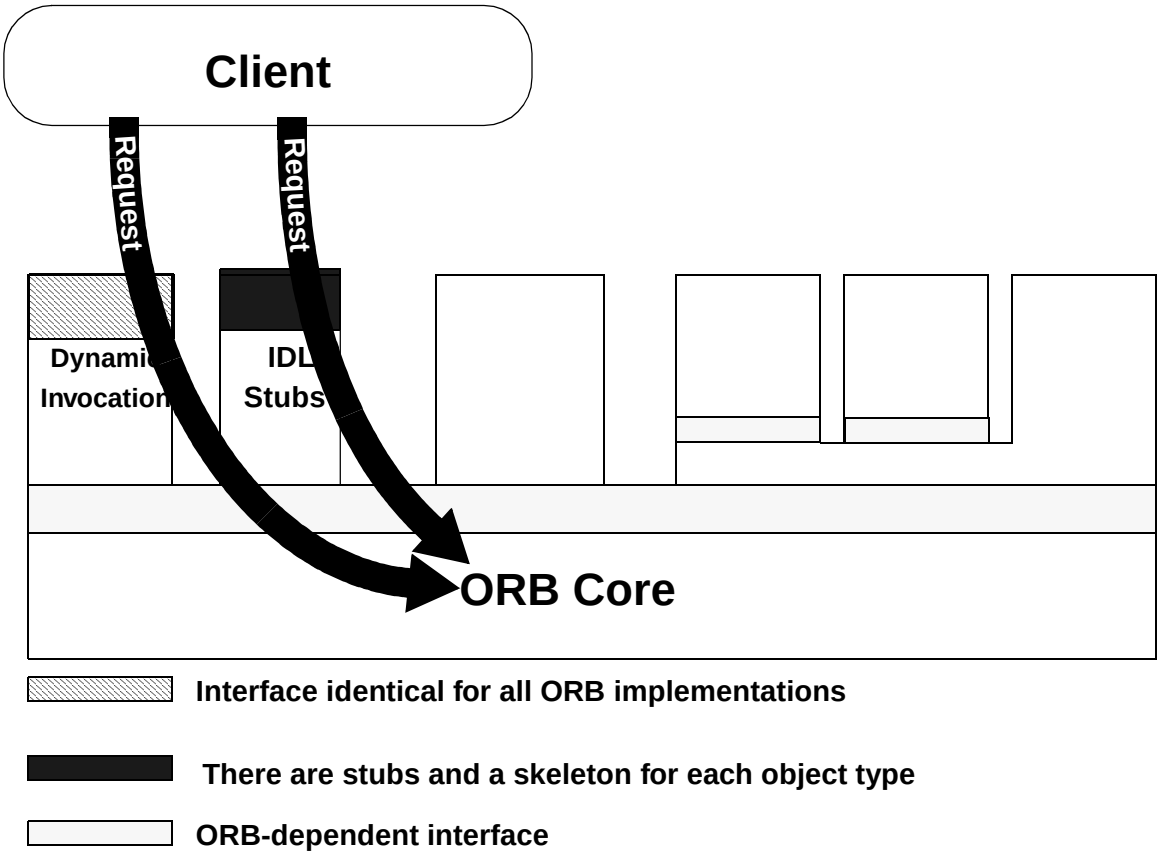


Figure 2-3 A Client Using the Stub or Dynamic Invocation Interface

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.

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 Figure 2-4 on page 2-5). 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.

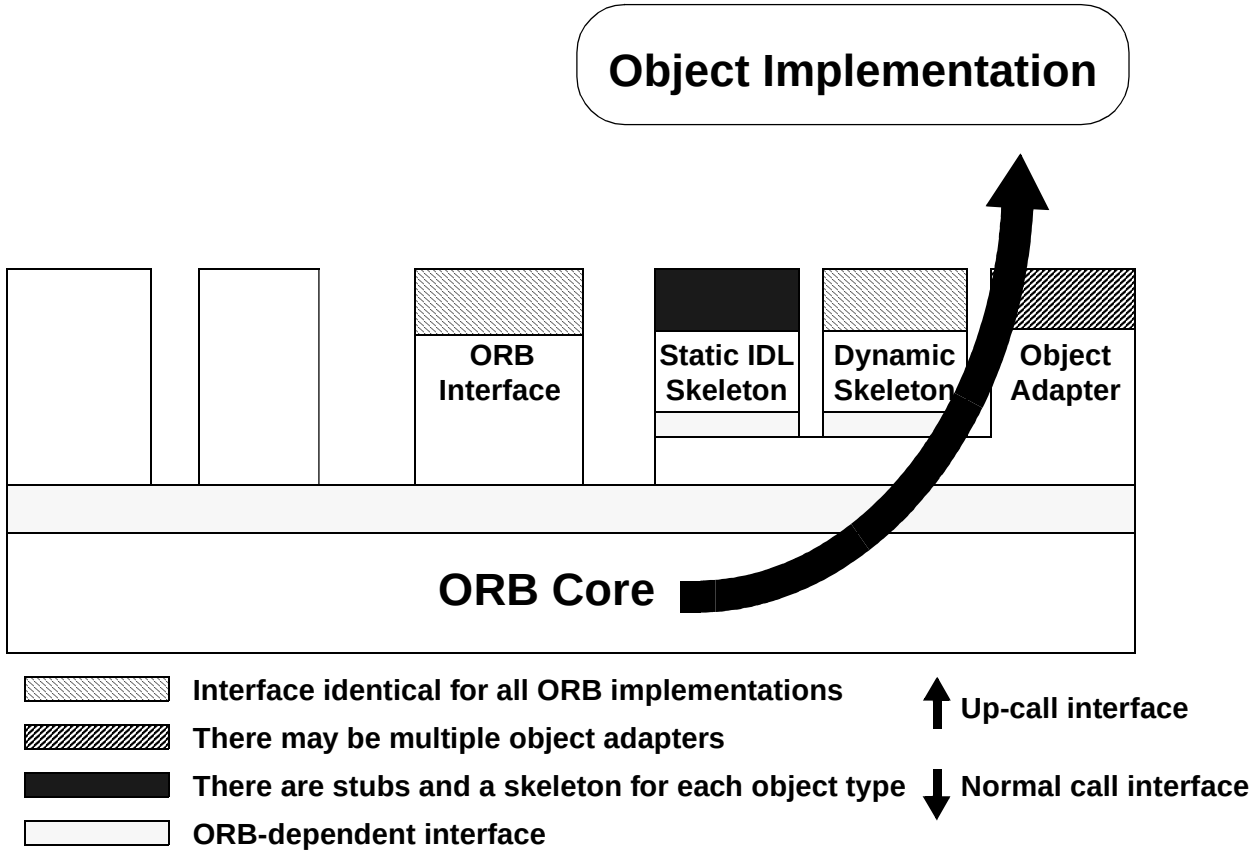


Figure 2-4 An Object Implementation Receiving a Request

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

Figure 2-5 on page 2-6 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.

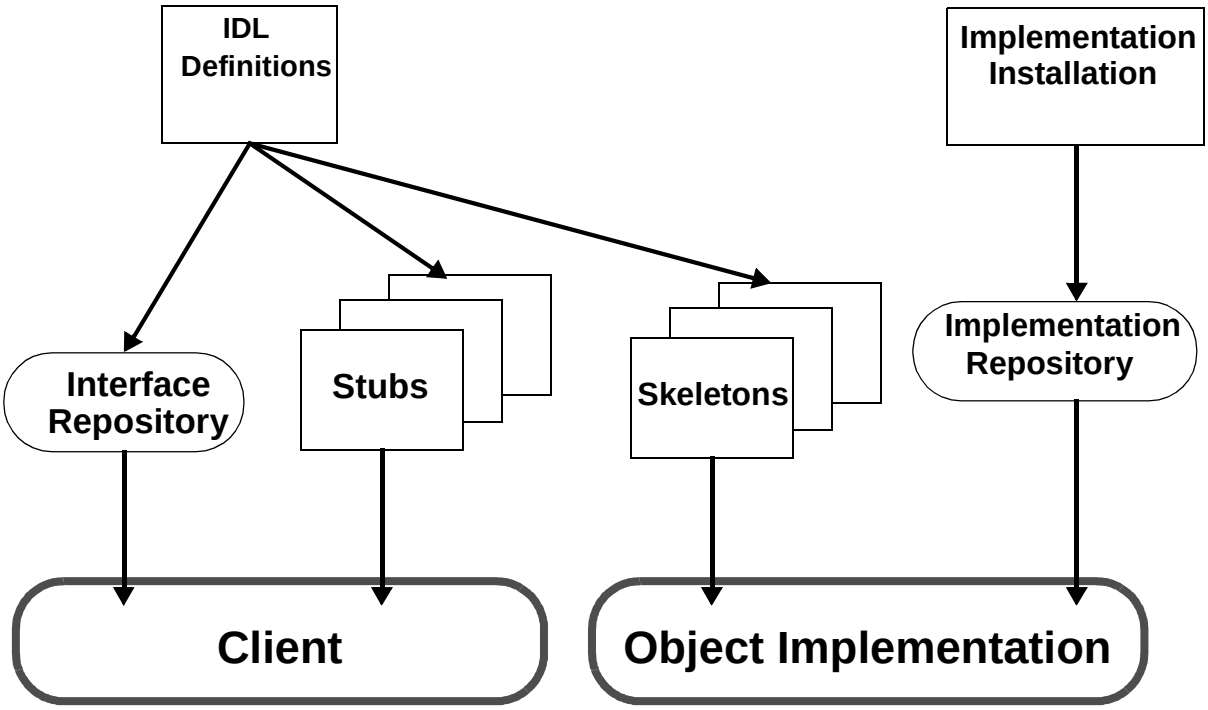


Figure 2-5 Interface and Implementation Repositories

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.

2.1.3 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.

2.1.4 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.

2.1.5 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.

2.1.6 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.

2.1.7 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 Smalltalk, do not require stub interfaces.

2.1.8 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 run-time source). The nature of the dynamic invocation interface may vary substantially from one programming language mapping to another.

2.1.9 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.

2.1.10 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.

2.1.11 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.

2.1.12 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.

2.1.13 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 might be associated with the Interface Repository.

2.1.14 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 Figure 2-6 on page 2-13). 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 the C Language Mapping chapter for additional, general information on language mapping of object references.)

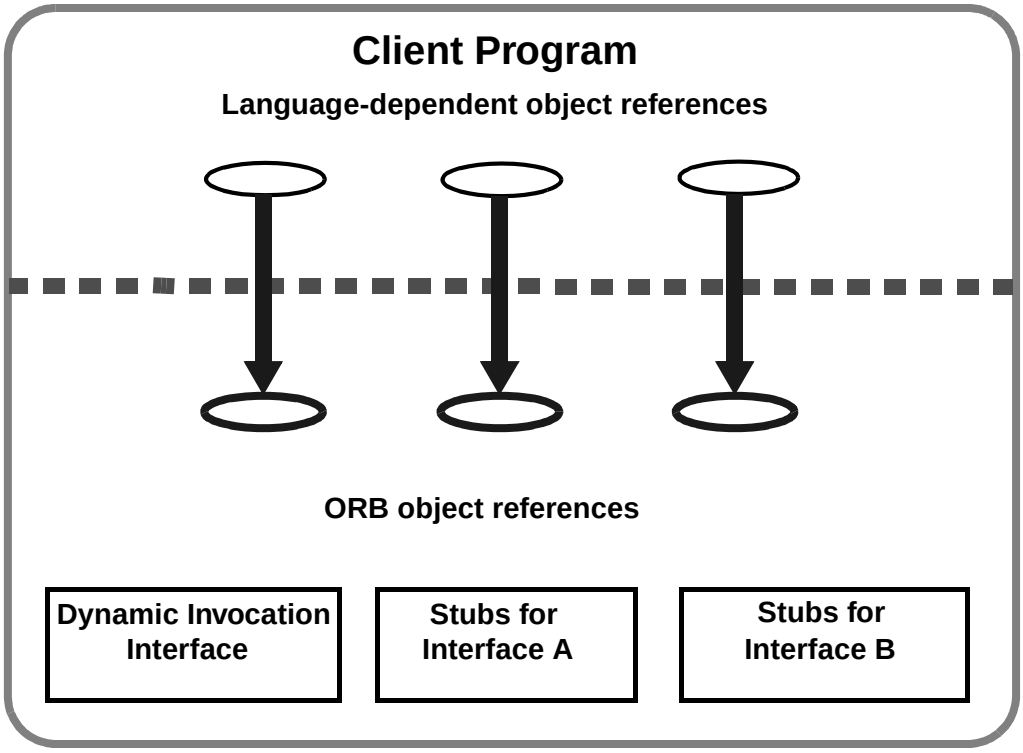


Figure 2-6 The Structure of a Typical Client

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 Figure 2-7) 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.

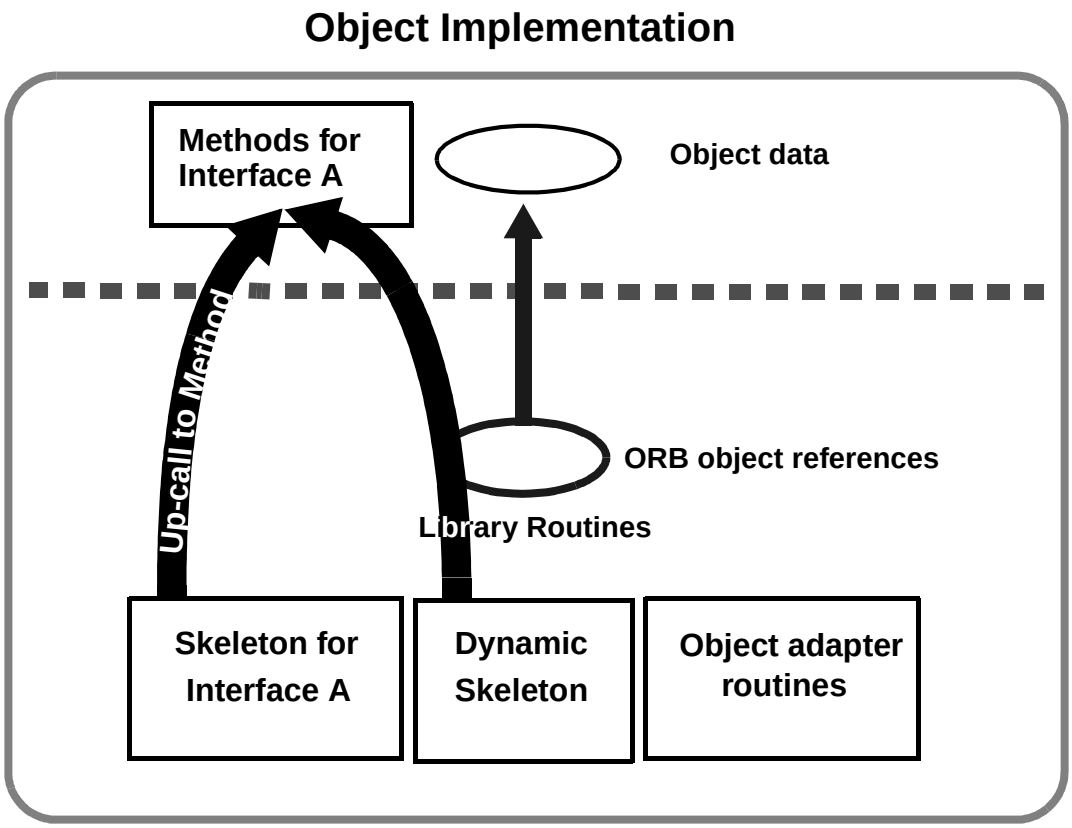


Figure 2-7 The Structure of a Typical Object Implementation

Because of the range of possible object implementations, it is difficult to be definitive about how an object implementation is structured. See the chapters on the Portable 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 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 Portable Object Adapter provides some persistent data associated with an object (its OID or Object ID), 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 Figure 2-8 on page 2-16) 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.

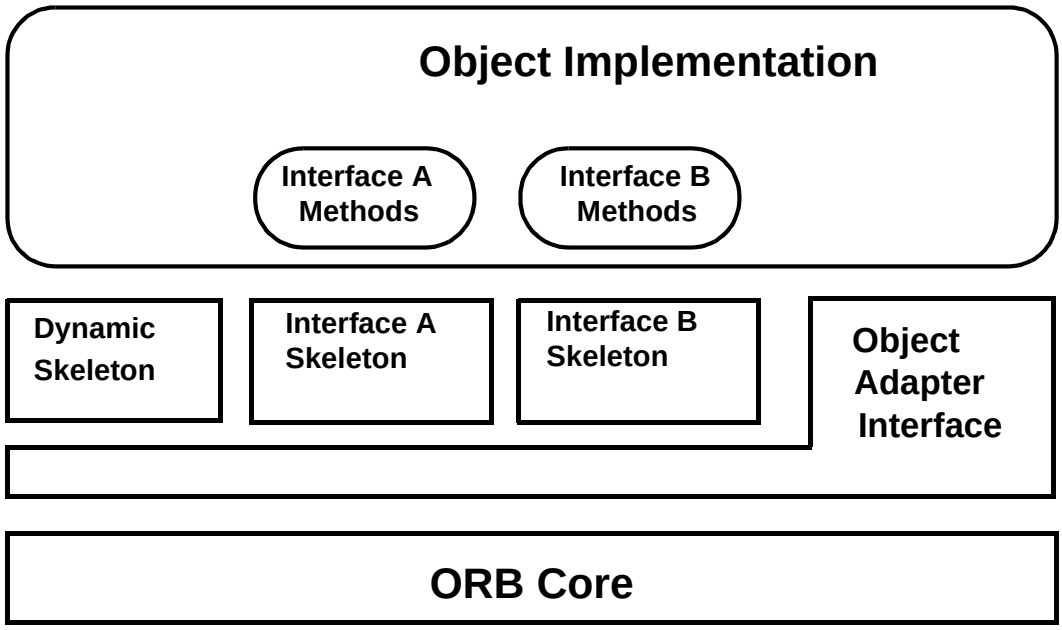


Figure 2-8 The Structure of a Typical Object Adapter

As shown in Figure 2-8, 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.

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 CORBA Required Object Adapter

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 briefly describe the object adapter defined in this specification.

2.6.1 Portable Object Adapter

This specification defines a Portable Object Adapter that can be used for most ORB objects with conventional implementations. (See the Portable Object Adapter chapter for more information.) The intent of the POA, as its name suggests, is to provide an Object Adapter that can be used with multiple ORBs with a minimum of rewriting needed to deal with different vendors' implementations.

This specification allows several ways of using servers but it does not deal with the administrative issues of starting server programs. Once started, however, there can be a servant started and ended for a single method call, a separate servant for each object, or a shared servant for all instances of the object type. It allows for groups of objects to be associated by means of being registered with different instances of the POA object and allows implementations to specify their own activation techniques. If the implementation is not active when an invocation is performed, the POA will start one. The POA is specified in IDL, so its mapping to languages is largely automatic, following the language mapping rules. (The primary task left for a language mapping is the definition of the Servant type.)

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 Figure 2-9 on page 2-18). 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 throughout this manual.

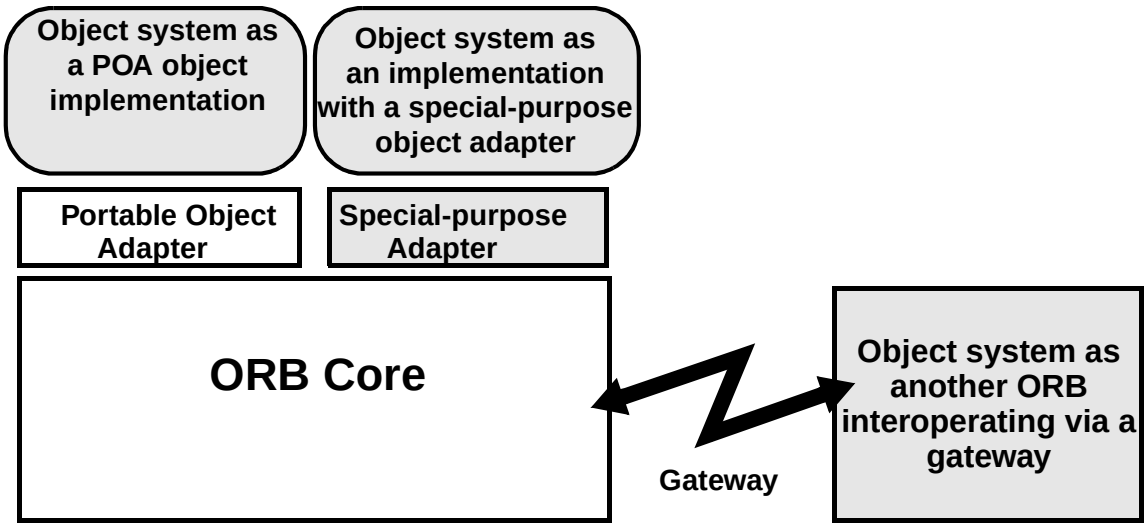


Figure 2-9 Different Ways to Integrate Foreign Object Systems

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 POA 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

This chapter has been updated based on CORE changes from ptc/98-09-04 and the Objects by Value documents (ptc/98-07-05 and orbos/98-01-18). This chapter describes OMG Interface Definition Language (IDL) semantics and gives the syntax for OMG IDL grammatical constructs.

Contents

This chapter contains the following sections.

3.1 Overview

The OMG Interface Definition Language (IDL) 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 several programming languages is described in this manual.

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.

The description of OMG IDL’s lexical conventions is presented in Section 3.2, “Lexical Conventions,” on page 3-3. A description of OMG IDL preprocessing is presented in Section 3.3, “Preprocessing,” on page 3-12. The scope rules for identifiers in an OMG IDL specification are described in Section 3.14, “CORBA Module,” on page 3-44.

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 Section 3.4, “OMG IDL Grammar,” on page 3-12.

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.

---

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

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

The description of OMG IDL grammar uses a syntax notation that is similar to Extended Backus-Naur Format (EBNF). Table 3-1 lists the symbols used in this format and their meaning.

Table 3-1 IDL EBNF

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.

OMG IDL uses the ASCII character set, except for string literals and character literals, which use the ISO Latin-1 (8859.1) character set. The ISO Latin-1 character set is divided into alphabetic characters (letters) digits, graphic characters, the space (blank)

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

character, and formatting characters. Table 3-2 shows the ISO Latin-1 alphabetic characters; upper and lower case equivalences are paired. The ASCII alphabetic characters are shown in the left-hand column of Table 3-2.

Table 3-2 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	Ññ	Upper/Lower-case N with tilde
Rr	Upper/Lower-case R	Òò	Upper/Lower-case O with grave accent
Ss	Upper/Lower-case S	Óó	Upper/Lower-case O with acute accent
Tt	Upper/Lower-case T	Ôô	Upper/Lower-case O with circumflex accent
Uu	Upper/Lower-case U	Õõ	Upper/Lower-case O with tilde
Vv	Upper/Lower-case V	Öö	Upper/Lower-case O with diaeresis
Ww	Upper/Lower-case W	Øø	Upper/Lower-case O with oblique stroke
Xx	Upper/Lower-case X	Ùù	Upper/Lower-case U with grave accent
Yy	Upper/Lower-case Y	Úú	Upper/Lower-case U with acute accent
Zz	Upper/Lower-case Z	Ûû	Upper/Lower-case U with circumflex accent
		Üü	Upper/Lower-case U with diaeresis
		ß	Lower-case German sharp S
		ÿ	Lower-case Y with diaeresis

Table 3-3 lists the decimal digit characters.

Table 3-3 Decimal Digits

0 1 2 3 4 5 6 7 8 9

Table 3-4 shows the graphic characters.

Table 3-4 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	¼	vulgar fraction 1/4
{	left curly bracket	½	vulgar fraction 1/2
|	vertical line	¾	vulgar fraction 3/4
}	right curly bracket	¿	inverted question mark
~	tilde	×	multiplication sign
		÷	division sign

The formatting characters are shown in Table 3-5.

Table 3-5 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 ASCII alphabetic, digit, and underscore (“_”) characters. The first character must be an ASCII alphabetic character. All characters are significant.

When comparing two identifiers to see if they collide:

- Upper- and lower-case letters are treated as the same letter. Table 3-2 on page 3-4 defines the equivalence mapping of upper- and lower-case letters.
- All characters are significant.

Identifiers that differ only in case collide, and will yield a compilation error under certain circumstances. An identifier for a given definition must be spelled identically (e.g., with respect to case) throughout a specification.

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

For example:

```
module M {
    typedef long Foo;
    const long thing = 1;
    interface thing {          // error: reuse of identifier
        void doit (
            in Foo foo         // error: Foo and foo collide and refer to
                                   different things
        );

        readonly attribute long Attribute; // error: Attribute collides with
                                                keyword attribute
    };
};
```

3.2.3.1 *Escaped Identifiers*

As IDL evolves, new keywords that are added to the IDL language may inadvertently collide with identifiers used in existing IDL and programs that use that IDL. Fixing these collisions will require not only the IDL to be modified, but programming language code that depends upon that IDL will have to change as well. The language mapping rules for the renamed IDL identifiers will cause the mapped identifier names (e.g., method names) to be changed.

To minimize the amount of work, users may lexically “escape” identifiers by prepending an underscore (_) to an identifier. This is a purely lexical convention which ONLY turns off keyword checking. The resulting identifier follows all the other rules for identifier processing. For example, the identifier _**AnIdentifier** is treated as if it were **AnIdentifier**.

The following is a non-exclusive list of implications of these rules:

- The underscore does not appear in the Interface Repository.
- The underscore is not used in the DII and DSI.
- The underscore is not transmitted over “the wire.”
- Case sensitivity rules are applied to the identifier after stripping off the leading underscore.

For example:

```
module M {
    interface thing {
        attribute boolean abstract;      // error: abstract collides with
                                         //    keyword abstract
        attribute boolean _abstract;     // ok: abstract is an identifier
    };
};
```

To avoid unnecessary confusion for readers of IDL, it is recommended that interfaces only use the escaped form of identifiers when the unescaped form clashes with a newly introduced IDL keyword. It is also recommended that interface designers avoid defining new identifiers that are known to require escaping. Escaped literals are only recommended for IDL that expresses legacy interface, or for IDL that is mechanically generated.

3.2.4 Keywords

The identifiers listed in Table 3-6 are reserved for use as keywords and may not be used otherwise, unless escaped with a leading underscore**.**

Table 3-6 Keywords

abstract	double	long	readonly	unsigned
any	enum	module	sequence	union
attribute	exception	native	short	ValueBase
boolean	factory	Object	string	valuetype
case	FALSE	octet	struct	void
char	fixed	oneway	supports	wchar
const	float	out	switch	wstring
context	in	private	TRUE	
custom	inout	public	truncatable	
default	interface	raises	typedef	

Keywords must be written exactly as shown in the above list. Identifiers that collide with keywords (see Section 3.2.3, “Identifiers,” on page 3-6) are illegal. For example, “**boolean**” is a valid keyword; “**Boolean**” and “**BOOLEAN**” are illegal identifiers.

For example:

```
module M {
    typedef Long Foo;                // Error: keyword is long not Long
    typedef boolean BOOLEAN;         // Error: BOOLEAN collides with
                                     // the keyword boolean;
};
```

OMG IDL specifications use the characters shown in Table 3-7 as punctuation.

Table 3-7 Punctuation Characters

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

In addition, the tokens listed in Table 3-8 are used by the preprocessor.

Table 3-8 Preprocessor Tokens

#	##	!	||	&&

3.2.5 Literals

This section describes the following literals:

- Integer
- Character
- Floating-point
- String
- Fixed-point

3.2.5.1 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.

3.2.5.2 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 Table 3-2 on page 3-4, Table 3-3 on page 3-4, and Table 3-4 on page 3-5). 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 Table 3-5 on page 3-6). The meaning of all other characters is implementation-dependent.

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

Table 3-9 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
unicode character	\uhhhh

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.

The escape \uhhhh consists of a backslash followed by the character ‘u’, followed by one, two, three or four hexadecimal digits. This represents a unicode character literal. Thus the literal “\u002E” represents the unicode period ‘.’ character and the literal “\u3BC” represents the unicode greek small letter ‘mu’. The \u escape is valid only with wchar and wstring types. Since wide string literal is defined as a sequence of wide character literals a sequence of \u literals can be used to define a wide string literal. Attempt to set a char type to a \u defined literal or a string type to a sequence of \u literals result in an error.

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.

Wide character literals have an **L** prefix, for example:

```
const wchar C1 = L'X';
```

Attempts to assign a wide character literal to a non-wide character constant or to assign a non-wide character literal to a wide character constant result in a compile-time diagnostic.

Both wide and non-wide character literals must be specified using characters from the ISO 8859-1 character set.

3.2.5.3 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.

3.2.5.4 String Literals

A string literal is a sequence of characters (as defined in Section 3.2.5.2, “Character Literals,” on page 3-9) 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'.

Wide string literals have an L prefix, for example:

```
const wstring S1 = L"Hello";
```

Attempts to assign a wide string literal to a non-wide string constant or to assign a non-wide string literal to a wide string constant result in a compile-time diagnostic.

Both wide and non-wide string literals must be specified using characters from the ISO 8859-1 character set.

A wide string literal shall not contain the wide character with value zero.

3.2.5.5 Fixed-Point Literals

A fixed-point decimal literal consists of an integer part, a decimal point, a fraction part and a d or D. The integer and fraction parts both consist of a sequence of decimal (base 10) digits. Either the integer part or the fraction part (but not both) may be missing; the decimal point (but not the letter d (or D)) may be missing.

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 (see Section 3.2.1, “Tokens,” on page 3-6), 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*. The #pragma directive that is used to include RepositoryIds is described in Section 10.6, “RepositoryIds,” on page 10-39.

3.4 OMG IDL Grammar

```
(1)   <specification>  ::= <definition>+
(2)    <definition>    ::= <type_dcl> ";"
                         |  <const_dcl> ";"
                         |  <except_dcl> ";"
                         |  <interface> ";"
                         |  <module> ";"
                         |  <value> ";"
(3)        <module>    ::= "module" <identifier> "{" <definition>+ "}"
```

```
(4)                 <interface>  ::= <interface_dcl>
                                 |   <forward_dcl>
(5)               <interface_dcl>  ::= <interface_header> “{” <interface_body> “}”
(6)               <forward_dcl>  ::= [ “abstract” ] “interface” <identifier>
(7)            <interface_header>  ::= [ “abstract” ] “interface” <identifier>
                                       [ <interface_inheritance_spec> ]
(8)              <interface_body>  ::= <export>*
(9)                    <export>  ::= <type_dcl> “;”
                                 |   <const_dcl> “;”
                                 |   <except_dcl> “;”
                                 |   <attr_dcl> “;”
                                 |   <op_dcl> “;”
(10)<interface_inheritance_spec>::=“:” <interface_name>
                                       { “,” <interface_name> }*
(11)            <interface_name>  ::= <scoped_name>
(12)              <scoped_name>  ::= <identifier>
                                 |   “::” <identifier>
                                 |   <scoped_name> “::” <identifier>
(13)                    <value>  ::= ( <value_dcl> | <value_abs_dcl> |
                                       <value_box_dcl> |  <value_forward_dcl>)
(14)         <value_forward_dcl>  ::= [ “abstract” ] “valuetype” <identifier>
(15)             <value_box_dcl>  ::= “valuetype”  <identifier> <type_spec>
(16)             <value_abs_dcl>  ::= “abstract” “valuetype” <identifier>
                                       [ <value_inheritance_spec> ]
                                       “{“ <export>* “}”
(17)                  <value_dcl>  ::= <value_header> “{“  < value_element>* “}”
(18)               <value_header>  ::= [“custom” ] “valuetype” <identifier>
                                       [ <value_inheritance_spec> ]
(19)<value_inheritance_spec>  ::= [ “:” [ “truncatable” ] <value_name>
                                       { “,”  <value_name> }* ]
                                       [ “supports” <interface_name>
                                       { “,” <interface_name> }*  ]
(20)                <value_name>  ::= <scoped_name>
(21)             <value_element>  ::= <export> | < state_member> | <init_dcl>
(22)              <state_member>  ::= ( “public” | “private” )
                                       <type_spec> <declarators> “;”
(23)                   <init_dcl>  ::= “factory” <identifier>
                                       “(“  [ <init_param_decls> ]  “)” “;”
(24)             <init_param_decls>  ::= <init_param_decl> { “,” <init_param_decl> }
(25)              <init_param_decl>  ::= <init_param_attribute> <param_type_spec>
                                       <simple_declarator>
(26)          <init_param_attribute>  ::= “in”
(27)                 <const_dcl>  ::= <const_dcl>::=“const” <const_type>
                                       <identifier> “=” <const_exp>
(28)                 <const_type>  ::= <integer_type>
                                 |   <char_type>
                                 |   <wide_char_type>
```

```
                                    |   <boolean_type>
                                    |   <floating_pt_type>
                                    |   <string_type>
                                    |   <wide_string_type>
                                    |   <fixed_pt_const_type>
                                    |   <scoped_name>
                                    |   <octet_type>
(29)                <const_exp>     ::= <or_expr>
(30)                  <or_expr>     ::= <xor_expr>
                                    |   <or_expr> "|" <xor_expr>
(31)                 <xor_expr>     ::= <and_expr>
                                    |   <xor_expr> "^" <and_expr>
(32)                 <and_expr>     ::= <shift_expr>
                                    |   <and_expr> "&" <shift_expr>
(33)               <shift_expr>     ::= <add_expr>
                                    |   <shift_expr> ">>" <add_expr>
                                    |   <shift_expr> "<<" <add_expr>
(34)                 <add_expr>     ::= <mult_expr>
                                    |   <add_expr> "+" <mult_expr>
                                    |   <add_expr> "-" <mult_expr>
(35)                <mult_expr>     ::= <unary_expr>
                                    |   <mult_expr> "*" <unary_expr>
                                    |   <mult_expr> "/" <unary_expr>
                                    |   <mult_expr> "%" <unary_expr>
(36)               <unary_expr>     ::= <unary_operator> <primary_expr>
                                    |   <primary_expr>
(37)            <unary_operator>    ::= "-"
                                    |   "+"
                                    |   "~"
(38)             <primary_expr>     ::= <scoped_name>
                                    |   <literal>
                                    |   "(" <const_exp> ")"
(39)                  <literal>     ::= <integer_literal>
                                    |   <string_literal>
                                    |   <wide_string_literal>
                                    |   <character_literal>
                                    |   <wide_character_literal>
                                    |   <fixed_pt_literal>
                                    |   <floating_pt_literal>
                                    |   <boolean_literal>
(40)            <boolean_literal>   ::= "TRUE"
                                    |   "FALSE"
(41)         <positive_int_const>   ::= <const_exp>
(42)                  <type_dcl>    ::= "typedef" <type_declarator>
                                    |   <struct_type>
                                    |   <union_type>
                                    |   <enum_type>
                                    |   "native" <simple_declarator>
```

```
(43)          <type_declarator>  ::= <type_spec> <declarators>
(44)                <type_spec>  ::= <simple_type_spec>
                                 |   <constr_type_spec>
(45)         <simple_type_spec>  ::= <base_type_spec>
                                 |   <template_type_spec>
                                 |   <scoped_name>
(46)          <base_type_spec>   ::= <floating_pt_type>
                                 |   <integer_type>
                                 |   <char_type>
                                 |   <wide_char_type>
                                 |   <boolean_type>
                                 |   <octet_type>
                                 |   <any_type>
                                 |   <object_type>
                                 |   <value_base_type>
(47)      <template_type_spec>   ::= <sequence_type>
                                 |   <string_type>
                                 |   <wide_string_type>
                                 |   <fixed_pt_type>
(48)        <constr_type_spec>   ::= <struct_type>
                                 |   <union_type>
                                 |   <enum_type>
(49)              <declarators>  ::= <declarator> { “,” <declarator> }*
(50)              <declarator>   ::= <simple_declarator>
                                 |   <complex_declarator>
(51)        <simple_declarator>  ::= <identifier>
(52)       <complex_declarator>  ::= <array_declarator>
(53)          <floating_pt_type> ::= “float”
                                 |   “double”
                                 |   “long” “double”
(54)              <integer_type> ::= <signed_int>
                                 |   <unsigned_int>
(55)                <signed_int> ::= <signed_short_int>
                                 |   <signed_long_int>
                                 |   <signed_longlong_int>
(56)          <signed_short_int> ::= “short”
(57)           <signed_long_int> ::= “long”
(58)       <signed_longlong_int> ::= “long” “long”
(59)              <unsigned_int> ::= <unsigned_short_int>
                                 |   <unsigned_long_int>
                                 |   <unsigned_longlong_int>
(60)        <unsigned_short_int> ::= “unsigned” “short”
(61)         <unsigned_long_int> ::= “unsigned” “long”
(62)     <unsigned_longlong_int> ::= “unsigned” “long” “long”
(63)              <char_type>    ::= “char”
(64)         <wide_char_type>    ::= “wchar”
(65)              <boolean_type> ::= “boolean”
```

```
(66)          <octet_type>  ::= "octet"
(67)          <any_type>    ::= "any"
(68)          <object_type> ::= "Object"
(69)          <struct_type> ::= "struct" <identifier> "{" <member_list> "}"
(70)          <member_list> ::= <member>+
(71)          <member>      ::= <type_spec> <declarators> ";"
(72)          <union_type>  ::= "union" <identifier> "switch"
                                "(" <switch_type_spec> ")"
                                "{" <switch_body> "}"
(73)   <switch_type_spec>   ::= <integer_type>
                              |  <char_type>
                              |  <boolean_type>
                              |  <enum_type>
                              |  <scoped_name>
(74)          <switch_body> ::= <case>+
(75)          <case>        ::= <case_label>+ <element_spec> ";"
(76)          <case_label>  ::= "case" <const_exp> ":"
                              |  "default" ":"
(77)          <element_spec>::= <type_spec> <declarator>
(78)          <enum_type>   ::= "enum" <identifier>
                                "{" <enumerator> { "," <enumerator> }* "}"
(79)          <enumerator>  ::= <identifier>
(80)       <sequence_type>  ::= "sequence" "<" <simple_type_spec> ","
                                <positive_int_const> ">"
                              |  "sequence" "<" <simple_type_spec> ">"
(81)          <string_type> ::= "string" "<" <positive_int_const> ">"
                              |  "string"
(82)     <wide_string_type> ::= "wstring" "<" <positive_int_const> ">"
                              |  "wstring"
(83)     <array_declarator> ::= <identifier> <fixed_array_size>+
(84)     <fixed_array_size> ::= "[" <positive_int_const> "]"
(85)          <attr_dcl>    ::= [ "readonly" ] "attribute"
                                <param_type_spec> <simple_declarator>
                                {  "," <simple_declarator> }*
(86)          <except_dcl>  ::= "exception" <identifier> "{" <member>* "}"
(87)          <op_dcl>      ::= [ <op_attribute> ] <op_type_spec>
                                <identifier> <parameter_dcls>
                                [ <raises_expr> ] [ <context_expr> ]
(88)          <op_attribute>::= "oneway"
(89)          <op_type_spec>::= <param_type_spec>
                              |  "void"
(90)       <parameter_dcls> ::= "(" <param_dcl> { "," <param_dcl> }* ")"
                              |  "(" ")"
(91)          <param_dcl>   ::= <param_attribute> <param_type_spec>
                                <simple_declarator>
(92)       <param_attribute>::= "in"
```

```
                                  |   “out”
                                  |   “inout”
(93)              <raises_expr>   ::= “raises” “(” <scoped_name>
                                      { “,”  <scoped_name> }* “)”
(94)              <context_expr>  ::= “context” “(” <string_literal>
                                      { “,” <string_literal> }* “)”
(95)           <param_type_spec>  ::= <base_type_spec>
                                  |   <string_type>
                                  |   <wide_string_type>
                                  |   <scoped_name>
(96)              <fixed_pt_type> ::= “fixed” “<“ <positive_int_const> “,”
                                      <positive_int_const> “>”
(97)     <fixed_pt_const_type>    ::= “fixed”
(98)          <value_base_type>   ::= “ValueBase”
```

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> “;”
                  |   <value> “;”
```

See Section 3.9, “Constant Declaration,” on page 3-28, Section 3.10, “Type Declaration,” on page 3-31, and Section 3.11, “Exception Declaration,” on page 3-40 respectively for specifications of **<const_dcl>**, **<type_dcl>**, and **<except_dcl>**.

See Section 3.7, “Interface Declaration,” on page 3-18 for the specification of <interface>.

See Section 3.6, “Module Declaration,” on page 3-17 for the specification of <module>.

See Section 3.8, “Value Declaration,” on page 3-23 for the specification of <value>.

3.6 Module Declaration

A module definition satisfies the following syntax:

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

The module construct is used to scope OMG IDL identifiers; see Section 3.14, “CORBA Module,” on page 3-44 for details.

3.7 Interface Declaration

An interface definition satisfies the following syntax:

```
<interface>          ::= <interface_dcl>
                       |  <forward_dcl>
<interface_dcl>      ::= <interface_header> "{" <interface_body> "}"
<forward_dcl>        ::= [ "abstract" ] "interface" <identifier>
<interface_header>   ::= [ "abstract" ] "interface" <identifier>
                           [ <interface_inheritance_spec> ]
<interface_body>     ::= <export>*
<export>             ::= <type_dcl> ";"
                       |  <const_dcl> ";"
                       |  <except_dcl> ";"
                       |  <attr_dcl> ";"
                       |  <op_dcl> ";"
```

3.7.1 Interface Header

The interface header consists of three elements:

- An optional modifier specifying if the interface is an abstract interface.
- 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.

Abstract interfaces have slightly different rules and semantics from "regular" interfaces as described in Chapter 6, "Abstract Interface Semantics". They also follow different language mapping rules.

3.7.2 Interface Inheritance Specification

The syntax for inheritance is as follows:

```
<interface_inheritance_spec>::= ":" <interface_name>
                                    {"," <interface_name>}*
<interface_name>             ::= <scoped_name>
```

<scoped_name> **::= <identifier>**
| “::” <identifier>
| <scoped_name> “::” <identifier>

Each **<scoped_name>** in an **<interface_inheritance_spec>** must denote a previously defined interface. See Section 3.7.5, “Interface Inheritance,” on page 3-20 for the description of inheritance.

3.7.3 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 Section 3.9, “Constant Declaration,” on page 3-28.
- Type declarations, which specify the type definitions that the interface exports; type declaration syntax is described in Section 3.10, “Type Declaration,” on page 3-31.
- Exception declarations, which specify the exception structures that the interface exports; exception declaration syntax is described in Section 3.11, “Exception Declaration,” on page 3-40.
- Attribute declarations, which specify the associated attributes exported by the interface; attribute declaration syntax is described in Section 3.13, “Attribute Declaration,” on page 3-43.
- 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 Section 3.12, “Operation Declaration,” on page 3-41.

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

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

3.7.4 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.

It is illegal to inherit from a forward-declared interface whose definition has not yet been seen:

```
module Example {
    interface base;              // Forward declaration

    // ...

    interface derived : base {};// Error
    interface base {};           // Define base
    interface derived : base {};// OK
};
```

3.7.5 Interface 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 Section 3.14, “CORBA Module,” on page 3-44.

An interface is called a direct base if it is mentioned in the **<interface_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 **<interface_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 abstract interface may only inherit from other abstract interfaces.

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 { ... }
interface E: A, B { ... };                    // OK
```

The relationships between these interfaces is shown in Figure 3-1. This “diamond” shape is legal, as is the definition of E on the right.

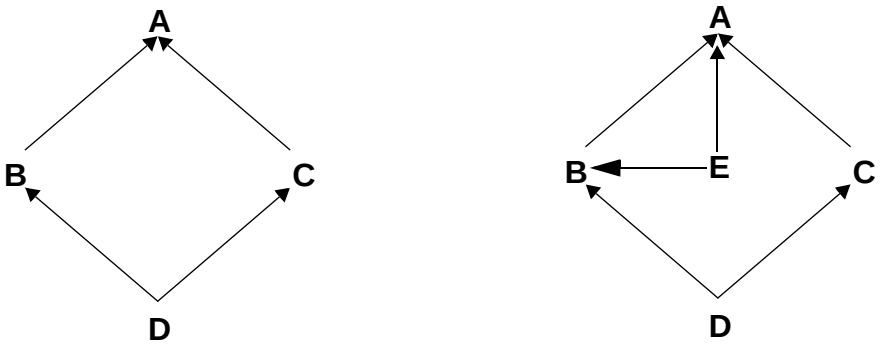


Figure 3-1 Legal Multiple Inheritance Example

References to base interface elements must be unambiguous. A Reference to a base interface element is ambiguous if the name is declared as a constant, type, or exception in more than one base interface. Ambiguities can be resolved by qualifying a name with its interface name (that is, using a **<scoped_name>**). It is 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.

So for example in:

```
interface A {
    typedef long L1;
    short opA(in L1 l_1);
};

interface B {
    typedef short L1;
    L1 opB(in long l);
};

interface C: B, A {
    typedef L1 L2;          // Error: L1 ambiguous
    typedef A::L1 L3;       // A::L1 is OK
    B::L1 opC(in L3 l_3);   // all OK no ambiguities
};
```

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 {
    typedef float coord[L]:
    void f (in coord s);          // s has three floats
};

interface B {
    const long L = 4;
};

interface C: B, A { };                    // what is C::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

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

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 produces a compilation error. Thus in

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

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

interface C: A, B {
    attribute string_t      Title;      // Error: string_t ambiguous
    attribute A::string_t   Name;       // OK
    attribute B::string_t   City;       // OK
};
```

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

```
interface A {
    void make_it_so();
};

interface B: A {
    short make_it_so(in long times);   // Error: redefinition of make_it_so
};
```

3.8 Value Declaration

There are several kinds of value type declarations: “regular” value types, boxed value types, abstract value types, and forward declarations.

A value declaration satisfies the following syntax:

```
<value>     ::= ( <value_dcl>
            |   <value_abs_dcl>
            |   <value_box_dcl>
            |   <value_forward_dcl>) “;”
```

3.8.1 Regular Value Type

A regular value type satisfies the following syntax:

```
<value_dcl>     ::= <value_header> “{“  < value_element >* “}”

<value_header>  ::= [“custom” ] “valuetype” <identifier>
                    [ <value_inheritance_spec> ]

<value_element> ::= <export>
                |   < state_member>
                |   <init_dcl>
```

3.8.1.1 Value Header

The value header consists of two elements:

- The value type’s name and optional modifier specifying whether the value type uses custom marshaling.
- An optional value inheritance specification. The value inheritance specification is described in the next section.

3.8.1.2 Value Element

A value can contain all the elements that an interface can as well as the definition of state members, and initializers for that state.

3.8.1.3 Value Inheritance Specification

```
<value_inheritance_spec> ::=  [ “:” [ “truncatable” ] <value_name>
                                { “,” <value_name> }* ]
```

[“supports” <interface_name>
{ “,” interface_name> }*]

<value_name> **::= <scoped_name>**

Each **<value_name>** and **<interface_name>** in a **<value_inheritance_spec>** must denote previously defined value type or interface. See Section 3.8.5, “Valuetype Inheritance,” on page 3-27 for the description of value type inheritance.

The **truncatable** modifier may not be used if the value type being defined is a custom value.

3.8.1.4 State Members

<state_member> ::=(“public” | “private”) <type_spec> <declarators> “;”

Each **<state_member>** defines an element of the state, which is marshaled and sent to the receiver when the value type is passed as a parameter. A state member is either public or private. The annotation directs the language mapping to hide or expose the different parts of the state to the clients of the value type. The private part of the state is only accessible to the implementation code and the marshaling routines.

Note that certain programming languages may not have the built in facilities needed to distinguish between the public and private members. In these cases, the language mapping specifies the rules that programmers are responsible for following.

3.8.1.5 Initializers

<init_dcl> **::= “factory” <identifier>**
“(“ [<init_param_decls>] “)” “;”

<init_param_decls> **::= <init_param_decl> { “,” <init_param_decl> }**

<init_param_decl> **::= <init_param_attribute> <param_type_spec>**
<simple_declarator>

<init_param_attribute> ::= “in”

In order to ensure portability of value implementations, designers may also define the signatures of initializers (or constructors) for non abstract value types. Syntactically these look like local operation signatures except that they are prefixed with the keyword **factory**, have no return type, and must use only in parameters. There may be any number of factory declarations. The names of the initializers are part of the name scope of the value type.

If no initializers are specified in IDL, the value type does not provide a portable way of creating a runtime instance of its type. There is no default initializer. This allows the definition of IDL value types which are not intended to be directly instantiated by client code.

3.8.1.6 *Value Type Example*

```
interface Tree {
    void print()
};

valuetype WeightedBinaryTree {
    // state definition
        private unsigned long weight;
        private WeightedBinaryTree left;
        private WeightedBinaryTree right;
    // initializer
        factory init(in unsigned long w);
    // local operations
        WeightSeq pre_order();
        WeightSeq post_order();
};
valuetype WTree: WeightedBinaryTree supports Tree {};
```

3.8.2 *Boxed Value Type*

<value_box_dcl> ::= “valuetype” <identifier> <type_spec>

It is often convenient to define a value type with no inheritance or operations and with a single state member. A shorthand IDL notation is used to simplify the use of value types for this kind of simple containment, referred to as a “value box.”

This is particularly useful for strings and sequences. Basically one does not have to create what is in effect an additional namespace that will contain only one name.

An example is the following IDL:

```
module Example {
    interface Foo {
        ... /* anything */
    };
    valuetype FooSeq sequence<Foo>;
    interface Bar {
        void doIt (in FooSeq seq1);
    };
};
```

The above IDL provides similar functionality to writing the following IDL. However the type identities (repository ID’s) would be different.

```
module Example {
    interface Foo {
        ... /* anything */
    };
    valuetype FooSeq {
        public sequence<Foo> data;
    };
    interface Bar {
        void doIt (in FooSeq seq);
    };
};
```

The former is easier to manipulate after it is mapped to a concrete programming language.

The declaration of a boxed value type does not open a new scope.Thus a construction such as:

```
valuetype FooSeq sequence <FooSeq>;
```

is not legal IDL. The identifier being declared as a boxed value type cannot be used subsequent to its initial use and prior to the completion of the boxed value declaration.

3.8.3 Abstract Value Type

<value_abs_dcl> ::= “abstract” “valuetype” <identifier>
[<value_inheritance_spec>] “{“ <export>* “}”

Value types may also be abstract. They are called abstract because an abstract value type may not be instantiated. No <state_member> or <initializers> may be specified. However, local operations may be specified. Essentially they are a bundle of operation signatures with a purely local implementation.

Note that a concrete value type with an empty state is not an abstract value type.

3.8.4 Value Forward Declaration

<value_forward_dcl> ::= [“abstract”] “valuetype” <identifier>

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

Multiple forward declarations of the same value type name are legal.

Boxed value types cannot be forward declared; such a forward declaration would refer to a normal value type.

It is illegal to inherit from a forward-declared value type whose definition has not yet been seen.

3.8.5 Valuetype Inheritance

The terminology that is used to describe value type inheritance is directly analogous to that used to describe interface inheritance (see Section 3.7.5, “Interface Inheritance,” on page 3-20).

The name scoping and name collision rules for valuetypes are identical to those for interfaces. In addition, no valuetype may be specified as a direct abstract base of a derived valuetype more than once; it may be an indirect abstract base more than once. See Section 3.7.5, “Interface Inheritance,” on page 3-20 for a detailed description of the analogous properties for interfaces.

Values may be derived from other values and can support an interface and any number of abstract interfaces.

Once implementation (state) is specified at a particular point in the inheritance hierarchy, all derived value types (which must of course implement the state) may only derive from a single (concrete) value type. They can however derive from other additional abstract values and support an additional interface.

The single immediate base concrete value type, if present, must be the first element specified in the inheritance list of the value declaration’s IDL. It may be followed by other abstract values from which it inherits. The interface and abstract interfaces that it supports are listed following the **supports** keyword.

A stateful value that derives from another stateful value may specify that it is truncatable. This means that it is to “truncate” (see Section 5.2.5.2, “Value instance -> Value type,” on page 5-5) an instance to be an instance of any of its truncatable parent (stateful) value types under certain conditions. Note that all the intervening types in the inheritance hierarchy must be truncatable in order for truncation to a particular type to be allowed.

Because custom values require an exact type match between the sending and receiving context, **truncatable** may not be specified for a custom value type.

Non-custom value types may not (transitively) inherit from custom value types.

Boxed value types may not be derived from, nor may they derive from anything else.

These rules are summarized in the following table:

Table 3-10 Allowable Inheritance Relationships

May inherit from:	Interface	Abstract Interface	Abstract Value	Stateful Value	Boxed value
Interface	multiple	multiple	no	no	no
Abstract Interface	no	multiple	no	no	no
Abstract Value	supports	supports	multiple	no	no
Stateful Value	supports single	supports	multiple	single (may be truncatable)	no
Boxed Value	no	no	no	no	no

3.9 Constant Declaration

This section describes the syntax for constant declarations.

3.9.1 Syntax

The syntax for a constant declaration is:

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

<const_type>       ::= <integer_type>
                   |   <char_type>
                   |   <wide_char_type>
                   |   <boolean_type>
                   |   <floating_pt_type>
                   |   <string_type>
                   |   <wide_string_type>
                   |   <fixed_pt_const_type>
                   |   <scoped_name>
                   |   <octet_type>

<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>
```

3.9.2 Semantics

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

An infix operator can combine two integers, floats or fixeds, but not mixtures of these. Infix operators are applicable only to integer, float and fixed types.

If the type of an integer constant is **long** or **unsigned long**, then each subexpression of the associated constant expression is treated as an **unsigned long** by default, or a signed **long** for negated literals or negative integer constants. It is an error if any subexpression values exceed the precision of the assigned type (**long** or **unsigned long**), or if a final expression value (of type **unsigned long**) exceeds the precision of the target type (**long**).

If the type of an integer constant is **long long** or **unsigned long long**, then each subexpression of the associated constant expression is treated as an **unsigned long long** by default, or a signed **long long** for negated literals or negative integer constants. It is an error if any subexpression values exceed the precision of the assigned type (**long long** or **unsigned long long**), or if a final expression value (of type **unsigned long long**) exceeds the precision of the target type (**long long**).

If the type of a floating-point constant is **double**, then each subexpression of the associated constant expression is treated as a **double.** It is an error if any subexpression value exceeds the precision of **double**.

If the type of a floating-point constant is **long double**, then each subexpression of the associated constant expression is treated as a **long double**. It is an error if any subexpression value exceeds the precision of **long double**.

Fixed-point decimal constant expressions are evaluated as follows. A fixed-point literal has the apparent number of total and fractional digits, except that leading and trailing zeros are factored out, including non-significant zeros before the decimal point. For example, **0123.450d** is considered to be **fixed<5,2>** and **3000.00D** is **fixed<1,-3>**. Prefix operators do not affect the precision; a prefix **+** is optional, and does not change

the result. The upper bounds on the number of digits and scale of the result of an infix expression, **fixed<d1,s1> op fixed<d2,s2>**, are shown in the following table:

Op	Result: fixed<d,s>
+	fixed<max(d1-s1,d2-s2) + max(s1,s2) + 1, max(s1,s2)>
-	fixed<max(d1-s1,d2-s2) + max(s1,s2) + 1, max(s1,s2)>
*	fixed<d1+d2, s1+s2>
/	**fixed<(d1-s1+s2) + s_{inf}, s_{inf}>**

A quotient may have an arbitrary number of decimal places, denoted by a scale of s_{inf}. The computation proceeds pairwise, with the usual rules for left-to-right association, operator precedence, and parentheses. All intermediate computations shall be performed using double precision (i.e., 62 digit) arithmetic. If an individual computation between a pair of fixed-point literals actually generates more than 31 significant digits, then a 31-digit result is retained as follows:

fixed<d,s> => fixed<31, 31-d+s>

Leading and trailing zeros are not considered significant. The omitted digits are discarded; rounding is not performed. The result of the individual computation then proceeds as one literal operand of the next pair of fixed-point literals to be computed.

Unary (**+ -**) and binary (*** / + -**) operators are applicable in floating-point and fixed-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:

Integer Constant Expression Type	Generated 2's Complement Numbers
long	long -(value+1)
unsigned long	unsigned long (2**32-1) - value
long long	long long -(value+1)
unsigned long long	unsigned long (2**64-1) - 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 < 64.

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 < 64.

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.

An octet constant can be defined using an integer literal or an integer constant expression, for example:

```
const octet O1 = 0x1;
const long L = 3;
const octet O2 = 5 + L;
```

Values for an octet constant outside the range 0 - 255 shall cause a compile-time error.

An enum constant can only be defined using a scoped name for the enumerator. The scoped name is resolved using the normal scope resolution rules Section 3.15, “Names and Scoping,” on page 3-45. For example:

```
enum Color { red, green, blue };
const Color FAVORITE_COLOR = red;

module M {
    enum Size { small, medium, large };
};
const M::Size MYSIZE = M::medium;
```

The constant name for the RHS of an enumerated constant definition must denote one of the enumerators defined for the enumerated type of the constant. For example:

```
const Color col = red;    // is OK but
const Color another = M::medium; // is an error
```

3.10 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**, **enum**, and **native** declarations; the syntax is:

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

<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>
                    |   <wide_char_type>
                    |   <boolean_type>
                    |   <octet_type>
                    |   <any_type>
                    |   <object-type>
                    |   <value_base_type>

<template_type_spec>::= <sequence_type>
                    |   <string_type>
                    |   <wide_string_type>
                    |   <fixed_pt_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.

3.10.1 Basic Types

The syntax for the supported basic types is as follows:

```
<floating_pt_type>      ::= "float"
                            | "double"
                            | "long" "double"
<integer_type>:         :=  <signed_int>
                            | <unsigned_int>
<signed_int>            ::= <signed_long_int
                            | <signed_short_int>
                            | <signed_longlong_int>
<signed_long_int>       ::= "long"
<signed_short_int>      ::= "short"
<signed_longlong_int>::= "long" "long"
<unsigned_int>          ::= <unsigned_long_int>
                            | <unsigned_short_int>
                            | <unsigned_longlong_int>
<unsigned_long_int>     ::= "unsigned" "long"
<unsigned_short_int>    ::= "unsigned" "short"
<unsigned_longlong_int>::= "unsigned" "long" "long"
<char_type>             ::= "char"
<wide_char_type>        ::= "wchar"
<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 Section 3.17, “Standard Exceptions,” on page 3-51.

3.10.1.1 Integer Types

OMG IDL integer types are **short**, **unsigned short**, **long**, **unsigned long**, **long long** and **unsigned long long**, representing integer values in the range indicated below in Table 3-11.

Table 3-11 Range of integer types

short	-2^{15} .. $2^{15} - 1$
long	-2^{31} .. $2^{31} - 1$
long long	-2^{63} .. $2^{63} - 1$
unsigned short	0 .. $2^{16} - 1$

Table 3-11 Range of integer types

unsigned long	0 .. 2^{32} - 1
unsigned long long	0 .. 2^{64} - 1

3.10.1.2 Floating-Point Types

OMG IDL floating-point types are **float**, **double** and **long double**. The **float** type represents IEEE single-precision floating point numbers; the **double** type represents IEEE double-precision floating point numbers.The **long double** data type represents an IEEE double-extended floating-point number, which has an exponent of at least 15 bits in length and a signed fraction of at least 64 bits. See *IEEE Standard for Binary Floating-Point Arithmetic*, ANSI/IEEE Standard 754-1985, for a detailed specification.

3.10.1.3 Char Type

OMG IDL defines a **char** data type that is an 8-bit quantity which (1) encodes a single-byte character from any byte-oriented code set, or (2) when used in an array, encodes a multi-byte character from a multi-byte code set. In other words, an implementation is free to use any code set internally for encoding character data, though conversion to another form may be required for transmission.

The ISO 8859-1 (Latin1) character set standard defines the meaning and representation of all possible graphic characters used in OMG IDL (i.e., the space, alphabetic, digit and graphic characters defined in Table 3-2 on page 3-4, Table 3-3 on page 3-4, and Table 3-4 on page 3-5). The meaning and representation of the null and formatting characters (see Table 3-5 on page 3-6) 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.

3.10.1.4 Wide Char Type

OMG IDL defines a **wchar** data type which encodes wide characters from any character set. As with character data, an implementation is free to use any code set internally for encoding wide characters, though, again, conversion to another form may be required for transmission. The size of **wchar** is implementation-dependent.

3.10.1.5 Boolean Type

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

3.10.1.6 Octet Type

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

3.10.1.7 Any Type

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

An **any** logically contains a TypeCode (see Section 3.10, “Type Declaration,” on page 3-31) and a value that is described by the TypeCode. Each IDL language mapping provides operations that allow programers to insert and access the TypeCode and value contained in an any.

3.10.2 Constructed Types

The constructed types are:

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

Although the IDL syntax allows the generation of recursive constructed type specifications, the only recursion permitted for constructed types is through the use of the **sequence** template type. For example, the following is legal:

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

See Section 3.10.3.1, “Sequences,” on page 3-37 for details of the **sequence** template type.

3.10.2.1 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.

3.10.2.2 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 Table 3-12.

Table 3-12 Case Label Matching

Discriminator Type	Matched By
long	any integer value in the value range of long
long long	any integer value in the range of long long
short	any integer value in the value range of short
unsigned long	any integer value in the value range of unsigned long
unsigned long long	any integer value in the range of unsigned long long
unsigned short	any integer value in the value range of unsigned short
char	char
wchar	wchar
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.

3.10.2.3 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.

3.10.3 Template Types

The template types are:

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

3.10.3.1 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.

3.10.3.2 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.

3.10.3.3 Wstrings

The **wstring** data type represents a sequence of wchar, except the wide character null. The type wstring is similar to that of type string, except that its element type is wchar instead of char. The actual length of a wstring is set at run-time and, if the bounded form is used, must be less than or equal to the bound.

The syntax for defining a wstring is:

<wide_string_type> ::= "wstring" "<" <positive_int_const> ">"
| "wstring"

3.10.3.4 Fixed Type

The **fixed** data type represents a fixed-point decimal number of up to 31 significant digits. The scale factor is a non-negative integer less than or equal to the total number of digits (note that constants with effectively negative scale, such as 10000, are always permitted).

The **fixed** data type will be mapped to the native fixed point capability of a programming language, if available. If there is not a native fixed point type, then the IDL mapping for that language will provide a fixed point data type. Applications that use the IDL fixed point type across multiple programming languages must take into account differences between the languages in handling rounding, overflow, and arithmetic precision.

3.10.4 Complex Declarator

3.10.4.1 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.

3.10.5 Native Types

OMG IDL provides a declaration for use by object adapters to define an opaque type whose representation is specified by the language mapping for that object adapter.

The syntax is:

<type_dcl> **::= “native” <simple_declarator>**

<simple_declarator> **::= <identifier>**

This declaration defines a new type with the specified name. A native type is similar to an IDL basic type. The possible values of a native type are language-mapping dependent, as are the means for constructing them and manipulating them. Any interface that defines a native type requires each language mapping to define how the native type is mapped into that programming language.

A native type may be used to define operation parameters and results. However, there is no requirement that values of the type be permitted in remote invocations, either directly or as a component of a constructed type. Any attempt to transmit a value of a native type in a remote invocation may raise the MARSHAL standard exception.

It is recommended that native types be mapped to equivalent type names in each programming language, subject to the normal mapping rules for type names in that language. For example, in a hypothetical Object Adapter IDL module

```
module HypotheticalObjectAdapter {
    native Servant;
    interface HOA {
        Object activate_object(in Servant x);
    };
};
```

the IDL type Servant would map to **HypotheticalObjectAdapter::Servant** in C++ and the **activate_object** operation would map to the following C++ member function signature:

```
CORBA::Object_ptr activate_object(
HypotheticalObjectAdapter::Servant x);
```

The definition of the C++ type **`HypotheticalObjectAdapter::Servant`** would be provided as part of the C++ mapping for the HypotheticalObjectAdapter module.

Note – The native type declaration is provided specifically for use in object adapter interfaces, which require parameters whose values are concrete representations of object implementation instances. It is strongly recommended that it not be used in service or application interfaces. The native type declaration allows object adapters to define new primitive types without requiring changes to the OMG IDL language or to OMG IDL compiler.

3.11 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>** 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 Section 3.17, “Standard Exceptions,” on page 3-51.

3.12 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 Section 3.12.1, “Operation Attribute,” on page 3-42.
- 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 Section 3.12.2, “Parameter Declarations,” on page 3-42.
- 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 3.12.3, “Raises Expressions,” on page 3-43.
- 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 Section 3.12.4, “Context Expressions,” on page 3-43.

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

3.12.1 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.

3.12.2 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.

3.12.3 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 Section 3.17, “Standard Exceptions,” on page 3-51. 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.

3.12.4 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.

3.13 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, assuming that one of the leading '_'s is removed by application of the Escaped Identifier rule described in Section 3.2.3.1, "Escaped Identifiers," on page 3-7:

```
• • •
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 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 Section 3.14, "CORBA Module," on page 3-44 for more information on redefinition constraints and the handling of ambiguity.

3.14 CORBA Module

Names defined by the CORBA specification are in 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.

For example in:

```
#include <orb.idl>
module M {
    typedef CORBA::Object myObjRef;     // Error: keyword Object scoped
    typedef TypeCode myTypeCode;        // Error: TypeCode undefined
    typedef CORBA::TypeCode TypeCode;// OK

};
```

The file **orb.idl** contains the IDL definitions for the CORBA module. The file **orb.idl** must be included in IDL files that use names defined in the CORBA module.

The version of **CORBA** specified in this release of the specification is version **<x.y>**, and this is reflected in the IDL for the **CORBA** module by including the following pragma version (see Section 10.6.5.3, “The Version Pragma,” on page 10-45):

```
#pragma version CORBA <x.y>
```

as the first line immediately following the very first **CORBA** module introduction line, which in effect associates that version number with the **CORBA** entry in the **IR**. The version number in that version pragma line must be changed whenever any changes are made to any remotely accessible parts of the **CORBA** module in an officially released OMG standard.

3.15 Names and Scoping

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. So for example:

```
module M {
    typedef long Long;      // Error: Long clashes with keyword long
    typedef long TheThing;
    interface I {
        typedef long MyLong;
        myLong op1(                 // Error: inconsistent capitalization
            in TheThing thething;  // Error: TheThing clashes with thething
        );
    };
};
```

3.15.1 Qualified Names

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 Section 10.5.1, “Supporting Type Definitions,” on page 10-10).

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 3-1 on page 3-21) 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;      // Error: Ambiguous
    attribute A::string_t   Name;       // OK
    attribute B::string_t   City;       // OK
};
```

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

3.15.2 Scoping Rules and Name Resolution

Contents of an entire OMG IDL file, together with the contents of any files referenced by #include statements, forms a naming scope. Definitions that do not appear inside a scope are part of the global scope. There is only a single global scope, irrespective of the number of source files that form a specification.

The following kinds of definitions form scopes:

- module
- interface
- valuetype
- struct
- union
- operation
- exception

The scope for module, interface, valuetype, struct and exception begins immediately following its opening '{' and ends immediately preceding its closing '}'. The scope of an operation begins immediately following its '(' and ends immediately preceding its closing ')'. The scope of an union begins immediately following the '(' following the keyword **switch**, and ends immediately preceding its closing '}'. The appearance of the declaration of any of these kinds in any scope, subject to semantic validity of such declaration, opens a nested scope associated with that declaration.

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 with the same identifier within the same scope reopens the module and hence its scope, allowing additional definitions to be added to it.

The name of an interface, value type, struct, union, exception or a module may not be redefined within the immediate scope of the interface, value type, struct, union, exception, or the module. For example:

```
module M {
    typedef short M;    // Error: M is the name of the module
                        //        in the scope of which the typedef is.
    interface I {
        void i (in short j);    // Error: i clashes with the interface name I
    };
};
```

An identifier from a surrounding scope is introduced into a scope if it is used in that scope. An identifier is not introduced into a scope by merely being visible in that scope. The use of a scoped name introduces the identifier of the outermost scope of the scoped name. For example in:

```
module M {
    module Inner1 {
        typedef string S1;
    };

    module Inner2 {
        typedef string inner1;      // OK
    };
}
```

The declaration of **Inner2::inner1** is OK because the identifier **Inner1**, while visible in module **Inner2**, has not been introduced into module **Inner2** by actual use of it. On the other hand, if module **Inner2** were:

```
module Inner2{
    typedef Inner1::S1 S2;      // Inner1 introduced
    typedef string inner1;      // Error
    typedef string S1;          // OK
};
```

The definition of **inner1** is now an error because the identifier **Inner1** referring to the **module Inner1** has been introduced in the scope of module **Inner2** in the first line of the module declaration. Also, the declaration of **S1** in the last line is OK since the identifier **S1** was not introduced into the scope by the use of **Inner1::S1** in the first line.

Enumeration value names are introduced into the enclosing scope and then are treated like any other declaration in that scope. For example:

```
interface A {
    enum E { E1, E2, E3 };       // line 1

    enum BadE { E3, E4, E5 }; // Error: E3 is already introduced
                              // into the A scope in line 1 above
};

interface C {
    enum AnotherE { E1, E2, E3 };
};

interface D : C, A {
    union U switch ( E ) {
        case A::E1 : boolean b;// OK.
        case E2 : long l;        // Error: E2 is ambiguous (notwithstanding
                                 // the switch type specification!!)
    };
};
```

Type names defined in a scope are available for immediate use within that scope. In particular, see Section 3.10.2, “Constructed Types,” on page 3-35 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, while taking into consideration inheritance relationships among interfaces. For example:

```
module M {
    typedef long ArgType;
    typedef ArgType AType;          // line l1
    interface B {
        typedef string ArgType;     // line l3
        ArgType opb(in AType i);    // line l2
    };
};

module N {
    typedef char ArgType;           // line l4
    interface Y : M::B {
        void opy(in ArgType i);     // line l5
    };
};
```

The following scopes are searched for the declaration of **ArgType** used on **line l5**:

1. Scope of **N::Y** before the use of **ArgType**.

2. Scope of **N::Y**’s base interface **M::B**. (inherited scope)

3. Scope of **module N** before the definition of **N::Y**.

4. Global scope before the definition of **N**.

M::B::ArgType is found in **step 2** in **line l3**, and that is the definition that is used in **line l5**, hence **ArgType** in **line l5** is **string**. It should be noted that **ArgType** is not **char** in **line l5**. Now if **line l3** were removed from the definition of interface **M::B** then **ArgType** on **line l5** would be **char** from **line l4** which is found in **step 3**.

Following analogous search steps for the types used in the operation **M::B::opb** on **line l2**, the type of **AType** used on **line l2** is **long** from the **typedef** in **line l1** and the return type **ArgType** is **string** from **line l3.**

3.15.3 Special Scoping Rules for Type Names

Once a type has been *defined* anywhere within the scope of a module, interface or valuetype, it may not be redefined except within the scope of a nested module or interface. For example:

```
module M {
    typedef long ArgType;
    interface A {
        typedef string ArgType;    // OK, redefined in nested scope
        struct S {
            ArgType x;             // x is a string
        };
    };
    typedef double ArgType;        // Error: redefinition in same scope
};
```

Once a type identifier has been *used* anywhere within the scope of an interface or valuetype, it may not be redefined within the scope of that interface or valuetype. Use of type names within nested scopes created by structs, unions, and exceptions, as well as within the unnamed scope created by an operation parameter list, are for these purposes considered to occur within the scope of the enclosing interface or valuetype. For example:

```
module M {
    typedef long ArgType;
    const long I = 10;
    typedef short Y;

    interface A {
        struct S {
            ArgType x[I];    // x is a long[10], ArgType and I are used
            long y;      // Note: a new y is defined; the existing Y is not used
        };
        typedef string ArgType;    // Error: ArgType redefined after use
        enum I {I1, I2};           // Error: I redefined after use
        typedef short Y;           // OK because Y has not been used yet!
    };
};
```

Note that redefinition of a type after use in a module is OK as in the example:

```
typedef long ArgType;
module M {
    struct S {
        ArgType x;          // x is a long
    };
    typedef string ArgType;    // OK!
    struct T {
        ArgType y;          // Ugly but OK, y is a string
    };
};
```

3.16 Differences from C++

The OMG IDL grammar, while attempting to conform to the C++ syntax, is somewhat more restrictive. The 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, long** or **long long**.
- **char** cannot be qualified by **signed** or **unsigned** keywords.

3.17 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 marshaling, unmarshaling, 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.

Minor exception codes are of type **unsigned long** and consist of a 20-bit “Vendor Minor Codeset ID”(**VMCID**), which occupies the high order 20 bits, and the minor code which occupies the low order 12 bits.

Minor codes for the standard exceptions are prefaced by the **VMCID** assigned to OMG, defined as the unsigned long constant **CORBA::OMGVMCID**, which has the VMCID allocated to OMG occupying the high order 20 bits. The minor exception codes associated with the standard exceptions that are found in Table 3-13 on page 3-58 are or-ed with **OMGVMCID** to get the minor code value that is returned in the **ex_body** structure (see Section 3.17.1, “Standard Exception Definitions,” on page 3-52 and Section 3.17.2, “Standard Minor Exception Codes,” on page 3-58).

Within a vendor assigned space, the assignment of values to minor codes is left to the vendor. Vendors may request allocation of **VMCID**s by sending email to tag-*request@omg.org*.

The **VMCID** *0* and *\xfffff* are reserved for experimental use. The **VMCID OMGVMCID** (Section 3.17.1, “Standard Exception Definitions,” on page 3-52) and *1 through \xf* are reserved for OMG use.

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.

3.17.1 Standard Exception Definitions

The standard exceptions are defined below. Clients must be prepared to handle system exceptions that are not on this list, both because future versions of this specification may define additional standard exceptions, and because ORB implementations may raise non-standard system exceptions.

```
module CORBA {
    const unsigned long OMGVMCID = \x4f4d0000;

#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 marshaling
                                                    // 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
    exception TRANSACTION_REQUIRED
                                    ex_body;        // transaction required
    exception TRANSACTION_ROLLEDBACK
                                    ex_body;        // transaction rolled
                                                    // back
    exception INVALID_TRANSACTION
                                    ex_body;        // invalid transaction
    exception INV_POLICY            ex_body;        // invalid policy
    exception CODESET_INCOMPATIBLE
                                    ex_body         // incompatible code set
};
```

3.17.1.1 UNKNOWN

This exception is raised if an operation implementation throws a non-CORBA exception (such as an exception specific to the implementation's programming language), or if an operation raises a user exception that does not appear in the operation's raises expression. UNKNOWN is also raised if the server returns a system exception that is unknown to the client. (This can happen if the server uses a later version of CORBA than the client and new system exceptions have been added to the later version.)

3.17.1.2 BAD_PARAM

A parameter passed to a call is out of range or otherwise considered illegal. An ORB may raise this exception if null values or null pointers are passed to an operation (for language mappings where the concept of a null pointers or null values applies). BAD_PARAM can also be raised as a result of client generating requests with incorrect parameters using the DII.

3.17.1.3 NO_MEMORY

The ORB run time has run out of memory.

3.17.1.4 IMP_LIMIT

This exception indicates that an implementation limit was exceeded in the ORB run time. For example, an ORB may reach the maximum number of references it can hold simultaneously in an address space, the size of a parameter may have exceeded the allowed maximum, or an ORB may impose a maximum on the number of clients or servers that can run simultaneously.

3.17.1.5 COMM_FAILURE

This exception is raised if communication is lost while an operation is in progress, after the request was sent by the client, but before the reply from the server has been returned to the client.

3.17.1.6 INV_OBJREF

This exception indicates that an object reference is internally malformed. For example, the repository ID may have incorrect syntax or the addressing information may be invalid. This exception is raised by **ORB::string_to_object** if the passed string does not decode correctly.

An ORB may choose to detect calls via nil references (but is not obliged to do detect them). INV_OBJREF is used to indicate this.

3.17.1.7 NO_PERMISSION

An invocation failed because the caller has insufficient privileges.

3.17.1.8 INTERNAL

This exception indicates an internal failure in an ORB, for example, if an ORB has detected corruption of its internal data structures.

3.17.1.9 MARSHAL

A request or reply from the network is structurally invalid. This error typically indicates a bug in either the client-side or server-side run time. For example, if a reply from the server indicates that the message contains 1000 bytes, but the actual message is shorter or longer than 1000 bytes, the ORB raises this exception. MARSHAL can also be caused by using the DII or DSI incorrectly, for example, if the type of the actual parameters sent does not agree with IDL signature of an operation.

3.17.1.10 INITIALIZE

An ORB has encountered a failure during its initialization, such as failure to acquire networking resources or detecting a configuration error.

3.17.1.11 NO_IMPLEMENT

This exception indicates that even though the operation that was invoked exists (it has an IDL definition), no implementation for that operation exists. NO_IMPLEMENT can, for example, be raised by an ORB if a client asks for an object's type definition from the interface repository, but no interface repository is provided by the ORB.

3.17.1.12 BAD_TYPECODE

The ORB has encountered a malformed type code (for example, a type code with an invalid **TCKind** value).

3.17.1.13 BAD_OPERATION

This indicates that an object reference denotes an existing object, but that the object does not support the operation that was invoked.

3.17.1.14 NO_RESOURCES

The ORB has encountered some general resource limitation. For example, the run time may have reached the maximum permissible number of open connections.

3.17.1.15 NO_RESPONSE

This exception is raised if a client attempts to retrieve the result of a deferred synchronous call, but the response for the request is not yet available.

3.17.1.16 PERSIST_STORE

This exception indicates a persistent storage failure, for example, failure to establish a database connection or corruption of a database.

3.17.1.17 BAD_INV_ORDER

This exception indicates that the caller has invoked operations in the wrong order. For example, it can be raised by an ORB if an application makes an ORB-related call without having correctly initialized the ORB first.

3.17.1.18 TRANSIENT

TRANSIENT indicates that the ORB attempted to reach an object and failed. It is not an indication that an object does not exist. Instead, it simply means that no further determination of an object's status was possible because it could not be reached. This exception is raised if an attempt to establish a connection fails, for example, because the server or the implementation repository is down.

3.17.1.19 FREE_MEM

The ORB failed in an attempt to free dynamic memory, for example because of heap corruption or memory segments being locked.

3.17.1.20 INV_IDENT

This exception indicates that an IDL identifier is syntactically invalid. It may be raised if, for example, an identifier passed to the interface repository does not conform to IDL identifier syntax, or if an illegal operation name is used with the DII.

3.17.1.21 INV_FLAG

An invalid flag was passed to an operation (for example, when creating a DII request).

3.17.1.22 INTF_REPOS

An ORB raises this exception if it cannot reach the interface repository, or some other failure relating to the interface repository is detected.

3.17.1.23 BAD_CONTEXT

An operation may raise this exception if a client invokes the operation but the passed context does not contain the context values required by the operation.

3.17.1.24 OBJ_ADAPTER

This exception typically indicates an administrative mismatch. For example, a server may have made an attempt to register itself with an implementation repository under a name that is already in use, or is unknown to the repository. OBJ_ADAPTER is also raised by the POA to indicate problems with application-supplied servant managers.

3.17.1.25 DATA_CONVERSION

This exception is raised if an ORB cannot convert the representation of data as marshaled into its native representation or vice-versa. For example, DATA_CONVERSION can be raised if wide character codeset conversion fails, or if an ORB cannot convert floating point values between different representations.

3.17.1.26 OBJECT_NOT_EXIST

The OBJECT_NOT_EXIST 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.

3.17.1.27 TRANSACTION_REQUIRED

The TRANSACTION_REQUIRED exception indicates that the request carried a null transaction context, but an active transaction is required.

3.17.1.28 TRANSACTION_ROLLEDBACK

The TRANSACTION_ROLLEDBACK exception indicates that the transaction associated with the request has already been rolled back or marked to roll back. Thus, the requested operation either could not be performed or was not performed because further computation on behalf of the transaction would be fruitless.

3.17.1.29 INVALID_TRANSACTION

The INVALID_TRANSACTION indicates that the request carried an invalid transaction context. For example, this exception could be raised if an error occurred when trying to register a resource.

3.17.1.30 INV_POLICY

INV_POLICY is raised when an invocation cannot be made due to an incompatibility between Policy overrides that apply to the particular invocation.

3.17.1.31 CODESET_INCOMPATIBLE

This exception is raised whenever meaningful communication is not possible between client and server native code sets. See Section 13.7.2.6, “Code Set Negotiation,” on page 13-34.

3.17.2 Standard Minor Exception Codes

The following table specifies standard minor exception codes that have been assigned for the standard exceptions. The actual value that is to be found in the **minor** field of the **ex_body** structure is obtained by or-ing the values in this table with the **OMGVMCID** constant. For example “Missing local value implementation” for the exception NO_IMPLEMENT would be denoted by the **minor** value **\x4f4d0001**.

Table 3-13 Minor Exception Codes

SYSTEM EXCEPTION	MINOR CODE	EXPLANATION
BAD_PARAM	1	Failure to register, unregister or lookup value factory
	2	RID already defined in IFR
	3	Name already used in the context in IFR
	4	Target is not a valid container
	5	Name clash in inherited context
	6	Incorrect type for abstract interface
MARSHAL	1	Unable to locate value factory
NO_IMPLEMENT	1	Missing local value implementation
	2	Incompatible value implementation version
BAD_INV_ORDER	1	Dependency exists in IFR preventing destruction of this object
	2	Attempt to destroy indestructible objects in IFR
	3	Operation would deadlock
	4	ORB has shutdown
OBJECT_NOT_EXIST	1	Attempt to pass an unactivated (unregistered) value as an object reference

ORB Interface 4

The ORB Interface chapter has been updated based on the CORE changes from (ptc/98-09-04) and the Objects by Value documents (ptc/98-07-06) and (orbos/98-01-18). Changes from RTF 2.4 (ptc/99-03-01) and policy management related material from the Messaging specification (orbos/98-05-05) have also been incorporated.

Contents

This chapter contains the following sections.

4.1 Overview

This chapter introduces the operations that are implemented by the ORB core, and describes some basic ones, while providing reference to the description of the remaining operations that are described elsewhere. 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. The Object interface contains operations that are implemented by the ORB, and are accessed as implicit operations of the Object Reference. The ValueBase interface contains operations that are implemented by the ORB, and are accessed as implicit operations of the ValueBase Reference.

Because the operations in this section are implemented by the ORB itself, they are not in fact operations on objects, although they are described that way for the Object or ValueBase interface operations and the language binding will, for consistency, make them appear that way.

4.2 The ORB Operations

The ORB interface contains the operations that are available to both clients and servers. These operations do not depend on any specific object adapter or any specific object reference.

```
module CORBA {

    interface NVList;           // forward declaration
    interface OperationDef;     // forward declaration
    interface TypeCode;         // forward declaration

    typedef short PolicyErrorCode;
    // for the definition of consts see “PolicyErrorCode” on page 4-22

    interface Request;                  // forward declaration
    typedef sequence <Request> RequestSeq;

    native AbstractBase;

    exception PolicyError {PolicyErrorCode reason;};

    typedef string RepositoryId;
    typedef string Identifier;

    // StructMemberSeq defined in Chapter 10
    // UnionMemberSeq defined in Chapter 10
    // EnumMemberSeq defined in Chapter 10

    typedef unsigned short ServiceType;
    typedef unsigned long ServiceOption;
    typedef unsigned long ServiceDetailType;
```

```
    const ServiceType Security = 1;

    struct ServiceDetail {
        ServiceDetailType service_detail_type;
        sequence <octet> service_detail;
    };

    struct ServiceInformation {
        sequence <ServiceOption> service_options;
        sequence <ServiceDetail> service_details;
    };

    native ValueFactory;

    interface ORB {                                          // PIDL
#pragma version ORB 2.3

        typedef string ObjectId;
        typedef sequence <ObjectId> ObjectIdList;

        exception InvalidName {};

        string object_to_string (
            in Object           obj
        );

        Object string_to_object (
            in string           str
        );

        // Dynamic Invocation related operations

        void create_list (
            in long             count,
            out NVList          new_list
        );

        void create_operation_list (
            in OperationDef     oper,
            out NVList          new_list
        );

        void get_default_context (
            out Context          ctx
        );

        void send_multiple_requests_oneway(
            in RequestSeq       req
        );

        void send_multiple_requests_deferred(
```

```
        in RequestSeq       req
    );

    boolean poll_next_response();

    void get_next_response(
        out Request         req
    );

    // Service information operations

    boolean get_service_information (
        in ServiceType service_type,
        out ServiceInformation service_information
    );

    ObjectIdList list_initial_services ();

    // Initial reference operation

    Object resolve_initial_references (
        in ObjectId identifier
    ) raises (InvalidName);

    // Type code creation 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_wstring_tc (
    in unsigned long bound
);

TypeCode create_fixed_tc (
    in unsigned short digits,
    in short scale
);

TypeCode create_sequence_tc (
    in unsigned long bound,
    in TypeCode element type
);

TypeCode create_recursive_sequence_tc // deprecated
    in unsigned long bound,
    in unsigned long offset
);

TypeCode create_array_tc (
    in unsigned long length,
    in TypeCode element_type
);

TypeCode create_value_tc (
    in RepositoryId         id,
    in Identifier           name,
    in ValueModifier        type_modifier,
    in TypeCode             concrete_base,
    in ValueMembersSeq      members
);

TypeCode create_value_box_tc (
    in RepositoryId         id,
    in Identifier           name,
    in TypeCode             boxed_type
```

```
    );
    TypeCode create_native_tc (
        in RepositoryId         id,
        in Identifier           name
    );

    TypeCode create_recursive_tc(
        in RepositoryId         id
    );

    TypeCode create_abstract_interface_tc(
        in RepositoryId         id,
        in Identifier           name
    );

    // Thread related operations

    boolean work_pending( );

    void perform_work();

    void run();

    void shutdown(
        in boolean          wait_for_completion
    );

    void destroy();

    // Policy related operations

    Policy create_policy(
        in PolicyType       type,
        in any              val
    ) raises (PolicyError);

    // Dynamic Any related operations deprecated and removed
    // from primary list of ORB operations

    // Value factory operations

    ValueFactory register_value_factory(
        in RepositoryId id,
        in ValueFactory factory
    );

    void unregister_value_factory(in RepositoryId id);

    ValueFactory lookup_value_factory(in RepositoryId id);
};
```

```
};
```

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::**".

The operations **object_to_string** and **string_to_object** are described in "Converting Object References to Strings" on page 4-7.

For a description of the **create_list** and **create_operation_list** operations, see Section 7.4, "List Operations," on page 7-10. The **get_default_context** operation is described in the section Section 7.6.1, "get_default_context," on page 7-14. The **send_multiple_requests_oneway** and **send_multiple_requests_deferred** operations are described in the section Section 7.3.2, "send_multiple_requests," on page 7-9. The **poll_next_response** and **get_next_response** operations are described in the section Section 7.3.5, "get_next_response," on page 7-10.

The **list_intial_services** and **resolve_initial_references** operations are described in "Obtaining Initial Object References" on page 4-18.

The Type code creation operations with names of the form **create_<type>_tc** are described in Section 10.7.3, "Creating TypeCodes," on page 10-53.

The **work_pending**, **perform_work**, **shutdown**, **destroy** and **run** operations are described in "Thread-Related Operations" on page 4-33.

The **create_policy** operations is described in "Create_policy" on page 4-23.

The **register_value_factory**, **unregister_value_factory** and **lookup_value_factory** operations are described in Section 5.4.3, "Language Specific Value Factory Requirements," on page 5-9.

4.2.1 Converting Object References to Strings

4.2.1.1 object_to_string

```
string object_to_string (
    in Object           obj
);
```

4.2.1.2 string_to_object

```
Object string_to_object (
    in string           str
);
```

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.

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. For all conforming ORBs, if **obj** is a valid reference to an object, then **string_to_object(object_to_string(obj))** will return a valid reference to the same object, if the two operations are performed on the same ORB. For all conforming ORB's supporting IOP, this remains true even if the two operations are performed on different ORBs.

4.2.2 Getting Service Information

4.2.2.1 get_service_information

```
boolean get_service_information (
    in ServiceType service_type;
    out ServiceInformation service_information;
);
```

The **get_service_information** operation is used to obtain information about CORBA facilities and services that are supported by this ORB. The service type for which information is being requested is passed in as the in parameter **service_type**, the values defined by constants in the CORBA module. If service information is available for that type, that is returned in the out parameter **service_information**, and the operation returns the value TRUE. If no information for the requested services type is available, the operation returns FALSE (i.e., the service is not supported by this ORB).

4.3 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 define an interface for **Object**:

```
module CORBA {

    interface DomainManager;            // forward declaration
    typedef sequence <DomainManager> DomainManagersList;
```

```
interface Policy;                   // forward declaration
typedef sequence <Policy> PolicyList;
typedef unsigned long PolicyType;

interface Context;                  // forward declaration

typedef string Identifier;
interface Request;                  // forward declaration
interface NVList;                   // forward declaration
struct NamedValue{};                // an implicitly well known type
typedef unsigned long Flags;
interface InterfaceDef;

enum SetOverrideType {SET_OVERRIDE, ADD_OVERRIDE};

interface Object {                                          // PIDL

    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
    );

    void create_request (
        in Context              ctx
        in Identifier           operation,
        in NVList               arg_list,
        inout NamedValue        result,
        out Request             request,
        in Flags                req_flag
    );

    Policy get_policy (
        in PolicyType           policy_type
    );
```

```
        DomainManagersList get_domain_managers ();

        Object set_policy_overrides(
            in PolicyList       policies,
            in SetOverrideType  set_add
        );
    };
};
```

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 Section 7.2, “Request Operations,” on page 7-4.

Unless otherwise stated below, the operations in the IDL above do not require access to remote information.

4.3.1 Determining the Object Interface

4.3.1.1 get_interface

```
InterfaceDef get_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 the Interface Repository chapter for a definition of operations on the Interface Repository. The implementation of this operation may involve contacting the ORB that implements the target object.

4.3.2 Duplicating and Releasing Copies of Object References

4.3.2.1 duplicate

```
Object duplicate();
```

4.3.2.2 release

```
void release();
```

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.

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.

4.3.3 Nil Object References

4.3.3.1 is_nil

```
boolean is_nil();
```

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.

4.3.4 Equivalence Checking Operation

4.3.4.1 is_a

```
boolean is_a(
    in RepositoryId          logical_type_id
);
```

An operation is defined to facilitate maintaining type-safety for object references over the scope of an ORB.

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.

Determining whether an object's type is compatible with the **logical_type_id** may require contacting a remote ORB or interface repository. Such an attempt may fail at either the local or the remote end. If **is_a** cannot make a reliable determination of type compatibility due to failure, it raises an exception in the calling application code. This enables the application to distinguish among the **TRUE**, **FALSE**, and indeterminate cases.

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.

4.3.5 Probing for Object Non-Existence

4.3.5.1 non_existent

```
boolean non_existent ();
```

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; 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.

Probing for object non-existence may require contacting the ORB that implements the target object. Such an attempt may fail at either the local or the remote end. If non-existent cannot make a reliable determination of object existence due to failure, it raises an exception in the calling application code. This enables the application to distinguish among the true, false, and indeterminate cases.

4.3.6 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.

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.

4.3.6.1 Hashing Object Identifiers

hash

```
unsigned long hash(
    in unsigned long        maximum
);
```

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.

4.3.6.2 Equivalence Testing

is_equivalent

```
boolean is_equivalent(
    in Object                   other_object
);
```

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 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.

4.3.7 Getting Policy Associated with the Object

4.3.7.1 get_policy

The **get_policy** operation returns the policy object of the specified type (see “Policy Object” on page 4-20), which applies to this object. It returns the *effective* **Policy** for the object reference. The effective **Policy** is the one that would be used if a request were made. This **Policy** is determined first by obtaining the *effective override* for the **PolicyType** as returned by **get_client_policy**. The effective override is then compared with the **Policy** as specified in the IOR. The effective **Policy** is the

intersection of the values allowed by the effective override and the IOR-specified **Policy**. If the intersection is empty, the system exception INV_POLICY is raised. Otherwise, a **Policy** with a value legally within the intersection is returned as the effective **Policy**. The absence of a **Policy** value in the IOR implies that any legal value may be used. Invoking **non_existent** on an object reference prior to **get_policy** ensures the accuracy of the returned effective **Policy**. If **get_policy** is invoked prior to the object reference being bound, the returned effective **Policy** is implementation dependent. In that situation, a compliant implementation may do any of the following: raise the system exception BAD_INV_ORDER, return some value for that **PolicyType** which may be subject to change once a binding is performed, or attempt a binding and then return the effective **Policy**. Note that if the effective **Policy** may change from invocation to invocation due to transparent rebinding.

```
Policy get_policy (
    in PolicyType       policy_type
);
```

Parameter(s)

policy_type - The type of policy to be obtained.

Return Value

A **Policy** object of the type specified by the **policy_type** parameter.

Exception(s)

CORBA::INV_POLICY - raised when the value of policy type is not valid either because the specified type is not supported by this ORB or because a policy object of that type is not associated with this Object.

The implementation of this operation may involve remote invocation of an operation (e.g. **DomainManager::get_domain_policy** for some security policies) for some policy types.

4.3.8 Overriding Associated Policies on an Object Reference

4.3.8.1 set_policy_overrides

The **set_policy_overrides** operation returns a new object reference with the new policies associated with it. It takes two input parameters. The first parameter **policies** is a sequence of references to **Policy** objects. The second parameter **set_add** of type **SetOverrideType** indicates whether these policies should be added onto any other overrides that already exist (**ADD_OVERRIDE**) in the object reference, or they should be added to a clean override free object reference (**SET_OVERRIDE**). This operation associates the policies passed in the first parameter with a newly created object reference that it returns. Only certain policies that pertain to the invocation of an operation at the client end can be overridden using this operation. Attempts to override any other policy will result in the raising of the CORBA::NO_PERMISSION exception.

```
enum SetOverrideType {SET_OVERRIDE, ADD_OVERRIDE};

Object set_policy_overrides(
    in PolicyList           policies,
    in SetOverrideType      set_add
);
```

Parameter(s)

policies - a sequence of **Policy** objects that are to be associated with the new copy of the object reference returned by this operation

set_add - whether the association is in addition to (**ADD_OVERRIDE**) or as replacement of (**SET_OVERRIDE**) any existing overrides already associated with the object reference.

Return Value

A copy of the object reference with the overrides from **policies** associated with it in accordance with the value of **set_add**.

Exception(s)

CORBA::NO_PERMISSION - raised when an attempt is made to override any policy that cannot be overridden.

4.3.9 Getting the Domain Managers Associated with the Object

4.3.9.1 get_domain_managers

The get_domain_managers operation allows administration services (and applications) to retrieve the domain managers (see “Management of Policy Domains” on page 4-28), and hence the security and other policies applicable to individual objects that are members of the domain.

```
typedef sequence <DomainManager> DomainManagersList;

DomainManagersList get_domain_managers ();
```

Return Value

The list of immediately enclosing domain managers of this object. At least one domain manager is always returned in the list since by default each object is associated with at least one domain manager at creation.

The implementation of this operation may involve contacting the ORB that implements the target object.

4.4 ValueBase Operations

ValueBase serves a similar role for value types that **Object** serves for interfaces. Its mapping is language-specific and must be explicitly specified for each language.

Typically it is mapped to a concrete language type which serves as a base for all value types. Any operations that are required to be supported for all values are conceptually defined on **ValueBase**, although in reality their actual mapping depends upon the specifics of any particular language mapping.

Analogous to the definition of the **Object** interface for implicit operations of object references, the implicit operations of **ValueBase** are defined on a pseudo-**valuetype** as follows:

```
module CORBA {
    valuetype ValueBase{                                    PIDL
        ValueDef get_value_def();
    };
};
```

The **get_value_def()** operation returns a description of the value's definition as described in the interface repository (Section 10.5.24, "ValueDef," on page 10-34).

4.5 ORB and OA Initialization and Initial References

Before an application can enter the CORBA environment, it must first:

- Be initialized into the ORB and possibly the object adapter (POA) environments.
- Get references to ORB pseudo-object (for use in future ORB operations) and perhaps other objects (including the root POA or some Object Adapter objects).

The following operations are provided to initialize applications and obtain the appropriate object references:

- Operations providing access to the ORB. These operations reside in the CORBA module, but not in the ORB interface and are described in Section 4.6, "ORB Initialization," on page 4-16.
- Operations providing access to Object Adapters, Interface Repository, Naming Service, and other Object Services. These operations reside in the ORB interface and are described in Section 4.7, "Obtaining Initial Object References," on page 4-18.

4.6 ORB Initialization

When an application requires a CORBA environment it needs a mechanism to get the ORB pseudo-object reference and possibly an OA object reference (such as the root POA). This serves two purposes. First, it initializes an application into the ORB and OA environments. Second, it returns the ORB pseudo-object reference and the OA object reference to the application for use in future ORB and OA operations.

The ORB and OA 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 **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 other 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);
};
```

The identifier for the ORB will be a name of type **CORBA::ORBid**. All **ORBid** strings other than the empty string are allocated by ORB administrators and are not managed by the OMG. **ORBid** strings other than the empty string are intended to be used to uniquely identify each ORB used within the same address space in a multi-ORB application. These special **ORBid** strings are specific to each ORB implementation and the ORB administrator is responsible for ensuring that the names are unambiguous.

If an empty **ORBid** string is passed to **ORB_init**, then the **arg_list** arguments shall be examined to determine if they indicate an ORB reference that should be returned. This is achieved by searching the **arg_list** parameters for one preceded by "**-ORBid**" for example, "**-ORBid example_orb**" (the white space after the "**-ORBid**" tag is ignored) or "**-ORBidMyFavoriteORB**" (with no white space following the "**-ORBid**" tag). Alternatively, two sequential parameters with the first being the string "**-ORBid**" indicates that the second is to be treated as an **ORBid** parameter. If an empty string is passed and no **arg_list** parameters indicate the ORB reference to be returned, the default ORB for the environment will be returned.

Other parameters of significance to the ORB can also be identified in **arg_list**, 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 parameter format that ORB parameters may take. In general, parameters shall be formatted as either one single **arg_list** parameter:

–ORB<suffix><optional white space> <value>

or as two sequential **arg_list** parameters:

-ORB<suffix>

<value>

Regardless of whether an empty or non-empty **ORBid** string is passed to **ORB_init**, the **arg_list** arguments are examined to determine if any ORB parameters are given. If a non-empty **ORBid** string is passed to **ORB_init**, all **ORBid** parameters in the **arg_list** are ignored. All other **-ORB<suffix>** parameters in the **arg_list** may be of significance during the ORB initialization process.

Before **ORB_init** returns, it will remove from the **arg_list** parameter all strings that match the **-ORB<suffix>** pattern described above and that are recognized by that ORB implementation, along with any associated sequential parameter strings. If any strings in **arg_list** that match this pattern are not recognized by the ORB implementation, **ORB_init** will raise the BAD_PARAM system exception instead.

The **ORB_init** operation may be called any number of times and shall return the same ORB reference when the same **ORBid** string is passed, either explicitly as an argument to **ORB_init** or through the **arg_list**. All other **-ORB<suffix>** parameters in the **arg_list** may be considered on subsequent calls to **ORB_init**.

4.7 Obtaining Initial Object References

Applications require a portable means by which to obtain their initial object references. References are required for the root POA, POA Current, Interface Repository and various Object Services instances. (The POA is described in the Portable Object Adaptor chapter; the Interface Repository is described in the Interface Repository chapter; Object Services are described in *CORBAservices: Common Object Services Specification*.) 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 operations are provided in the ORB pseudo-object interface, providing facilities to list and resolve initial object references.

list_initial_services

```
typedef string ObjectId;
typedef sequence <ObjectId> ObjectIdList;
ObjectIdList list_initial_services ();
```

resolve_initial_references

```
exception InvalidName {};

Object resolve_initial_references (
    in ObjectId identifier
) raises (InvalidName);
```

The **resolve_initial_references** operation 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 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 **RootPOA**, **POACurrent**, **InterfaceRepository,** **NameService**, **TradingService**, **SecurityCurrent**, **TransactionCurrent,** and **DynAnyFactory.**

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 an 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. For example, for InterfaceRepository the object returned would be narrowed to **Repository** type.

In the future, specifications for Object Services (in *CORBAservices: Common Object Services Specification*) 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).

4.8 Current Object

ORB and CORBA services may wish to provide access to information (context) associated with the thread of execution in which they are running. This information is accessed in a structured manner using interfaces derived from the **Current** interface defined in the CORBA module.

Each ORB or CORBA service that needs its own context derives an interface from the CORBA module's **Current**. Users of the service can obtain an instance of the appropriate **Current** interface by invoking **ORB::resolve_initial_references**. For example the Security service obtains the **Current** relevant to it by invoking

ORB::resolve_initial_references(“SecurityCurrent”)

A CORBA service does not have to use this method of keeping context but may choose to do so.

```
module CORBA {
    // interface for the Current object
    interface Current {
    };
};
```

Operations on interfaces derived from **Current** access state associated with the thread in which they are invoked, not state associated with the thread from which the **Current** was obtained. This prevents one thread from manipulating another thread's state, and avoids the need to obtain and narrow a new **Current** in each method's thread context.

Current objects must not be exported to other processes, or externalized with **ORB::object_to_string**. If any attempt is made to do so, the offending operation will raise a MARSHAL system exception. **Current**s are per-process singleton objects, so no destroy operation is needed.

4.9 Policy Object

4.9.1 Definition of Policy Object

An ORB or CORBA service may choose to allow access to certain choices that affect its operation. This information is accessed in a structured manner using interfaces derived from the **Policy** interface defined in the CORBA module. A CORBA service does not have to use this method of accessing operating options, but may choose to do so. The *Security Service* in particular uses this technique for associating *Security Policy* with objects in the system.

```
module CORBA {
    typedef unsigned long PolicyType;

    // Basic IDL definition
    interface Policy {
        readonly attribute PolicyType policy_type;
        Policy copy();
        void destroy();
    };

    typedef sequence <Policy> PolicyList;
};
```

PolicyType defines the type of **Policy** object. In general the constant values that are allocated are defined in conjunction with the definition of the corresponding **Policy** object. The values of **PolicyTypes** for policies that are standardized by OMG are allocated by OMG. Additionally, vendors may reserve blocks of 4096 PolicyType values identified by a 20 bit *Vendor PolicyType Valueset ID* (**VPVID**) for their own use.

PolicyType which is an unsigned long consists of the 20-bit **VPVID** in the high order 20 bits, and the vendor assigned policy value in the low order 12 bits. The **VPVID**s *0 through \xf* are reserved for OMG. All values for the standard **PolicyTypes** are allocated within this range by OMG. Additionally, the **VPVID**s *\xfffff* is reserved for experimental use and **OMGVMCID** (Section 3.17.1, "Standard Exception Definitions," on page 3-52) is reserved for OMG use. These will not be allocated to anybody. Vendors can request allocation of **VPVID** by sending mail to tag-*request@omg.org*.

When a **VMCID** (Section 3.17, "Standard Exceptions," on page 3-51) is allocated to a vendor automatically the same value of **VPVID** is reserved for the vendor and vice versa. So once a vendor gets either a **VMCID** or a **VPVID** registered they can use that value for both their minor codes and their policy types.

4.9.1.1 Copy

```
Policy copy();
```

Return Value

This operation copies the policy object. The copy does not retain any relationships that the policy had with any domain, or object.

4.9.1.2 Destroy

```
void destroy();
```

This operation destroys the policy object. It is the responsibility of the policy object to determine whether it can be destroyed.

Exception(s)

CORBA::NO_PERMISSION - raised when the policy object determines that it cannot be destroyed.

4.9.1.3 Policy_type

```
readonly attribute policy_type
```

Return Value

This readonly attribute returns the constant value of type **PolicyType** that corresponds to the type of the **Policy** object.

4.9.2 Creation of Policy Objects

A generic ORB operation for creating new instances of Policy objects is provided as described in this section.

```
module CORBA {

    typedef short PolicyErrorCode;
    const PolicyErrorCode BAD_POLICY = 0;
    const PolicyErrorCode UNSUPPORTED_POLICY = 1;
    const PolicyErrorCode BAD_POLICY_TYPE = 2;
    const PolicyErrorCode BAD_POLICY_VALUE = 3;
    const PolicyErrorCode UNSUPPORTED_POLICY_VALUE = 4;

    exception PolicyError {PolicyErrorCode reason;};

    interface ORB {

        .....

        Policy create_policy(
            in PolicyType type,
            in any val
        ) raises(PolicyError);
    };
};
```

4.9.2.1 PolicyErrorCode

A request to create a **Policy** may be invalid for the following reasons:

BAD_POLICY - the requested **Policy** is not understood by the ORB.

UNSUPPORTED_POLICY - the requested **Policy** is understood to be valid by the ORB, but is not currently supported.

BAD_POLICY_TYPE - The type of the value requested for the **Policy** is not valid for that **PolicyType**.

BAD_POLICY_VALUE - The value requested for the **Policy** is of a valid type but is not within the valid range for that type.

UNSUPPORTED_POLICY_VALUE - The value requested for the **Policy** is of a valid type and within the valid range for that type, but this valid value is not currently supported.

4.9.2.2 PolicyError

```
exception PolicyError {PolicyErrorCode reason;};
```

PolicyError exception is raised to indicate problems with parameter values passed to the **ORB::create_policy** operation. Possible reasons are described above.

4.9.2.3 INV_POLICY

```
exception INV_POLICY
```

Due to an incompatibility between **Policy** overrides, the invocation cannot be made. This is a standard system exception that can be raised from any invocation.

4.9.2.4 Create_policy

The ORB operation **create_policy** can be invoked to create new instances of policy objects of a specific type with specified initial state. If **create_policy** fails to instantiate a new **Policy** object due to its inability to interpret the requested type and content of the policy, it raises the PolicyError exception with the appropriate reason as described in “PolicyErrorCode” on page 4-22.

```
Policy create_policy(
    in PolicyType type,
    in any val
) raises(PolicyError);
```

Parameter(s)

type - the **PolicyType** of the policy object to be created.

val - the value that will be used to set the initial state of the **Policy** object that is created.

ReturnValue

Reference to a newly created **Policy** object of type specified by the **type** parameter and initialized to a state specified by the **val** parameter.

Exception(s)

PolicyError - raised when the requested policy is not supported or a requested initial state for the policy is not supported.

When new policy types are added to CORBA or CORBA Services specification, it is expected that the IDL type and the valid values that can be passed to **create_policy** also be specified.

4.9.3 Usages of Policy Objects

Policy Objects are used in general to encapsulate information about a specific policy, with an interface derived from the policy interface. The type of the Policy object determines how the policy information contained within it is used. Usually a Policy object is associated with another object to associate the contained policy with that object.

Objects with which policy objects are typically associated are Domain Managers, POA, the execution environment, both the process/capsule/ORB instance and thread of execution (Current object) and object references. Only certain types of policy object can be meaningfully associated with each of these types of objects.

These relationships are documented in sections that pertain to these individual objects and their usages in various core facilities and object services. The use of Policy Objects with the POA are discussed in the *Portable Object Adaptor* chapter. The use of Policy objects in the context of the Security services, involving their association with Domain Managers as well as with the Execution Environment are discussed in *CORBAservices*, *Security Service* chapter.

In the following section the association of Policy objects with the Execution Environment is discussed. In “Management of Policy Domains” on page 4-28 the use of Policy objects in association with Domain Managers is discussed.

4.9.4 Policy Associated with the Execution Environment

Certain policies that pertain to services like security (e.g., QOP, Mechanism, invocation credentials etc.) are associated by default with the process/capsule(RM-ODP)/ORB instance (hereinafter referred to as “capsule”) when the application is instantiated together with the capsule. By default these policies are applicable whenever an invocation of an operation is attempted by any code executing in the said capsule. The Security service provides operations for modulating these policies on a per-execution thread basis using operations in the **Current** interface. Certain of these policies (e.g., invocation credentials, qop, mechanism etc.) which pertain to the invocation of an operation through a specific object reference can be further modulated at the client end, using the **set_policy_overrides** operation of the **Object** reference. For a description of this operation see “Overriding Associated Policies on an Object Reference” on page 4-14. It associates a specified set of policies with a newly created object reference that it returns.

The association of these overridden policies with the object reference is a purely local phenomenon. These associations are never passed on in any IOR or any other marshaled form of the object reference. the associations last until the object reference in the capsule is destroyed or the capsule in which it exists is destroyed.

The policies thus overridden in this new object reference and all subsequent duplicates of this new object reference apply to all invocations that are done through these object references. The overridden policies apply even when the default policy associated with **Current** is changed. It is always possible that the effective policy on an object reference at any given time will fail to be successfully applied, in which case the invocation attempt using that object reference will fail and return a CORBA::NO_PERMISSION exception. Only certain policies that pertain to the invocation of an operation at the client end can be overridden using this operation. These are listed in the Security specification. Attempts to override any other policy will result in the raising of the CORBA::NO_PERMISSION exception.

In general the policy of a specific type that will be used in an invocation through an specific object reference using a specific thread of execution is determined first by determining if that policy type has been overridden in that object reference. if so then the overridden policy is used. if not then if the policy has been set in the thread of execution then that policy is used. If not then the policy associated with the capsule is used. For policies that matter, the ORB ensures that there is a default policy object of each type that matters associated with each capsule (ORB instance). Hence, in a correctly implemented ORB there is no case when a required type policy is not available to use with an operation invocation.

4.9.5 *Specification of New Policy Objects*

When new **PolicyType**s are added to CORBA specifications, the following details must be defined. It must be clearly stated which particular uses of a new policy are legal and which are not:

- Specify the assigned **CORBA::PolicyType** and the policy's interface definition.
- If the **Policy** can be created through **CORBA::ORB::create_policy**, specify the allowable values for the any argument 'val' and how they correspond to the initial state/behavior of that **Policy** (such as initial values of attributes). For example, if a Policy has multiple attributes and operations, it is most likely that create_policy will receive some complex data for the implementation to initialize the state of the specific policy:

```
//IDL
struct MyPolicyRange {
      long low;
      long high;
};

const CORBA::PolicyType MY_POLICY_TYPE = 666;
interface MyPolicy : Policy {
      readonly attribute long low;
      readonly attribute long high;
};
```

  If this sample **MyPolicy** can be constructed via create_policy, the specification of **MyPolicy** will have a statement such as: “When instances of **MyPolicy** are created, a value of type **MyPolicyRang**e is passed to **CORBA::ORB::create_policy** and the resulting MyPolicy's attribute 'low' has the same value as the **MyPolicyRange** member 'low' and attribute 'high' has the same value as the **MyPolicyRange** member 'high'.

- If the **Policy** can be passed as an argument to **POA::create_POA**, specify the effects of the new policy on that **POA**. Specifically define incompatibilities (or inter-dependencies) with other **POA** policies, effects on the behavior of invocations on objects activated with the **POA**, and whether or not presence of the POA policy implies some **IOR** profile/component contents for object references created with

that **POA**. If the **POA** policy implies some addition/modification to the object reference it is marked as “client-exposed” and the exact details are specified including which profiles are affected and how the effects are represented.

- If the component which is used to carry this information. can be set within a client to tune the client's behavior, specify the policy's effects on the client specifically with respect to (a) establishment of connections and reconnections for an object reference; (b) effects on marshaling of requests; (c) effects on insertion of service contexts into requests; (d) effects upon receipt of service contexts in replies. In addition, incompatibilities (or inter-dependencies) with other client-side policies are stated. For policies that cause service contexts to be added to requests, the exact details of this addition are given.
- If the **Policy** can be used with **POA** creation to tune **IOR** contents and can also be specified (overridden) in the client, specify how to reconcile the policy's presence from both the client and server. It is strongly recommended to avoid this case! As an exercise in completeness, most **POA** policies can probably be extended to have some meaning in the client and vice versa, but this does not help make usable systems, it just makes them more complicated without adding really useful features. There are very few cases where a policy is really appropriate to specify in both places, and for these policies the interaction between the two must be described.
- Pure client-side policies are assumed to be immutable. This allows efficient processing by the runtime that can avoid re-evaluating the policy upon every invocation and instead can perform updates only when new overrides are set (or policies change due to rebind). If the newly specified policy is mutable, it must be clearly stated what happens if non-readonly attributes are set or operations are invoked that have side-effects.
- For certain policy types, override operations may be disallowed. If this is the case, the policy specification must clearly state what happens if such overrides are attempted.

4.9.6 Standard Policies

Table 4-1 below lists the standard policy types that are defined by various parts of CORBA and CORBA Services in this version of CORBA.

Table 4-1 Standard Policy Types

Policy Type	Policy Interface	Defined in Sect./Page	Uses create_ policy
SecClientInvocationAccess	SecurityAdmin::AccessPolicy	Security Service	No
SecTargetInvocationAccess	SecurityAdmin::AccessPolicy	Security Service	No
SecApplicationAccess	SecurityAdmin::AccessPolicy	Security Service	No
SecClientInvocationAudit	SecurityAdmin::AuditPolicy	Security Service	No
SecTargetInvocationAudit	SecurityAdmin::AuditPolicy	Security Service	No

Table 4-1 Standard Policy Types

Policy Type	Policy Interface	Defined in Sect./Page	Uses create_ policy
SecApplicationAudit	SecurityAdmin::AuditPolicy	Security Service	No
SecDelegation	SecurityAdmin::DelegationPolicy	Security Service	No
SecClientSecureInvocation	SecurityAdmin::SecureInvocationPolicy	Security Service	No
SecTargetSecureInvocation	SecurityAdmin::SecureInvocationPolicy	Security Service	No
SecNonRepudiation	NRService::NRPolicy	Security Service	No
SecConstruction	CORBA::SecConstruction	CORBA Core - ORB Interface chapter	No
SecMechanismPolicy	SecurityLevel2::MechanismPolicy	Security Service	Yes
SecInvocationCredentialsPolicy	SecurityLevel2::InvocationCredentialsPolicy	Security Service	Yes
SecFeaturesPolicy	SecurityLevel2::FeaturesPolicy	Security Service	Yes
SecQOPPolicy	SecurityLevel2::QOPPolicy	Security Service	Yes
THREAD_POLICY_ID	PortableServer::ThreadPolicy	CORBA Core - Portable Object Adapter chapter	Yes
LIFESPAN_POLICY_ID	PortableServer::LifespanPolicy	CORBA Core - Portable Object Adapter chapter Core Chapter 11	Yes
ID_UNIQUENESS_POLICY_ID	PortableServer::IdUniquenessPolicy	CORBA Core - Portable Object Adapter chapter Core Chapter 11	Yes
ID_ASSIGNMENT_POLICY_ID	PortableServer::IdAssignmentPolicy	CORBA Core - Portable Object Adapter chapter	Yes
IMPLICIT_ACTIVATION_POLICY_ID	PortableServer::ImplicitActivationPolicy	CORBA Core - Portable Object Adapter chapter	Yes
SERVENT_RETENTION_POLICY_ID	PortableServer::ServentRetentionPolicy	CORBA Core - Portable Object Adapter chapter	Yes
REQUEST_PROCESSING_POLICY_ID	PortableServer::RequestProcessingPolicy	CORBA Core - Portable Object Adapter chapter	Yes
BIDIRECTIONAL_POLICY_TYPE	BiDirPolicy::BidirectionalPolicy	CORBA Core - General Inter-ORB Protocol chapter	Yes
SecDelegationDirectivePolicy	SecurityLevel2::DelegtionDirectivePolicy	Security Service	Yes
SecEstablishTrustPolicy	SecurityLevel2::EstablishTrustPolicy	Security Service	Yes

4.10 Management of Policy Domains

4.10.1 Basic Concepts

This section describes how policies, such as security policies, are associated with objects that are managed by an ORB. The interfaces and operations that facilitate this aspect of management is described in this section together with the section describing **Policy** objects.

4.10.1.1 Policy Domain

A policy domain is a set of objects to which the policies associated with that domain apply. These objects are the domain members. The policies represent the rules and criteria that constrain activities of the objects which belong to the domain. On object reference creation, the ORB implicitly associates the object reference with one or more policy domains. Policy domains provide leverage for dealing with the problem of scale in policy management by allowing application of policy at a domain granularity rather than at an individual object instance granularity.

4.10.1.2 Policy Domain Manager

A policy domain includes a unique object, one per policy domain, called the domain manager, which has associated with it the policy objects for that domain. The domain manager also records the membership of the domain and provides the means to add and remove members. The domain manager is itself a member of a domain, possibly the domain it manages.

4.10.1.3 Policy Objects

A policy object encapsulates a policy of a specific type. The policy encapsulated in a policy object is associated with the domain by associating the policy object with the domain manager of the policy domain.

There may be several policies associated with a domain, with a policy object for each. There is at most one policy of each type associated with a policy domain. The policy objects are thus shared between objects in the domain, rather than being associated with individual objects. Consequently, if an object needs to have an individual policy, then it must be a singleton member of a domain.

4.10.1.4 Object Membership of Policy Domains

Since the only way to access objects is through object references, associating object references with policy domains, implicitly associates the domain policies with the object associated with the object reference. Care should be taken by the application that is creating object references using POA operations to ensure that object references

to the same object are not created by the server of that object with different domain associations. Henceforth whenever the concept of “object membership” is used, it actually means the membership of an object reference to the object in question.

An object can simultaneously be a member of more than one policy domain. In that case the object is governed by all policies of its enclosing domains. The reference model allows an object to be a member of multiple domains, which may overlap for the same type of policy (for example, be subject to overlapping access policies). This would require conflicts among policies defined by the multiple overlapping domains to be resolved. The specification does not include explicit support for such overlapping domains and, therefore, the use of policy composition rules required to resolve conflicts at policy enforcement time.

Policy domain managers and policy objects have two types of interfaces:

- The operational interfaces used when enforcing the policies. These are the interfaces used by the ORB during an object invocation. Some policy objects may also be used by applications, which enforce their own policies.

  The caller asks for the policy of a particular type (e.g., the delegation policy), and then uses the policy object returned to enforce the policy. The caller finding a policy and then enforcing it does not see the domain manager objects and the domain structure.

- The administrative interfaces used to set policies (e.g., specifying which events to audit or who can access objects of a specified type in this domain). The administrator sees and navigates the domain structure, so he is aware of the scope of what he is administering.

Note – This specification does not include any explicit interfaces for managing the policy domains themselves: creating and deleting them; moving objects between them; changing the domain structure and adding, changing, and removing policies applied to the domains.

4.10.1.5 Domains Association at Object Reference Creation

When a new object reference is created, the ORB implicitly associates the object reference (and hence the object that it is associated with) with the following elements forming its environment:

- One or more *Policy Domains*, defining all the policies to which the object associated with the object reference is subject.
- The *Technology Domains*, characterizing the particular variants of mechanisms (including security) available in the ORB.

The ORB will establish these associations when one of the object reference creation operations of the POA is called. Some or all of these associations may subsequently be explicitly referenced and modified by administrative or application activity, which might be specifically security-related but could also occur as a side-effect of some other activity, such as moving an object to another host machine.

In some cases, when a new object reference is created, it needs to be associated with a new domain. Within a given domain a construction policy can be associated with a specific object type thus causing a new domain (i.e., a domain manager object) to be created whenever an object reference of that type is created and the newly created object reference associated with the new domain manager. This construction policy is enforced at the same time as the domain membership (i.e., by the POA when it creates an object reference).

4.10.1.6 Implementor's View of Object Creation

For policy domains, the construction policy of the application or factory creating the object proceeds as follows. The application (which may be a generic factory) calls one of the object reference creation operations of the POA to create the new object reference. The ORB obtains the construction policy associated with the creating object, or the default domain absent a creating object.

By default, the new object reference that is created is made a member of the domain to which the parent belongs. Non-object applications on the client side are associated with a default, per-ORB instance policy domain by the ORB.

Each domain manager has a construction policy associated with it, which controls whether, in addition to creating the specified new object reference, a new domain manager is created with it. This object provides a single operation **make_domain_manager** which can be invoked with the **constr_policy** parameter set to TRUE to indicate to the ORB that new object references of the specified type are to be associated their own separate domains. Once such a construction policy is set, it can be reversed by invoking **make_domain_manager** again with the **constr_policy** parameter set to FALSE.

When creating an object reference of the type specified in the **make_domain_manager** call with **constr_policy** set to TRUE, the ORB must also create a new domain for the newly created object reference. If a new domain is needed, the ORB creates both the requested object reference and a domain manager object. A reference to this domain manager can be found by calling **get_domain_managers** on the newly created object reference.

While the management interface to the construction policy object is standardized, the interface from the ORB to the policy object is assumed to be a private one, which may be optimized for different implementations.

If a new domain is created, the policies initially applicable to it are the policies of the enclosing domain. The ORB will always arrange to provide a default enclosing domain with default ORB policies associated with it, in those cases where there would be no such domain as in the case of a non-object client invoking object creation operations.

The calling application, or an administrative application later, can change the domains to which this object belongs, using the domain management interfaces, which will be defined in the future.

Since the ORB has control only over domain associations with object references, it is the responsibility of the creator of new object to ensure that the object references that are created to the new object are associated meaningfully with domains.

4.10.2 Domain Management Operations

This section defines the interfaces and operations needed to find domain managers and find the policies associated with these. However, it does not include operations to manage domain membership, structure of domains, or to manage which policies are associated with domains.

This section also includes the interface to the construction policy object, as that is relevant to domains. The basic definitions of the interfaces and operations related to these are part of the CORBA module, since other definitions in the CORBA module depend on these.

```
module CORBA {
    interface DomainManager {
        Policy get_domain_policy (
            in PolicyType policy_type
        );
    };

    const PolicyType SecConstruction = 11;

    interface ConstructionPolicy: Policy{
        void make_domain_manager(
            in CORBA::InterfaceDef object_type,
            in boolean constr_policy
        );
    };

    typedef sequence <DomainManager> DomainManagersList;
};
```

4.10.2.1 Domain Manager

The domain manager provides mechanisms for:

- Establishing and navigating relationships to superior and subordinate domains.
- Creating and accessing policies.

There should be no unnecessary constraints on the ordering of these activities, for example, it must be possible to add new policies to a domain with a preexisting membership. In this case, some means of determining the members that do not conform to a policy that may be imposed is required. It should be noted that interfaces for adding new policies to domains or for changing domain memberships have not currently been standardized.

All domain managers provide the **get_domain_policy** operation. By virtue of being an object, the Domain Managers also have the **get_policy** and **get_domain_managers** operations, which is available on all objects (see “Getting Policy Associated with the Object” on page 4-13 and “Getting the Domain Managers Associated with the Object” on page 4-15).

CORBA::DomainManager::get_domain_policy

This returns the policy of the specified type for objects in this domain.

```
Policy get_domain_policy (
    in PolicyType policy_type
);
```

Parameter(s)

policy_type - The type of policy for objects in the domain which the application wants to administer. For security, the possible policy types are described in *CORBAservices: Common Object Services Specification*, Security chapter, Security Policies Introduction section.

Return Value

A reference to the policy object for the specified type of policy in this domain.

Exception(s)

CORBA::INV_POLICY - raised when the value of policy type is not valid either because the specified type is not supported by this ORB or because a policy object of that type is not associated with this Object.

4.10.2.2 Construction Policy

The construction policy object allows callers to specify that when instances of a particular object reference are created, they should be automatically assigned membership in a newly created domain at creation time.

CORBA::ConstructionPolicy::make_domain_manager

This operation enables the invoker to set the construction policy that is to be in effect in the domain with which this **ConstructionPolicy** object is associated. Construction Policy can either be set so that when an object reference of the type specified by the input parameter is created, a new domain manager will be created and the newly created object reference will respond to **get_domain_managers** by returning a reference to this domain manager. Alternatively the policy can be set to associate the newly created object reference with the domain associated with the creator. This policy is implemented by the ORB during execution of any one of the object reference creation operations of the POA, and results in the construction of the application-specified object reference and a Domain Manager object if so dictated by the policy in effect at the time of the creation of the object reference.

```
void make_domain_manager (
    in InterfaceDef object_type,
    in boolean constr_policy
);
```

Parameter(s)

object_type - The type of the object references for which Domain Managers will be created. If this is nil, the policy applies to all object references in the domain.

constr_policy - If TRUE the construction policy is set to create a new domain manager associated with the newly created object reference of this type in this domain. If FALSE construction policy is set to associate the newly created object references with the domain of the creator or a default domain as described above.

4.11 Thread-Related Operations

To support single-threaded ORBs, as well as multi-threaded ORBs that run multi-thread-unaware code, several operations are included in the ORB interface. These operations can be used by single-threaded and multi-threaded applications. An application that is a pure ORB client would not need to use these operations. Both the **ORB::run** and **ORB::shutdown** are useful in fully multi-threaded programs.

Note – These operations are defined on the ORB rather than on an object adapter to allow the main thread to be used for all kinds of asynchronous processing by the ORB. Defining these operations on the ORB also allows the ORB to support multiple object adapters, without requiring the application main to know about all the object adapters. The interface between the ORB and an object adapter is not standardized.

4.11.1 work_pending

```
boolean work_pending( );
```

This operation returns an indication of whether the ORB needs the main thread to perform some work.

A result of TRUE indicates that the ORB needs the main thread to perform some work and a result of FALSE indicates that the ORB does not need the main thread.

4.11.2 perform_work

```
void perform_work();
```

If called by the main thread, this operation performs an implementation-defined unit of work; otherwise, it does nothing.

It is platform-specific how the application and ORB arrange to use compatible threading primitives.

The **work_pending()** and **perform_work()** operations can be used to write a simple polling loop that multiplexes the main thread among the ORB and other activities. Such a loop would most likely be needed in a single-threaded server. A multi-threaded server would need a polling loop only if there were both ORB and other code that required use of the main thread.

Here is an example of such a polling loop:

```
// C++
for (;;) {
   if (orb->work_pending()) {
      orb->perform_work();
   };
   // do other things
   // sleep?
};
```

Once the ORB has shutdown, **work_pending** and **perform_work** will raise the BAD_INV_ORDER exception with minor code 4. An application can detect this exception to determine when to terminate a polling loop.

4.11.3 run

void run();

This operation provides execution resources to the ORB so that it can perform its internal functions. Single threaded ORB implementations, and some multi-threaded ORB implementations, need the use of the main thread in order to function properly. For maximum portability, an application should call either **run** or **perform_work** on its main thread. **run** may be called by multiple threads simultaneously.

This operation will block until the ORB has completed the shutdown process, initiated when some thread calls **shutdown**.

4.11.4 shutdown

```
void shutdown(
    in boolean          wait_for_completion
);
```

This operation instructs the ORB to shut down, that is, to stop processing in preparation for destruction.

Shutting down the ORB causes all object adapters to be destroyed, since they cannot exist in the absence of an ORB. Shut down is complete when all ORB processing (including request processing and object deactivation or other operations associated with object adapters) has completed and the object adapters have been destroyed. In the case of the **POA**, this means that all object etherealizations have finished and root **POA** has been destroyed (implying that all descendent **POA**s have also been destroyed).

If the **wait_for_completion** parameter is **TRUE**, this operation blocks until the shut down is complete. If an application does this in a thread that is currently servicing an invocation, the BAD_INV_ORDER system exception will be raised with the OMG minor code 3, since blocking would result in a deadlock.

If the **wait_for_completion** parameter is **FALSE**, then **shutdown** may not have completed upon return. An ORB implementation may require the application to call (or have a pending call to) **run** or **perform_work** after **shutdown** has been called with its parameter set to **FALSE**, in order to complete the shutdown process.

While the ORB is in the process of shutting down, the ORB operates as normal, servicing incoming and outgoing requests until all requests have been completed. An implementation may impose a time limit for requests to complete while a **shutdown** is pending.

Once an ORB has shutdown, only object reference management operations(**duplicate**, **release** and **is_nil**) may be invoked on the ORB or any object reference obtained from it. An application may also invoke the destroy operation on the ORB itself. Invoking any other operation will raise the BAD_INV_ORDER system exception with the OMG minor code 4.

4.11.5 destroy

void destroy();

This operation destroys the ORB so that its resources can be reclaimed by the application. Any operation invoked on a destroyed ORB reference will raise the OBJECT_NOT_EXIST exception. Once an ORB has been destroyed, another call to **ORB_init** with the same **ORBid** will return a reference to a newly constructed ORB.

If **destroy** is called on an ORB that has not been shut down, it will start the shut down process and block until the ORB has shut down before it destroys the ORB. If an application calls **destroy** in a thread that is currently servicing an invocation, the BAD_INV_ORDER system exception will be raised with the OMG minor code 3, since blocking would result in a deadlock.

For maximum portability and to avoid resource leaks, an application should always call **shutdown** and **destroy** on all ORB instances before exiting.

Value Type Semantics 5

The Value Type Semantics chapter includes information from the Objects by Value documents and describes the semantics of value types. Details specific to particular aspects of the ORB may be found in other chapters.

Contents

This chapter contains the following sections.

5.1 Overview

Objects, more specifically, interface types that objects support, are defined by an IDL interface, allowing arbitrary implementations. There is great value, which is described in great detail elsewhere, in having a distributed object system that places almost no constraints on implementations.

However there are many occasions in which it is desirable to be able to pass an object by value, rather than by reference. This may be particularly useful when an object’s primary “purpose” is to encapsulate data, or an application explicitly wishes to make a “copy” of an object.

The semantics of passing an object by value are similar to that of standard programming languages. The receiving side of a parameter passed by value receives a description of the “state” of the object. It then instantiates a new instance with that state but having a separate identity from that of the sending side. Once the parameter passing operation is complete, no relationship is assumed to exist between the two instances.

Because it is necessary for the receiving side to instantiate an instance, it must necessarily know something about the object’s state and implementation.

Valuetype(s) provide semantics that bridge between CORBA structs and CORBA interfaces:

- They support description of complex state (i.e., arbitrary graphs, with recursion and cycles)
- Their instances are always local to the context in which they are used (because they are always copied when passed as a parameter to a remote call)
- They support both public and private (to the implementation) data members.
- They can be used to specify the state of an object implementation (i.e., they can support an interface).
- They support single inheritance (of **valuetype**) and can support an **interface**.
- They may be also be **abstract**.

5.2 Architecture

The basic notion is relatively simple. A **value type** is, in some sense, half way between a “regular” IDL interface type and a struct. The use of a value type is a signal from the designer that some additional properties (state) and implementation details be specified beyond that of an interface type. Specification of this information puts some additional constraints on the implementation choices beyond that of interface types. This is reflected in both the semantics specified herein, and in the language mappings.

An essential property of value types is that their implementations are always local. That is, the explicit use of value type in a concrete programming language is always guaranteed to use a local implementation, and will not require a remote call. They have no identity (their value is their identity) and they are not “registered” with the ORB.

There are two kinds of value types, concrete (or stateful) value types, and abstract (stateless) ones. As explained below the essential characteristics of both are the same. The differences between them result from the differences in the way they are mapped in the language mappings. In this specification the semantics of value types apply to both kinds, unless specifically stated otherwise.

Concrete (stateful) values add to the expressive power of (IDL) structs by supporting:

- single derivation (from other value types)
- support of multiple interfaces
- arbitrary recursive value type definitions, with sharing semantics providing the ability to define lists, trees, lattices and more generally arbitrary graphs using value types.

- null value semantics

When an instance of such a type is passed as a parameter, the sending context marshals the state (data) and passes it to the receiving context. The receiving context instantiates a new instance using the information in the GIOP request and unmarshals the state. It is assumed that the receiving context has available to it an implementation that is consistent with the sender's (i.e., only needs the state information), or that it can somehow download a usable implementation. Provision is made in the on-the-wire format to support the carrying of an optional call back object (**CodeBase**) to the sending context which enables such downloading when it is appropriate.

It should be noted that it is possible to define a concrete value type with an empty state as a degenerate case.

5.2.1 Abstract Values

Value types may also be abstract. They are called abstract because an abstract value type may not be instantiated. Only concrete types derived from them may be actually instantiated and implemented. Their implementation, of course, is still local. However, because no state information may be specified (only local operations are allowed), abstract value types are not subject to the single inheritance restrictions placed upon concrete value types. Essentially they are a bundle of operation signatures with a purely local implementation. This distinction is made clear in the language mappings for abstract values.

Note that a concrete value type with an empty state is not an abstract value type. They are considered to be stateful, may be instantiated, marshaled and passed as actual parameters. Consider them to be a degenerate case of stateful values.

5.2.2 Operations

Operations defined on a value type specify signatures whose implementation can only be local. Because these operations are local, they must be directly implemented by a body of code in the language mapping (no proxy or indirection is involved).

The language mappings of such operations require that instances of value types passed into and returned by such local methods are passed by reference (programming language reference semantics, not CORBA object reference semantics) and that a copy is not made. Note, such a (local) invocation is not a CORBA invocation. Hence it is not mediated by the ORB, although the API to be used is specified in the language mapping.

The (copy) semantics for instances of value type are only guaranteed when instances of these value types are passed as a parameter to an operation defined on a CORBA interface, and hence mediated by the ORB. If an instance of a value type is passed as a parameter to a method of another value type in an invocation, then this call is a "normal" programming language call. In this case both of the instances are local programming language constructs. No CORBA style copy semantics are used and programming language reference semantics apply.

Operations on the value type are supported in order to guarantee the portability of the client code for these value types. They have no representation on the wire and hence no impact on interoperability.

5.2.3 Value Type vs. Interfaces

By default value types are not CORBA Objects. In particular instances of value types do not inherit from CORBA::Object and do not support normal object reference semantics. However it is always possible to explicitly declare that a given value type supports an interface type. In this case instances of the type may support CORBA object reference semantics (if they are registered with the ORB using an object adapter).

5.2.4 Parameter Passing

This section describes semantics when a value instance is passed as parameter in a CORBA invocation. It does not deal with the case of calling another non-CORBA (i.e., local) programming method which happens to have a parameter of the same type.

5.2.4.1 Value vs. Reference Semantics

Determination of whether a parameter is to be passed by value or reference is made by examining the parameter's formal type (i.e., the signature of the operation it is being passed to). If it is a value type then it is passed by value. If it is an ordinary interface then it is passed by reference (the case today for all CORBA objects). This rule is simple and consistent with the handling of the same situation in recursive state definitions or in structs.

In the case of abstract interfaces, the determination is made at runtime. See Section 6.2, "Semantics of Abstract Interfaces," on page 6-1 for a description of the rules.

5.2.4.2 Sharing Semantics

In order to be expressive enough to describe arbitrary graphs, lattice, trees etc., value types support sharing and null semantics. Instances of a value type can be shared by others across or within other instances. They can also be null. This is unlike other IDL data types such as structs, unions, and sequences which can never be shared. The sharing of values within and between the parameters to an operation, is preserved across an invocation (i.e., the graph which is reconstructed in the receiving context is structurally isomorphic to the sending context's).

5.2.4.3 Identity Semantics

When an instance of the value type is passed as a parameter, an independent copy of the instance is instantiated in the receiving context. That copy is a separate independent entity and there is no explicit or implicit sharing of state.

5.2.4.4 Any parameter type

When an instance of a value type is passed to an **any**, as with all cases of passing instances to an **any**, it is the responsibility of the implementer to insert and extract the value according to the language mapping specification.

5.2.5 Substitutability Issues

The substitutability requirements for CORBA require the definition of what happens when an instance of a derived value type is passed as a parameter that is declared to be a base value type or an instance of a value type that supports an interface is passed as a parameter that is declared as the interface type.

There are two cases to consider: the parameter type is an interface, and the parameter type is a value type.

5.2.5.1 Value instance -> Interface type

Assume that we have an instance of a value type that supports an interface type. We have an IDL operation whose signature contains a parameter whose formal type is the interface. The following rule applies to this situation:

- If the value type instance (in the sending context) has not been registered with ORB, then an OBJECT_NOT_EXIST exception with an identified minor code (see Section 3.17.2, “Standard Minor Exception Codes,” on page 3-58) is raised. Otherwise the instance’s object reference is used and it is passed as normal.

5.2.5.2 Value instance -> Value type

In this case the receiving context is expecting to receive a value type. If the receiving context currently has the appropriate implementation class then there is no problem.

If the receiving context does not currently hold an implementation with which to reconstruct the original type then the following algorithm is used to find such an implementation:

1. ***Load*** - Attempt to load (locally in C/C++, possibly remotely in Java and other “portable” languages) the real type of the object (with its methods). If this succeeds, OK

2. ***Truncate*** - Truncate the type of the object to the base type (if specified as **truncatable** in the IDL). Truncation can never lead to faulty programs because, from a structural point view base types structurally subsume a derived type and an object created in the receiving context bears no relationship with the original one. However, it might be semantically puzzling, as the derived type may completely re-interpret the meaning of the state of the base. For that reason a derived value needs to indicate if it is safe to truncate to its immediate non-abstract parent.

3. ***Raise Exception*** - If none of these work or are possible, then raise the NO_IMPLEMENT exception.

Truncatability is a transitive property.

Example

```
valuetype EmployeeRecord {   // note this is not a CORBA::Object
    // state definition
    private string name;
    private string email;
    private string SSN;
    // initializer
    factory init(in string name, in string SSN);
};

valuetype ManagerRecord: truncatable EmployeeRecord {
    // state definition
    private sequence<EmployeeRecord> direct_reports;
};
```

5.2.6 Widening/Narrowing

As has been described above, value type instances may be widened/narrowed to other value types. Each language mapping is responsible for specifying how these operations are made available to the programmer.

Narrowing from an interface type instance to a value type instance is not allowed. If the interface designer wants to allow the receiving context to create a local implementation of the value type (i.e., a value representing the interface) an operation which returns the appropriate value type may be defined.

5.2.7 Value Base Type

All value types have a conventional base type called **ValueBase.** This is a type which fulfills a role that is similar to that played by **Object**. Conceptually it supports the common operations available on all value types. See Section 4.4, “ValueBase Operations,” on page 4-16 for a description of those operations. In each language mapping **ValueBase** will be mapped to an appropriate base type that supports the marshaling/unmarshaling protocol as well as the model for custom marshaling.

The mapping for other operations which all value types must support, such as getting meta information about the type, may be found in the specifics for each language mapping.

5.2.8 Life Cycle issues

Value type instances are always local to their creating context. For example, in a given language mapping an instance of a value type is always created as a local “language” object with no POA semantics attached to it initially.

When passed using a CORBA invocation, a copy of the value is made in the receiving context and that copy starts its life as a local programming language entity with no POA semantics attached to it.

If a value type supports an ordinary interface type, its instances may also be passed by reference when the formal parameter type is an interface type (see Section 5.2.4, “Parameter Passing,” on page 5-4). In this case they behave like ordinary object implementations and must be associated with a POA policy and also be registered with the ORB (e.g., POA::activate_object() before they can be passed by reference. Not registering the value as a CORBA object and/or not associating an appropriate policy with it results in an exception when trying to use it as a remote object, the “normal” behavior. The exception raised shall be OBJECT_NOT_EXIST with an identified minor code (see Section 3.17.2, “Standard Minor Exception Codes,” on page 3-58).

5.2.8.1 Creation and Factories

When an instance of a value type is received by the ORB, it must be unmarshaled and an appropriate factory for its actual type found in order for the new instance to be created. The type is encoded by the RepositoryID which is passed over the wire as part of an invocation. The mapping between the type (as specified by the RepositoryID) and the factory is language specific. In certain languages it may be possible to specify default policies that are used to find the factory, without requiring that specific routines be called. In others the runtime and/or generated code may have to explicitly specify the mapping on a per type basis. In others a combination may be used. In any event the ORB implementation is responsible for maintaining this mapping See Section 5.4.3, “Language Specific Value Factory Requirements,” on page 5-9 for more details on the requirements for each language mapping.

5.2.9 Security Considerations

The addition of value types has few impacts on the CORBA security model. In essence, the security implications in defining and using value types are similar to those involved with the use of IDL structs. Instances of value types are mapped to local, concrete programming language constructs. Except for providing the marshaling mechanisms, the ORB is not directly involved with accessing value type implementations. This specification is mostly about two things: how value types manifest themselves as concrete programming language constructs and how they are transmitted.

To see this consider how value types are actually used. The IDL definition of a value type in conjunction with a programming language mapping is used to generate the concrete programming language definitions for that type.

Let us consider its life cycle. In order to use it, the programmer uses the mechanisms in the programming language to instantiate an instance. This is instance is a local programming language construct. It is not “registered” with the ORB, object adapter, etc. The programmer may manipulate this programming construct just like any other programming language construct. So far there are no security implications. As long as no ORB-mediated invocations are made, the programmer may manipulate the

construct. Note, this includes making “local,” non ORB-mediated calls to any locally implemented operations. Any assignments to the construct are the responsibility of the programmer and have no special security implications.

Things get interesting when the program attempts to pass one of these constructs through an orb-mediated invocation (i.e., calls a stub which uses it as a parameter type, or uses the DII). There are two cases to consider:

5.2.9.1 Value as Value

The formal type of the parameter is a value. This case is no different from using any other kind of a value (long, string, struct) in a CORBA invocation, with respect to security. The value (data) is marshaled and delivered to the receiving context. On the receiving context, the knowledge of the type is used (at least implicitly) to find the factory to create the correct local programming language construct. The data is then unmarshaled to fill in the newly created construct. This is similar to using other values (longs, strings, structs) except that the knowledge of the factory is not “built-in” to the ORB’s skeleton/DSI engine.

5.2.9.2 Value as Object Reference

The formal type of the parameter is an interface type which is supported by a value. The program must have “registered” the value with an object adapter and is really using the returned object reference (see for the specific rules.) Thus this case “reduces” to a regular CORBA invocation, using a regular object reference. An IOR is passed to the receiving context. All the “normal” security considerations apply. From the point of view of the receiving context, the IOR is a “normal” object reference. No “special” rules, with respect to security or otherwise, apply to it. The fact that it is ultimately a reference to an implementation that was created from instantiating and registering an value type implementation is not relevant.

In both of these cases, security considerations are involved with the decision to allow the ORB-mediated invocation to proceed. The fact that a value type is involved is not material.

5.3 Standard Value Box Definitions

For some CORBA-defined types for which preservation of sharing and transmission of nulls are likely to be important, the following value box type definitions are added to the CORBA module:

```
module CORBA {
    valuetype StringValue string;
    valuetype WStringValue wstring;
};
```

5.4 Language Mappings

5.4.1 General Requirements

A concrete value is mapped to a concrete usable “class” construct in each programming language, plus possibly some helper classes where appropriate. In Java, C++, and Smalltalk this is a real concrete class. In C it is a struct.

An abstract value is mapped to some sort of an abstract construct--an interface in Java, and an abstract class with pure virtual function members in C++.

Tools that implement the language mapping are free to “extend” the implementation classes with “extra” data members and methods. When an instance of such a class is used as a parameter, only the portions that correspond directly to the IDL declaration, are marshaled and delivered to the receiving context. This allows freedom of implementations while preserving the notion of contract and type safety in IDL.

5.4.2 Language Specific Marshaling

Each language mapping defines an appropriate marshaling/unmarshaling API and the entry point for custom marshaling/unmarshaling.

5.4.3 Language Specific Value Factory Requirements

Each language mapping specifies the algorithm and means by which RepositoryIDs are used to find the appropriate factory for an instance of a value type so that it may be created as it is unmarshaled “off the wire.”

It is desirable, where it makes sense, to specify a “default” policy for automatically using RepositoryIDs that are in common formats to find the appropriate factory. Such a policy can be thought of as an implicit registration.

Each language mapping specifies how and when the registration occurs, both explicit and implicit. The registration must occur before an attempt is made to unmarshal an instance of a value type. If the ORB is unable to locate and use the appropriate factory, then a MARSHAL exception with an identified minor code (see Section 3.17.2, “Standard Minor Exception Codes,” on page 3-58) is raised.

Because the type of the factory is programming language specific and each programming language platform has different policies, the factory type is specified as native. It is the responsibility of each language mapping to specify the actual programming language type of the factory.

```
module CORBA {

    // IDL
    native ValueFactory;
};
```

5.4.4 Value Method Implementation

The mapped class must support method bodies (i.e., code) that implement the required IDL operations. The means by which this association is accomplished is a language mapping “detail” in much the same way that an IDL compiler is.

5.5 Custom Marshaling

Value types can override the default marshaling/unmarshaling model and provide their own way to encode/decode their state. Custom marshaling is intended to be used to facilitate integration of existing “class libraries” and other legacy systems. It is explicitly not intended to be a standard practice, nor used in other OMG specifications to avoid “standard ORB” marshaling.

The fact that a value type has some custom marshaling code is declared explicitly in the IDL. This explicit declaration has two goals:

- ***type safety*** - stub and skeleton can know statically that a given type is custom marshaled and can then do sanity check on what is coming over the wire.
- ***efficiency*** - for value types that are not custom marshaled no run time test is necessary in the marshaling code.

If a custom marshaled value type has a state definition, the state definition is treated the same as that of a non custom value type for mapping purposes (i.e., the fields show up in the same fashion in the concrete programming language). It is provided to help with application portability.

A custom marshaled value type is always a stateful value type.

```
// Example IDL

custom valuetype T {
    // optional state definition
        ...
};
```

Custom value types can never be safely truncated to base (i.e., they always require an exact match for their RepositoryId in the receiving context).

Once a value type has been marked as custom, it needs to provide an implementation which marshals and unmarshals the valuetype. The marshaling code encapsulates the application code that can marshal and unmarshal instances of the value type over a stream using the CDR encoding. It is the responsibility of the implementation to marshal the state of all of its base types.

The following sections define the operations and streams that are used for custom marshaling.

5.5.1 Implementation of Custom Marshaling

Once a value type has been marked as custom, an implementation of the custom marshaling code must be provided. This is specified by providing a concrete implementation of an abstract value type, **CustomMarshal**, as part of the implementation of the value type. **CustomMarshal** encapsulates the application code that can marshal and unmarshal instances of the value type over a stream using the CDR encoding.

The following IDL defines the interfaces that are used to support the definition and use of custom marshaling.

```
module CORBA {
    abstract valuetype CustomMarshal {
        void marshal (in DataOutputStream os);
        void unmarshal (in DataInputStream is);
    };
};
```

CustomMarshal is an abstract value type that is meant to be used by the ORB, not the user. Semantically it is treated as a custom valuetype's implicit base class, although the custom valuetype does not actually inherit it in IDL. The implementor of a custom value type provides an implementation of the **CustomMarshal** operations. The manner in which this is done is specified for each language mapping. Each custom marshaled value type has its own implementation. The interface is exposed in the CORBA module so that the implementor can use the skeletons generated by the IDL compiler as the basis for the implementation. Hence there is no need for the application to acquire a reference to a Stream.

Note that while nothing prevents a user from writing IDL that inherits from **CustomMarshal**, doing so will not make the type custom, nor will it cause the ORB to treat it as custom.

The implementation requirements of the streaming mechanism require that the implementations must be local since local memory addresses (i.e., the marshal buffers) have to be manipulated.

5.5.2 Marshaling Streams

The streams used for marshaling are defined below. They are responsible for marshaling and demarshaling the data that makes up a custom value in CDR format.

```
module CORBA {

    typedef sequence<any> AnySeq;
    typedef sequence<boolean> BooleanSeq;
    typedef sequence<char> CharSeq;
    typedef sequence<wchar> WCharSeq;
    typedef sequence<octet> OctetSeq;
    typedef sequence<short> ShortSeq;
    typedef sequence<unsigned short> UShortSeq;
```

```
typedef sequence<long> LongSeq;
typedef sequence<unsigned long> ULongSeq;
typedef sequence<long long> LongLongSeq;
typedef sequence<unsigned long long> ULongLongSeq;
typedef sequence<float> FloatSeq;
typedef sequence<double> DoubleSeq;

abstract valuetype DataOutputStream {
    void write_any                  (in any value);
    void write_boolean              (in boolean value);
    void write_char                 (in char value);
    void write_wchar                (in wchar value);
    void write_octet                (in octet value);
    void write_short                (in short value);
    void write_ushort               (in unsigned short value);
    void write_long                 (in long value);
    void write_ulong                (in unsigned long value);
    void write_longlong             (in long long value);
    void write_ulonglong            (in unsigned long long value);
    void write_float                (in float value);

    void write_double               (in double value);
    void write_longdouble           (in long double value);
    void write_string               (in string value);
    void write_wstring              (in wstring value);
    void write_Object               (in Object value);
    void write_Abstract             (in AbstractBase value);

    void write_Value                (in ValueBase value);
    void write_TypeCode             (in TypeCode value);

    void write_any_array(           in AnySeq seq,
                                    in unsigned long offset,
                                    in unsigned long length);
    void write_boolean_array(       in BooleanSeq seq,
                                    in unsigned long offset,
                                    in unsigned long length);
    void write_char_array(          in CharSeq seq,
                                    in unsigned long offset,
                                    in unsigned long length);
    void write_wchar_array(         in WCharSeq seq,
                                    in unsigned long offset,
                                    in unsigned long length);
    void write_octet_array(         in OctetSeq seq,
                                    in unsigned long offset,
                                    in unsigned long length);
    void write_short_array(         in ShortSeq seq,
                                    in unsigned long offset,
                                    in unsigned long length);
    void write_ushort_array(        in UShortSeq seq,
                                    in unsigned long offset,
                                    in unsigned long length);
```

```
        void write_long_array(          in LongSeq seq,
                                        in unsigned long offset,
                                        in unsigned long length);
        void write_ulong_array(         in ULongSeq seq,
                                        in unsigned long offset,
                                        in unsigned long length);
        void write_ulonglong_array(     in ULongLongSeq seq,
                                        in unsigned long offset,
                                        in unsigned long length);
        void write_longlong_array(      in LongLongSeq seq,
                                        in unsigned long offset,
                                        in unsigned long length);
        void write_float_array(         in FloatSeq seq,
                                        in unsigned long offset,
                                        in unsigned long length);
        void write_double_array(        in DoubleSeq seq,
                                        in unsigned long offset,
                                        in unsigned long length);
};

abstract valuetype DataInputStream {
        any read_any();
        boolean read_boolean();
        char read_char();
        wchar read_wchar();
        octet read_octet();
        short read_short();
        unsigned short read_ushort();
        long read_long();
        unsigned long read_ulong();
        long long read_longlong();
        unsigned long long read_ulonglong();
        float read_float();
        double read_double();
        long double read_longdouble();
        string read_string();
        wstring read_wstring();
        Object read_Object();
        AbstractBase read_Abstract();
        ValueBase read_Value();
        TypeCode read_TypeCode();

        void read_any_array(        inout AnySeq seq,
                                    in unsigned long offset,
                                    in unsigned long length);
        void read_boolean_array(    inout BooleanSeq seq,
                                    in unsigned long offset,
                                    in unsigned long length);
        void read_char_array(       inout CharSeq seq,
                                    in unsigned long offset,
                                    in unsigned long length);
```

```
            void read_wchar_array(    inout WCharSeq seq,
                                      in unsigned long offset,
                                      in unsigned long length);
            void read_octet_array(    inout OctetSeq seq,
                                      in unsigned long offset,
                                      in unsigned long length);
            void read_short_array(    inout ShortSeq seq,
                                      in unsigned long offset,
                                      in unsigned long length);
            void read_ushort_array(   inout UShortSeq seq,
                                      in unsigned long offset,
                                      in unsigned long length);
            void read_long_array(     inout LongSeq seq,
                                      in unsigned long offset,
                                      in unsigned long length);
            void read_ulong_array(    inout ULongSeq seq,
                                      in unsigned long offset,
                                      in unsigned long length);
            void read_ulonglong_array(inout ULongLongSeq seq,
                                      in unsigned long offset,
                                      in unsigned long length);
            void read_longlong_array(inout LongLongSeq seq,
                                      in unsigned long offset,
                                      in unsigned long length);
            void read_float_array(    inout FloatSeq seq,
                                      in unsigned long offset,
                                      in unsigned long length);
            void read_double_array(   inout DoubleSeq seq,
                                      in unsigned long offset,
                                      in unsigned long length);
        };
    };
```

Note that the Data streams are abstract value types. This ensures that their implementation will be local, which is required in order for them to properly flatten and encode nested value types.

The ORB (i.e., the CDR encoding engine) is responsible for actually constructing the value's encoding. The application marshaling code merely calls the above operations. The details of writing the value tag, header information, end tag(s) are specifically not exposed to the application code. In particular the size of the custom data is not written by the application. This guarantees that the custom marshaling (and unmarshaling code) cannot corrupt the other parameters of the call.

If an inconsistency is detected, then the standard system exception MARSHAL is raised.

A possible implementation might have the engine determine that a custom marshal parameter is "next." It would then write the value tag and other header information and then return control back to the application defined marshaling policy which would do the marshaling by calling the DataOutputStream operations to write the data as

appropriate. (Note the stream takes care of breaking the data into chunks, if necessary.) When control was returned back to the engine, it performs any other cleanup activities to complete the value type, and then proceeds onto the next parameter. How this is actually accomplished is an implementation detail of the ORB.

The Data Streams shall test for possible shared or null values and place appropriate indirections or null encodings (even when used from the custom streaming policy).

There are no explicit operations for creating the streams. It is assumed that the ORB implicitly acts as a factory. In a sense they are always available.

5.6 Access to the Sending Context Run Time

There are two cases where a receiving context might want to access the run time environment of the sending context:

- To attempt the downloading of some missing implementation for the value.
- To access some meta information about the version of the value just received.

In order to provide that kind of service a call back object interface is defined. It may optionally be supported by the sending context (it can be seen as a service). If such a callback object is supported its IOR may be added to an optional service context in the GIOP header passed from the sending context to the receiving context.

A service context tagged with the ServiceID **SendingContextRunTime** (see Section 13.6.7, “Object Service Context,” on page 13-22) contains an encapsulation of the IOR for a **SendingContext::RunTime** object. Because ORBs are always free to skip a service context they don’t understand, this addition does not impact IIOP interoperability.

```
module SendingContext {

    interface RunTime {}; // so that we can provide more
                          // sending context run time
                          // services in the future

    interface CodeBase: RunTime {
        typedef string URL; // blank-separated list of one or more URLs
        typedef sequence<URL> URLSeq;
        typedef sequence
            <CORBA::ValueDef::FullValueDescription> ValueDescSeq;

        // Operation to obtain the IR from the sending context
        CORBA::Repository get_ir();

        // Operations to obtain a location of the implementation code
        URL implementation(in CORBA::RepositoryId x);
        URLSeq implementations(in CORBA::RepositoryIdSeq x);

        // Operations to obtain complete meta information about a Value
        // This is just a performance optimization the IR can provide
```

```
            // the same information
            CORBA::FullValueDescription meta(in CORBA::RepositoryId x);
            ValueDescSeq metas(in CORBA::RepositoryIdSeq x);

            // To obtain a type graph for a value type
            // same comment as before the IR can provide similar
            // information
            CORBA::RepositoryIdSeq bases(in CORBA::RepositoryId x);
        };
};
```

Supporting the CodeBase interface for a given ORB run time is an issue of quality of service. The point here is that if the sending context does not support a CodeBase then the receiving context will simply raise an exception with which the sending context had to be prepared to deal. There will always be cases where a receiving context will get a value type and won't be able to interpret it because:

- It can't get a legal implementation for it (even if it knows where it is, possibly due to security and/or resource access issues).
- Its local version is so radically different that it cannot make sense out of the piece of state being provided.

These two failure modes will be represented by the CORBA system exception NO_IMPLEMENT with identified minor codes, for a missing local value implementation and for incompatible versions (see Section 3.17.2, "Standard Minor Exception Codes," on page 3-58).

Under certain conditions it is possible that when several values of the same CORBA type (same repository id) are sent in either a request or reply, that the reality is that they have distinct implementations. In this case, in addition to the codebase URL(s) sent in the service context, each value which has a different codebase may have codebase URL(s) associated with it. This is encoded by using a different tag to encode the value on the wire.

Abstract Interface Semantics 6

This chapter describes the semantics of abstract interfaces. Other details specific to particular aspects of the ORB may be found in other chapters.

Contents

This chapter contains the following sections.

6.1 Overview

In many cases it may be useful to defer the determination of whether an object is passed by reference or by value until runtime. An IDL abstract interface provides this capability. See Section 6.4, “Example,” on page 3 for an example of when this might be useful.

6.2 Semantics of Abstract Interfaces

Abstract interfaces differ from regular IDL interfaces in the following ways:

1. When used in an operation signature, they do not determine whether actual parameters are passed as an object reference or by value. Instead, the type of the actual parameter (regular interface or value) is used to make this determination using the following rules:

   - The actual parameter is passed as an object reference if it is a regular interface type (or a subtype of a regular interface type), and that regular interface type is a subtype of the signature abstract interface type, and the object is already registered with the ORB/OA.
   - The actual parameter is passed as a value if it cannot be passed as an object reference but can be passed as a value. Otherwise, a BAD_PARAM exception is raised.

2. The GIOP encoding of an abstract interface type is a union with a boolean discriminator (TRUE if it is an IOR, FALSE if it is a value) followed by either the IOR or the value. This allows the demarshaling code to determine whether an object reference or a value was passed.

3. Abstract interfaces do not implicitly inherit from **CORBA::Object.** This is because they can represent either value types or CORBA object references, and value types do not necessarily support the object reference operations (see Section 4.3, “Object Reference Operations,” on page 4-8). If an IDL abstract interface type can be successfully narrowed to an object reference type (a regular IDL interface), then the **CORBA::Object** operations can be invoked on the narrowed object reference.

4. Abstract interfaces implicitly inherit from **CORBA::AbstractBase**. This type is defined as native. It is the responsibility of each language mapping to specify the actual programming language type that is used for this type.

   ```
   module CORBA {
   // IDL
       native AbstractBase;
   };
   ```

5. Abstract interfaces do not imply copy semantics for value types passed as arguments to their operations. This is because their operations may be either CORBA invocations (for abstract interfaces that represent CORBA object references) or local programming language calls (for abstract interfaces that represent CORBA value types). See Section 5.2.2, “Operations,” on page 5-3 and Section 5.2.4, “Parameter Passing,” on page 5-4 for details of these differences.

6. Abstract interfaces may only inherit from other abstract interfaces.

7. Value types may support any number of abstract interfaces, but no more than one regular interface.

8. In other respects, abstract interfaces are identical to regular IDL interfaces. For example, consider the following operation **m1()** in abstract interface **foo**:

   ```
   abstract interface foo {
           void m1(in AnInterfaceType x, in AnAbstractInterfaceType y,
               in AValueType z);
   ```

```
};
```

x's are always passed by reference,

z's are:

- passed as copied values if **foo** refers to an ordinary interface.
- passed as non-copied values if **foo** refers to a value type

y's are:

- passed as reference if their concrete type is an ordinary interface subtype of **AnAbstractInterfaceType** (registered with the ORB), no matter what **foo**'s concrete type is.
- passed as copied values if their concrete type is value and **foo**'s concrete type is ordinary interface.
- passed as non-copied values if their concrete type is value and **foo**'s concrete type is value.

6.3 Usage Guidelines

Abstract interfaces are intended for situations where it cannot be known at compile time whether an object reference or a value will be passed. In other cases, a regular interface or value type should be used. Abstract interfaces are not intended to replace regular CORBA interfaces in situations where there is no clear need to provide runtime flexibility to pass either an object reference or a value. If reference semantics are intended, regular interfaces should be used.

6.4 Example

For example, in a business application it is extremely common to need to display a list of objects of a given type, with some identifying attribute like account number and a translated text description such as "Savings Account." A developer might define an interface such as **Describable** whose methods provide this information, and implement this interface on a wide range of types. This allows the method that displays items to take an argument of type **Describable** and query it for the necessary information. The **Describable** objects passed in to the **display** method may be either CORBA interface types (passed in as object references) or CORBA value types (passed in by value).

In this example, **Describable** is used as a polymorphic abstract type. No instances of type **Describable** exist, but many different instances have interfaces that support the **Describable** type abstraction. In C++, **Describable** would be an abstract base class; in Java, an interface. In statically typed languages, the compiler can check that the actual parameter type passed by callers of **display** is a valid subtype of **Describable** and therefore supports the methods defined by **Describable**. The **display** method can simply invoke the methods of **Describable** on the objects that it receives, without concern for any details of their implementation.

Describable could not be declared as a regular IDL interface. This is because arguments of declared interface type are always passed as object references (see Section 5.2.4, “Parameter Passing,” on page 5-4) and we also want the **display** method to be able to accept value type objects that can only be passed by value. Similarly we cannot define **Describable** as a value type because then the **display** method would not be able to accept actual parameter objects that only support passing as an object reference. Abstract interfaces are needed to cover such cases.

The **Describable** abstract interface could be defined and used by the following IDL:

```
abstract interface Describable {
        string get_description();
};

interface Example {
        void display (in Describable anObject);
};

interface Account : Describable {// passed by reference
   // add Account methods here
};

valuetype Currency supports Describable {// passed by value
   // add Currency methods here
};
```

If **Describable** were defined as a regular interface instead of an abstract interface, then it would not be possible to pass a **Currency** value to the display method, even though the **Currency** IDL type supports the **Describable** interface.

6.5 Security Considerations

Security considerations for abstract interfaces are similar to those for regular interfaces and values (see Section 5.2.9, “Security Considerations,” on page 5-7). This is because an abstract interface formal parameter type allows either a regular interface (IOR) or a value to be passed. Likewise, an operation defined in an abstract interface can be implemented by either a regular interface (with “normal” security considerations) or by a value type (in which case it is a local call, not mediated by the ORB). The security implication of making the choice between these alternatives a runtime determination is that the programmer must ensure that for both alternatives, no security violations can occur. For example, a technique similar to that described in Section 6.5.1, “Passing Values to Trusted Domains,” on page 6-4 could be used to avoid inadvertently passing values outside a domain of trust.

6.5.1 Passing Values to Trusted Domains

When a server passes an object reference, it can be sure that access control policies will apply to any attempt to access anything through that object reference. When the underlying object is passed as a value, the granularity and level/semantics of access

control are different. In the “by value” case, all the data for the object is passed, and method invocations on the passed object are local calls that are not mediated by the ORB. Whether the server wants to use the (potentially more permissive) pass by value access control or not could depend on the security domain which is receiving the said object or object reference.

Consider the case where the server S has an object O that it is willing to pass only in the form of an object reference Or' to a domain Du that it does not trust, but is willing to pass the object by value Ow to another domain Ot that it trusts.

This flexibility is not possible without abstract interfaces. Signatures would have to be written to either always pass references or always pass values, irrespective of the level of trust of the invocation target domain. However, abstract interfaces provide the necessary flexibility. The formal parameter type **`MyType`** can be declared as an abstract interface and the method invocation can be coded along the lines of

```
myExample->foo(security_check(myExample,mydata));
```

where the **`security_check`** function determines the level of trust of **`myExample`**'s domain and returns an regular interface subtype of **`MyType`** for untrusted domains and a value subtype of **`MyType`** for trusted domains. The rules for abstract interfaces will then pass the correct thing in both these cases.

Dynamic Invocation Interface 7

This chapter has been updated based on the CORE document (ptc/98-09-04).

The Dynamic Invocation Interface (DII) describes the client's side of the interface that allows dynamic creation and invocation of request to objects. 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::**".

Contents

This chapter contains the following sections.

7.1 Overview

The Dynamic Invocation Interface (DII) 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 Section 7.1.1, “Common Data Structures,” on page 7-2). 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.

The user exception WrongTransaction is defined in the CORBA module, prior to the definitions of the ORB and Request interfaces, as follows:

```
exception WrongTransaction {};
```

This exception can be raised only if the request is implicitly associated with a transaction (the current transaction at the time that the request was issued).

7.1.1 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 as:

```
typedef unsigned long Flags;

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

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.

For out parameters, applications can set the **argument** member of the **NamedValue** structure to a value that includes either a NULL or a non-NULL storage pointer. If a non-null storage pointer is provided for an out parameter, the ORB will attempt to use the storage pointed to for holding the value of the out parameter. If the storage pointed to is not sufficient to hold the value of the out parameter, the behavior is undefined.

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 19.7, “Mapping for Basic Data Types,” on page 19-10, 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. See this section for a full description of TypeCodes.

For most data types, **len** is the actual number of bytes that the value occupies. For object references, **len** is 1. Table 7-1shows 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 7-1 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)
long long	sizeof (CORBA_long_long)
unsigned long long	sizeof (CORBA_unsigned_long_long)
float	sizeof (CORBA_float)
double	sizeof (CORBA_double)
long double	sizeof (CORBA_long_double)
fixed<d,s>	sizeof (CORBA_fixed_d_s)
char	sizeof (CORBA_char)
wchar	sizeof (CORBA_wchar)
boolean	sizeof (char)
octet	sizeof (CORBA_octet)
string	strlen (string) /* does NOT include ‘\0’ byte! */
wstring	number of wide characters in string, not including wide null terminator
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.

7.1.2 Memory Usage

The values for output argument data types that are unbounded strings or unbounded sequences are returned as pointers to dynamically allocated memory. 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 the *C Language Mapping* specification (*Mapping for Structure Types* section).

7.1.3 Return Status and Exceptions

In the Dynamic Invocation interface, routines typically indicate errors or exceptional conditions either via programming language exception mechanisms, or via an Environment parameter for those languages that do not support exceptions. Thus, the return type of these routines is void.

7.2 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 {

    native OpaqueValue;

    interface Request {                                    // PIDL

        void add_arg (
```

```
            in Identifier       name,       // argument name
            in TypeCode         arg_type,   // argument datatype
            in OpaqueValue      value,      // argument value to be added
            in long             len,        // length/count of argument
                                            value
            in Flags            arg_flags   // argument flags
        );

        void invoke (
            in Flags            invoke_flags // invocation flags
        );

        void delete ();

        void send (
            in Flags            invoke_flags // invocation flags
        );

        void get_response () raises (WrongTransaction);

        boolean poll_response();
    };
};
```

In IDL, The **native** type **OpaqueValue** is used to identify the type of the implementation language representation of the value that is to be passed as a parameter. For example in the C language this is the C language type **`(void *)`**. Each language mapping specifies what **OpaqueValue** maps to in that specific language.

7.2.1 create_request

Because it creates a pseudo-object, this operation is defined in the **Object** interface (see Section 4.3, “Object Reference Operations,” on page 4-8 for the complete interface definition). The **create_request** operation is performed on the **Object** which is to be invoked.

```
void 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 8.3, “ServerRequestPseudo-Object,” on page 8-3.

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 data type, 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_MEMORY indicates 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 the *C Language Mapping* specification, *Argument Passing Considerations* section); *C++ Language Mapping* specification, *NVList* section; and the *COBOL Language Mapping* specification, *Argument Passing Considerations* section).

The implicit object reference operations **non_existent**, **is_a**, and **get_interface** may be invoked using DII. No other implicit object reference operations may be invoked via DII.

To create a request for any one of these allowed implicit object reference operations, **create_request** must be passed the name of the operation with a "_" prepended, in the parameter "operation." For example to create a DII request for "**is_a**", the name passed to **create_request** must be "**_is_a**". If the name of an implicit operation that is not ivocable through DII is passed to **create_request** with a "_" prepended, **create_request** shall raise a BAD_PARAM exception. For example, if "**_is_equivalent**" is passed to create_request as the "**operation**" parameter will cause **create_request** to raise the BAD_PARAM exception.

7.2.2 add_arg

```
void add_arg (                                          // PIDL
    in Identifier       name,       // argument name
    in TypeCode         arg_type,   // argument datatype
    in OpaqueValue      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 data type, name, and usage flags (i.e., in, out, inout) may also be specified. If so indicated, arguments are validated for data type, 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 supported.

In addition to the argument modes defined in "Common Data Structures" on page 7-2, **arg_flags** may also take the flag value **IN_COPY_VALUE**. The argument passing flags defined in "Common Data Structures" on page 7-2 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.

7.2.3 invoke

```
void 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 **create_request**. The behavior is undefined if the **Request** pseudo-object has already been used with a previous call to **invoke**, **send**, or **send_multiple_requests**.

7.2.4 delete

```
void 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.

7.3 Deferred Synchronous Operations

7.3.1 send

```
void 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 flag is 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.**

7.3.2 send_multiple_requests

```
interface Request;       // forward declaration
typedef sequence <Request> RequestSeq;

void send_multiple_requests_oneway(in RequestSeq req);
void send_multiple_requests_deferred(in RequestSeq req);
```

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.

7.3.3 poll_response

```
// PIDL
boolean poll_response ( in Request    req);
```

poll_response determines whether the request has completed. A **TRUE** return indicates that it has; **FALSE** indicates it has not.

Return is immediate, whether the response has completed or not. Values in the request are not changed.

7.3.4 get_response

```
//PIDL
void get_response () raises (WrongTransaction);
```

get_response returns the result of a request. If **get_response** is called before the request has completed, it blocks until the request has completed. Upon return, the out parameters and return values defined in the **Request** are set appropriately and they may be treated as if the **Request** invoke operation had been used to perform the request.

A request has an associated transaction context if the thread originating the request had a non-null transaction context and the target object is a transactional object. The **get_response** operation may raise the WrongTransaction exception if the request has an associated transaction context, and the thread invoking **get_response** either has a null transaction context or a non-null transaction context that differs from that of the request.

7.3.5 get_next_response

```
interface Request;       // forward declaration

boolean poll_next_response();
void get_next_response(out Request req) raises (WrongTransaction);
```

Poll_next_response determines whether any request has completed. A **TRUE** return indicates that at least one has; **FALSE** indicates that none have completed. Return is immediate, whether any response has completed or not.

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 finish.

A request has an associated transaction context if the thread originating the request had a non-null transaction context and the target object is a transactional object. The **get_next_response** operation may raise the WrongTransaction exception if the request has an associated transaction context, and the thread invoking **get_next_response** has a non-null transaction context that differs from that of the request.

7.4 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 the ORB Interface chapter, but are described in this section. The **NVList** interface is shown below.

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

Interface **NVList** is defined in the CORBA module.

7.4.1 create_list

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

```
void 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 and clears it for initial use. The specified count is a "hint" to help with the storage allocation. List items may be added to the list using the **add_item** routine. Items are added starting with the "**slot()**," in the next available slot.

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

7.4.2 add_item

```
void add_item (                                             // PIDL
    in Identifier       item_name,      // name of item
    in TypeCode         item_type,      // item datatype
    in OpaqueValue      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 7.1.1, "Common Data Structures," on page 7-2, **item_flags** may also take the following flag values: **IN_COPY_VALUE**, **DEPENDENT_LIST**. The argument passing flags defined in Section 7.1.1, "Common Data Structures," on page 7-2 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.

7.4.3 free

```
void free ( );                                              // PIDL
```

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

7.4.4 free_memory

```
void free_memory ( );                                       // PIDL
```

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

7.4.5 get_count

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

This operation returns the total number of items added to the list.

7.4.6 create_operation_list

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

```
void 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.

7.5 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 to 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 OMG 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.

7.6 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 Section 3.2.3, "Identifiers," on page 3-6 for the valid set of characters that are allowed).

The **Context** interface is shown below.

```
module CORBA {

    interface Context {                                          // PIDL
        void set_one_value (
            in Identifier       prop_name,    // property name to add
            in string           value         // property value to add
        );
        void set_values (
```

```
            in NVList       values      // property values to be
                                            changed
        );
        void 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)
        );
        void delete_values (
            in Identifier   prop_name    // name of property(s) to
                                            delete
        );
        void create_child (
            in Identifier   ctx_name,    // name of context object
            out Context     child_ctx    // newly created context
                                            object
        );
        void delete (
            in Flags        del_flags    // flags controlling deletion
        );
    };
};
```

7.6.1 get_default_context

This operation, which creates a **Context** pseudo-object, is defined in the **ORB** interface (see Section 4.2.1, “Converting Object References to Strings,” on page 4-7 for the complete ORB definition).

```
void 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.

7.6.2 set_one_value

```
void 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.

7.6.3 set_values

```
void 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**.

7.6.4 get_values

```
void 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 **prop_name** is an empty string then the CORBA::BAD_PARAM exception is raised. If a property named by **prop_name** is not found then the CORBA::BAD_CONTEXT and no property list is returned. The CORBA::NO_MEMORY exception is raised if dynamic memory allocation fails.

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.

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 flag may be specified:

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

7.6.5 delete_values

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

This operation deletes the specified property value(s) 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 **prop_name** is an empty string the CORBA::BAD_PARAM exception is raised. If no matching property is found, the CORBA::BAD_CONTEXT exception is raised.

7.6.6 create_child

```
void 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 Section 3.2.3, “Identifiers,” on page 3-6).

7.6.7 delete

```
void 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_DESCENDENTS deletes the indicated context object and all of its descendent context objects, as well.

The exception CORBA::BAD_PARAM is raised if there are one or more child context objects and the **CTX_DELETE_DESCENDENTS** flag was not set.

7.7 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 8

The Dynamic Skeleton Interface (DSI) 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 also be used, to determine the parameters.

Contents

This chapter contains the following sections.

8.1 Introduction

The Dynamic Skeleton Interface 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.

.

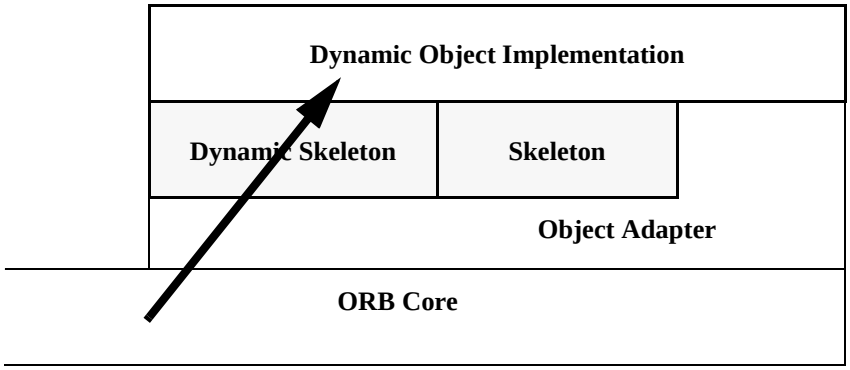


Figure 8-1 Requests are delivered through skeletons, including dynamic ones

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.

8.2 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.

This chapter describes the elements of the DSI that are common to all object adapters that provide a DSI. See Section 11.6.11, "Single Servant, Many Objects and Types, Using DSI," on page 11-59 for the specification of the DSI for the Portable Object Adapter.

8.3 ServerRequestPseudo-Object

8.3.1 ExplicitRequest State: ServerRequestPseudo-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 object adapter dispatches an invocation to a DSI-based object implementation by passing an instance of ServerRequest to the DIR associated with the object implementation. The following shows how it provides access to the request information:

```
module CORBA {
    ...
    interface ServerRequest { //                          PIDL
        readonly attribute Identifier    operation;
        void          arguments(inout NVList nv);
        Context       ctx();
        void          set_result(in Any val);
        void          set_exception(in Any val);
    };
};
```

The identity and/or reference of the target object of the invocation is provided by the object adapter and its language mapping. In the context of a bridge, the target object will typically be a proxy for an object in some other ORB.

The **operation** attribute provides the identifier naming the operation being invoked; according to OMG IDL's rules, these names must be unique among all operations supported by the object's "most-derived" interface. Note that the operation names for getting and setting attributes are **_get_<attribute_name>** and **_set_<attribute_name>**, respectively. The operation attribute can be accessed by the DIR at any time.

Operation parameter types will be specified, and "in" and "inout" argument values will be retrieved, with **arguments**. Unless it calls **set_exception**, the DIR must call **arguments** exactly once, even if the operation signature contains no parameters. Once **arguments** or **set_exception** has been called, calling **arguments** on the same **ServerRequest** will result in a BAD_INV_ORDER system exception. The DIR must pass in to **arguments** an **NVList** initialized with TypeCodes and Flags describing the parameter types for the operation, in the order in which they appear in the IDL specification (left to right). A potentially-different **NVList** will be returned from **arguments**, with the "in" and "inout" argument values supplied. If it does not call **set_exception**, the DIR must supply the returned **NVList** with return values for any "out" arguments before returning, and may also change the return values for any "inout" arguments.

When the operation is not an attribute access, and the operation's IDL definition contains a context expression, **ctx** will return the context information specified in IDL for the operation. Otherwise it will return a nil **Context** reference. Calling **ctx** before **arguments** has been called or after **ctx**, **set_result**, or **set_exception** has been called will result in a BAD_INV_ORDER system exception.

The **set_result** operation is used to specify any return value for the call. Unless **set_exception** is called, if the invoked operation has a non-void result type, **set_result** must be called exactly once before the DIR returns. If the operation has a void result type, **set_result** may optionally be called once with an **Any** whose type is **tk_void**. Calling **set_result** before **arguments** has been called or after **set_result** or **set_exception** has been called will result in a BAD_INV_ORDER system exception. Calling **set_result** without having previously called **ctx** when the operation IDL contains a context expression, or when the **NVList** passed to **arguments** did not describe all parameters passed by the client, may result in a MARSHAL system exception.

The DIR may call **set_exception** at any time to return an exception to the client. The **Any** passed to **set_exception** must contain either a system exception or one of the user exceptions specified in the **raises** expression of the invoked operation's IDL definition. Passing in an **Any** that does not contain an exception will result in a BAD_PARAM system exception. Passing in an unlisted user exception will result in either the DIR receiving a BAD_PARAM system exception or in the client receiving an UNKNOWN_EXCEPTION system exception.

See each language mapping for a description of the memory management aspects of the parameters to the **ServerRequest** operations.

8.4 DSI: 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. Each language provides its own mapping for DSI.

8.4.1 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.

8.4.2 Registering Dynamic Implementation Routines

In an ORB implementation, the Dynamic Skeleton Interface is supported entirely through the Object Adapter. An Object Adapter does not have to support the Dynamic Skeleton Interface but, if it does, the Object Adapter is responsible for the details.

Dynamic Management of Any Values 9

Please note that this document is based on ptc/99-03-02. All changes took place in the 2.4 RTF.

An **any** can be passed to a program that doesn't have any static information for the type of the **any** (code generated for the type by an IDL compiler has not been compiled with the object implementation). As a result, the object receiving the **any** does not have a portable method of using it.

The facility presented here enables traversal of the data value associated with an **any** at runtime and extraction of the primitive constituents of the data value. This is especially helpful for writing powerful generic servers (bridges, event channels supporting filtering).

Similarly, this facility enables the construction of an **any** at runtime, without having static knowledge of its type. This is especially helpful for writing generic clients (bridges, browsers, debuggers, user interface tools).

Contents

This chapter contains the following sections.

9.1 Overview

Unless explicitly stated otherwise, all IDL presented in section 9.1 through section 9.3 is part of the **DynamicAny** module.

Any values can be dynamically interpreted (traversed) and constructed through **DynAny** objects. A **DynAny** object is associated with a data value which corresponds to a copy of the value inserted into an **any**.

A **DynAny** object may be viewed as an ordered collection of component **DynAny**s. For **DynAny**s representing a basic type, such as **long**, or a type without components, such as an empty exception, the ordered collection of components is empty. Each **DynAny** object maintains the notion of a current position into its collection of component **DynAny**s. The current position is identified by an index value that runs from 0 to n–1, where n is the number of components. The special index value –1 indicates a current position that points nowhere. For values that cannot have a current position (such as an empty exception), the index value is fixed at –1. If a **DynAny** is initialized with a value that has components, the index is initialized to 0. After creation of an uninitialized **DynAny** (that is, a **DynAny** that has no value but a **TypeCode** that permits components), the current position depends on the type of value represented by the **DynAny**. (The current position is set to 0 or –1, depending on whether the new **DynAny** gets default values for its components.)

The iteration operations **rewind**, **seek**, and **next** can be used to change the current position and the **current_component** operation returns the component at the current position. The **component_count** operation returns the number of components of a **DynAny**. Collectively, these operations enable iteration over the components of a **DynAny**, for example, to (recursively) examine its contents.

A constructed **DynAny** object is a **DynAny** object associated with a constructed type. There is a different interface, inheriting from the **DynAny** interface, associated with each kind of constructed type in IDL (fixed, enum, struct, sequence, union, array, exception, and valuetype).

A constructed **DynAny** object exports operations that enable the creation of new **DynAny** objects, each of them associated with a component of the constructed data value.

As an example, a **DynStruct** is associated with a struct value. This means that the **DynStruct** may be seen as owning an ordered collection of components, one for each structure member. The **DynStruct** object exports operations that enable the creation of new **DynAny** objects, each of them associated with a member of the struct.

If a **DynAny** object has been obtained from another (constructed) **DynAny** object, such as a **DynAny** representing a structure member that was created from a **DynStruct**, the member **DynAny** is logically contained in the **DynStruct**.

Destroying a top-level **DynAny** object (one that was not obtained as a component of another **DynAny**) also destroys any component **DynAny** objects obtained from it. Destroying a non-top level **DynAny** object does nothing. Invoking operations on a destroyed top-level **DynAny** or any of its descendants raises OBJECT_NOT_EXIST.

Note that simply releasing all references to a **DynAny** object does not delete the **DynAny** or components; each **DynAny** created with one of the create operations or with the **copy** operation must be explicitly destroyed to avoid memory leaks.

If the programmer wants to destroy a **DynAny** object but still wants to manipulate some component of the data value associated with it, then he or she should first create a **DynAny** for the component and, after that, make a copy of the created **DynAny** object.

The behavior of **DynAny** objects has been defined in order to enable efficient implementations in terms of allocated memory space and speed of access. **DynAny** objects are intended to be used for traversing values extracted from **any**s or constructing values of **any**s at runtime. Their use for other purposes is not recommended.

9.2 DynAny API

The **DynAny** API comprises the following IDL definitions, located in the **DynamicAny** module:

```
// IDL
// File: DynamicAny.idl
#ifndef _DYNAMIC_ANY_IDL_
#define _DYNAMIC_ANY_IDL_
#pragma prefix “omg.org”
#include <orb.idl>

module DynamicAny {

    interface DynAny {
        exception InvalidValue {};
        exception TypeMismatch {};

        CORBA::TypeCode type();

        void assign(in DynAny dyn_any) raises(TypeMismatch);
        void from_any(in any value) raises(TypeMismatch, InvalidValue);
        any to_any();

        boolean equal(in DynAny dyn_any);

        void destroy();
        DynAny copy();
```

```
void insert_boolean(in boolean value)
      raises(TypeMismatch, InvalidValue);
void insert_octet(in octet value)
      raises(TypeMismatch, InvalidValue);
void insert_char(in char value)
      raises(TypeMismatch, InvalidValue);
void insert_short(in short value)
      raises(TypeMismatch, InvalidValue);
void insert_ushort(in unsigned short value)
      raises(TypeMismatch, InvalidValue);
void insert_long(in long value)
      raises(TypeMismatch, InvalidValue);
void insert_ulong(in unsigned long value)
      raises(TypeMismatch, InvalidValue);
void insert_float(in float value)
      raises(TypeMismatch, InvalidValue);
void insert_double(in double value)
      raises(TypeMismatch, InvalidValue);
void insert_string(in string value)
      raises(TypeMismatch, InvalidValue);
void insert_reference(in Object value)
      raises(TypeMismatch, InvalidValue);
void insert_typecode(in CORBA::TypeCode value)
      raises(TypeMismatch, InvalidValue);
void insert_longlong(in long long value)
      raises(TypeMismatch, InvalidValue);
void insert_ulonglong(in unsigned long long value)
      raises(TypeMismatch, InvalidValue);
void insert_longdouble(in long double value)
      raises(TypeMismatch, InvalidValue);
void insert_wchar(in wchar value)
      raises(TypeMismatch, InvalidValue);
void insert_wstring(in wstring value)
      raises(TypeMismatch, InvalidValue);
void insert_any(in any value)
      raises(TypeMismatch, InvalidValue);
void insert_dyn_any(in DynAny value)
      raises(TypeMismatch, InvalidValue);
void insert_val(in ValueBase value)
      raises(TypeMismatch, InvalidValue);
```

```
boolean get_boolean()
      raises(TypeMismatch, InvalidValue);
octet get_octet()
      raises(TypeMismatch, InvalidValue);
char get_char()
      raises(TypeMismatch, InvalidValue);
short get_short()
      raises(TypeMismatch, InvalidValue);
unsigned short get_ushort()
      raises(TypeMismatch, InvalidValue);
```

```
        long get_long()
            raises(TypeMismatch, InvalidValue);
        unsigned long get_ulong()
            raises(TypeMismatch, InvalidValue);
        float get_float()
            raises(TypeMismatch, InvalidValue);
        double get_double()
            raises(TypeMismatch, InvalidValue);
        string get_string()
            raises(TypeMismatch, InvalidValue);
        Object get_reference()
            raises(TypeMismatch, InvalidValue);
        CORBA::TypeCode get_typecode()
            raises(TypeMismatch, InvalidValue);
        long long get_longlong()
            raises(TypeMismatch, InvalidValue);
        unsigned long long get_ulonglong()
            raises(TypeMismatch, InvalidValue);
        long double get_longdouble()
            raises(TypeMismatch, InvalidValue);
        wchar get_wchar()
            raises(TypeMismatch, InvalidValue);
        wstring get_wstring()
            raises(TypeMismatch, InvalidValue);
        any get_any()
            raises(TypeMismatch, InvalidValue);
        DynAny get_dyn_any()
            raises(TypeMismatch, InvalidValue);
        ValueBase get_val()
            raises(TypeMismatch, InvalidValue);

        boolean seek(in long index);
        void rewind();
        boolean next();
        unsigned long component_count();
        DynAny current_component() raises(TypeMismatch);
    };

    interface DynFixed : DynAny {
        string get_value();
        boolean set_value(in string val) raises(TypeMismatch, InvalidValue);
    };

    interface DynEnum : DynAny {
        string get_as_string();
        void set_as_string(in string value) raises(InvalidValue);
        unsigned long get_as_ulong();
        void set_as_ulong(in unsigned long value) raises(InvalidValue);
    };
```

```
typedef string FieldName;

struct NameValuePair {
    FieldName id;
    any value;
};
typedef sequence<NameValuePair> NameValuePairSeq;

struct NameDynAnyPair {
    FieldName id;
    DynAny value;
};
typedef sequence<NameDynAnyPair> NameDynAnyPairSeq;

interface DynStruct : DynAny {
    FieldName current_member_name()
        raises(TypeMismatch, InvalidValue);
    CORBA::TCKind current_member_kind()
        raises(TypeMismatch, InvalidValue);
    NameValuePairSeq get_members();
    void set_members(in NameValuePairSeq value)
        raises(TypeMismatch, InvalidValue);
    NameDynAnyPairSeq get_members_as_dyn_any();
    void set_members_as_dyn_any(in NameDynAnyPairSeq value)
        raises(TypeMismatch, InvalidValue);
};

interface DynUnion : DynAny {
    DynAny get_discriminator();
    void set_discriminator(in DynAny d) raises(TypeMismatch);
    void set_to_default_member() raises(TypeMismatch);
    void set_to_no_active_member() raises(TypeMismatch);
    boolean has_no_active_member();
    CORBA::TCKind discriminator_kind();
    DynAny member() raises(InvalidValue);
    FieldName member_name() raises(InvalidValue);
    CORBA::TCKind member_kind() raises(InvalidValue);
};
```

```
        typedef sequence<any> AnySeq;
        typedef sequence<DynAny> DynAnySeq;

        interface DynSequence : DynAny {
            unsigned long get_length();
            void set_length(in unsigned long len) raises(InvalidValue);
            AnySeq get_elements();
            void set_elements(in AnySeq value)
                raises(TypeMismatch, InvalidValue);
            DynAnySeq get_elements_as_dyn_any();
            void set_elements_as_dyn_any(in DynAnySeq value)
                raises(TypeMismatch, InvalidValue);
        };

        interface DynArray : DynAny {
            AnySeq get_elements();
            void set_elements(in AnySeq value)
                raises(TypeMismatch, InvalidValue);
            DynAnySeq get_elements_as_dyn_any();
            void set_elements_as_dyn_any(in DynAnySeq value)
                raises(TypeMismatch, InvalidValue);
        };

        interface DynValue : DynAny {
            FieldName current_member_name()
                raises(TypeMismatch, InvalidValue);
            CORBA::TCKind current_member_kind()
                raises(TypeMismatch, InvalidValue);
            NameValuePairSeq get_members();
            void set_members(in NameValuePairSeq value)
                raises(TypeMismatch, InvalidValue);
            NameDynAnyPairSeq get_members_as_dyn_any();
            void set_members_as_dyn_any(in NameDynAnyPairSeq value)
                raises(TypeMismatch, InvalidValue);
        };

        interface DynAnyFactory {
            exception InconsistentTypeCode {};
            DynAny create_dyn_any(in any value)
                raises(InconsistentTypeCode);
            DynAny
                create_dyn_any_from_type_code(in CORBA::TypeCode type)
                raises(InconsistentTypeCode);
        };

    }; // module DynamicAny

    #endif // _DYNAMIC_ANY_IDL_
```

9.2.1 Locality and usage constraints

DynAny and **DynAnyFactory** objects are intended to be local to the process in which they are created and used. This means that references to **DynAny** and **DynAnyFactory** objects cannot be exported to other processes, or externalized with **ORB::object_to_string**. If any attempt is made to do so, the offending operation will raise a MARSHAL system exception.

Since their interfaces are specified in IDL, **DynAny** objects export operations defined in the standard **CORBA::Object** interface. However, any attempt to invoke operations exported through the **Object** interface may raise the standard NO_IMPLEMENT exception.

An attempt to use a **DynAny** object with the DII may raise the NO_IMPLEMENT exception.

9.2.2 Creating a DynAny object

A **DynAny** object can be created as a result of:

- invoking an operation on an existing **DynAny** object
- invoking an operation on a **DynAnyFactory** object

A constructed **DynAny** object supports operations that enable the creation of new **DynAny** objects encapsulating access to the value of some constituent. **DynAny** objects also support the **copy** operation for creating new **DynAny** objects.

In addition, **DynAny** objects can be created by invoking operations on the **DynAnyFactory** object. A reference to the **DynAnyFactory** object is obtained by calling **CORBA::ORB::resolve_initial_references** with the **identifier** parameter set to **"DynAnyFactory"**.

```
interface DynAnyFactory {
    exception InconsistentTypeCode {};
    DynAny create_dyn_any(in any value)
        raises(InconsistentTypeCode);
    DynAny create_dyn_any_from_type_code(in CORBA::TypeCode type)
        raises(InconsistentTypeCode);
};
```

The **create_dyn_any** operation creates a new **DynAny** object from an **any** value. A copy of the **TypeCode** associated with the **any** value is assigned to the resulting **DynAny** object. The value associated with the **DynAny** object is a copy of the value in the original any. The **create_dyn_any** operation sets the current position of the created **DynAny** to zero if the passed value has components; otherwise, the current position is set to −1. The operation raises **InconsistentTypeCode** if **value** has a **TypeCode** with a **TCKind** of **tk_Principal**, **tk_native**, or **tk_abstract_interface**.

The **create_dyn_any_from_type_code** operation creates a **DynAny** from a **TypeCode**. Depending on the **TypeCode**, the created object may be of type **DynAny**, or one of its derived types, such as **DynStruct**. The returned reference can be narrowed to the derived type.

In all cases, a **DynAny** constructed from a **TypeCode** has an initial default value. The default values of basic types are:

- **FALSE** for **Boolean**
- zero for numeric types
- zero for types **octet**, **char**, and **wchar**
- the empty string for **string** and **wstring**
- nil for object references
- a type code with a **TCKind** value of **tk_null** for type codes
- for **any** values, an **any** containing a type code with a **TCKind** value of **tk_null** type and no value

For complex types, creation of the corresponding **DynAny** assigns a default value as follows:

- For **DynSequence**, the operation sets the current position to –1 and creates an empty sequence.
- For **DynEnum**, the operation sets the current position to –1 and sets the value of the enumerator to the first enumerator value indicated by the **TypeCode**.
- For **DynFixed**, operations set the current position to –1 and sets the value zero.
- For **DynStruct**, the operation sets the current position to –1 for empty exceptions and to zero for all other **TypeCode**s. The members (if any) are (recursively) initialized to their default values.
- For **DynArray**, the operation sets the current position to zero and (recursively) initializes elements to their default value.
- For **DynUnion**, the operation sets the current position to zero. The discriminator value is set to a value consistent with the first named member of the union. That member is activated and (recursively) initialized to its default value.
- For **DynValue**, the members are initialized as for **DynStruct**.

Dynamic interpretation of an **any** usually involves creating a **DynAny** object using **DynAnyFactory::create_dyn_any** as the first step. Depending on the type of the **any**, the resulting **DynAny** object reference can be narrowed to a **DynFixed**, **DynStruct**, **DynSequence**, **DynArray**, **DynUnion**, **DynEnum**, or **DynValue** object reference.

Dynamic creation of an **any** involves creating a **DynAny** object using **DynAnyFactory::create_dyn_any_from_type_code**, passing the **TypeCode** associated with the value to be created. The returned reference is narrowed to one of

the complex types, such as **DynStruct**, if appropriate. Then, the value can be initialized by means of invoking operations on the resulting object. Finally, the **to_any** operation can be invoked to create an **any** value from the constructed **DynAny**.

9.2.3 The DynAny interface

The following operations can be applied to a **DynAny** object:

- Obtaining the **TypeCode** associated with the **DynAny** object
- Generating an **any** value from the **DynAny** object
- Comparing two **DynAny** objects for equality
- Destroying the **DynAny** object
- Creating a **DynAny** object as a copy of the **DynAny** object
- Inserting/getting a value of some basic type into/from the **DynAny** object
- Iterating through the components of a **DynAny**
- Initializing a **DynAny** object from another **DynAny** object
- Initializing a **DynAny** object from an **any** value

9.2.3.1 Obtaining the TypeCode associated with a DynAny object

```
CORBA::TypeCode type();
```

A **DynAny** object is created with a **TypeCode** value assigned to it. This **TypeCode** value determines the type of the value handled through the **DynAny** object. The **type** operation returns the **TypeCode** associated with a **DynAny** object.

Note that the **TypeCode** associated with a **DynAny** object is initialized at the time the **DynAny** is created and cannot be changed during lifetime of the **DynAny** object.

9.2.3.2 Initializing a DynAny object from another DynAny object

```
void assign(in DynAny dyn_any) raises(TypeMismatch);
```

The **assign** operation initializes the value associated with a **DynAny** object with the value associated with another **DynAny** object.

If the type of the passed **DynAny** is not equivalent to the type of target **DynAny**, the operation raises TypeMismatch. The current position of the target **DynAny** is set to zero for values that have components and to −1 for values that do not have components.

9.2.3.3 Initializing a DynAny object from an any value

void from_any(in any value) raises(TypeMismatch, InvalidValue);

The **from_any** operation initializes the value associated with a **DynAny** object with the value contained in an **any**.

If the type of the passed **Any** is not equivalent to the type of target **DynAny**, the operation raises TypeMismatch. If the passed **Any** does not contain a legal value (such as a null string), the operation raises InvalidValue. The current position of the target **DynAny** is set to zero for values that have components and to –1 for values that do not have components.

9.2.3.4 Generating an any value from a DynAny object

any to_any();

The **to_any** operation creates an **any** value from a **DynAny** object. A copy of the **TypeCode** associated with the **DynAny** object is assigned to the resulting **any**. The value associated with the **DynAny** object is copied into the **any**.

9.2.3.5 Comparing DynAny values

boolean equal(in DynAny dyn_any);

The **equal** operation compares two **DynAny** values for equality and returns true of the **DynAny**s are equal, false otherwise. Two **DynAny** values are equal if their **TypeCodes** are equivalent and, recursively, all component **DynAnys** have equal values. The current position of the two **DynAny**s being compared has no effect on the result of **equal**.

9.2.3.6 Destroying a DynAny object

void destroy();

The **destroy** operation destroys a **DynAny** object. This operation frees any resources used to represent the data value associated with a **DynAny** object. **destroy** must be invoked on references obtained from one of the creation operations on the **ORB** interface or on a reference returned by **DynAny::copy** to avoid resource leaks. Invoking **destroy** on component **DynAny** objects (for example, on objects returned by the **current_component** operation) does nothing.

Destruction of a **DynAny** object implies destruction of all **DynAny** objects obtained from it. That is, references to components of a destroyed **DynAny** become invalid; invocations on such references raise OBJECT_NOT_EXIST.

It is possible to manipulate a component of a **DynAny** beyond the life time of the **DynAny** from which the component was obtained by making a copy of the component with the **copy** operation before destroying the **DynAny** from which the component was obtained.

9.2.3.7 Creating a copy of a DynAny object

```
DynAny copy();
```

The **copy** operation creates a new **DynAny** object whose value is a deep copy of the **DynAny** on which it is invoked. The operation is polymorphic, that is, invoking it on one of the types derived from **DynAny**, such as **DynStruct**, creates the derived type but returns its reference as the **DynAny** base type.

9.2.3.8 Accessing a value of some basic type in a DynAny object

The insert and get operations enable insertion/extraction of basic data type values into/from a **DynAny** object.

Both bounded and unbounded strings are inserted using **insert_string** and **insert_wstring**. These operations raise the **InvalidValue** exception if the string inserted is longer than the bound of a bounded string.

Calling an insert or get operation on a **DynAny** that has components but has a current position of −1 raises **InvalidValue**.

Get operations raise **TypeMismatch** if the accessed component in the **DynAny** is of a type that is not equivalent to the requested type. (Note that **get_string** and **get_wstring** are used for both unbounded and bounded strings.)

A type is consistent for inserting or extracting a value if its **TypeCode** is equivalent to the **TypeCode** contained in the **DynAny** or, if the **DynAny** has components, is equivalent to the **TypeCode** of the **DynAny** at the current position.

The **get_dyn_any** and **insert_dyn_any** operations are provided to deal with **any** values that contain another **any**.

Calling an insert or get operation leaves the current position unchanged.

These operations are necessary to handle basic **DynAny** objects but are also helpful to handle constructed **DynAny** objects. Inserting a basic data type value into a constructed **DynAny** object implies initializing the current component of the constructed data value associated with the **DynAny** object. For example, invoking **insert_boolean** on a **DynStruct** implies inserting a boolean data value at the current position of the associated struct data value. If **dyn_construct** points to a constructed **DynAny** object, then:

```
result = dyn_construct->get_boolean();
```

has the same effect as:

```
DynamicAny::DynAny_var temp =
   dyn_construct->current_component();
result = temp->get_boolean();
```

Calling an insert or get operation on a **DynAny** whose current component itself has components raises TypeMismatch.

In addition, availability of these operations enable the traversal of **any**s associated with sequences of basic data types without the need to generate a **DynAny** object for each element in the sequence.

9.2.3.9 Iterating through components of a DynAny

The **DynAny** interface allows a client to iterate through the components of the values pointed to by **DynStruct**, **DynSequence**, **DynArray**, **DynUnion**, **DynAny**, and **DynValue** objects.

As mentioned previously, a **DynAny** object may be seen as an ordered collection of components, together with a current position.

boolean seek(in long index);

The **seek** operation sets the current position to **index**. The current position is indexed 0 to n−1, that is, index zero corresponds to the first component. The operation returns true if the resulting current position indicates a component of the **DynAny** and false if **index** indicates a position that does not correspond to a component.

Calling **seek** with a negative index is legal. It sets the current position to −1 to indicate no component and returns false. Passing a non-negative index value for a **DynAny** that does not have a component at the corresponding position sets the current position to −1 and returns false.

void rewind();

The **rewind** operation is equivalent to calling **seek(0);**

boolean next();

The **next** operation advances the current position to the next component. The operation returns true while the resulting current position indicates a component, false otherwise. A false return value leaves the current position at −1. Invoking **next** on a **DynAny** without components leaves the current position at −1 and returns false.

unsigned long component_count();

The **component_count** operation returns the number of components of a **DynAny**. For a **DynAny** without components, it returns zero. The operation only counts the components at the top level. For example, if **component_count** is invoked on a **DynStruct** with a single member, the return value is 1, irrespective of the type of the member.

For sequences, the operation returns the current number of elements. For structures, exceptions, and valuetypes, the operation returns the number of members. For arrays, the operation returns the number of elements. For unions, the operation returns 2 if the discriminator indicates that a named member is active; otherwise, it returns 1. For **DynFixed** and **DynEnum**, the operation returns zero.

```
DynAny current_component() raises(TypeMismatch);
```

The **current_component** operation returns the **DynAny** for the component at the current position. It does not advance the current position, so repeated calls to **current_component** without an intervening call to **rewind**, **next**, or **seek** return the same component.

The returned **DynAny** object reference can be used to get/set the value of the current component. If the current component represents a complex type, the returned reference can be narrowed based on the **TypeCode** to get the interface corresponding to the to the complex type.

Calling **current_component** on a **DynAny** that cannot have components, such as a **DynEnum** or an empty exception, raises TypeMismatch. Calling **current_component** on a **DynAny** whose current position is –1 returns a nil reference.

The iteration operations, together with **current_component**, can be used to dynamically compose an **any** value. After creating a dynamic any, such as a **DynStruct**, **current_component** and **next** can be used to initialize all the components of the value. Once the dynamic value is completely initialized, **to_any** creates the corresponding **any** value.

9.2.4 The DynFixed interface

DynFixed objects are associated with values of the IDL **fixed** type.

```
interface DynFixed : DynAny {
    string get_value();
    boolean set_value(in string val)
        raises (TypeMismatch, InvalidValue);
};
```

Because IDL does not have a generic type that can represent fixed types with arbitrary number of digits and arbitrary scale, the operations use the IDL **string** type.

The **get_value** operation returns the value of a **DynFixed**.

The **set_value** operation sets the value of the **DynFixed**. The **val** string must contain a **fixed** string constant in the same format as used for IDL fixed-point literals. However, the trailing **d** or **D** is optional. If **val** contains a value whose scale exceeds that of the **DynFixed** or is not initialized, the operation raises InvalidValue. The return value is true if **val** can be represented as the **DynFixed** without loss of precision. If **val** has more fractional digits than can be represented in the **DynFixed**, fractional digits are truncated and the return value is false. If **val** does not contain a valid fixed-point literal or contains extraneous characters other than leading or trailing white space, the operation raises TypeMismatch.

9.2.5 The DynEnum interface

DynEnum objects are associated with enumerated values.

```
interface DynEnum : DynAny {
    string get_as_string();
    void set_as_string(in string value) raises(InvalidValue);
    unsigned long get_as_ulong();
    void set_as_ulong(in unsigned long value) raises(InvalidValue);
};
```

The **get_as_string** operation returns the value of the **DynEnum** as an IDL identifier.

The **set_as_string** operation sets the value of the **DynEnum** to the enumerated value whose IDL identifier is passed in the **value** parameter. If **value** contains a string that is not a valid IDL identifier for the corresponding enumerated type, the operation raises InvalidValue.

The **get_as_ulong** operation returns the value of the **DynEnum** as the enumerated value's ordinal value. Enumerators have ordinal values 0 to n–1, as they appear from left to right in the corresponding IDL definition.

The **set_as_ulong** operation sets the value of the **DynEnum** as the enumerated value's ordinal value. If **value** contains a value that is outside the range of ordinal values for the corresponding enumerated type, the operation raises InvalidValue.

The current position of a **DynEnum** is always –1.

9.2.6 The DynStruct interface

DynStruct objects are associated with struct values and exception values.

```
typedef string FieldName;

struct NameValuePair {
    FieldName id;
    any value;
};
typedef sequence<NameValuePair> NameValuePairSeq;

struct NameDynAnyPair {
    FieldName id;
    DynAny value;
};
typedef sequence<NameDynAnyPair> NameDynAnyPairSeq;
```

```
interface DynStruct : DynAny {
    FieldName current_member_name()
        raises(TypeMismatch, InvalidValue);
    CORBA::TCKind current_member_kind()
        raises(TypeMismatch, InvalidValue);
    NameValuePairSeq get_members();
    void set_members(in NameValuePairSeq value)
        raises(TypeMismatch, InvalidValue);
    NameDynAnyPairSeq get_members_as_dyn_any();
    void set_members_as_dyn_any(in NameDynAnyPairSeq value)
        raises(TypeMismatch, InvalidValue);
};
```

```
    FieldName current_member_name()
        raises(TypeMismatch, InvalidValue);
```

The **current_member_name** operation returns the name of the member at the current position. If the **DynStruct** represents an empty exception, the operation raises TypeMismatch. If the current position does not indicate a member, the operation raises InvalidValue.

This operation may return an empty string since the **TypeCode** of the value being manipulated may not contain the names of members.

```
    CORBA::TCKind current_member_kind()
        raises(TypeMismatch, InvalidValue);
```

current_member_kind returns the **TCKind** associated with the member at the current position. If the **DynStruct** represents an empty exception, the operation raises TypeMismatch. If the current position does not indicate a member, the operation raises InvalidValue.

```
    NameValuePairSeq get_members();
```

The **get_members** operation returns a sequence of name/value pairs describing the name and the value of each member in the struct associated with a **DynStruct** object. The sequence contains members in the same order as the declaration order of members as indicated by the **DynStruct**'s **TypeCode**. The current position is not affected. The member names in the returned sequence will be empty strings if the **DynStruct**'s **TypeCode** does not contain member names.

```
    void set_members(in NameValuePairSeq value)
        raises(TypeMismatch, InvalidValue);
```

The **set_members** operation initializes the struct data value associated with a **DynStruct** object from a sequence of name value pairs. The operation sets the current position to zero if the passed sequences has non-zero length; otherwise, if an empty sequence is passed, the current position is set to –1.

Members must appear in the **NameValuePairSeq** in the order in which they appear in the IDL specification of the struct. If one or more sequence elements have a type that is not equivalent to the **TypeCode** of the corresponding member, the operation raises

TypeMismatch. If the passed sequence has a number of elements that disagrees with the number of members as indicated by the **DynStruct**'s **TypeCode**, the operation raises InvalidValue.

If member names are supplied in the passed sequence, they must either match the corresponding member name in the **DynStruct**'s **TypeCode** or must be empty strings, otherwise, the operation raises TypeMismatch. Members must be supplied in the same order as indicated by the **DynStruct**'s **TypeCode**. (The operation makes no attempt to assign member values based on member names.)

The **get_members_as_dyn_any** and **set_members_as_dyn_any** operations have the same semantics as their **Any** counterparts, but accept and return values of type **DynAny** instead of **Any**.

DynStruct objects can also be used for handling exception values. In that case, members of the exceptions are handled in the same way as members of a struct.

9.2.7 The DynUnion interface

DynUnion objects are associated with unions.

```
interface DynUnion : DynAny {
    DynAny get_discriminator();
    void set_discriminator(in DynAny d)
        raises(TypeMismatch);
    void set_to_default_member()
        raises(TypeMismatch);
    void set_to_no_active_member()
        raises(TypeMismatch);
    boolean has_no_active_member()
        raises(InvalidValue);
    CORBA::TCKind discriminator_kind();
    DynAny member()
        raises(InvalidValue);
    FieldName member_name()
        raises(InvalidValue);
    CORBA::TCKind member_kind()
        raises(InvalidValue);
};
```

The **DynUnion** interface allows for the insertion/extraction of an OMG IDL union type into/from a **DynUnion** object.

A union can have only two valid current positions: zero, which denotes the discriminator, and one, which denotes the active member. The **component_count** value for a union depends on the current discriminator: it is 2 for a union whose discriminator indicates a named member, and 1 otherwise.

```
DynAny get_discriminator()
    raises(InvalidValue);
```

The **get_discriminator** operation returns the current discriminator value of the **DynUnion**.

```
void set_discriminator(in DynAny d)
    raises(TypeMismatch);
```

The **set_discriminator** operation sets the discriminator of the **DynUnion** to the specified value. If the **TypeCode** of the **d** parameter is not equivalent to the **TypeCode** of the union's discriminator, the operation raises TypeMismatch.

Setting the discriminator to a value that is consistent with the currently active union member does not affect the currently active member. Setting the discriminator to a value that is inconsistent with the currently active member deactivates the member and activates the member that is consistent with the new discriminator value (if there is a member for that value) by initializing the member to its default value.

Setting the discriminator of a union sets the current position to 0 if the discriminator value indicates a non-existent union member (**has_no_active_member** returns true in this case). Otherwise, if the discriminator value indicates a named union member, the current position is set to 1 (**has_no_active_member** returns false and **component_count** returns 2 in this case).

```
void set_to_default_member()
    raises(TypeMismatch);
```

The **set_to_default_member** operation sets the discriminator to a value that is consistent with the value of the **default** case of a union; it sets the current position to zero and causes **component_count** to return 2. Calling **set_to_default_member** on a union that does not have an explicit **default** case raises TypeMismatch.

```
void set_to_no_active_member()
    raises(TypeMismatch);
```

The **set_to_no_active_member** operation sets the discriminator to a value that does not correspond to any of the union's case labels; it sets the current position to zero and causes **component_count** to return 1. Calling **set_to_no_active_member** on a union that has an explicit **default** case or on a union that uses the entire range of discriminator values for explicit **case** labels raises TypeMismatch.

```
boolean has_no_active_member();
```

The **has_no_active_member** operation returns true if the union has no active member (that is, the union's value consists solely of its discriminator because the discriminator has a value that is not listed as an explicit **case** label). Calling this operation on a union that has a **default** case returns false. Calling this operation on a union that uses the entire range of discriminator values for explicit **case** labels returns false.

```
CORBA::TCKind discriminator_kind();
```

The **discriminator_kind** operation returns the **TCKind** value of the discriminator's **TypeCode**.

```
CORBA::TCKind member_kind()
    raises(InvalidValue);
```

The **member_kind** operation returns the **TCKind** value of the currently active member's **TypeCode**. Calling this operation on a union that does not have a currently active member raises InvalidValue.

```
DynAny member()
    raises(InvalidValue);
```

The **member** operation returns the currently active member. If the union has no active member, the operation raises InvalidValue. Note that the returned reference remains valid only for as long as the currently active member does not change. Using the returned reference beyond the life time of the currently active member raises OBJECT_NOT_EXIST.

```
FieldName member_name()
    raises(InvalidValue);
```

The **member_name** operation returns the name of the currently active member. If the union's **TypeCode** does not contain a member name for the currently active member, the operation returns an empty string. Calling **member_name** on a union without an active member raises InvalidValue.

```
CORBA::TCKind member_kind()
    raises(InvalidValue);
```

The **member_kind** operation returns the **TCKind** value of the **TypeCode** of the currently active member. If the union has no active member, the operation raises InvalidValue.

9.2.8 The DynSequence interface

DynSequence objects are associated with sequences.

```
typedef sequence<any> AnySeq;
typedef sequence<DynAny> DynAnySeq;

interface DynSequence : DynAny {
    unsigned long get_length();
    void set_length(in unsigned long len)
        raises(InvalidValue);
    AnySeq get_elements();
```

```
        void set_elements(in AnySeq value)
            raises(TypeMismatch, InvalidValue);
        DynAnySeq get_elements_as_dyn_any();
        void set_elements_as_dyn_any(in DynAnySeq value)
            raises(TypeMismatch, InvalidValue);
};

    unsigned long get_length();
```

The **get_length** operation returns the current length of the sequence.

```
    void set_length(in unsigned long len)
        raises(TypeMismatch, InvalidValue);
```

The **set_length** operation sets the length of the sequence. Increasing the length of a sequence adds new elements at the tail without affecting the values of already existing elements. Newly added elements are default-initialized.

Increasing the length of a sequence sets the current position to the first newly-added element if the previous current position was –1. Otherwise, if the previous current position was not –1, the current position is not affected.

Increasing the length of a bounded sequence to a value larger than the bound raises InvalidValue.

Decreasing the length of a sequence removes elements from the tail without affecting the value of those elements that remain. The new current position after decreasing the length of a sequence is determined as follows:

- If the length of the sequence is set to zero, the current position is set to –1.
- If the current position is –1 before decreasing the length, it remains at –1.
- If the current position indicates a valid element and that element is not removed when the length is decreased, the current position remains unaffected.
- If the current position indicates a valid element and that element is removed, the current position is set to –1.

```
    DynAnySeq get_elements();
```

The **get_elements** operation returns the elements of the sequence.

```
    void set_elements(in AnySeq value)
        raises(TypeMismatch, InvalidValue);
```

The **set_elements** operation sets the elements of a sequence. The length of the **DynSequence** is set to the length of **value**. The current position is set to zero if **value** has non-zero length and to –1 if **value** is a zero-length sequence.

If value contains one or more elements whose **TypeCode** is not equivalent to the element **TypeCode** of the **DynSequence**, the operation raises TypeMismatch. If the length of value exceeds the bound of a bounded sequence, the operation raises InvalidValue.

The **get_elements_as_dyn_any** and **set_elements_as_dyn_any** operations have the same semantics, but accept and return values of type **DynAny** instead of **Any**.

9.2.9 The DynArray interface

DynArray objects are associated with arrays.

```
interface DynArray : DynAny {
    AnySeq get_elements();
    void set_elements(in AnySeq value)
        raises(TypeMismatch, InvalidValue);
    DynAnySeq get_elements_as_dyn_any();
    void set_elements_as_dyn_any(in DynAnySeq value)
        raises(TypeMismatch, InvalidValue);
};
```

```
    DynAnySeq get_elements();
```

The **get_elements** operation returns the elements of the **DynArray**.

```
    void set_elements(in DynAnySeq value)
        raises(TypeMismatch, InvalidValue);
```

The **set_elements** operation sets the **DynArray** to contain the passed elements. If the sequence does not contain the same number of elements as the array dimension, the operation raises InvalidValue. If one or more elements have a type that is inconsistent with the **DynArray**'s **TypeCode**, the operation raises TypeMismatch.

The **get_elements_as_dyn_any** and **set_elements_as_dyn_any** operations have the same semantics as their **Any** counterparts, but accept and return values of type **DynAny** instead of **Any**.

Note that the dimension of the array is contained in the **TypeCode** which is accessible through the **type** attribute. It can also be obtained by calling the **component_count** operation.

9.2.10 The DynValue interface

DynValue objects are associated with value types.

```
interface DynValue : DynAny {
    FieldName current_member_name()
        raises(TypeMismatch, InvalidValue);
    CORBA::TCKind current_member_kind()
        raises(TypeMismatch, InvalidValue);
```

```
    NameValuePairSeq get_members();
    void set_members(in NameValuePairSeq value)
        raises(TypeMismatch, InvalidValue);
    NameDynAnyPairSeq get_members_as_dyn_any();
    void set_members_as_dyn_any(in NameDynAnyPairSeq value)
        raises(TypeMismatch, InvalidValue);
};
```

Operations on the **DynValue** interface have semantics as for **DynStruct**.

9.3 Usage in C++ Language

9.3.1 Dynamic creation of CORBA::Any values

9.3.1.1 Creating an any which contains a struct

Consider the following IDL definition:

```
// IDL
struct MyStruct {
    long member1;
    boolean member2;
};
```

The following example illustrates how a **CORBA::Any** value may be constructed on the fly containing a value of type **MyStruct**:

```
// C++
CORBA::ORB_var orb = ...;
DynamicAny::DynAnyFactory_var dafact
   = orb->resolve_initial_references("DynAnyFactory");
CORBA::StructMemberSeq mems(2);
CORBA::Any_var result;
CORBA::Long    value1 = 99;
CORBA::Boolean value2 = 1;
mems.length(2);
mems[0].name = CORBA::string_dup("member1");
mems[0].type = CORBA::TypeCode::_duplicate(CORBA::_tc_long);
mems[1].name = CORBA::string_dup("member2");
mems[1].type
   = CORBA::TypeCode::_duplicate(CORBA::_tc_boolean);

CORBA::TypeCode_var new_tc = orb->create_struct_tc(
         "IDL:MyStruct:1.0",
         "MyStruct",
         mems
);
```

```
// Construct the DynStruct object. Values for members are
// the value1 and value2 variables

DynamicAny::DynAny_ptr dyn_any
   = dafact->create_dyn_any(new_tc);
DynamicAny::DynStruct_ptr dyn_struct
   = DynamicAny::DynStruct::_narrow(dyn_any);
CORBA::release(dyn_any);
dyn_struct->insert_long(value1);

dyn_struct->next();
dyn_struct->insert_boolean(value2);
result = dyn_struct->to_any();
dyn_struct->destroy();
CORBA::release(dyn_struct);
```

9.3.2 Dynamic interpretation of CORBA::Any values

9.3.2.1 Filtering of events

Suppose there is a CORBA object which receives events and prints all those events which correspond to a data structure containing a member called **`is_urgent`** whose value is true.

The following fragment of code corresponds to a method which determines if an event should be printed or not. Note that the program allows several struct events to be filtered with respect to some common member.

```
// C++
CORBA::Boolean Tester::eval_filter(
   DynamicAny::DynAnyFactory_ptr dafact,
   const CORBA::Any & event
)
{
   CORBA::Boolean success = FALSE;
   DynamicAny::DynAny_var;
   try {
      // First, convert the event to a DynAny.
      // Then attempt to narrow it to a DynStruct.
      // The _narrow only returns a reference
      // if the event is a struct.
```

```
            dyn_var = dafact->create_dyn_any(event);
            DynamicAny::DynStruct_var dyn_struct
               = DynamicAny::DynStruct::_narrow(dyn_any);
            if (!CORBA::is_nil(dyn_struct)) {
               CORBA::Boolean found = FALSE;
               do {
                  CORBA::String_var member_name
                     = dyn_struct->current_member_name();
                  found = (strcmp(member_name, "is_urgent") == 0);
               } while (!found && dyn_struct->next());
               if (found) {
                  // We only create a DynAny object for the member
                  // we were looking for:
                  DynamicAny::DynAny_var dyn_member
                     = dyn_struct->current_component();
                  success = dyn_member->get_boolean();
               }
            }
         }
         catch(...) {};
         if (!CORBA::is_nil(dyn_var))
            dyn_var->destroy();
         return success;
      }
```

The Interface Repository 10

The Interface Repository chapter has been updated based on CORE changes from ptc/98-09-04 and the Object by Value documents (orbos/98-01-18 and ptc/98-07-06).

Contents

This chapter contains the following sections.

10.1 Overview

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.

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).
- Assist in checking the correctness of interface inheritance graphs.
- 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.
- Provide components of a CASE environment (for example, an interface browser).
- Provide interface information to language bindings (such as a compiler).
- Provide components of end-user environments (for example, a menu bar constructor).

The complete OMG IDL specification for the Interface Repository is in Section 10.8, "OMG IDL for Interface Repository," on page 10-56; however, fragments of the specification are used throughout this chapter as necessary.

10.2 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 and value 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 consists of a set of *interface repository 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,
- an object implementation (such as an OODB) prefers to provide its own type information, or
- 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 10-1 on page 10-4, 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.

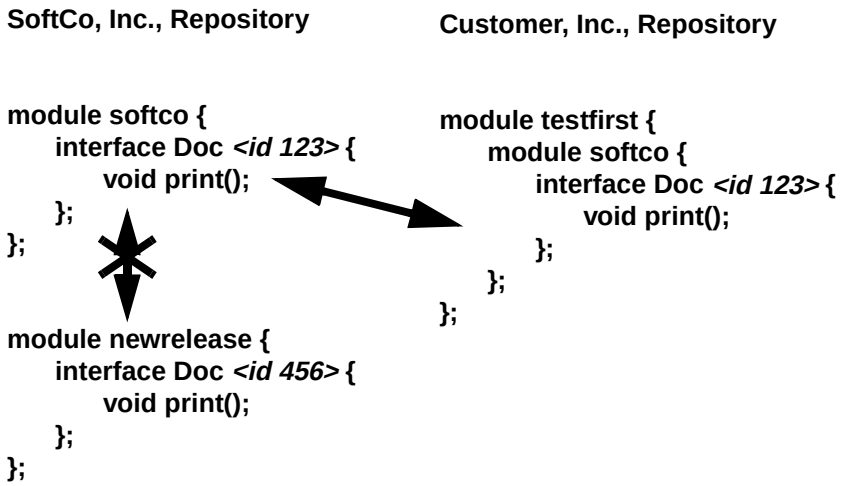


Figure 10-1 Using Repository IDs to establish correspondence between repositories

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).

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.

10.3 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.

10.3.1 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.

10.4 Basics

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

- Names and Identifiers
- Types and TypeCodes
- Interface Repository Objects
- Structure and Navigation of the Interface Repository

10.4.1 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, struct, union, exception and value type definitions are considered implicit modules.

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

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

10.4.2 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 Section 10.7, “TypeCodes,” on page 10-48 for more information on the internal structure of TypeCodes.

10.4.3 Interface Repository Objects

Information about the entities that are managed in an Interface Repository is maintained as a collection of *interface repository objects* of the following types:

- **Repository**: the top-level module for the repository name space; it contains constants, typedefs, exceptions, interface or value type definitions, and modules.
- **ModuleDef**: a logical grouping of interfaces and value types; it contains constants, typedefs, exceptions, interface or value type definitions, and other modules.
- **InterfaceDef**: an interface definition; it contains lists of constants, types, exceptions, operations, and attributes.
- **ValueDef**: a value type definition which contains lists of constants, types, exceptions, operations, attributes and members
- **ValueBoxDef**: the definition of a boxed value type.
- **ValueMemberDef**: the definition of a member of the value type.
- **AttributeDef**: the definition of an attribute of the interface or value type.
- **OperationDef**: the definition of an operation of the interface or value type; it contains lists of parameters and exceptions raised by this operation.
- **TypedefDef**: base interface for definitions of named types that are not interfaces or value types.
- **ConstantDef**: the definition of a named constant.
- **ExceptionDef**: the definition of an exception that can be raised by an operation.

The interface specifications for each *interface repository object* lists the attributes maintained by that object (see Section 10.5, “Interface Repository Interfaces,” on page 10-9). 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 shall do so by deriving new interfaces, not by modifying the standard interfaces.

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

10.4.4 Structure and Navigation of the Interface Repository

The definitions in the Interface Repository are structured as a set of *interface repository objects*. These objects are structured the same way definitions are structured—some objects (definitions) "contain" other objects.

The containment relationships for the *interface repository objects* types in the Interface Repository are shown in Figure 10-2

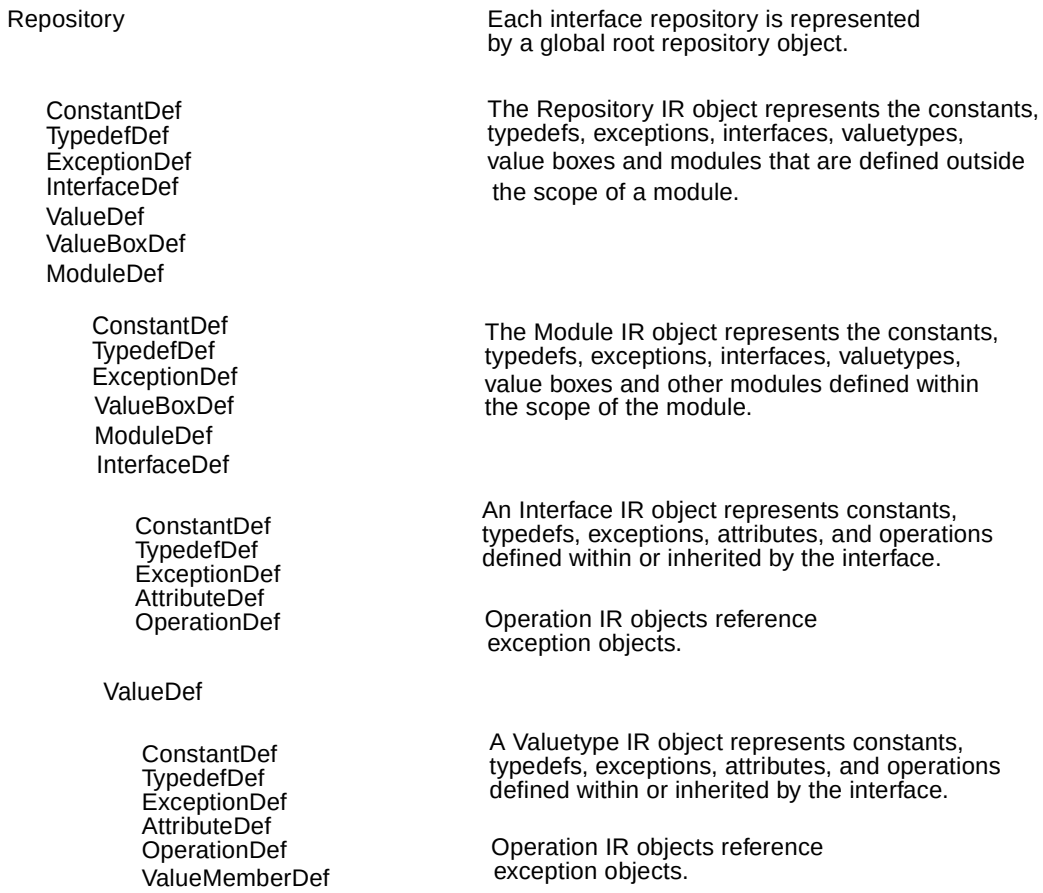


Figure 10-2 Interface Repository Object Containment

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

1. Obtaining an **InterfaceDef** object directly from the ORB.
2. Navigating through the module name space using a sequence of names.
3. 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.

Analogous operations are provided for manipulating value types.

10.5 Interface Repository Interfaces

Several interfaces are used as *base interfaces* for objects in the IR. These *base interface*s are not instantiable.

A common set of operations is used to locate objects within the Interface Repository. These operations are defined in the 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 *base interfaces* **IRObject**, **Contained**, **Container**, **IDLType**, and **TypedefDef** are not instantiable.

All string data in the Interface Repository are encoded as defined by the ISO 8859-1 coded character set.

Interface Repository operations indicate error conditions using the system exceptions BAD_PARAM and BAD_INV_ORDER with specific minor codes. Which specific operations raise these exceptions are documented in the description of the operations. For a description of how these minor codes are encoded in the ex_body of standard

exceptions see Section 3.17, “Standard Exceptions,” on page 3-51 and Section 3.17.2, “Standard Minor Exception Codes,” on page 3-58. The exceptions and minor codes that are used by Interface Repository interfaces are as follows:

Table 10-1 Standard Exceptions used by the Interface Repository Operations

Exception	Minor Code	Explanation
BAD_PARAM	2	RID is already defined in IFR
	3	Name already used in the context in IFR
	4	Target is not a valid container
	5	Name clash in inherited context
BAD_INV_ORDER	1	Dependency exists in IFR preventing destruction of this object
	2	Attempt to destroy indestructible objects in IFR

10.5.1 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,
        dk_Wstring, dk_Fixed,
        dk_Value, dk_ValueBox, dk_ValueMember,
        dk_Native
    };
};
```

Identifiers are the simple names that identify modules, interfaces, value types, value members, value boxes, constants, typedefs, exceptions, attributes, operations and native types. 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**, **ValueDef** or **OperationDef**.

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. A *relative* **ScopedName** 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, value type, value member, value box, native type, 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.

10.5.2 IRObject

The *base interface* **IRObject** 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 ();
    };
};
```

10.5.2.1 Read Interface

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

10.5.2.2 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. If **destroy** is invoked on a **Repository** or on a **PrimitiveDef** then the BAD_INV_ORDER exception is raised with minor value 2. Implementations may vary in their handling of references to an object that is being destroyed, but the Repository should not be left in an incoherent state. Attempt to destroy an object that would leave the repository in an incoherent state shall cause BAD_INV_ORDER exception to be raised with the minor code 1.

10.5.3 Contained

The *base interface* **Contained** 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, WstringDef, FixedDef 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
        );
    };
};
```

10.5.3.1 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 will require mechanisms 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** or **ValueDef** 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 **within** operation returns the list of objects that contain the object. If the object is an interface or module it can be contained only by the object that defines it. Other objects can be contained by the objects that define them and by the objects that inherit them.

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 name of the structure returned is provided with the returned structure. The **kind** field of the returned **Description** struct shall give the **DefinitionKind** for the most derived type of the object. For example, if the **describe** operation is invoked on an attribute object, the **kind** field contains **dk_Attribute name** field contains "AttributeDescription" and the **value** field contains an **any**, which contains the **AttributeDescription** structure. The **kind** field in this must contain **dk_attribute** and not the kind of any **IRObject** from which the **attribute** object is derived. For example returning **dk_all** would be an error.

10.5.3.2 Write Interface

Setting the **id** attribute changes the global identity of this definition. A BAD_PARAM exception is raised with minor code 2 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**. A BAD_PARAM exception is raised with minor code 1 if an object with the specified **name** attribute already exists within 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** must satisfy the following conditions:

- It must be in the same **Repository.** If it is not, then BAD_PARAM exception is raised with minor code 4.
- It must be capable of containing this object's type (see Section 10.4.4, "Structure and Navigation of the Interface Repository," on page 10-8). If it is not, then BAD_PARAM exception is raised with minor code 4.
- It must not already contain an object with this object's name (unless multiple versions are supported by the IR). If this condition is not satisfied, then BAD_PARAM exception is raised with minor code 3.

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.

10.5.4 Container

The *base interface* **Container** 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,
                in boolean              is_abstract
            );

            ExceptionDef create_exception(
                in RepositoryId         id,
                in Identifier           name,
                in VersionSpec          version,
                in StructMemberSeq      members
            );

            ValueDef create_value(
                in RepositoryId         id,
                in Identifier           name,
                in VersionSpec          version,
                in boolean              is_custom,
                in boolean              is_abstract,
                in ValueDef             base_value,
                in boolean              is_truncatable,
                in ValueDefSeq          abstract_base_values,
                in InterfaceDefSeq      supported_interfaces,
                in InitializerSeq       initializers
            );

            ValueBoxDef create_value_box(
                in RepositoryId         id,
                in Identifier           name,
                in VersionSpec          version,
                in IDLType              original_type_def
            );

            NativeDef create_native(
                in RepositoryId         id,
                in Identifier           name,
                in VersionSpec          version
            );
        };
    };
```

10.5.4.1 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 and value types within a specific module, and so on.

limit_type	If **limit_type** is set to **dk_all** "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** "AttributeDef".
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. Use of values of **levels_to_search** of 0 or of negative numbers other than -1 is undefined.

search_name	Specified 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. Use of values of **levels_to_search** of 0 or of negative numbers other than -1 is undefined.

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.

10.5.4.2 Write Interface

The **Container** interface provides operations to create **ModuleDef**s, **ConstantDef**s, **StructDef**s, **UnionDef**s, **EnumDef**s, **AliasDef**s, **InterfaceDef**s, **ValueDef**s **ValueBoxDef**s and **NativeDef**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. A BAD_PARAM exception is raised with minor code 2 if an object with the specified **id** already exists in the **Repository**. A BAD_PARAM exception with minor code 3 is raised if the specified **name** already exists within this **Container and** multiple versions are not supported.Certain interfaces derived from **Container** may restrict the types of definitions that they may contain. Any **create_<type>** operation that would insert a definition that is not allowed by a Container will raise the BAD_PARAM exception with minor code 4.

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 10-23 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 10-24 for more information.

The **create_enum** operation returns a new **EnumDef** with the specified **members**. See "EnumDef" on page 10-25 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.

The **create_value** operation returns a new empty **ValueDef** with the specified base interfaces and values (**base_value**, **supported_interfaces**, and **abstract_base_values**) as well as the other information describing the new values characteristics (**is_custom**, **is_abstract**, **is_truncatable**, and **initializers**). Type, exception, and constant definitions can be added using **Container::create_<type>** operations on the new **ValueDef**. **OperationDefs** can be added using **ValueDef::create_operation** and **AttributeDefs** can be added using **Value::create_attribute**. Definitions can also be added using the **Contained::move** operation.

The **create_value_box** operation returns a new **ValueBoxDef** with the specified **original_type_def**.

The **create_exception** operation returns a new **ExceptionDef** with the specified members. The **type** member of the **StructMember** structures should be set to **TC_void**.

The **create_native** operation returns a new **NativeDef** with the specified **name**.

10.5.5 IDLType

The *base interface* **IDLType** is 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**.

10.5.6 Repository

Repository is an interface that provides global access to the Interface Repository. The **Repository** object can contain constants, typedefs, exceptions, interfaces, value types, value boxes, native types 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**.

Since **Repository** derives only from **Container** and not from **Contained**, it does not have a **RepositoryId** associated with it. By default it is deemed to have the **RepositoryId** "" (the empty string) for purposes of assigning a value to the

defined_in field of the **description** structure of **ModuleDef**, **InterfaceDef**, **ValueDef**, **ValueBoxDef, TypedefDef**, **ExceptionDef** and **ConstantDef** that are contained immediately in the Repository object.

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);

        TypeCode get_canonical_typecode(in TypeCode tc);

        PrimitiveDef get_primitive (in PrimitiveKind kind);

        // write interface

        StringDef create_string (in unsigned long bound);

        WstringDef create_wstring(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
        );

        FixedDef create_fixed(
            in unsigned short digits,
            in short scale
        );
    };
};
```

10.5.6.1 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_canonical_typecode** operation looks up the **TypeCode** in the Interface Repository and returns an equivalent **TypeCode** that includes all **repository ids**, **names**, and **member_names**. If the top level **TypeCode** does not contain a **RepositoryId**, such as array and sequence **TypeCodes**, or **TypeCodes** from older ORBs, or if it contains a **RepositoryId** that is not found in the target **Repository**, then a new **TypeCode** is constructed by recursively calling **get_canonical_typecode** on each member **TypeCode** of the original **TypeCode**.

The **get_primitive** operation returns a reference to a **PrimitiveDef** (see Section 10.5.14, “PrimitiveDef,” on page 10-26) with the specified **kind** attribute. All **PrimitiveDef**s are immutable and are owned by the **Repository**.

10.5.6.2 Write Interface

The five **create_<type>** operations that create new IR 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.

1. 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.
2. The **create_wstring** operation returns a new **WstringDef** with the specified **bound**, which must be non-zero. The **get_primitive** operation is used for unbounded strings.
3. The **create_sequence** operation returns a new **SequenceDef** with the specified **bound** and **element_type**.
4. The **create_array** operation returns a new **ArrayDef** with the specified **length** and **element_type**.
5. The **create_fixed** operation returns a new **FixedDef** with the specified number of digits and scale. The number of digits must be from 1 to 31, inclusive.

10.5.7 ModuleDef

A **ModuleDef** can contain constants, typedefs, exceptions, interfaces, value types, value boxes, native types 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**.

10.5.8 ConstantDef

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;
    };
};
```

10.5.8.1 Read Interface

The **type** attribute specifies the **TypeCode** describing the type of the constant. The type of a constant must be one of the primitive types allowed in constant declarations (see Section 3.9, “Constant Declaration,” on page 3-28). 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**.

10.5.8.2 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**.

10.5.9 TypedefDef

The *base interface* **TypedefDef** is inherited by 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**.

10.5.10 StructDef

A **StructDef** represents an OMG IDL structure definition. It can contain structs, unions, and enums.

```
module CORBA {

    struct StructMember {
        Identifier      name;
        TypeCode        type;
        IDLType         type_def;
    };

    typedef sequence <StructMember> StructMemberSeq;

    interface StructDef : TypedefDef, Container {
        attribute StructMemberSeq       members;
    };
};
```

10.5.10.1 Read Interface

The **members** attribute contains a description of each structure member.

The inherited **type** attribute is a **tk_struct TypeCode** describing the structure.

10.5.10.2 Write Interface

Setting the **members** attribute also updates the **type** attribute. When setting the **members** attribute, the **type** member of the **StructMember** structure should be set to **TC_void**.

A **StructDef** used as a **Container** may only contain **StructDef**, **UnionDef**, or **EnumDef** definitions.

10.5.11 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, Container {
        readonly attribute TypeCode   discriminator_type;
        attribute IDLType             discriminator_type_def;
        attribute UnionMemberSeq      members;
    };
};
```

10.5.11.1 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.

10.5.11.2 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 should be set to **TC_void**.

A **UnionDef** used as a **Container** may only contain **StructDef**, **UnionDef**, or **EnumDef** definitions.

10.5.12 EnumDef

An **EnumDef** represents an OMG IDL enumeration definition.

```
module CORBA {
    typedef sequence <Identifier> EnumMemberSeq;

    interface EnumDef : TypedefDef {
        attribute EnumMemberSeq     members;
    };
};
```

10.5.12.1 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.

10.5.12.2 Write Interface

Setting the **members** attribute also updates the **type** attribute.

10.5.13 AliasDef

An **AliasDef** represents an OMG IDL typedef that aliases another definition.

```
module CORBA {
    interface AliasDef : TypedefDef {
        attribute IDLType   original_type_def;
    };
};
```

10.5.13.1 Read Interface

The **original_type_def** attribute identifies the type being aliased.

The inherited **type** attribute is a **tk_alias TypeCode** describing the alias.

10.5.13.2 Write Interface

Setting the **original_type_def** attribute also updates the **type** attribute.

10.5.14 PrimitiveDef

A **PrimitiveDef** represents one of the OMG 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,
        pk_longlong, pk_ulonglong, pk_longdouble, pk_wchar, pk_wstring,
        pk_value_base
    };

    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**. A **PrimitiveDef** with **kind pk_value_base** represents the IDL type **ValueBase**.

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**.

10.5.15 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.

10.5.16 WstringDef

A **WstringDef** represents an IDL wide string. The unbounded wide string type is represented as a **PrimitiveDef**. As wide string types are anonymous, this interface is not derived from **TypedefDef** or **Contained.**

```
module CORBA {
    interface WstringDef  :  IDLType {
        attribute unsigned long  bound;
    };
};
```

The **bound** attribute specifies the maximum number of wide characters in a wide string, and must not be zero.

The inherited **type** attribute is a **tk_wstring TypeCode** describing the wide string.

10.5.17 FixedDef

A **FixedDef** represents an IDL fixed point type.

```
module CORBA {
    interface FixedDef  :  IDLType {
        attribute unsigned short digits;
        attribute short scale;
    };
};
```

The **digits** attribute specifies the total number of decimal digits in the number, and must be from 1 to 31, inclusive. The **scale** attribute specifies the position of the decimal point.

The inherited **type** attribute is a **tk_fixed TypeCode**, which describes a fixed-point decimal number.

10.5.18 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;
    };
};
```

10.5.18.1 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.

10.5.18.2 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.

10.5.19 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;
    };
};
```

10.5.19.1 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.

10.5.19.2 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.

10.5.20 ExceptionDef

An **ExceptionDef** represents an exception definition. It can contain structs, unions, and enums.

```
module CORBA {
    interface ExceptionDef : Contained, Container {
        readonly attribute TypeCode   type;
        attribute StructMemberSeq     members;
    };

    struct ExceptionDescription {
        Identifier        name;
        RepositoryId      id;
        RepositoryId      defined_in;
        VersionSpec       version;
        TypeCode          type;
    };
};
```

10.5.20.1 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**.

10.5.20.2 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**.

A **ExceptionDef** used as a **Container** may only contain **StructDef**, **UnionDef**, or **EnumDef** definitions.

10.5.21 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;
        };
    };
```

10.5.21.1 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.

The **describe** operation for an **AttributeDef** object returns an **AttributeDescription**.

10.5.21.2 Write Interface

Setting the **type_def** attribute also updates the **type** attribute.

10.5.22 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;
    };
};
```

10.5.22.1 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 **kind** attribute indicates whether the **OperationDef** represents an IDL operation (**OP_IDL**), or an accessor for a an IDL attribute (**OP_ATTR**). For an **OperationDef** representing an attribute accessor, the **name** parameter is generated by concatenating either "_get_" or "_set_" with the **name** attribute of the corresponding **AttributeDef**. Only the "_get_" accessor is provided for readonly attributes. A "_get_" accessor takes no parameters and its result type is the attribute type. A "_set_" accessor takes a single in parameter of the attribute type, and its result type is void. The **mode** attribute of accessor operations is **OP_NORMAL**. Accessor **OperationDef**s are contained in the same **OperationDef**s as their corresponding **AttributeDef**s.

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**.

10.5.22.2 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**.

10.5.23 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;
        attribute boolean               is_abstract;

        // 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;
            boolean                     is_abstract;
        };

        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;
        boolean             is_abstract;
    };
};
```

10.5.23.1 Read Interface

The **base_interfaces** attribute lists all the interfaces from which this interface inherits.

The **is_abstract** attribute is **TRUE** if the interface is an abstract interface type.

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 **operations** and **attributes** fields of the **FullInterfaceDescription** structure include descriptions of all of the operations and attributes in the transitive closure of the inheritance graph of the interface being described.

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.

10.5.23.2 Write Interface

Setting the **base_interfaces** attribute causes a BAD_PARAM exception with minor code 5 to be raised 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**. A BAD_PARAM exception with minor code 2 is raised if an object with the specified **id** already exists in the **Repository**. BAD_PARAM exception with minor code 3 is raised if an object with the same **name** already exists in 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**.

An **InterfaceDef** used as a **Container** may only contain **TypedefDef**, (including definitions derived from **TypedefDef**), **ConstantDef**, and **ExceptionDef** definitions.

10.5.24 ValueDef

A **ValueDef** object represents a value definition. It can contain constants, typedefs, exceptions, operations, and attributes.

```
module CORBA {
    interface ValueDef;
    typedef sequence <ValueDef> ValueDefSeq;

    struct Initializer {
        StructMemberSeq  members;
        Identifier       name;
    };

    typedef sequence<Initializer> InitializerSeq;

    typedef short Visibility;
        const Visibility PRIVATE_MEMBER = 0;
        const Visibility PUBLIC_MEMBER = 1;
```

```
struct ValueMember {
    Identifier              name;
    RepositoryId            id;
    RepositoryId            defined_in;
    VersionSpec             version;
    TypeCode                type;
    IDLType                 type_def;
    Visibility              access;
};

typedef sequence <ValueMember> ValueMemberSeq;

interface ValueMemberDef : Contained {
    readonly attribute TypeCode   type;
    attribute IDLType             type_def;
    attribute Visibility          access;
};

interface ValueDef : Container, Contained, IDLType {
    // read/write interface

    attribute InterfaceDefSeq  supported_interfaces;
    attribute InitializerSeq   initializers;
    attribute ValueDef         base_value;
    attribute ValueDefSeq      abstract_base_values;
    attribute boolean          is_abstract;
    attribute boolean          is_custom;
    attribute boolean          is_truncatable;

    // read interface
    boolean is_a(
        in RepositoryId        id
    );

    struct FullValueDescription {
        Identifier             name;
        RepositoryId           id;
        boolean                is_abstract;
        boolean                is_custom;
        RepositoryId           defined_in;
        VersionSpec            version;
        OpDescriptionSeq       operations;
        AttrDescriptionSeq     attributes;
        ValueMemberSeq         members;
        InitializerSeq         initializers;
        RepositoryIdSeq        supported_interfaces;
        RepositoryIdSeq        abstract_base_values;
        boolean                is_truncatable;
        RepositoryId           base_value;
        TypeCode               type;
    };
```

```
        FullValueDescription describe_value();

        ValueMemberDef create_value_member(
            in RepositoryId         id,
            in Identifier           name,
            in VersionSpec          version,
            in IDLType              type,
            in Visibility           access
        );

        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 ValueDescription {
        Identifier              name;
        RepositoryId            id;
        boolean                 is_abstract;
        boolean                 is_custom;
        RepositoryId            defined_in;
        VersionSpec             version;
        RepositoryIdSeq         supported_interfaces;
        RepositoryIdSeq         abstract_base_values;
        boolean                 is_truncatable;
        RepositoryId            base_value;
    };
};
```

10.5.24.1 Read Interface

The **supported_interfaces** attribute lists the interfaces which this value type supports.

The **initializers** attribute lists the initializers this value type supports.

The **base_value** attribute describes the value type from which this value inherits.

The **abstract_base_values** attribute lists the abstract value types from which this value inherits.

The **is_abstract** attribute is **TRUE** if the value is an abstract value type.

The **is_custom** attribute is **TRUE** if the value uses custom marshaling.

The **is_truncatable** attribute is **TRUE** if the value inherits "safely" (i.e., supports truncation) from another value.

The **is_a** operation returns **TRUE** if the value on which it is invoked either is identical to or inherits, directly or indirectly, from the interface or value identified by its **id** parameter. Otherwise it returns **FALSE**.

The **describe_value** operation returns a **FullValueDescription** describing the value, including its operations and attributes.

The inherited **describe** operation for an **ValueDef** returns an **ValueDescription**.

The inherited **contents** operation returns the list of constants, typedefs, and exceptions defined in this **ValueDef** and the list of attributes, operations and members either defined or inherited in this **ValueDef**. If the **exclude_inherited** parameter is set to **TRUE**, only attributes, operations and members defined within this value are returned. If the **exclude_inherited** parameter is set to **FALSE**, all attributes, operations and members are returned.

10.5.24.2 Write Interface

Setting the **supported_interfaces, base_value, or abstract_base_values** attribute causes a BAD_PARAM exception with minor code 5 to be raised if the **name** attribute of any object contained by this **ValueDef** conflicts with the **name** attribute of any object contained by any of the specified bases.

The **create_value_member** operation returns a new **ValueMemberDef** contained in the **ValueDef** on which it is invoked. The **id**, **name**, **version, type_def**, and **access** attributes are set as specified. The **type** attribute is also set. The **defined_in** attribute is initialized to identify the containing **ValueDef**. A BAD_PARAM exception with minor code 2 is raised if an object with the specified **id** already exists in the **Repository**. A BAD_PARAM exception with minor code 3 is raised if an object with the same **name** already exists in this **ValueDef**.

The **create_attribute** operation returns a new **AttributeDef** contained in the **ValueDef** 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 **ValueDef**. A BAD_PARAM exception with minor code 2 is raised if an object with the specified **id** already exists in the **Repository**. A BAD_PARAM exception with minor code 3 is raised if an object with the same **name** already exists in this **ValueDef**.

The **create_operation** operation returns a new **OperationDef** contained in the **ValueDef** 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 **ValueDef**. A BAD_PARAM exception with minor code 2 is raised if an object with the specified **id** already exists in the **Repository**. A BAD_PARAM exception with minor code 3 is raised if an object with the same **name** already exists in this **ValueDef**.

A **ValueDef** used as a **Container** may only contain **TypedefDef**, (including definitions derived from **TypedefDef**), **ConstantDef**, and **ExceptionDef** definitions.

10.5.25 ValueBoxDef

A **ValueBoxDef** object represents a value box definition. It merely identifies the IDL **type_def** that is being "boxed."

```
module CORBA {
    interface ValueBoxDef : TypedefDef {
        attribute IDLType original_type_def;
    };
};
```

10.5.25.1 Read Interface

The **original_type_def** attribute identifies the type being boxed.

The inherited **type** attribute is a **tk_value_box TypeCode** describing the value box.

10.5.25.2 Write Interface

Setting the **original_type_def** attribute also updates the **type** attribute.

10.5.26 NativeDef

A **NativeDef** object represents a native definition.

```
module CORBA {
    interface NativeDef : TypedefDef {};
};
```

The inherited **type** attribute is a **tk_native TypeCode** describing the native type.

10.6 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. Ensuring consistency of **RepositoryId**s with the IDL source or the IR contents is the responsibility of the programmer allocating **Repositoryid**s.

The format of the id is a short format name followed by a colon (":") followed by characters according to the format. This specification defines four formats: one derived from OMG IDL names, one that uses Java class names and Java serialization version UIDs, one that uses DCE UUIDs, and another intended for short-term use, such as in a development environment.

Since new repository ID formats may be added from time to time, compliant IDL compilers must accept any string value of the form

"<format>:<string>"

provided as the argument to the id pragma and use it as the repository ID. The OMG maintains a registry of allocated format identifiers. The **<format>** part of the ID may not contain a colon (:) character.

The version and prefix pragmas only affect default repository IDs that are generated by the IDL compiler using the IDL format.

10.6.1 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** consists 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.

10.6.2 RMI Hashed Format

The OMG IDL format defined above does not include any structural information. Identity of IDL types determined for this format depends upon the names used in the **RepositoryID** being correct. For interfaces, if stubs and skeletons are not actually in synch, even though the **RepositoryIds** report they are, the worst that can happen is that the result of an invocation is a BAD_OPERATION exception. With value types, these kinds of errors are more problematic. An inconsistency between the stub and skeleton marshaling/unmarshaling code can confuse the marshaling engine and may even corrupt memory and/or cause a crash failure.

The RMI Hashed format is used for Java RMI values mapped to IDL using the Java to IDL Mapping (see the Java/IDL Language Mapping document). It is computed based upon the structural information of the original Java definition. Whenever the Java definition changes, the hash function will (statistically) produce a hash code which is different from the previous one. When an ORB run time receives a **value** with a different hash from what is expected, it is free to raise a BAD_PARAM exception. It may also try to resolve the incompatibility by some means. If it is not successful, then it shall raise the BAD_PARAM exception.

An RMI Hashed **RepositoryId** consists of either three or four components, separated by colons:

RMI: <class name> : <hash code> [: <serialization version UID>]

The class name is a Java class name as returned by the **`getName`** method of **`java.lang.Class`**. Any characters not in *ISO Latin 1* are replaced by "**\U**" followed by the 4 hexadecimal characters (in upper case) representing the *Unicode* value.

For classes that do not implement **`java.io.Serializable`**, and for interfaces, the hash code is always zero, and the **RepositoryID** does not contain a *serial version UID*.

For classes that implement **`java.io.Externalizable`**, the hash code is always the *64-bit value 1*.

For classes that implement **`java.io.Serializable`** but not **`java.io.Externalizable`**, the hash code is a *64-bit hash of a stream of bytes*. An instance of **`java.lang.DataOutputStream`** is used to convert primitive data types to a sequence of bytes. The sequence of items in the stream is as follows:

1. The hash code of the superclass, written as a 64-bit long.
2. The value 1 if the class has no **`writeObject`** method, or the value 2 if the class has a **`writeObject`** method, written as a 32-bit integer.
3. For each field of the class that is mapped to IDL, sorted lexicographically by Java field name, in increasing order:
    a. Java field name, in *UTF encoding*

b. field descriptor, as defined by the *Java Virtual Machine Specification*, in *UTF encoding*

The *National Institute of Standards and Technology* (*NIST*) *Secure Hash Algorithm* (*SHA-1*) is executed on the stream of bytes produced by **DataOutputStream**, producing a *20 byte array of values, sha[0..19*]. The hash code is assembled from the *first 8 bytes* of this array as follows:

```
long hash = 0;
for (int i = 0; i < Math.min(8, sha.length); i++) {
   hash += (long)(sha[i] & 255) << (i * 8);
}
```

If the actual serialization version **UID** for the Java class differs from the hash code, a colon and the actual serialization version **UID** (transcribed as a 16 digit upper-case hex string) shall be appended to the **RepositoryId** after the hash code.

Examples for the valuetype **::foo::bar** would be

RMI:foo/bar;:1234567812345678
RMI:foo/bar;:1234567812345678:ABCD123456781234

An example of a Java array of valuetype **::foo::bar** would be

RMI:[Lfoo.bar;:1234567812345678:ABCD123456781234

For a Java class **x\u03bCy** which contains a Unicode character not in ISO Latin 1, an example **RepositoryId** is

RMI:foo.x\U03BCy:8765432187654321

A conforming implementation which uses this format shall implement the standard hash algorithm defined above.

10.6.3 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”.

10.6.4 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.

10.6.5 Pragma Directives for RepositoryId

Three pragma directives (id, prefix, and version), are specified to accommodate arbitrary **RepositoryId** formats and still support the OMG IDL **RepositoryId** format with minimal annotation. The prefix and version pragma directives apply only to the IDL format. An IDL compiler must interpret these annotations as specified. Conforming IDL compilers may support additional non-standard pragmas, but must not refuse to compile IDL source containing non-standard pragmas that are not understood by the compiler.

10.6.5.1 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 **<id>** must be a repository ID of the form described in Section 10.6, “RepositoryIds,” on page 10-39.

If an attempt is made to assign a repository ID to the same IDL construct a second time, a compile-time diagnostic shall be emitted, regardless of whether the second ID is in conflict or not:

```
interface A {};
#pragma ID A “IDL:A:1.1”
#pragma ID A “IDL:X:1.1”       // Compile-time error

interface B {};
#pragma ID B “IDL:BB:1.1”
#pragma ID B “IDL:BB:1.1”      // Compile-time error
```

It is also an error to apply an ID to a forward-declared interface and then later assign the same or a different ID to that interface.

10.6.5.2 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 RepositoryIds generated after the pragma until the end of the current scope is reached or another prefix pragma is encountered. An IDL file forms a scope for this purpose, so a prefix resets to the previous prefix at the end of the scope of an included file:

```
// A.idl
#pragma prefix “A”
interface A {};

// B.idl
#pragma prefix “B”
#include “A.idl”
interface B {};
```

The repository IDs for interfaces A and B in this case are:

```
IDL:A/A:1.0
IDL:B/B:1.0
```

Similarly, a prefix in an including file does not affect the prefix of an included file:

```
// C.idl
interface C {};

// D.idl
#pragma prefix “D”
#include “C.idl”
interface D {};
```

The repository IDs for interface C and D in this case are:

```
IDL:C:1.0
IDL:D/D:1.0
```

If an included file does not contain a #pragma prefix, the current prefix implicitly resets to the empty prefix:

```
// E.idl
interface E {};

// F.idl
module M {
  #include <E.idl>
};
```

The repository IDs for module M and interface E in this case are:

```
IDL:M:1.0
IDL:E:1.0
```

If a #include directive appears at non-global scope and the included file contains a prefix pragma, the included file's prefix takes precedence, for example:

```
// A.idl
#pragma prefix “A”
interface A {};
```

```
// B.idl
#pragma prefix “B”
module M {
#include “A.idl”
};
```

The repository ID for module M and interface A in this case are:

```
IDL:B/M:1.0
IDL:A/A:1.0
```

Attempts to assign a prefix to a forward-declared interface and a different prefix to that interface later result in a compile-time diagnostic:

```
#pragma prefix “A”
interface A;          // Forward decl.

#pragma prefix “B”
interface A;          // Compile-time error

#pragma prefix “C”
interface A {         // Compile-time error
    void op();
};
```

A prefix pragma of the form

```
#pragma prefix “”
```

resets the prefix to the empty string. For example:

```
#pragma prefix “X”
interface X {};
#pragma prefix “”
interface Y {};
```

The repository IDs for interface X and Y in this case are:

```
IDL:X/X:1.0
IDL:Y:1.0
```

If a specification contains both a prefix pragma and an ID or version pragma, the prefix pragma does not affect the repository ID for an ID pragma, but does affect the repository ID for a version pragma:

```
#pragma prefix “A”
interface A {};
interface B {};
interface C {};
#pragma ID B “IDL:myB:1.0”
#pragma version C 9.9
```

The repository IDs for this specification are

```
IDL:A/A:1.0
IDL:myB:1.0
IDL:A/C:9.9
```

A #pragma prefix must appear before the beginning of an IDL definition. Placing a #pragma prefix elsewhere has undefined behavior, for example:

```
interface Bar
    #pragma prefix “foo”    // Undefined behavior
    {
    // ...
};
```

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.”

10.6.5.3 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.

If an attempt is made to change the version of a repository ID that was specified with an ID pragma, a compliant compiler shall emit a diagnostic:

```
interface A {};
#pragma ID A “IDL:myA:1.1”
#pragma version A 9.9          // Compile-time error
```

If an attempt is made to assign a version to the same IDL construct a second time, a compile-time diagnostic shall be emitted, regardless of whether the second version is in conflict or not:

```
interface A {};
```

```
#pragma version A 1.1
#pragma version A 2.2        // Compile-time error

interface B {};
#pragma version B 1.1
#pragma version B 1.1        // Compile-time error
```

10.6.5.4 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::T1IDL:M1/T1:1.0

::M1::T2 DCE:d62207a2-011e-11ce-88b4-0800090b5d3e:3

::M2::M3::T3IDL:P2/T3:1.0

::M2::T4IDL: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.

10.6.6 For More Information

“OMG IDL for Interface Repository” on page 10-56 shows the OMG IDL specification of the IR, including the #pragma directive. Section 3.3, “Preprocessing,” on page 3-12 contains additional, general information on the pragma directive.

10.6.7 RepositoryIDs for OMG-Specified Types

Interoperability between implementations of official OMG specifications, including but not limited to CORBA, CORBAservices, and CORBAfacilities, depends on unambiguous specification of **RepositoryId**s for all IDL-defined types in such specifications.

All official OMG IDL files shall contain the following pragma prefix directive:

```
#pragma prefix “omg.org”
```

unless said file already contains a pragma prefix identifying the original source of the file (e.g., “**w3c.org**”).

Revisions to existing OMG specifications must not change the definition of an existing type in any way. Two types with different repository Ids are considered different types, regardless of which part of the repository Id differs.

If an implementation must extend an OMG-specified interface, interoperability requires it to derive a new interface from the standard interface, rather than modify the standard definition.

10.7 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.

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**.

10.7.1 The TypeCode Interface

The PIDL interface for **TypeCodes** is as follows:

```
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,
        tk_longlong, tk_ulonglong, tk_longdouble,
        tk_wchar, tk_wstring, tk_fixed,
        tk_value, tk_value_box,
        tk_native,
        tk_abstract_interface
    };

    typedef short ValueModifier;
        const ValueModifier VM_NONE = 0;
        const ValueModifier VM_CUSTOM = 1;
        const ValueModifier VM_ABSTRACT = 2;
        const ValueModifier VM_TRUNCATABLE = 3;

    interface TypeCode {
        exception       Bounds {};
        exception       BadKind {};

        // for all TypeCode kinds
        boolean equal (in TypeCode tc);

        boolean equivalent(in TypeCode tc);
        TypeCode get_compact_typecode();
```

```
        TCKind kind ();

        // for tk_objref, tk_struct, tk_union, tk_enum, tk_alias,
        // tk_value, tk_value_box, tk_native, tk_abstract_interface
        // and tk_except
        RepositoryId id () raises (BadKind);

        // for tk_objref, tk_struct, tk_union, tk_enum, tk_alias,
        // tk_value, tk_value_box, tk_native, tk_abstract_interface
        // and tk_except
        Identifier name () raises (BadKind);

        // for tk_struct, tk_union, tk_enum, tk_value,
        // and tk_except
        unsigned long member_count () raises (BadKind);
        Identifier member_name (in unsigned long index)
            raises(BadKind, Bounds);

        // for tk_struct, tk_union, tk_value,
        // 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, tk_value_box and tk_alias
        TypeCode content_type () raises (BadKind);

        // for tk_fixed
        unsigned short fixed_digits() raises(BadKind);
        short fixed_scale() raises(BadKind);

        // for tk_value
        Visibility member_visibility(in unsigned long index)
            raises(BadKind, Bounds);
        ValueModifier type_modifier() raises(BadKind);
        TypeCode concrete_base_type() raises(BadKind);
    };
};
```

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 it invoked.

The **equal** operation can be invoked on any **TypeCode**. The **equal** operation returns **TRUE** if and only if for the target **TypeCode** and the **TypeCode** passed through the parameter **tc,** the set of legal operations is the same and invoking any operation from that set on the two **TypeCodes** return identical results.

The **equivalent** operation is used by the ORB when determining type equivalence for values stored in an IDL **any**. **TypeCodes** are considered equivalent based on the following semantics:

- If the result of the **kind** operation on either **TypeCode** is **tk_alias**, recursively replace the **TypeCode** with the result of calling **content_type**, until the kind is no longer **tk_alias**.
- If results of the **kind** operation on each typecode differ, **equivalent** returns false.
- If the **id** operation is valid for the **TypeCode kind**, **equivalent** returns **TRUE** if the results of **id** for both **TypeCodes** are non-empty strings and both strings are equal. If both ids are non-empty but are not equal, then **equivalent** returns **FALSE**. If either or both id is an empty string, or the **TypeCode kind** does not support the **id** operation, **equivalent** will perform a structural comparison of the **TypeCodes** by comparing the results of the other **TypeCode** operations in the following bullet items (ignoring aliases as described in the first bullet.). The structural comparison only calls operations that are valid for the given **TypeCode kind**. If any of these operations do not return equal results, then **equivalent** returns **FALSE**. If all comparisons are equal, **equivalent** returns true.
- The results of the **name** and **member_name** operations are ignored and not compared.
- The results of the **member_count**, **default_index**, **length**, **digits**, and **scale** operations are compared.
- The results of the **member_label** operation for each member index of a **union TypeCode** are compared for equality. Note that this means that **unions** whose members are not defined in the same order are not considered structurally equivalent.
- The results of the **discriminator_type** and **member_type** operation for each member index, and the result of the **content_type** operation are compared by recursively calling **equivalent**.

Applications that need to distinguish between a type and different aliases of that type can supplement **equivalent** by directly invoking the **id** operation and comparing the results.

The **get_compact_typecode** operation strips out all optional **name** & **member name** fields, but it leaves all alias typecodes intact.

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, valuetype, boxed valuetype, native 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, value type, boxed valuetype, native 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 order in which members are presented in the interface repository is the same as the order in which they appeared in the IDL specification, and this ordering determines the index value for each member. The first member has index value 0. For example for a structure definition:

```
struct example {
    short   member1;
    short   member2;
    long    member3;
};
```

In this example **member1** has **index** = 0, **member2** has **index** = 1, and **member3** has **index** = 2. The value of **member_count** in this case is 3.

The **member_count** and **member_name** operations can be invoked on structure, union, non-boxed valuetype, exception 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, non-boxed valuetype, exception 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_visibility** operation can only be invoked on non-boxed valuetype **TypeCodes**. It returns the **Visibility** of the valuetype member identified by index.

The **member_name**, **member_type**, **member_label** and **member_visibility** operations raise **Bounds** if the index parameter is greater than or equal to the number of members constituting the type.

The **content_type** operation can be invoked on sequence, array, boxed valuetype and alias **TypeCode**s. For sequences and arrays, it returns the element type. For aliases, it returns the original type. For boxed valuetype, it returns the boxed type.

An array **TypeCode** only describes a single dimension of an OMG IDL array. Multi-dimensional 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 **type_modifier** and **concrete_base_type** operations can be invoked on non-boxed valuetype **TypeCodes**. The **type_modifier** operation returns the **ValueModifier** that applies to the valuetype represented by the target **TypeCode**. If the valuetype represented by the target **TypeCode** has a concrete base valuetype, the **concrete_base_type** operation returns a **TypeCode** for the concrete base, otherwise it returns a nil **TypeCode** reference.

The **length** operation can be invoked on string, wide 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 the number of elements in the array. For wide strings, it returns the bound, or zero for unbounded wide strings.

10.7.2 TypeCode Constants

For IDL type declarations, the IDL compiler produces (if asked) a declaration of a **TypeCode** constant. See the language mapping rules for more information about the names of the generated **TypeCode** constants. **TypeCode** constants include tk_alias definitions wherever an IDL typedef is referenced. These constants can be used with the dynamic invocation interface and other routines that require **TypeCodes**.

The predefined **TypeCode** constants, named according to the C language mapping, are:

TC_null
TC_void
TC_short
TC_long
TC_longlong
TC_ushort
TC_ulong
TC_ulonglong
TC_float
TC_double
TC_longdouble
TC_boolean
TC_char
TC_wchar
TC_octet
TC_any
TC_TypeCode

TC_Object = tk_objref {Object}
TC_string= tk_string {0} // unbounded
TC_wstring = tk_wstring{0} // unbounded

10.7.3 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_wstring_tc (
        in unsigned long        bound
);

TypeCode create_fixed_tc (
        in unsigned short       digits,
        in unsigned short       scale
);

TypeCode create_sequence_tc (
        in unsigned long        bound,
        in TypeCode             element_type
);

TypeCode create_recursive_sequence_tc (// deprecated
        in unsigned long        bound,
        in unsigned long        offset
);

TypeCode create_array_tc (
        in unsigned long        length,
        in TypeCode             element_type
);

TypeCode create_value_tc (
        in RepositoryId         id,
        in Identifier           name,
        in ValueModifier        type_modifier,
        in TypeCode             concrete_base,
        in ValueMemberSeq       members
);

TypeCode create_value_box_tc (
        in RepositoryId         id,
        in Identifier           name,
        in TypeCode             boxed_type
);

TypeCode create_native_tc (
        in RepositoryId         id,
        in Identifier           name
);
```

```
        TypeCode create_recursive_tc(
            in RepositoryId         id
        );

        TypeCode create_abstract_interface_tc(
            in RepositoryId         id,
            in Identifier           name
        );
    };
};
```

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**, **UnionMember** and **ValueMember** structures, only the **type** is used, and the **type_def** should be set to nil.

Note – The **create_recursive_sequence_tc** operation is deprecated. No new code should make use of this operation. Its functionality is subsumed by the new operation **create_recursive_tc**. The **create_recursive_sequence_tc** operation will be removed from a future revision of the standard.

The **create_recursive_sequence_tc** operation is used to create **TypeCodes** describing recursive sequences that are members of structs or unions. The result of this operation should be used as the typecode in the **StructMemberSeq** or **UnionMemberSeq** arguments of the **create_struct_tc** or **create_union_tc** operations. The **offset** parameter specifies which enclosing struct or union is the target of the recursion, with the value **1** indicating the most immediate enclosing struct or union, and larger values indicating successive enclosing struct or unions. For example, the offset would be **1** for the following IDL structure:

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

Once the recursive sequence **TypeCode** has been properly embedded in its enclosing **TypeCodes**, it will function as a normal sequence **TypeCode**. Invoking operations on the recursive sequence **TypeCode** before it has been embedded in the required number of enclosing **TypeCodes** will result in undefined behavior.

For **create_value_tc** operation, the **concrete_base** parameter is a **TypeCode** for the immediate concrete valuetype base of the valuetype for which the **TypeCode** is being created. If the valuetype does not have a concrete base, the **concrete_base** parameter is a nil **TypeCode** reference.

The **create_recursive_tc** operation is used to create a recursive **TypeCode**, which serves as a place holder for a concrete **TypeCode** during the process of creating **TypeCode**s that contain recursion. The **id** parameter specifies the repository id of the type for which the recursive **TypeCode** is serving as a place holder. Once the recursive **TypeCode** has been properly embedded in the enclosing **TypeCode**, which

corresponds to the specified repository id, it will function as a normal **TypeCode**. Invoking operations on the recursive **TypeCode** before it has been embedded in the enclosing **TypeCode** will result in undefined behavior. For example, the following IDL type declarations contain recursion:

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

valuetype V {
    public V member;
};
```

To create a **TypeCode** for **valuetype V**, you would invoke the **TypeCode** creation operations as shown below:

```
// C++
TypeCode_var recursive_tc
   = orb->create_recursive_tc("IDL:V:1.0");

ValueMemberSeq v_seq;
v_seq.length(1);
v_seq[0].name = string_dup("member");
v_seq[0].type = recursive_tc;
v_seq[0].access = PUBLIC_MEMBER;
TypeCode_var v_val_tc
   = orb->create_value_tc("IDL:V:1.0",
                          "V",
                          VM_NONE,
                          TypeCode::_nil(),
                          v_seq);
```

10.8 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 {
#         pragma version DefinitionKind 2.3
          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,
        dk_Wstring, dk_Fixed,
        dk_Value, dk_ValueBox, dk_ValueMember,
        dk_Native
    };

    interface IRObject {
#       pragma version IRObject 2.3
        // read interface
        readonly attribute DefinitionKind def_kind;
        // write interface
        void destroy ();
    };

    typedef string VersionSpec;

    interface Contained;
    interface Repository;
    interface Container;

    interface Contained : IRObject {
#       pragma version Contained 2.3


        // 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;
    interface ExceptionDef;
    interface NativeDef;
    typedef sequence <InterfaceDef> InterfaceDefSeq;
    interface ValueDef;
    typedef sequence <ValueDef> ValueDefSeq;
    interface ValueBoxDef;

    typedef sequence <Contained> ContainedSeq;
    struct StructMember {
        Identifier          name;
        TypeCode            type;
        IDLType             type_def;
    };

    typedef sequence <StructMember> StructMemberSeq;

    struct Initializer {
#       pragma version Initializer 2.3
        StructMemberSeq  members;
        Identifier          name;
    };
    typedef sequence <Initializer> InitializerSeq;

     struct UnionMember {
        Identifier      name;
        any             label;
        TypeCode        type;
        IDLType         type_def;
    };

    typedef sequence <UnionMember> UnionMemberSeq;

    typedef sequence <Identifier> EnumMemberSeq;

    interface Container : IRObject {
#       pragma version Container 2.3
        // 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,
        in boolean              is_abstract
    );

    ValueDef create_value(
        in RepositoryId         id,
        in Identifier           name,
        in VersionSpec          version,
        in boolean              is_custom,
        in boolean              is_abstract,
        in ValueDef             base_value,
        in boolean              is_truncatable,
        in ValueDefSeq          abstract_base_values,
        in InterfaceDefSeq      supported_interfaces,
        in InitializerSeq       initializers
    );

    ValueBoxDef create_value_box(
        in RepositoryId         id,
        in Identifier           name,
        in VersionSpec          version,
        in IDLType              original_type_def
    );

    ExceptionDef create_exception(
        in RepositoryId         id,
        in Identifier           name,
```

```
            in VersionSpec          version,
            in StructMemberSeq      members
        );

        NativeDef create_native(
            in RepositoryId         id,
            in Identifier           name,
            in VersionSpec          version,
        );
    };

    interface IDLType : IRObject {
#       pragma version IDLType 2.3
        readonly attribute TypeCode type;
    };

    interface PrimitiveDef;
    interface StringDef;
    interface SequenceDef;
    interface ArrayDef;
    interface WstringDef;
    interface FixedDef;

    enum PrimitiveKind {
#       pragma version PrimitiveKind 2.3
        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,
        pk_longlong, pk_ulonglong, pk_longdouble,
        pk_wchar, pk_wstring, pk_value_base
    };

    interface Repository : Container {
#       pragma version Repository 2.3
        // read interface

        Contained lookup_id (in RepositoryId search_id);

        TypeCode get_canonical_typecode(in TypeCode tc);

        PrimitiveDef get_primitive (in PrimitiveKind kind);

        // write interface

        StringDef create_string (in unsigned long bound);

        WstringDef create_wstring (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
        );

        FixedDef create_fixed (
            in unsigned short       digits,
            in short                scale
        );
    };

    interface ModuleDef : Container, Contained {
#       pragma version ModuleDef 2.3
    };

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

    interface ConstantDef : Contained {
#       pragma version ConstantDef 2.3
        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 {
#       pragma version TypedefDef 2.3
    };

    struct TypeDescription {
        Identifier              name;
        RepositoryId            id;
        RepositoryId            defined_in;
        VersionSpec             version;
        TypeCode                type;
    };
```

```
    interface StructDef : TypedefDef, Container {
#       pragma version StructDef 2.3
        attribute StructMemberSeq     members;
    };

    interface UnionDef : TypedefDef, Container {
#       pragma version UnionDef 2.3
        readonly attribute TypeCode   discriminator_type;
        attribute IDLType             discriminator_type_def;
        attribute UnionMemberSeq      members;
    };

    interface EnumDef : TypedefDef {
#       pragma version EnumDef 2.3
        attribute EnumMemberSeq       members;
    };

    interface AliasDef : TypedefDef {
#       pragma version AliasDef 2.3
        attribute IDLType             original_type_def;
    };

    interface NativeDef : TypedefDef {
#       pragma version NativeDef 2.3
    };

    interface PrimitiveDef: IDLType {
#       pragma version PrimitiveDef 2.3
        readonly attribute PrimitiveKind kind;
    };

    interface StringDef : IDLType {
#       pragma version StringDef 2.3
        attribute unsigned long       bound;
    };

    interface WstringDef : IDLType {
#       pragma version WstringDef 2.3
        attribute unsigned long       bound;
    };

    interface FixedDef : IDLType {
#       pragma version FixedDef 2.3
        attribute unsigned short      digits;
        attribute short               scale;
    };

    interface SequenceDef : IDLType {
#       pragma version SequenceDef 2.3
        attribute unsigned long       bound;
        readonly attribute TypeCode   element_type;
```

```
        attribute IDLType               element_type_def;
    };

    interface ArrayDef : IDLType {
#       pragma version ArrayDef 2.3
        attribute unsigned long         length;
        readonly attribute TypeCode     element_type;
        attribute IDLType               element_type_def;
    };

    interface ExceptionDef : Contained, Container {
#       pragma version ExceptionDef 2.3
        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 {
#       pragma version AttributeDef 2.3
        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 {
#   pragma version OperationDef 2.3
    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 {
#   pragma version InterfaceDef 2.3
    // read/write interface

    attribute InterfaceDefSeq       base_interfaces;
    attribute boolean               is_abstract;

    // read interface

    boolean is_a (
        in RepositoryId     interface_id
    );

    struct FullInterfaceDescription {
#       pragma version FullInterfaceDescription 2.3
        Identifier          name;
        RepositoryId        id;
        RepositoryId        defined_in;
        VersionSpec         version;
```

```
            OpDescriptionSeq    operations;
            AttrDescriptionSeq  attributes;
            RepositoryIdSeq     base_interfaces;
            TypeCode            type;
            boolean             is_abstract;
        };

        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 {
#       pragma version InterfaceDescription 2.3
        Identifier              name;
        RepositoryId            id;
        RepositoryId            defined_in;
        VersionSpec             version;
        RepositoryIdSeq         base_interfaces;
        boolean                 is_abstract;
    };

    typedef short Visibility;
    const Visibility PRIVATE_MEMBER = 0;
    const Visibility PUBLIC_MEMBER = 1;

    struct ValueMember {
#       pragma version ValueMember 2.3
        Identifier              name;
        RepositoryId            id;
        RepositoryId            defined_in;
        VersionSpec             version;
        TypeCode                type;
```

```
        IDLType                 type_def;
        Visibility              access;
    };

    typedef sequence <ValueMember> ValueMemberSeq;

    interface ValueMemberDef : Contained {
#       pragma version ValueMemberDef 2.3
        readonly attribute TypeCode type;
        attribute IDLType type_def;
        attribute Visibility access;
    };

    interface ValueDef : Container, Contained, IDLType {
#       pragma version ValueDef 2.3
        // read/write interface

        attribute InterfaceDefSeq supported_interfaces;
        attribute InitializerSeq initializers;
        attribute ValueDef base_value;
        attribute ValueDefSeq abstract_base_values;
        attribute boolean is_abstract;
        attribute boolean is_custom;
        attribute boolean is_truncatable;

        // read interface
        boolean is_a(
            in RepositoryId         id
        );

        struct FullValueDescription {
#           pragma version FullValueDescription 2.3
            Identifier              name;
            RepositoryId            id;
            boolean                 is_abstract;
            boolean                 is_custom;
            RepositoryId            defined_in;
            VersionSpec             version;
            OpDescriptionSeq        operations;
            AttrDescriptionSeq      attributes;
            ValueMemberSeq          members;
            InitializerSeq          initializers;
            RepositoryIdSeq         supported_interfaces;
            RepositoryIdSeq         abstract_base_values;
            boolean                 is_truncatable;
            RepositoryId            base_value;
            TypeCode                type;
        };

        FullValueDescription describe_value();
```

```
            ValueMemberDef create_value_member(
                in RepositoryId         id,
                in Identifier           name,
                in VersionSpec          version,
                in IDLType              type,
                in Visibility           access
            );

            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 ValueDescription {
#           pragma version ValueDescription 2.3
            Identifier          name;
            RepositoryId        id;
            boolean             is_abstract;
            boolean             is_custom;
            RepositoryId        defined_in;
            VersionSpec         version;
            RepositoryIdSeq     supported_interfaces;
            RepositoryIdSeq     abstract_base_values;
            boolean             is_truncatable;
            RepositoryId        base_value;
        };

        interface ValueBoxDef : TypedefDef {
#           pragma version ValueBoxDef 2.3
            attribute IDLType original_type_def;
        };

        enum TCKind {                                   // PIDL
#           pragma version TCKind 2.3
            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,
        tk_longlong, tk_ulonglong, tk_longdouble,
        tk_wchar, tk_wstring, tk_fixed,
        tk_value, tk_value_box,
        tk_native,
        tk_abstract_interface
    };

    typedef short ValueModifier;                    // PIDL
        const ValueModifier VM_NONE = 0;
        const ValueModifier VM_CUSTOM = 1;
        const ValueModifier VM_ABSTRACT = 2;
        const ValueModifier VM_TRUNCATABLE = 3;

    interface TypeCode {                            // PIDL
#       pragma version TypeCode 2.3
        exception Bounds {};
        exception BadKind {};

        // for all TypeCode kinds
        boolean equal (in TypeCode tc);

        boolean equivalent(in TypeCode tc);
        TypeCode get_compact_typecode();

        TCKind kind ();

        // for tk_objref, tk_struct, tk_union, tk_enum, tk_alias,
        // tk_value, tk_value_box, tk_native, tk_abstract_interface
        // and tk_except
        RepositoryId id () raises (BadKind);

        // for tk_objref, tk_struct, tk_union, tk_enum, tk_alias,
        // tk_value, tk_value_box, tk_native, tk_abstract_interface
        // and tk_except
        Identifier name () raises (BadKind);

        // for tk_struct, tk_union, tk_enum, tk_value,
        // and tk_except
        unsigned long member_count () raises (BadKind);
        Identifier member_name (in unsigned long index)
            raises (BadKind, Bounds);

        // for tk_struct, tk_union, tk_value, 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, tk_value_box, and tk_alias
            TypeCode content_type () raises (BadKind);

            // for tk_fixed
            unsigned short fixed_digits() raises (BadKind);
            short fixed_scale() raises (BadKind);

            // for tk_value
            Visibility member_visibility(in unsigned long index)
                raises(BadKind, Bounds);
            ValueModifier type_modifier() raises(BadKind);
            TypeCode concrete_base_type() raises(BadKind);
        };

    // Only the TypeCode related part of interface ORB shown below.
    // For complete description of interface ORB see Chapter 4.

        interface ORB {                                          // PIDL
    #       pragma version ORB 2.3
            // 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_wstring_tc (
        in unsigned long        bound
    );

    TypeCode create_fixed_tc (
        in unsigned short       digits,
        in unsigned short       scale
    );

    TypeCode create_sequence_tc (
        in unsigned long        bound,
        in TypeCode             element_type
    );

    TypeCode create_recursive_sequence_tc (// deprecated
        in unsigned long        bound,
        in unsigned long        offset
    );

    TypeCode create_array_tc (
        in unsigned long        length,
        in TypeCode             element_type
    );

    TypeCode create_value_tc (
        in RepositoryId         id,
        in Identifier           name,
        in ValueModifier        type_modifier,
        in TypeCode             concrete_base,
        in ValueMemberSeq       members
    );

    TypeCode create_value_box_tc (
```

```
                in RepositoryId         id,
                in Identifier           name,
                in TypeCode             boxed_type
            );

            TypeCode create_native_tc (
                in RepositoryId         id,
                in Identifier           name
            );

            TypeCode create_recursive_tc(
                in RepositoryId         id
            );

            TypeCode create_abstract_interface_tc(
                in RepositoryId         id,
                in Identifier           name
            );
        };
    };
```

The Portable Object Adaptor 11

The Portable Object Adaptor chapter has been updated based on CORE changes from ptc/98-09-04.

This chapter describes the Portable Object Adapter, or POA. It presents the design goals, a description of the abstract model of the POA and its interfaces, followed by a detailed description of the interfaces themselves.

Contents

This chapter contains the following sections.

11.1 Overview

The POA is designed to meet the following goals:

- Allow programmers to construct object implementations that are portable between different ORB products.

- Provide support for objects with persistent identities. More precisely, the POA is designed to allow programmers to build object implementations that can provide consistent service for objects whose lifetimes (from the perspective of a client holding a reference for such an object) span multiple server lifetimes.
- Provide support for transparent activation of objects.
- Allow a single servant to support multiple object identities simultaneously.
- Allow multiple distinct instances of the POA to exist in a server.
- Provide support for transient objects with minimal programming effort and overhead.
- Provide support for implicit activation of servants with POA-allocated Object Ids.
- Allow object implementations to be maximally responsible for an object's behavior. Specifically, an implementation can control an object's behavior by establishing the datum that defines an object's identity, determining the relationship between the object's identity and the object's state, managing the storage and retrieval of the object's state, providing the code that will be executed in response to requests, and determining whether or not the object exists at any point in time.
- Avoid requiring the ORB to maintain persistent state describing individual objects, their identities, where their state is stored, whether certain identity values have been previously used or not, whether an object has ceased to exist or not, and so on.
- Provide an extensible mechanism for associating policy information with objects implemented in the POA.
- Allow programmers to construct object implementations that inherit from static skeleton classes, generated by OMG IDL compilers, or a DSI implementation.

11.2 Abstract Model Description

The POA interfaces described in this chapter imply a particular abstract computational model. This section presents that model and defines terminology and basic concepts that will be used in subsequent sections.

This section provides the rationale for the POA design, describes some of its intended uses, and provides a background for understanding the interface descriptions.

11.2.1 Model Components

The model supported by the POA is a specialization of the general object model described in the OMA guide. Most of the elements of the CORBA object model are present in the model described here, but there are some new components, and some of the names of existing components are defined more precisely than they are in the CORBA object model. The abstract model supported by the POA has the following components:

- *Client*—A client is a computational context that makes requests on an object through one of its references.

- *Server*—A server is a computational context in which the implementation of an object exists. Generally, a server corresponds to a process. Note that *client* and *server* are roles that programs play with respect to a given object. A program that is a client for one object may be the server for another. The same process may be both client and server for a single object.
- *Object*—In this discussion, we use *object* to indicate a CORBA object in the abstract sense, that is, a programming entity with an identity, an interface, and an implementation. From a client's perspective, the object's identity is encapsulated in the object's reference. This specification defines the server's view of object identity, which is explicitly managed by object implementations through the POA interface.
- *Servant*—A servant is a programming language object or entity that implements requests on one or more objects. Servants generally exist within the context of a server process. Requests made on an object's references are mediated by the ORB and transformed into invocations on a particular servant. In the course of an object's lifetime it may be associated with (that is, requests on its references will be targeted at) multiple servants.
- *Object Id*—An Object Id is a value that is used by the POA and by the user-supplied implementation to identify a particular abstract CORBA object. Object Id values may be assigned and managed by the POA, or they may be assigned and managed by the implementation. Object Id values are hidden from clients, encapsulated by references. Object Ids have no standard form; they are managed by the POA as uninterpreted octet sequences.

  Note that the Object Id defined in this specification is a mechanical device used by an object implementation to correlate incoming requests with references it has previously created and exposed to clients. It does not constitute a unique logical identity for an object in any larger sense. The assignment and interpretation of Object Id values is primarily the responsibility of the application developer, although the **SYSTEM_ID** policy enables the POA to generate Object Id values for the application.
- *Object Reference*—An object reference in this model is the same as in the CORBA object model. This model implies, however, that a reference specifically encapsulates an Object Id and a POA identity.

  Note that a concrete reference in a specific ORB implementation will contain more information, such as the location of the server and POA in question. For example, it might contain the full name of the POA (the names of all POAs starting from the root and ending with the specific POA). The reference might not, in fact, actually contain the Object Id, but instead contain more compact values managed by the ORB which can be mapped to the Object Id. This is a description of the abstract information model implied by the POA. Whatever encoding is used to represent the POA name and the Object Id must not restrict the ability to use any legal character in a POA name or any legal octet in an Object Id.

- *POA*—A POA is an identifiable entity within the context of a server. Each POA provides a namespace for Object Ids and a namespace for other (nested or child) POAs. Policies associated with a POA describe characteristics of the objects implemented in that POA. Nested POAs form a hierarchical name space for objects within a server.
- *Policy*—A Policy is an object associated with a POA by an application in order to specify a characteristic shared by the objects implemented in that POA. This specification defines policies controlling the POA's threading model as well as a variety of other options related to the management of objects. Other specifications may define other policies that affect how an ORB processes requests on objects implemented in the POA.
- *POA Manager*—A POA manager is an object that encapsulates the processing state of one or more POAs. Using operations on a POA manager, the developer can cause requests for the associated POAs to be queued or discarded. The developer can also use the POA manager to deactivate the POAs.
- *Servant Manager*—A servant manager is an object that the application developer can associate with a POA. The ORB will invoke operations on servant managers to activate servants on demand, and to deactivate servants. Servant managers are responsible for managing the association of an object (as characterized by its Object Id value) with a particular servant, and for determining whether an object exists or not. There are two kinds of servant managers, called **ServantActivator** and **ServantLocator**; the type used in a particular situation depends on policies in the POA.
- *Adapter Activator*—An adapter activator is an object that the application developer can associate with a POA. The ORB will invoke an operation on an adapter activator when a request is received for a child POA that does not currently exist. The adapter activator can then create the required POA on demand.

11.2.2 Model Architecture

This section describes the architecture of the abstract model implied by the POA, and the interactions between various components. The ORB is an abstraction visible to both the client and server. The POA is an object visible to the server. User-supplied implementations are registered with the POA (this statement is a simplification; more detail is provided below). Clients hold references upon which they can make requests. The ORB, POA, and implementation all cooperate to determine which servant the operation should be invoked on, and to perform the invocation.

Figure 11-1 shows the detail of the relationship between the POA and the implementation. Ultimately, a POA deals with an Object Id and an active servant. By *active servant*, we mean a programming object that exists in memory and has been presented to the POA with one or more associated object identities. There are several ways for this association to be made.

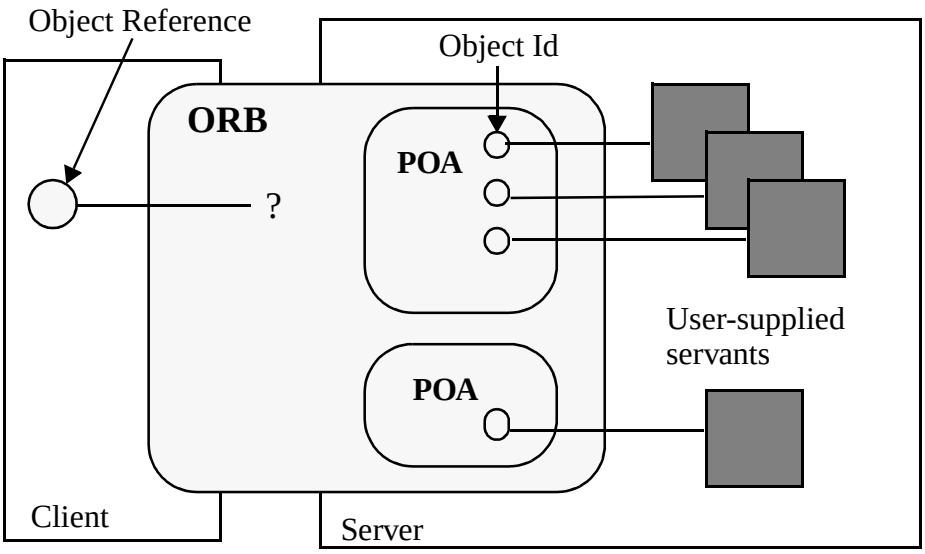


Figure 11-1 Abstract POA Model

If the POA supports the **RETAIN** policy, it maintains a map, labeled *Active Object Map*, that associates Object Ids with active servants, each association constituting an active object. If the POA has the **USE_DEFAULT_SERVANT** policy, a default servant may be registered with the POA. Alternatively, if the POA has the **USE_SERVANT_MANAGER** policy, a user-written servant manager may be registered with the POA. If the Active Object Map is not used, or a request arrives for an object not present in the Active Object Map, the POA either uses the default servant to perform the request or it invokes the servant manager to obtain a servant to perform the request. If the **RETAIN** policy is used, the servant returned by a servant manager is retained in the Active Object Map. Otherwise, the servant is used only to process the one request.

In this specification, the term *active* is applied equally to servants, Object Ids, and objects. An object is active in a POA if the POA's Active Object Map contains an entry that associates an Object Id with an existing servant. When this specification refers to *active Object Ids* and *active servants*, it means that the Object Id value or servant in question is part of an entry in the Active Object Map. An Object Id can appear in a POA's Active Object Map only once.

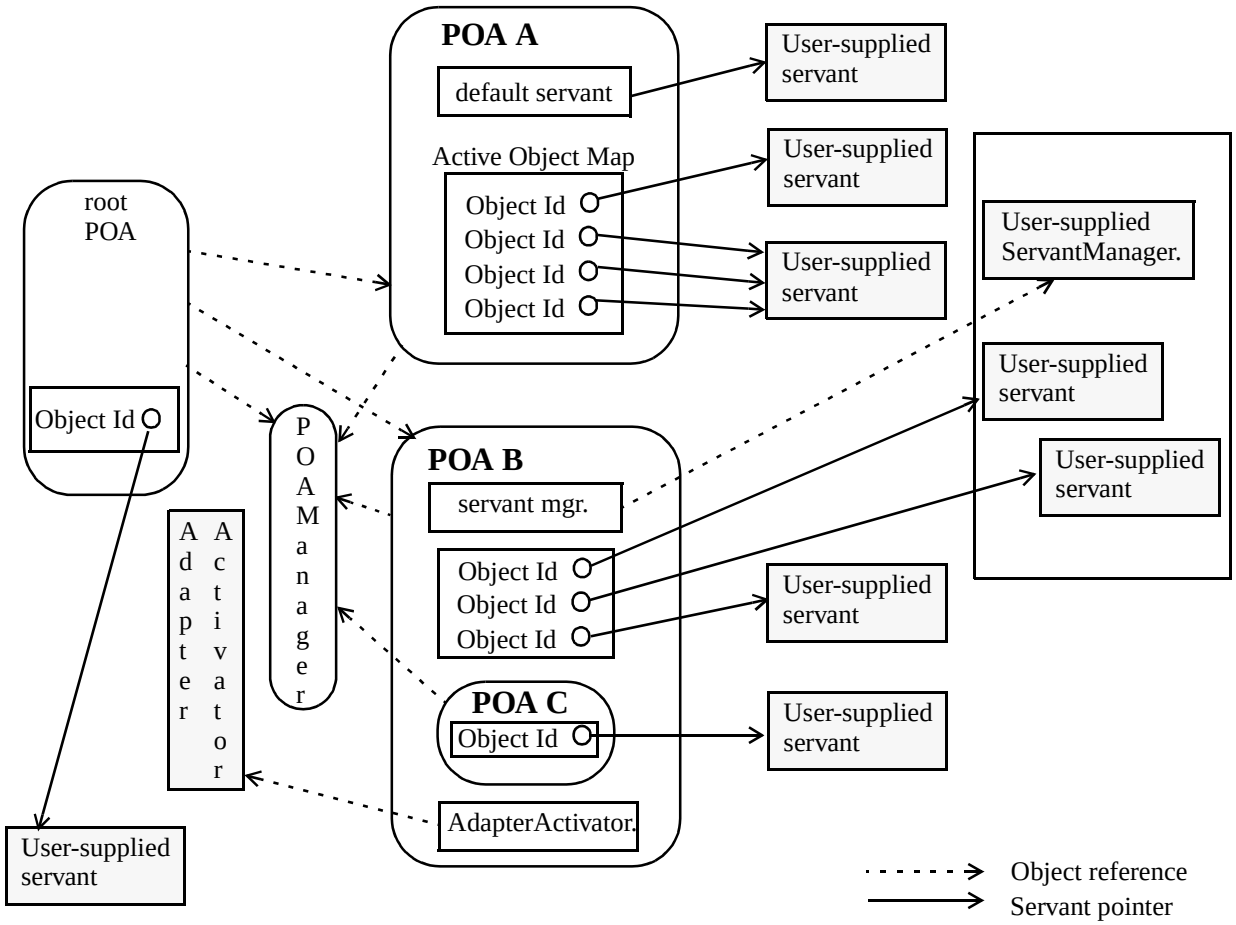


Figure 11-2 POA Architecture

11.2.3 POA Creation

To implement an object using the POA requires that the server application obtain a POA object. A distinguished POA object, called the *root POA*, is managed by the ORB and provided to the application using the ORB initialization interface under the initial object name “RootPOA.” The application developer can create objects using the root POA if those default policies are suitable. The root POA has the following policies.

- Thread Policy: **ORB_CTRL_MODEL**
- Lifespan Policy: **TRANSIENT**
- Object Id Uniqueness Policy: **UNIQUE_ID**
- Id Assignment Policy: **SYSTEM_ID**
- Servant Retention Policy: **RETAIN**
- Request Processing Policy: **USE_ACTIVE_OBJECT_MAP_ONLY**
- Implicit Activation Policy: **IMPLICIT_ACTIVATION**

The developer can also create new POAs. Creating a new POA allows the application developer to declare specific policy choices for the new POA and to provide a different adapter activator and servant manager (these are callback objects used by the POA to activate objects and nested POAs on demand). Creating new POAs also allows the application developer to partition the name space of objects, as Object Ids are interpreted relative to a POA. Finally, by creating new POAs, the developer can independently control request processing for multiple sets of objects.

A POA is created as a child of an existing POA using the **create_POA** operation on the parent POA. When a POA is created, the POA is given a name that must be unique with respect to all other POAs with the same parent.

POA objects are not persistent. No POA state can be assumed to be saved by the ORB. It is the responsibility of the server application to create and initialize the appropriate POA objects during server initialization or to set an AdapterActivater to create POA objects needed later.

Creating the appropriate POA objects is particularly important for persistent objects, objects whose existence can span multiple server lifetimes. To support an object reference created in a previous server process, the application must recreate the POA that created the object reference as well as all of its ancestor POAs. To ensure portability, each POA must be created with the same name as the corresponding POA in the original server process and with the same policies. (It is the user's responsibility to create the POA with these conditions.)

A portable server application can presume that there is no conflict between its POA names and the POA names chosen by other applications. It is the responsibility of the ORB implementation to provide a way to support this behavior.

11.2.4 Reference Creation

Object references are created in servers. Once they are created, they may be exported to clients.

From this model's perspective, object references encapsulate object identity information and information required by the ORB to identify and locate the server and POA with which the object is associated (that is, in whose scope the reference was created.) References are created in the following ways:

- The server application may directly create a reference with the **create_reference** and **create_reference_with_id** operations on a POA object. These operations collect the necessary information to constitute the reference, either from information associated with the POA or as parameters to the operation. These operations only create a reference. In doing so, they bring the abstract object into existence, but do not associate it with an active servant.
- The server application may explicitly activate a servant, associating it with an object identity using the **activate_object** or **activate_object_with_id** operations. Once a servant is activated, the server application can map the servant to its corresponding reference using the **servant_to_reference** or **id_to_reference** operations.

- The server application may cause a servant to implicitly activate itself. This behavior can only occur if the POA has been created with the **IMPLICIT_ACTIVATION** policy. If an attempt is made to obtain an object reference corresponding to an inactive servant, the POA may automatically assign a generated unique Object Id to the servant and activate the resulting object. The reference may be obtained by invoking **POA::servant_to_reference** with an inactive servant, or by performing an explicit or implicit type conversion from the servant to a reference type in programming language mappings that permit this conversion.

Once a reference is created in the server, it can be made available to clients in a variety of ways. It can be advertised through the OMG Naming and Trading Services. It can be converted to a string via **ORB::object_to_string** and published in some way that allows the client to discover the string and convert it to a reference using **ORB::string_to_object**. It can be returned as the result of an operation invocation.

Once a reference becomes available to a client, that reference constitutes the identity of the object from the client's perspective. As long as the client program holds and uses that reference, requests made on the reference should be sent to the "same" object.

Note – The meaning of object identity and "sameness" is at present the subject of debate in the OMG. This specification does not attempt to resolve that debate in any way, particularly by defining a concrete notion of identity that is exposed to clients, beyond the existing notions of identity described in the CORBA specifications and the OMA guide.

The states of servers and implementation objects are opaque to clients. This specification deals primarily with the view of the ORB from the server's perspective.

11.2.5 Object Activation States

At any point in time, a CORBA object may or may not be associated with an active servant.

If the POA has the **RETAIN** policy, the servant and its associated Object Id are entered into the Active Object Map of the appropriate POA. This type of activation can be accomplished in one of the following ways.

- The server application itself explicitly activates individual objects (via the **activate_object** or **activate_object_with_id** operations).
- The server application instructs the POA to activate objects on demand by having the POA invoke a user-supplied servant manager. The server application registers this servant manager with **set_servant_manager**.
- Under some circumstances (when the **IMPLICIT_ACTIVATION** policy is also in effect and the language binding allows such an operation), the POA may implicitly activate an object when the server application attempts to obtain a reference for a servant that is not already active (that is, not associated with an Object Id).

If the **USE_DEFAULT_SERVANT** policy is also in effect, the server application instructs the POA to activate unknown objects by having the POA invoke a single servant no matter what the Object Id is. The server application registers this servant with **set_servant**.

If the POA has the **NON_RETAIN** policy, for every request, the POA may use either a default servant or a servant manager to locate an active servant. From the POA's point of view, the servant is active only for the duration of that one request. The POA does not enter the servant-object association into the Active Object Map.

11.2.6 Request Processing

A request must be capable of conveying the Object Id of the target object as well as the identification of the POA that created the target object reference. When a client issues a request, the ORB first locates an appropriate server (perhaps starting one if needed) and then it locates the appropriate POA within that server.

If the POA does not exist in the server process, the application has the opportunity to re-create the required POA by using an adapter activator. An adapter activator is a user-implemented object that can be associated with a POA. It is invoked by the ORB when a request is received for a non-existent child POA. The adapter activator has the opportunity to create the required POA. If it does not, the client receives the OBJECT_NOT_EXIST exception.

Once the ORB has located the appropriate POA, it delivers the request to that POA. The further processing of that request depends both upon the policies associated with that POA as well as the object's current state of activation.

If the POA has the **RETAIN** policy, the POA looks in the Active Object Map to find out if there is a servant associated with the Object Id value from the request. If such a servant exists, the POA invokes the appropriate method on the servant.

If the POA has the **NON_RETAIN** policy or has the **RETAIN** policy but didn't find a servant in the Active Object Map, the POA takes the following actions:

- If the POA has the **USE_DEFAULT_SERVANT** policy, a default servant has been associated with the POA so the POA will invoke the appropriate method on that servant. If no servant has been associated with the POA, the POA raises the OBJ_ADAPTER system exception.
- If the POA has the **USE_SERVANT_MANAGER** policy, a servant manager has been associated with the POA so the POA will invoke **incarnate** or **preinvoke** on it to find a servant that may handle the request. (The choice of method depends on the **NON_RETAIN** or **RETAIN** policy of the POA.) If no servant manager has been associated with the POA, the POA raises the OBJ_ADAPTER system exception.
- If the **USE_OBJECT_MAP_ONLY** policy is in effect, the POA raises the OBJECT_NOT_EXIST system exception.

If a servant manager is located and invoked, but the servant manager is not directly capable of incarnating the object, it (the servant manager) may deal with the circumstance in a variety of ways, all of which are the application's responsibility.

Any system exception raised by the servant manager will be returned to the client in the reply. In addition to standard CORBA exceptions, a servant manager is capable of raising a ForwardRequest exception. This exception includes an object reference. The ORB will process this exception as stated below.

11.2.7 Implicit Activation

A POA can be created with a policy that indicates that its objects may be implicitly activated. This policy, **IMPLICIT_ACTIVATION**, also requires the **SYSTEM_ID** and **RETAIN** policies. When a POA supports implicit activation, an inactive servant may be implicitly activated in that POA by certain operations that logically require an Object Id to be assigned to that servant. Implicit activation of an object involves allocating a system-generated Object Id and registering the servant with that Object Id in the Active Object Map. The interface associated with the implicitly activated object is determined from the servant (using static information from the skeleton, or, in the case of a dynamic servant, using the **_primary_interface()** operation).

The operations that support implicit activation include:

- The **POA::servant_to_reference** operation, which takes a servant parameter and returns a reference.
- The **POA::servant_to_id** operation, which takes a servant parameter and returns an Object Id.
- Operations supported by a language mapping to obtain an object reference or an Object Id for a servant. For example, the **`_this()`** servant member function in C++ returns an object reference for the servant.
- Implicit conversions supported by a language mapping that convert a servant to an object reference or an Object Id.

The last two categories of operations are language-mapping-dependent.

If the POA has the **UNIQUE_ID** policy, then implicit activation will occur when any of these operations are performed on a servant that is not currently active (that is, it is associated with no Object Id in the POA's Active Object Map).

If the POA has the **MULTIPLE_ID** policy, the **servant_to_reference** and **servant_to_id** operations will *always* perform implicit activation, even if the servant is already associated with an Object Id. The behavior of language mapping operations in the **MULTIPLE_ID** case is specified by the language mapping. For example, in C++, the **`_this()`** servant member function will not implicitly activate a **MULTIPLE_ID** servant if the invocation of **`_this()`** is immediately within the dynamic context of a request invocation directed by the POA to that servant; instead, it returns the object reference used to issue the request.

Note – The exact timing of implicit activation is ORB implementation-dependent. For example, instead of activating the object immediately upon creation of a local object reference, the ORB could defer the activation until the Object Id is actually needed (for example, when the object reference is exported outside the process).

11.2.8 Multi-threading

The POA does not require the use of threads and does not specify what support is needed from a threads package. However, in order to allow the development of portable servers that utilize threads, the behavior of the POA and related interfaces when used within a multiple-thread environment must be specified.

Specifying this behavior does not require that an ORB must support being used in a threaded environment, nor does it require that an ORB must utilize threads in the processing of requests. The only requirement given here is that if an ORB does provide support for multi-threading, these are the behaviors that will be supported by that ORB. This allows a programmer to take advantage of multiple ORBs that support threads in a portable manner across those ORBs.

The POA's processing is affected by the thread-related calls available in the ORB: **work_pending**, **perform_work**, **run**, and **shutdown**.

11.2.8.1 POA Threading Models

The POA supports two models of threading when used in conjunction with multi-threaded ORB implementations; ORB controlled and single thread behavior. The two models can be used together or independently. Either model can be used in environments where a single-threaded ORB is used.

The threading model associated with a POA is indicated when the POA is created by including a **ThreadPolicy** object in the policies parameter of the POA's **create_POA** operation. Once a POA is created with one model, it cannot be changed to the other. All uses of the POA within the server must conform to that threading model associated with the POA.

11.2.8.2 Using the Single Thread Model

Requests for a single-threaded POA are processed sequentially. In a multi-threaded environment, all upcalls made by this POA to implementation code (servants, servant managers, and adapter activators) are made in a manner that is safe for code that is multi-thread-unaware.

11.2.8.3 Using the ORB Controlled Model

The ORB controlled model of threading is used in environments where the developer wants the ORB/POA to control the use of threads in the manner provided by the ORB. This model can also be used in environments that do not support threads.

In this model, the ORB is responsible for the creation, management, and destruction of threads used with one or more POAs.

11.2.8.4 Limitations When Using Multiple Threads

There are no guarantees that the ORB and POA will do anything specific about dispatching requests across threads with a single POA. Therefore, a server programmer who wants to use one or more POAs within multiple threads must take on all of the serialization of access to objects within those threads.

There may be requests active for the same object being dispatched within multiple threads at the same time. The programmer must be aware of this possibility and code with it in mind.

11.2.9 Dynamic Skeleton Interface

The POA is designed to enable programmers to connect servants to:

- type-specific skeletons, typically generated by OMG IDL compilers, or
- dynamic skeletons.

Servants that are members of type-specific skeleton classes are referred to as type-specific servants. Servants connected to dynamic skeletons are used to implement the Dynamic Skeleton Interface (DSI) and are referred to as DSI servants.

Whether a CORBA object is being incarnated by a DSI servant or a type-specific servant is transparent to its clients. Two CORBA objects supporting the same interface may be incarnated, one by a DSI servant and the other with a type-specific servant. Furthermore, a CORBA object may be incarnated by a DSI servant only during some period of time, while the rest of the time is incarnated by a static servant.

The mapping for POA DSI servants is language-specific, with each language providing a set of interfaces to the POA. These interfaces are used only by the POA. The interfaces required are the following.

- Take a **CORBA::ServerRequest** object from the POA and perform the processing necessary to execute the request.
- Return the Interface Repository Id identifying the most-derived interface supported by the target CORBA object in a request.

The reason for the first interface is the entire reason for existence of the DSI: to be able to handle any request in the way the programmer wishes to handle it. A single DSI servant may be used to incarnate several CORBA objects, potentially supporting different interfaces.

The reason for the second interface can be understood by comparing DSI servants to type-specific servants.

A type-specific servant may incarnate several CORBA objects but all of them will support the same IDL interface as the most-derived IDL interface. In C++, for example, an IDL interface **`Window`** in module **`GraphicalSystem`** will generate a type-specific skeleton class called **`Window`** in namespace **`POA_GraphicalSystem`**. A type-specific servant which is directly derived from the

POA_GraphicalSystem::Window skeleton class may incarnate several CORBA objects at a time, but all those CORBA objects will support the **GraphicalSystem::Window** interface as the most-derived interface.

A DSI servant may incarnate several CORBA objects, not necessarily supporting the same IDL interface as the most-derived IDL interface.

In both cases (type-specific and DSI) the POA may need to determine, at runtime, the Interface Repository Id identifying the most-derived interface supported by the target CORBA object in a request. The POA should be able to determine this by asking the servant that is going to serve the CORBA object.

In the case of type-specific servants, the POA obtains that information from the type-specific skeleton class from which the servant is directly derived. In the case of DSI servants, the POA obtains that information by using the second language-specific interface above.

11.2.10 Location Transparency

The POA supports location transparency for objects implemented using the POA. Unless explicitly stated to the contrary, all POA behavior described in this specification applies regardless of whether the client is local (same process) or remote. For example, like a request from a remote client, a request from a local client may cause object activation if the object is not active, block indefinitely if the target object's POA is in the holding state, be rejected if the target object's POA is in the discarding or inactive states, be delivered to a thread-unaware object implementation, or be delivered to a different object if the target object's servant manager raises the ForwardRequest exception. The Object Id and POA of the target object will also be available to the server via the **Current** object, regardless of whether the client is local or remote.

Note – The implication of these requirements on the ORB implementation is to require the ORB to mediate all requests to POA-based objects, even if the client is co-resident in the same process. This specification is not intended to change CORBAServices specifications that allow for behaviors that are not location transparent. This specification does not prohibit (nonstandard) POA extensions to support object behavior that is not location-transparent.

11.3 Interfaces

The POA-related interfaces are defined in a module separate from the **CORBA** module, the **PortableServer** module. It consists of these interfaces:

- **POA**
- **POAManager**
- **ServantManager**
- **ServantActivator**
- **ServantLocator**

- **AdapterActivator**
- **ThreadPolicy**
- **LifespanPolicy**
- **IdUniquenessPolicy**
- **IdAssignmentPolicy**
- **ImplicitActivationPolicy**
- **ServantRetentionPolicy**
- **RequestProcessingPolicy**
- **Current**

In addition, the POA defines the **Servant** native type.

11.3.1 The Servant IDL Type

This specification defines a native type **PortableServer::Servant**. Values of the type **Servant** are programming-language-specific implementations of CORBA interfaces. Each language mapping must specify how **Servant** is mapped to the programming language data type that corresponds to an object implementation. The **Servant** type has the following characteristics and constraints.

- Values of type **Servant** are opaque from the perspective of CORBA application programmers. There are no operations that can be performed directly on them by user programs. They can be passed as parameters to certain POA operations. Some language mappings may allow **Servant** values to be implicitly converted to object references under appropriate conditions.
- Values of type **Servant** support a language-specific programming interface that can be used by the ORB to obtain a default POA for that servant. This interface is used only to support implicit activation. A language mapping may provide a default implementation of this interface that returns the root POA of a default ORB.
- Values of type Servant provide default implementations of the standard object reference operations **get_interface**, **is_a**, and **non_existent**. These operations can be overridden by the programmer to provide additional behavior needed by the object implementation. The default implementations of **get_interface** and **is_a** operations use the most derived interface of a static servant or the most derived interface retrieved from a dynamic servant to perform the operation. The default implementation of the **non_existent** operation returns **FALSE**. These operations are invoked by the POA just like any other operation invocation, so the **PortableServer::Current** interface and any language-mapping-provided method of accessing the invocation context are available.
- Values of type **Servant** must be testable for identity.
- Values of type **Servant** have no meaning outside of the process context or address space in which they are generated.

11.3.2 POAManager Interface

Each POA object has an associated **POAManager** object. A POA manager may be associated with one or more POA objects. A POA manager encapsulates the processing state of the POAs it is associated with. Using operations on the POA manager, an application can cause requests for those POAs to be queued or discarded, and can cause the POAs to be deactivated.

POA managers are created and destroyed implicitly. Unless an explicit POA manager object is provided at POA creation time, a POA manager is created when a POA is created and is automatically associated with that POA. A POA manager object is implicitly destroyed when all of its associated POAs have been destroyed.

11.3.2.1 Processing States

A POA manager has four possible processing states; *active*, *inactive*, *holding*, and *discarding*. The processing state determines the capabilities of the associated POAs and the disposition of requests received by those POAs. Figure 11-3 on page 11-16 illustrates the processing states and the transitions between them. For simplicity of presentation, this specification sometimes describes these states as POA states, referring to the POA or POAs that have been associated with a particular POA manager. A POA manager is created in the *holding* state. The root POA is therefore initially in the *holding* state.

For simplicity in the figure and the explanation, operations that would not cause a state change are not shown. For example, if a POA is in “active” state, it does not change state due to an activate operation. Such operations complete successfully with no special notice.

The only exception is the inactive state: a “deactivate” operation raises an exception just the same as every other attempted state change operation.

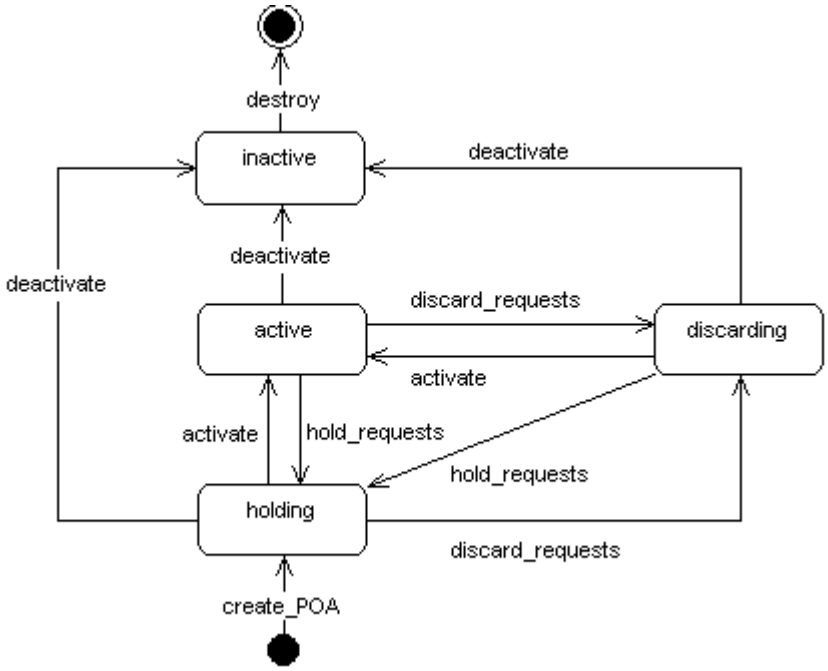


Figure 11-3 Processing States

Active State

When a POA manager is in the *active* state, the associated POAs will receive and start processing requests (assuming that appropriate thread resources are available). Note that even in the active state, a POA may need to queue requests depending upon the ORB implementation and resource limits. The number of requests that can be received and/or queued is an implementation limit. If this limit is reached, the POA should return a TRANSIENT system exception to indicate that the client should re-issue the request.

A user program can legally transition a POA manager from the *active* state to either the *discarding*, *holding*, or *inactive* state by calling the **discard_requests**, **hold_requests**, or **deactivate** operations, respectively. The POA enters the *active* state through the use of the **activate** operation when in the *discarding* or *holding* state.

Discarding State

When a POA manager is in the *discarding* state, the associated POAs will discard all incoming requests (whose processing has not yet begun). When a request is discarded, the TRANSIENT system exception must be returned to the client-side to indicate that the request should be re-issued. (Of course, an ORB may always reject a request for other reasons and raise some other system exception.)

In addition, when a POA manager is in the *discarding* state, the adapter activators registered with the associated POAs will not get called. Instead, requests that require the invocation of an adapter activator will be discarded, as described in the previous paragraph.

The primary purpose of the *discarding* state is to provide an application with flow-control capabilities when it determines that an object's implementation or POA is being flooded with requests. It is expected that the application will restore the POA manager to the *active* state after correcting the problem that caused flow-control to be needed.

A POA manager can legally transition from the *discarding* state to either the *active*, *holding*, or *inactive* state by calling the **activate**, **hold_requests**, or **deactivate** operations, respectively. The POA enters the *discarding* state through the use of the **discard_requests** operation when in the *active* or *holding* state.

Holding State

When a POA manager is in the *holding* state, the associated POAs will queue incoming requests. The number of requests that can be queued is an implementation limit. If this limit is reached, the POAs may discard requests and return the TRANSIENT system exception to the client to indicate that the client should reissue the request. (Of course, an ORB may always reject a request for other reasons and raise some other system exception.)

In addition, when a POA manager is in the *holding* state, the adapter activators registered with the associated POAs will not get called. Instead, requests that require the invocation of an adapter activator will be queued, as described in the previous paragraph.

A POA manager can legally transition from the *holding* state to either the *active*, *discarding*, or *inactive* state by calling the **activate**, **discard_requests**, or **deactivate** operations, respectively. The POA enters the *holding* state through the use of the **hold_requests** operation when in the *active* or *discarding* state. A POA manager is created in the holding state.

Inactive State

The *inactive* state is entered when the associated POAs are to be shut down. Unlike the *discarding* state, the *inactive* state is not a temporary state. When a POA manager is in the *inactive* state, the associated POAs will reject new requests. The rejection mechanism used is specific to the vendor. The GIOP location forwarding mechanism and CloseConnection message are examples of mechanisms that could be used to indicate the rejection. If the client is co-resident in the same process, the ORB could raise the OBJ_ADAPTER exception to indicate that the object implementation is unavailable.

In addition, when a POA manager is in the *inactive* state, the adapter activators registered with the associated POAs will not get called. Instead, requests that require the invocation of an adapter activator will be rejected, as described in the previous paragraph.

The *inactive* state is entered using the **deactivate** operation. It is legal to enter the *inactive* state from either the *active*, *holding*, or *discarding* states.

If the transition into the *inactive* state is a result of calling **deactivate** with an **etherealize_objects** parameter of

- TRUE - the associated POAs will call **etherealize** for each active object associated with the POA once all currently executing requests have completed processing (if the POAs have the **RETAIN** and **USE_SERVANT_MANAGER** policies). If a servant manager has been registered for the POA, the POA will get rid of the object. If there are any queued requests that have not yet started executing, they will be treated as if they were new requests and rejected.
- FALSE - No deactivations or etherealizations will be attempted.

11.3.2.2 Locality Constraints

A **POAManager** object must not be exported to other processes, or externalized with **ORB::object_to_string**. If any attempt is made to do so, the offending operation will raise a MARSHAL system exception. An attempt to use a **POAManager** object with the DII may raise the NO_IMPLEMENT exception.

11.3.2.3 activate

```
void activate()
    raises (AdapterInactive);
```

This operation changes the state of the POA manager to *active*. If issued while the POA manager is in the *inactive* state, the AdapterInactive exception is raised. Entering the *active* state enables the associated POAs to process requests.

11.3.2.4 hold_requests

```
void hold_requests( in boolean wait_for_completion )
    raises(AdapterInactive);
```

This operation changes the state of the POA manager to *holding*. If issued while the POA manager is in the *inactive* state, the AdapterInactive exception is raised. Entering the *holding* state causes the associated POAs to queue incoming requests. Any requests that have been queued but have not started executing will continue to be queued while in the *holding* state.

If the **wait_for_completion** parameter is **FALSE**, this operation returns immediately after changing the state. If the parameter is **TRUE** and the current thread is not in an invocation context dispatched by some POA belonging to the same ORB as this POA, this operation does not return until either there are no actively executing requests in any of the POAs associated with this POA manager (that is, all requests that were started prior to the state change have completed) or the state of the POA manager is changed to a state other than *holding*. If the parameter is **TRUE** and the current thread is in an invocation context dispatched by some POA belonging to the same ORB as this POA the BAD_INV_ORDER exception is raised and the state is not changed.

11.3.2.5 *discard_requests*

void discard_requests(in boolean wait_for_completion)
raises (AdapterInactive);

This operation changes the state of the POA manager to *discarding*. If issued while the POA manager is in the *inactive* state, the AdapterInactive exception is raised. Entering the *discarding* state causes the associated POAs to discard incoming requests. In addition, any requests that have been queued but have not started executing are discarded. When a request is discarded, a TRANSIENT system exception is returned to the client.

If the **wait_for_completion** parameter is FALSE, this operation returns immediately after changing the state. If the parameter is TRUE and the current thread is not in an invocation context dispatched by some POA belonging to the same ORB as this POA, this operation does not return until either there are no actively executing requests in any of the POAs associated with this POA manager (that is, all requests that were started prior to the state change have completed) or the state of the POA manager is changed to a state other than *discarding*. If the parameter is TRUE and the current thread is in an invocation context dispatched by some POA belonging to the same ORB as this POA the BAD_INV_ORDER exception is raised and the state is not changed.

11.3.2.6 *deactivate*

void deactivate(in boolean etherealize_objects,
in boolean wait_for_completion);
raises (AdapterInactive);

This operation changes the state of the POA manager to *inactive*. If issued while the POA manager is in the *inactive* state, the AdapterInactive exception is raised. Entering the inactive state causes the associated POAs to reject requests that have not begun to be executed as well as any new requests.

After changing the state, if the **etherealize_objects** parameter is

- TRUE - the POA manager will cause all associated POAs that have the **RETAIN** and **USE_SERVANT_MANAGER** policies to perform the **etherealize** operation on the associated servant manager for all active objects.
- FALSE - the **etherealize** operation is not called. The purpose is to provide developers with a means to shut down POAs in a crisis (for example, unrecoverable error) situation.

If the **wait_for_completion** parameter is FALSE, this operation will return immediately after changing the state. If the parameter is TRUE and the current thread is not in an invocation context dispatched by some POA belonging to the same ORB as this POA, this operation does not return until there are no actively executing requests in any of the POAs associated with this POA manager (that is, all requests that were started prior to the state change have completed) and, in the case of a TRUE **etherealize_objects**, all invocations of **etherealize** have completed for POAs having the **RETAIN** and **USE_SERVANT_MANAGER** policies. If the parameter is **TRUE**

and the current thread is in an invocation context dispatched by some POA belonging to the same ORB as this POA the BAD_INV_ORDER exception is raised and the state is not changed.

If the **ORB::shutdown** operation is called, it makes a call on **deactivate** with a TRUE **etherealize_objects** parameter for each POA manager known in the process; the **wait_for_completion** parameter to **deactivate** will be the same as the similarly named parameter of **ORB::shutdown**.

If **deactivate** is called multiple times before destruction is complete (because there are active requests), the **etherealize_objects** parameter applies only to the first call of **deactivate;** subsequent calls with conflicting **etherealize_objects** settings will use the value of the **etherealize_objects** from the first call. The **wait_for_completion** parameter will be handled as defined above for each individual call (some callers may choose to block, while others may not).

11.3.2.7 *get_state*

```
enum State {HOLDING, ACTIVE, DISCARDING, INACTIVE};
State get_state();
```

This operation returns the state of the POA manager.

11.3.3 *AdapterActivator Interface*

Adapter activators are associated with POAs. An adapter activator supplies a POA with the ability to create child POAs on demand, as a side-effect of receiving a request that names the child POA (or one of its children), or when **find_POA** is called with an activate parameter value of TRUE. An application server that creates all its needed POAs at the beginning of execution does not need to use or provide an adapter activator; it is necessary only for the case in which POAs need to be created during request processing.

While a request from the POA to an adapter activator is in progress, all requests to objects managed by the new POA (or any descendant POAs) will be queued. This serialization allows the adapter activator to complete any initialization of the new POA before requests are delivered to that POA.

11.3.3.1 *Locality Constraints*

An AdapterActivator object must be local to the process containing the POA objects it is registered with.

11.3.3.2 *unknown_adapter*

```
boolean unknown_adapter(in POA parent, in string name);
```

This operation is invoked when the ORB receives a request for an object reference that identifies a target POA that does not exist. The ORB invokes this operation once for each POA that must be created in order for the target POA to exist (starting with the

ancestor POA closest to the root POA). The operation is invoked on the adapter activator associated with the POA that is the parent of the POA that needs to be created. That parent POA is passed as the **parent** parameter. The name of the POA to be created (relative to the parent) is passed as the **name** parameter.

The implementation of this operation should either create the specified POA and return TRUE, or it should return FALSE. If the operation returns TRUE, the ORB will proceed with processing the request. If the operation returns FALSE, the ORB will return OBJECT_NOT_EXIST to the client. If multiple POAs need to be created, the ORB will invoke **unknown_adapter** once for each POA that needs to be created. If the parent of a nonexistent POA does not have an associated adapter activator, the ORB will return the OBJECT_NOT_EXIST exception.

If **unknown_adapter** raises a system exception, the ORB will report an OBJ_ADAPTER exception.

Note – It is possible for another thread to create the same POA the AdapterActivator is being asked to create if AdapterActivators are used in conjunction with other threads calling **create_POA** with the same POA name. Applications should be prepared to deal with failures from either the manual or automatic (AdapterActivator) POA creation request. There can be no guarantee of the order of such calls.

For example, if the target object reference was created by a POA whose full name is “A”, “B”, “C”, “D” and only POAs “A” and “B” currently exist, the **unknown_adapter** operation will be invoked on the adapter activator associated with POA “B” passing POA “B” as the parent parameter and “C” as the name of the missing POA. Assuming that the adapter activator creates POA “C” and returns TRUE, the ORB will then invoke **unknown_adapter** on the adapter activator associated with POA “C”, passing POA “C” as the parent parameter and “D” as the name.

The **unknown_adapter** operation is also invoked when **find_POA** is called on the POA with which the AdapterActivator is associated, the specified child does not exist, and the **activate_it** parameter to **find_POA** is TRUE. If **unknown_adapter** creates the specified POA and returns TRUE, that POA is returned from **find_POA**.

Note – This allows the same code, the **unknown_adapter** implementation, to be used to initialize a POA whether that POA is created explicitly by the application or as a side-effect of processing a request. Furthermore, it makes this initialization atomic with respect to delivery of requests to the POA.

11.3.4 ServantManager Interface

Servant managers are associated with POAs. A servant manager supplies a POA with the ability to activate objects on demand when the POA receives a request targeted at an inactive object. A servant manager is registered with a POA as a callback object, to be invoked by the POA when necessary. An application server that activates all its needed objects at the beginning of execution does not need to use a servant manager; it is used only for the case in which an object must be activated during request processing.

The **ServantManager** interface is itself empty. It is inherited by two other interfaces, **ServantActivator** and **ServantLocator**.

The two types of servant managers correspond to the POA's **RETAIN** policy (**ServantActivator**) and to the **NON_RETAIN** policy (**ServantLocator**). The meaning of the policies and the operations that are available for POAs using each policy are listed under the two types of derived interfaces.

Each servant manager type contains two operations, the first called to find and return a servant and the second to deactivate a servant. The operations differ according to the amount of information usable for their situation.

11.3.4.1 Common information for servant manager types

The two types of servant managers have certain semantics that are identical.

The **incarnate** and **preinvoke** operation may raise any system exception deemed appropriate (for example, OBJECT_NOT_EXIST if the object corresponding to the Object Id value has been destroyed).

Note – If a user-written routine (servant manager or method code) raises the OBJECT_NOT_EXIST exception, the POA does nothing but pass on that exception. It is the user's responsibility to deactivate the object if it had been previously activated.

The **incarnate** and **preinvoke** operation may also raise a ForwardRequest exception. If this occurs, the ORB is responsible for delivering the current request and subsequent requests to the object denoted in the **forward_reference** member of the exception. The behavior of this mechanism must be the functional equivalent of the GIOP location forwarding mechanism. If the current request was delivered via an implementation of the GIOP protocol (such as IIOP), the reference in the exception should be returned to the client in a reply message with **LOCATION_FORWARD** reply status. If some other protocol or delivery mechanism was used, the ORB is responsible for providing equivalent behavior, from the perspectives of the client and the object denoted by the new reference.

If a **ServantManager** returns a null Servant (or the equivalent in a language mapping) as the result of an **incarnate()** or **preinvoke()** operation, the POA will return the OBJ_ADAPTER system exception as the result of the request. If the **ServantManager** returns the wrong type of Servant, it is indeterminate when that error is detected. It is likely to result in a BAD_OPERATION or MARSHAL exception at the time of method invocation.

11.3.4.2 Locality Constraints

A ServantManager object must be local to the process containing the POA objects it is registered with.

11.3.5 ServantActivator Interface

When the POA has the **RETAIN** policy it uses servant managers that are ServantActivators. When using such servant managers, the following statements apply for a given **ObjectId** used in the **incarnate** and **etherealize** operations:

- Servants incarnated by the servant manager will be placed in the Active Object Map with objects they have activated.
- Invocations of **incarnate** on the servant manager are serialized.
- Invocations of **etherealize** on the servant manager are serialized.
- Invocations of **incarnate** and **etherealize** on the servant manager are mutually exclusive.
- Incarnations of a particular object may not overlap; that is, **incarnate** shall not be invoked with a particular **ObjectId** while, within the same POA, that **ObjectId** is in use as the **ObjectId** of an activated object or as the argument of a call to incarnate or etherealize that has not completed.

It should be noted that there may be a period of time between an object's deactivation and the etherealization (during which outstanding requests are being processed) in which arriving requests on that object should not be passed to its servant. During this period, requests targeted for such an object act as if the POA were in *holding* state until **etherealize** completes. If **etherealize** is called as a consequence of a **deactivate** call with an **etherealize_objects** parameter of TRUE, incoming requests are rejected.

It should also be noted that a similar situation occurs with **incarnate**. There may be a period of time after the POA invokes **incarnate** and before that method returns in which arriving requests bound for that object should not be passed to the servant.

A single servant manager object may be concurrently registered with multiple POAs. Invocations of **incarnate** and **etherealize** on a servant manager in the context of different POAs are not necessarily serialized or mutually exclusive. There are no assumptions made about the thread in which **etherealize** is invoked.

11.3.5.1 incarnate

```
Servant incarnate (
    in ObjectId         oid,
    in POA              adapter)
    raises (ForwardRequest);
```

This operation is invoked by the POA whenever the POA receives a request for an object that is not currently active, assuming the POA has the **USE_SERVANT_MANAGER** and **RETAIN** policies.

The **oid** parameter contains the **ObjectId** value associated with the incoming request. The **adapter** is an object reference for the POA in which the object is being activated.

The user-supplied servant manager implementation is responsible for locating or creating an appropriate servant that corresponds to the **ObjectId** value if possible. **incarnate** returns a value of type Servant, which is the servant that will be used to process the incoming request (and potentially subsequent requests, since the POA has the **RETAIN** policy).

The POA enters the returned Servant value into the Active Object Map so that subsequent requests with the same **ObjectId** value will be delivered directly to that servant without invoking the servant manager.

If the **incarnate** operation returns a servant that is already active for a different Object Id and if the POA also has the **UNIQUE_ID** policy, the **incarnate** has violated the POA policy and is considered to be in error. The POA will raise an OBJ_ADAPTER system exception for the request.

Note – If the same servant is used in two different POAs, it is legal for the POAs to use that servant even if the POAs have different Object Id uniqueness policies. The POAs do not interact with each other in this regard.

11.3.5.2 etherealize

```
void etherealize (
    in ObjectId         oid,
    in POA              adapter,
    in Servant          serv,
    in boolean          cleanup_in_progress,
    in boolean          remaining_activations);
```

This operation is invoked whenever a servant for an object is deactivated, assuming the POA has the **USE_SERVANT_MANAGER** and **RETAIN** policies. Note that an active servant may be deactivated by the servant manager via **etherealize** even if it was not incarnated by the servant manager.

The **oid** parameter contains the Object Id value of the object being deactivated. The **adapter** parameter is an object reference for the **POA** in whose scope the object was active. The **serv** parameter contains a reference to the servant that is associated with the object being deactivated. If the servant denoted by the **serv** parameter is associated with other objects in the **POA** denoted by the **adapter** parameter (that is, in the **POA**'s Active Object Map) at the time that **etherealize** is called, the **remaining_activations** parameter has the value **TRUE**. Otherwise, it has the value **FALSE**.

If the **cleanup_in_progress** parameter is **TRUE**, the reason for the **etherealize** operation is that either the **deactivate** or **destroy** operation was called with an **etherealize_objects** parameter of **TRUE**. If the parameter is **FALSE**, the **etherealize** operation is called for other reasons.

Deactivation occurs in the following circumstances:

- When an object is deactivated explicitly by an invocation of **POA::deactivate_object**.
- When the ORB or POA determines internally that an object must be deactivated. For example, an ORB implementation may provide policies that allow objects to be deactivated after some period of quiescence, or when the number of active objects reaches some limit.
- If **POAManager::deactivate** is invoked on a POA manager associated with a POA that has currently active objects.

Destroying a servant that is in the Active Object Map or is otherwise known to the POA can lead to undefined results.

In a multi-threaded environment, the **POA** makes certain guarantees that allow servant managers to safely destroy servants. Specifically, the servant's entry in the Active Object Map corresponding to the target object is removed before **etherealize** is called. Because calls to **incarnate** and **etherealize** are serialized, this prevents new requests for the target object from being invoked on the servant during etherealization. After removing the entry from the Active Object Map, if the **POA** determines before invoking **etherealize** that other requests for the same target object are already in progress on the servant, it delays the call to **etherealize** until all active methods for the target object have completed. Therefore, when **etherealize** is called, the servant manager can safely destroy the servant if it wants to, unless the **remaining_activations** argument is TRUE.

If the **etherealize** operation returns a system exception, the **POA** ignores the exception.

11.3.6 ServantLocator Interface

When the POA has the **NON_RETAIN** policy it uses servant managers that are **ServantLocators**. Because the POA knows that the servant returned by this servant manager will be used only for a single request, it can supply extra information to the servant manager's operations and the servant manager's pair of operations may be able to cooperate to do something different than a ServantActivator.

When the POA uses the **ServantLocator** interface, immediately after performing the operation invocation on the servant returned by **preinvoke**, the POA will invoke **postinvoke** on the servant manager, passing the **ObjectId** value and the **Servant** value as parameters (among others). The next request with this **ObjectId** value will then cause **preinvoke** to be invoked again. This feature may be used to force every request for objects associated with a **POA** to be mediated by the servant manager.

When using such a **ServantLocator**, the following statements apply for a given **ObjectId** used in the **preinvoke** and **postinvoke** operations:

- The servant returned by **preinvoke** is used only to process the single request that caused **preinvoke** to be invoked.
- No servant incarnated by the servant manager will be placed in the Active Object Map.

- When the invocation of the request on the servant is complete, **postinvoke** will be invoked for the object.
- No serialization of invocations of **preinvoke** or **postinvoke** may be assumed; there may be multiple concurrent invocations of **preinvoke** for the same **ObjectId**. (However, if the **SINGLE_THREAD_MODEL** policy is being used, that policy will serialize these calls.)
- The same thread will be used to **preinvoke** the object, process the request, and **postinvoke** the object.
- The **preinvoke** and postinvoke operations are always called in pairs in response to any ORB activity. In particular, for a response to a **GIOP Locate** message a **GIOP**-conforming ORB may (or may not) call preinvoke to determine whether the object could be served at this location. If the ORB makes such a call, whatever the result, the ORB does not invoke a method, but does call postinvoke before responding to the **Locate** message. (Note that the **ServantActivator** interface does not behave similarly with respect to a **GIOP Locate** message since the etherealize operation is not associated with request processing.)

11.3.6.1 preinvoke

```
Servant preinvoke(
    in ObjectId             oid,
    in POA                  adapter,
    in CORBA::Identifier    operation,
    out Cookie              the_cookie)
    raises (ForwardRequest);
```

This operation is invoked by the POA whenever the POA receives a request for an object that is not currently active, assuming the POA has the **USE_SERVANT_MANAGER** and **NON_RETAIN** policies.

The **oid** parameter contains the **ObjectId** value associated with the incoming request. The **adapter** is an object reference for the POA in which the object is being activated.

The user-supplied servant manager implementation is responsible for locating or creating an appropriate servant that corresponds to the **ObjectId** value if possible. **preinvoke** returns a value of type **Servant**, which is the servant that will be used to process the incoming request.

The **Cookie** is a type opaque to the **POA** that can be set by the servant manager for use later by **postinvoke**. The operation is the name of the operation that will be called by the **POA** when the servant is returned.

11.3.6.2 postinvoke

```
void postinvoke(
    in ObjectId             oid,
    in POA                  adapter,
    in CORBA::Identifier    operation,
    in Cookie               the_cookie,
```

in Servant the_servant);

This operation is invoked whenever a servant completes a request, assuming the POA has the **USE_SERVANT_MANAGER** and **NON_RETAIN** policies.

The **postinvoke** operation is considered to be part of a request on an object. That is, the request is not complete until postinvoke finishes. If the method finishes normally but postinvoke raises a system exception, the method's normal return is overridden; the request completes with the exception.

The **oid** parameter contains the Object Id value of the object on which the request was made. The **adapter** parameter is an object reference for the POA in whose scope the object was active. The **the_servant** parameter contains a reference to the servant that is associated with the object.

The **Cookie** is a type opaque to the **POA**; it contains any value that was set by the **preinvoke** operation. The operation is the name of the operation that was called by the **POA** for the request.

Destroying a servant that is known to the **POA** can lead to undefined results.

11.3.7 POA Policy Objects

Interfaces derived from **CORBA::Policy** are used with the **POA::create_POA** operation to specify policies that apply to a POA. Policy objects are created using factory operations on any pre-existing POA, such as the root POA. Policy objects are specified when a POA is created. Policies may not be changed on an existing POA. Policies are not inherited from the parent POA.

11.3.7.1 Thread Policy

Objects with the **ThreadPolicy** interface are obtained using the **POA::create_thread_policy** operation and passed to the **POA::create_POA** operation to specify the threading model used with the created POA. The value attribute of **ThreadPolicy** contains the value supplied to the **POA::create_thread_policy** operation from which it was obtained. The following values can be supplied.

- **ORB_CTRL_MODEL** - The ORB is responsible for assigning requests for an ORB-controlled POA to threads. In a multi-threaded environment, concurrent requests may be delivered using multiple threads.
- **SINGLE_THREAD_MODEL** - Requests for a single-threaded POA are processed sequentially. In a multi-threaded environment, all upcalls made by this POA to implementation code (servants and servant managers) are made in a manner that is safe for code that is multi-thread-unaware.

If no **ThreadPolicy** object is passed to **create_POA**, the thread policy defaults to **ORB_CTRL_MODEL**.

Note – In some environments, calling multi-thread-unaware code safely (that is, using the **SINGLE_THREAD_MODEL)** may mean that the POA will use only the main thread, in which case the application programmer is responsible to ensure that the main thread is given to the ORB, using **ORB::perform_work** or **ORB::run**.

POAs using the **SINGLE_THREAD_MODEL** may need to cooperate to ensure that calls are safe even when implementation code (such as a servant manager) is shared by multiple single-threaded POAs.

These models presume that the ORB and the application are using compatible threading primitives in a multi-threaded environment.

11.3.7.2 Lifespan Policy

Objects with the **LifespanPolicy** interface are obtained using the **POA::create_lifespan_policy** operation and passed to the **POA::create_POA** operation to specify the lifespan of the objects implemented in the created POA. The following values can be supplied.

- **TRANSIENT** - The objects implemented in the **POA** cannot outlive the **POA** instance in which they are first created. Once the **POA** is deactivated, use of any object references generated from it will result in an OBJECT_NOT_EXIST exception.
- **PERSISTENT** - The objects implemented in the **POA** can outlive the process in which they are first created.
  - Persistent objects have a **POA** associated with them (the **POA** which created them). When the ORB receives a request on a persistent object, it first searches for the matching **POA**, based on the names of the **POA** and all of its ancestors.
  - Administrative action beyond the scope of this specification may be necessary to inform the ORB's location service of the creation and eventual termination of existence of this **POA**, and optionally to arrange for on-demand activation of a process implementing this **POA**.
  - **POA** names must be unique within their enclosing scope (the parent **POA**). A portable program can assume that **POA** names used in other processes will not conflict with its own **POA** names. A conforming CORBA implementation will provide a method for ensuring this property.

If no **LifespanPolicy** object is passed to **create_POA**, the lifespan policy defaults to **TRANSIENT**.

11.3.7.3 Object Id Uniqueness Policy

Objects with the **IdUniquenessPolicy** interface are obtained using the **POA::create_id_uniqueness_policy** operation and passed to the **POA::create_POA** operation to specify whether the servants activated in the created **POA** must have unique object identities. The following values can be supplied.

- **UNIQUE_ID** - Servants activated with that POA support exactly one Object Id.
- **MULTIPLE_ID** - a servant activated with that POA may support one or more Object Ids.

If no **IdUniquenessPolicy** is specified at POA creation, the default is **UNIQUE_ID**.

11.3.7.4 Id Assignment Policy

Objects with the **IdAssignmentPolicy** interface are obtained using the **POA::create_id_assignment_policy** operation and passed to the **POA::create_POA** operation to specify whether Object Ids in the created **POA** are generated by the application or by the ORB. The following values can be supplied.

- **USER_ID** - Objects created with that **POA** are assigned Object Ids only by the application.
- **SYSTEM_ID** - Objects created with that **POA** are assigned Object Ids only by the **POA**. If the **POA** also has the **PERSISTENT** policy, assigned Object Ids must be unique across all instantiations of the same POA.

If no **IdAssignmentPolicy** is specified at POA creation, the default is **SYSTEM_ID**.

11.3.7.5 Servant Retention Policy

Objects with the **ServantRetentionPolicy** interface are obtained using the **POA::create_servant_retention_policy** operation and passed to the **POA::create_POA** operation to specify whether the created **POA** retains active servants in an Active Object Map. The following values can be supplied.

- **RETAIN -** The POA will retain active servants in its Active Object Map.
- **NON_RETAIN -** Servants are not retained by the POA.

If no **ServantRetentionPolicy** is specified at POA creation, the default is **RETAIN**.

Note – The **NON_RETAIN** policy requires either the **USE_DEFAULT_SERVANT** or **USE_SERVANT_MANAGER** policies.

11.3.7.6 Request Processing Policy

Objects with the RequestProcessingPolicy interface are obtained using the **POA::create_request_processing_policy** operation and passed to the **POA::create_POA** operation to specify how requests are processed by the created **POA**. The following values can be supplied.

- **USE_ACTIVE_OBJECT_MAP_ONLY** - If the Object Id is not found in the Active Object Map, an OBJECT_NOT_EXIST exception is returned to the client. The **RETAIN** policy is also required.

- **USE_DEFAULT_SERVANT** - If the Object Id is not found in the Active Object Map or the **NON_RETAIN** policy is present, and a default servant has been registered with the **POA** using the **set_servant** operation, the request is dispatched to the default servant. If no default servant has been registered, an OBJ_ADAPTER exception is returned to the client. The **MULTIPLE_ID** policy is also required.
- **USE_SERVANT_MANAGER** - If the Object Id is not found in the Active Object Map or the **NON_RETAIN** policy is present, and a servant manager has been registered with the **POA** using the **set_servant_manager** operation, the servant manager is given the opportunity to locate a servant or raise an exception. If no servant manager has been registered, an OBJECT_ADAPTER exception is returned to the client.

If no **RequestProcessingPolicy** is specified at **POA** creation, the default is **USE_ACTIVE_OBJECT_MAP_ONLY**.

By means of combining the **USE_ACTIVE_OBJECT_MAP_ONLY** / **USE_DEFAULT_SERVANT** / **USE_SERVANT_MANAGER** policies and the **RETAIN** / **NON_RETAIN** policies, the programmer is able to define a rich number of possible behaviors.

RETAIN and USE_ACTIVE_OBJECT_MAP_ONLY

This combination represents the situation where the **POA** does no automatic object activation (that is, the **POA** searches only the Active Object Map). The server must activate all objects served by the **POA** explicitly, using either the **activate_object** or **activate_object_with_id** operation.

RETAIN and USE_SERVANT_MANAGER

This combination represents a very common situation, where there is an Active Object Map and a **ServantManager**.

Because **RETAIN** is in effect, the application can call **activate_object** or **activate_object_with_id** to establish known servants in the Active Object Map for use in later requests.

If the **POA** doesn't find a servant in the Active Object Map for a given object, it tries to determine the servant by means of invoking incarnate in the **ServantManager** (specifically a ServantActivator) registered with the POA. If no **ServantManager** is available, the **POA** raises the OBJECT_ADAPTER system exception.

RETAIN and USE_DEFAULT_SERVANT

This combination represents the situation where there is a default servant defined for all requests involving unknown objects.

Because **RETAIN** is in effect, the application can call **activate_object** or **activate_object_with_id** to establish known servants in the Active Object Map for use in later requests.

The **POA** first tries to find a servant in the Active Object Map for a given object. If it does not find such a servant, it uses the default servant. If no default servant is available, the **POA** raises the OBJECT_ADAPTER system exception.

NON-RETAIN and USE_SERVANT_MANAGER

This combination represents the situation where one servant is used per method call.

The **POA** doesn't try to find a servant in the Active Object Map because the **ActiveObjectMap** does not exist. In every request, it will call preinvoke on the **ServantManager** (specifically a **ServantLocator**) registered with the **POA**. If no **ServantManager** is available, the **POA** will raise the OBJECT_ADAPTER system exception.

NON-RETAIN and USE_DEFAULT_SERVANT

This combination represents the situation where there is one single servant defined for all CORBA objects.

The **POA** does not try to find a servant in the Active Object Map because the **ActiveObjectMap** doesn't exist. In every request, the **POA** will invoke the appropriate operation on the default servant registered with the **POA**. If no default servant is available, the **POA** will raise the OBJECT_ADAPTER system exception.

11.3.7.7 Implicit Activation Policy

Objects with the **ImplicitActivationPolicy** interface are obtained using the **POA::create_implicit_activation_policy** operation and passed to the **POA::create_POA** operation to specify whether implicit activation of servants is supported in the created POA. The following values can be supplied.

- **IMPLICIT_ACTIVATION** - the POA will support implicit activation of servants. **IMPLICIT_ACTIVATION** also requires the **SYSTEM_ID** and **RETAIN** policies.
- **NO_IMPLICIT_ACTIVATION** - the POA will not support implicit activation of servants.

If no **ImplicitActivationPolicy** is specified at POA creation, the default is **NO_IMPLICIT_ACTIVATION**.

11.3.8 POA Interface

A POA object manages the implementation of a collection of objects. The POA supports a name space for the objects, which are identified by Object Ids.

A POA also provides a name space for POAs. A POA is created as a child of an existing POA, which forms a hierarchy starting with the root POA.

11.3.8.1 Locality Constraints

A **POA** object must not be exported to other processes, or externalized with **ORB::object_to_string**. If any attempt is made to do so, the offending operation will raise a MARSHAL system exception. An attempt to use a **POA** object with the DII may raise the NO_IMPLEMENT exception.

11.3.8.2 create_POA

```
POA create_POA(
    in string               adapter_name,
    in POAManager           a_POAManager,
    in CORBA::PolicyList    policies)
    raises (AdapterAlreadyExists, InvalidPolicy);
```

This operation creates a new POA as a child of the target POA. The specified name identifies the new POA with respect to other POAs with the same parent POA. If the target POA already has a child POA with the specified name, the AdapterAlreadyExists exception is raised.

If the **a_POAManager** parameter is null, a new **POAManager** object is created and associated with the new POA. Otherwise, the specified **POAManager** object is associated with the new POA. The **POAManager** object can be obtained using the attribute name **the_POAManager**.

The specified policy objects are associated with the POA and used to control its behavior. The policy objects are effectively copied before this operation returns, so the application is free to destroy them while the POA is in use. Policies are *not* inherited from the parent POA.

If any of the policy objects specified are not valid for the ORB implementation, if conflicting policy objects are specified, or if any of the specified policy objects require prior administrative action that has not been performed, an InvalidPolicy exception is raised containing the index in the policies parameter value of the first offending policy object.

Note – Creating a POA using a POA manager that is in the active state can lead to race conditions if the POA supports preexisting objects, because the new POA may receive a request before its adapter activator, servant manager, or default servant have been initialized. These problems do not occur if the POA is created by an adapter activator registered with a parent of the new POA, because requests are queued until the adapter activator returns. To avoid these problems when a POA must be explicitly initialized, the application can initialize the POA by invoking **find_POA** with a TRUE activate parameter.

11.3.8.3 find_POA

```
POA find_POA(
    in string       adapter_name,
```

```
    in boolean      activate_it)
    raises (AdapterNonExistent);
```

If the target **POA** is the parent of a child **POA** with the specified name (relative to the target **POA**), that child **POA** is returned. If a child **POA** with the specified name does not exist and the value of the **activate_it** parameter is **TRUE**, the target **POA**'s **AdapterActivator**, if one exists, is invoked, and, if it successfully activates the child **POA**, that child **POA** is returned. Otherwise, the AdapterNonExistent exception is raised.

11.3.8.4 destroy

```
void destroy(
    in boolean          etherealize_objects,
    in boolean          wait_for_completion);
```

This operation destroys the **POA** and all descendant **POA**s. All descendant **POA**s are destroyed (recursively) before the destruction of the containing **POA**. The **POA** so destroyed (that is, the **POA** with its name) may be re-created later in the same process. (This differs from the **POAManager::deactivate** operation that does not allow a re-creation of its associated **POA** in the same process. After a deactivate, re-creation is allowed only if the **POA** is later destroyed.)

When **destroy** is called the **POA** behaves as follows:

- The **POA** calls **destroy** on all of its immediate descendants.
- After all descendant **POA**s have been destroyed and their servants etherealized, the **POA** continues to process requests until there are no requests executing in the **POA**. The apparent destruction of the **POA** occurs only after all executing requests in the **POA** have completed. After destruction has become apparent, the **POA** may be re-created via either an **AdapterActivator** or a call to **create_POA**.
- If the **etherealize_objects** parameter is **TRUE**, the **POA** has the **RETAIN** policy, and a servant manager is registered with the **POA**, the etherealize operation on the servant manager is called for each active object in the Active Object Map. The apparent destruction of the POA occurs before any calls to etherealize are made. Thus, for example, an etherealize method that attempts to invoke operations on the **POA** receives the **OBJECT_NOT_EXIST** exception. Once apparent destruction has occurred, the **POA** behaves as if its **POAManager** is in the holding state until destruction is complete. Thus, for example, an invocation of **create_POA** with the same name blocks until **POA** destruction has finished.

The **wait_for_completion** parameter is handled as follows:

- If **wait_for_completion** is **TRUE** and the current thread is not in an invocation context dispatched from some **POA** belonging to the same ORB as this **POA**, the destroy operation returns only after all active requests have completed and all invocations of **etherealize** have completed.
- If **wait_for_completion** is **TRUE** and the current thread is in an invocation context dispatched from some **POA** belonging to the same ORB as this **POA**, the BAD_INV_ORDER exception is raised and **POA** destruction does not occur.

- If **wait_for_completion** is **FALSE**, the **destroy** operation destroys the POA and its children but waits neither for active requests to complete nor for etherealization to occur. If **destroy** is called multiple times before destruction is complete (because there are active requests), the **etherealize_objects** parameter applies only to the first call of **destroy**. Subsequent calls with conflicting **etherealize_objects** settings use the value of **etherealize_objects** from the first call. The **wait_for_completion** parameter is handled as defined above for each individual call (some callers may choose to block, while others may not).

11.3.8.5 *Policy Creation Operations*

```
ThreadPolicy create_thread_policy(
        in ThreadPolicyValue value);
LifespanPolicy create_lifespan_policy(
        in LifespanPolicyValue value);
IdUniquenessPolicy create_id_uniqueness_policy(
        in IdUniquenessPolicyValue value);
IdAssignmentPolicy create_id_assignment_policy(
        in IdAssignmentPolicyValue value);
ImplicitActivationPolicy create_implicit_activation_policy(
        in ImplicitActivationPolicyValue value);
ServantRetentionPolicy create_servant_retention_policy(
        in ServantRetentionPolicyValue value);
RequestProcessingPolicy create_request_processing_policy(
        in RequestProcessingPolicyValue value);
```

These operations each return a reference to a policy object with the specified value. The application is responsible for calling the inherited **destroy** operation on the returned reference when it is no longer needed.

11.3.8.6 *the_name*

```
readonly attribute string the_name;
```

This attribute identifies the POA relative to its parent. This name is assigned when the POA is created. The name of the root POA is system-dependent and should not be relied upon by the application.

11.3.8.7 *the_parent*

```
readonly attribute POA the_parent;
```

This attribute identifies the parent of the POA. The parent of the root POA is null.

11.3.8.8 *the_children*

```
readonly attribute POAList the_children;
```

This attribute identifies the current set of all child POAs of the POA. The set of child POAs includes only the POA's immediate children, and not their descendants.

11.3.8.9 the_POAManager

```
readonly attribute POAManager the_POAManager;
```

This attribute identifies the POA manager associated with the POA.

11.3.8.10 the_activator

```
attribute AdapterActivator the_activator;
```

This attribute identifies the adapter activator associated with the POA. A newly created POA has no adapter activator (the attribute is null). It is system-dependent whether the root POA initially has an adapter activator; the application is free to assign its own adapter activator to the root POA.

11.3.8.11 get_servant_manager

```
ServantManager get_servant_manager()
    raises(WrongPolicy);
```

This operation requires the **USE_SERVANT_MANAGER** policy; if not present, the WrongPolicy exception is raised.

If the **ServantRetentionPolicy** of the **POA** is **RETAIN**, then the **ServantManager** argument (**imgr**) shall support the ServantActivator interface (e.g., in C++ **imgr** is narrowable to **ServantActivator**). If the ServantRetentionPolicy of the POA is NON_RETAIN, then the **ServantManager** argument shall support the **ServantLocator** interface. If the argument is **nil**, or does not support the required interface, then the OBJ_ADAPTER exception is raised.

This operation returns the servant manager associated with the POA. If no servant manager has been associated with the POA, it returns a null reference.

11.3.8.12 set_servant_manager

```
void set_servant_manager(in ServantManager imgr)
    raises(WrongPolicy);
```

This operation requires the **USE_SERVANT_MANAGER** policy; if not present, the WrongPolicy exception is raised.

This operation sets the default servant manager associated with the POA. This operation may only be invoked once after a POA has been created. Attempting to set the servant manager after one has already been set will result in the BAD_INV_ORDER exception being raised.

11.3.8.13 get_servant

```
Servant get_servant()
    raises(NoServant, WrongPolicy);
```

This operation requires the **USE_DEFAULT_SERVANT** policy; if not present, the WrongPolicy exception is raised.

This operation returns the default servant associated with the POA. If no servant has been associated with the POA, the NoServant exception is raised.

11.3.8.14 set_servant

```
void set_servant(in Servant p_servant)
    raises(WrongPolicy);
```

This operation requires the **USE_DEFAULT_SERVANT** policy; if not present, the WrongPolicy exception is raised.

This operation registers the specified servant with the POA as the default servant. This servant will be used for all requests for which no servant is found in the Active Object Map.

11.3.8.15 activate_object

```
ObjectId activate_object(in Servant p_servant)
    raises (ServantAlreadyActive, WrongPolicy);
```

This operation requires the **SYSTEM_ID** and **RETAIN** policy; if not present, the WrongPolicy exception is raised.

If the POA has the **UNIQUE_ID** policy and the specified servant is already in the Active Object Map, the ServantAlreadyActive exception is raised. Otherwise, the **activate_object** operation generates an Object Id and enters the Object Id and the specified servant in the Active Object Map. The Object Id is returned.

11.3.8.16 activate_object_with_id

```
void activate_object_with_id(
    in          ObjectId oid,
    in Servant  p_servant)
    raises (ObjectAlreadyActive, ServantAlreadyActive, WrongPolicy);
```

This operation requires the **RETAIN** policy; if not present, the WrongPolicy exception is raised.

If the CORBA object denoted by the Object Id value is already active in this POA (there is a servant bound to it in the Active Object Map), the ObjectAlreadyActive exception is raised. If the POA has the **UNIQUE_ID** policy and the servant is already

in the Active Object Map, the ServantAlreadyActive exception is raised. Otherwise, the **activate_object_with_id** operation enters an association between the specified Object Id and the specified servant in the Active Object Map.

If the **POA** has the **SYSTEM_ID** policy and it detects that the Object Id value was not generated by the system or for this **POA**, the **activate_object_with_id** operation may raise the BAD_PARAM system exception. An ORB is not required to detect all such invalid Object Id values, but a portable application must not invoke **activate_object_with_id** on a **POA** that has the **SYSTEM_ID** policy with an Object Id value that was not previously generated by the system for that **POA**, or, if the **POA** also has the **PERSISTENT** policy, for a previous instantiation of the same **POA**.

11.3.8.17 deactivate_object

```
void deactivate_object(
    in ObjectId oid)
    raises (ObjectNotActive, WrongPolicy);
```

This operation requires the **RETAIN** policy; if not present, the WrongPolicy exception is raised.

This operation causes the **ObjectId** specified in the **oid** parameter to be deactivated. An **ObjectId** which has been deactivated continues to process requests until there are no active requests for that **ObjectId**. A deactivated **ObjectId** is removed from the Active Object Map when all requests executing for that **ObjectId** have completed. If a servant manager is associated with the **POA**, **ServantActivator::etherealize** is invoked with the **oid** and the associated servant after the **ObjectId** has been removed from the Active Object Map. Reactivation for the **ObjectId** blocks until etherealization (if necessary) is complete. This includes implicit activation (as described in etherealize) and explicit activation via **POA::activate_object_with_id**. Once an **ObjectId** has been removed from the Active Object Map and etherealized (if necessary) it may then be reactivated through the usual mechanisms.

The operation does not wait for requests or etherealization to complete and always returns immediately after deactivating the **ObjectId**.

Note – If the servant associated with the **oid** is serving multiple Object Ids, **ServantActivator::etherealize** may be invoked multiple times with the same servant when the other objects are deactivated. It is the responsibility of the object implementation to refrain from destroying the servant while it is active with any Id.

11.3.8.18 create_reference

```
Object create_reference (
    in CORBA::RepositoryId intf)
    raises (WrongPolicy);
```

This operation requires the **SYSTEM_ID** policy; if not present, the WrongPolicy exception is raised.

This operation creates an object reference that encapsulates a POA-generated Object Id value and the specified interface repository id. The specified repository id, which may be a null string, will become the **type_id** of the generated object reference. A repository id that does not identify the most derived interface of the object or one of its base interfaces will result in undefined behavior.

This operation does not cause an activation to take place. The resulting reference may be passed to clients, so that subsequent requests on those references will cause the appropriate servant manager to be invoked, if one is available. The generated Object Id value may be obtained by invoking **POA::reference_to_id** with the created reference.

11.3.8.19 create_reference_with_id

```
Object create_reference_with_id (
    in ObjectId                 oid,
    in CORBA::RepositoryId      intf);
```

This operation creates an object reference that encapsulates the specified Object Id and interface repository Id values. The specified repository id, which may be a null string, will become the **type_id** of the generated object reference. A repository id that does not identify the most derived interface of the object or one of its base interfaces will result in undefined behavior.

This operation does not cause an activation to take place. The resulting reference may be passed to clients, so that subsequent requests on those references will cause the object to be activated if necessary, or the default servant used, depending on the applicable policies.

If the **POA** has the **SYSTEM_ID** policy and it detects that the Object Id value was not generated by the system or for this POA, the **create_reference_with_id** operation may raise the BAD_PARAM system exception. An ORB is not required to detect all such invalid Object Id values, but a portable application must not invoke this operation on a POA that has the **SYSTEM_ID** policy with an Object Id value that was not previously generated by the system for that **POA**, or, if the **POA** also has the **PERSISTENT** policy, for a previous instantiation of the same **POA**.

11.3.8.20 servant_to_id

```
ObjectId servant_to_id(
    in Servant                  p_servant)
    raises (ServantNotActive, WrongPolicy);
```

This operation requires the **USE_DEFAULT_SERVANT** policy or a combination of the RETAIN policy and either the **UNIQUE_ID** or **IMPLICIT_ACTIVATION** policies; if not present, the WrongPolicy exception is raised.

This operation has four possible behaviors.

1. If the **POA** has the **UNIQUE_ID** policy and the specified servant is active, the Object Id associated with that servant is returned.

2. If the **POA** has the **IMPLICIT_ACTIVATION** policy and either the POA has the **MULTIPLE_ID** policy or the specified servant is not active, the servant is activated using a POA-generated Object Id and the Interface Id associated with the servant, and that Object Id is returned.

3. If the **POA** has the **USE_DEFAULT_SERVANT** policy, the servant specified is the default servant, and the operation is being invoked in the context of executing a request on the default servant, then the ObjectId associated with the current invocation is returned.

4. Otherwise, the ServantNotActive exception is raised.

11.3.8.21 servant_to_reference

```
Object servant_to_reference (
    in Servant                  p_servant)
    raises (ServantNotActive, WrongPolicy);
```

This operation requires the **RETAIN** policy and either the **UNIQUE_ID** or **IMPLICIT_ACTIVATION** policies if invoked outside the context of an operation dispatched by this POA. If this operation is not invoked in the context of executing a request on the specified servant and the policies specified previously are not present the WrongPolicy exception is raised.

This operation has four possible behaviors.

1. If the POA has both the **RETAIN** and the **UNIQUE_ID** policy and the specified servant is active, an object reference encapsulating the information used to activate the servant is returned.

2. If the **POA** has both the **RETAIN** and the **IMPLICIT_ACTIVATION** policy and either the **POA** has the **MULTIPLE_ID** policy or the specified servant is not active, the servant is activated using a POA-generated Object Id and the Interface Id associated with the servant, and a corresponding object reference is returned.

3. If the operation was invoked in the context of executing a request on the specified servant, the reference associated with the current invocation is returned.

4. Otherwise, the ServantNotActive exception is raised.

Note – The allocation of an Object Id value and installation in the Active Object Map caused by implicit activation may actually be deferred until an attempt is made to externalize the reference. The real requirement here is that a reference is produced that will behave appropriately (that is, yield a consistent Object Id value when asked politely).

11.3.8.22 reference_to_servant

```
Servant reference_to_servant (
    in Object                   reference)
    raises (ObjectNotActive, WrongAdapter, WrongPolicy);
```

This operation requires the **RETAIN** policy or the **USE_DEFAULT_SERVANT** policy. If neither policy is present, the WrongPolicy exception is raised.

If the **POA** has the **RETAIN** policy and the specified object is present in the Active Object Map, this operation returns the servant associated with that object in the Active Object Map. Otherwise, if the **POA** has the **USE_DEFAULT_SERVANT** policy and a default servant has been registered with the **POA**, this operation returns the default servant. Otherwise, the ObjectNotActive exception is raised.

If the object reference was not created by this **POA**, the WrongAdapter exception is raised.

11.3.8.23 reference_to_id

```
ObjectId reference_to_id(
    in Object                   reference)
    raises (WrongAdapter, WrongPolicy);
```

The WrongPolicy exception is declared to allow future extensions.

This operation returns the Object Id value encapsulated by the specified **reference**. This operation is valid only if the reference was created by the POA on which the operation is being performed. If the reference was not created by that POA, a WrongAdapter exception is raised. The object denoted by the reference does not have to be active for this operation to succeed.

11.3.8.24 id_to_servant

```
Servant id_to_servant(
    in ObjectId                 oid)
    raises (ObjectNotActive, WrongPolicy);
```

This operation requires the **RETAIN** policy or the **USE_DEFAULT_SERVANT** policy. If neither policy is present, the WrongPolicy exception is raised.

If the POA has the **RETAIN** policy and the specified ObjectId is in the Active Object Map, this operation returns the servant associated with that object in the Active Object Map. Otherwise, if the POA has the **USE_DEFAULT_SERVANT** policy and a default servant has been registered with the POA, this operation returns the default servant. Otherwise the ObjectNotActive exception is raised.

11.3.8.25 id_to_reference

```
Object id_to_reference(
    in ObjectId                 oid)
    raises (ObjectNotActive, WrongPolicy);
```

This operation requires the **RETAIN** policy; if not present, the WrongPolicy exception is raised.

If an object with the specified Object Id value is currently active, a reference encapsulating the information used to activate the object is returned. If the Object Id value is not active in the POA, an ObjectNotActive exception is raised.

11.3.9 Current operations

The **PortableServer::Current** interface, derived from **CORBA::Current**, provides method implementations with access to the identity of the object on which the method was invoked. The Current interface is provided to support servants that implement multiple objects, but can be used within the context of POA-dispatched method invocations on any servant. To provide location transparency, ORBs are required to support use of **Current** in the context of both locally and remotely invoked operations.

An instance of Current can be obtained by the application by issuing the **CORBA::ORB::resolve_initial_references("POACurrent")** operation. Thereafter, it can be used within the context of a method dispatched by the **POA** to obtain the **POA** and **ObjectId** that identify the object on which that operation was invoked.

11.3.9.1 get_POA

```
POA get_POA()
    raises (NoContext);
```

This operation returns a reference to the POA implementing the object in whose context it is called. If called outside the context of a POA-dispatched operation, a NoContext exception is raised.

11.3.9.2 get_object_id

```
ObjectId get_object_id()
    raises (NoContext);
```

This operation returns the ObjectId identifying the object in whose context it is called. If called outside the context of a POA-dispatched operation, a NoContext exception is raised.

11.4 IDL for PortableServer module

```
#pragma prefix "omg.org"
module PortableServer {
#   pragma version PortableServer 2.3
    interface POA;                  // forward declaration
    typedef sequence<POA> POAList;

    native Servant;

    typedef sequence<octet> ObjectId;
```

```
exception ForwardRequest {
    Object forward_reference;
};

// Policy interfaces

const CORBA::PolicyType THREAD_POLICY_ID = 16;
const CORBA::PolicyType LIFESPAN_POLICY_ID = 17;
const CORBA::PolicyType ID_UNIQUENESS_POLICY_ID = 18;
const CORBA::PolicyType ID_ASSIGNMENT_POLICY_ID = 19;
const CORBA::PolicyType IMPLICIT_ACTIVATION_POLICY_ID = 20;
const CORBA::PolicyType SERVANT_RETENTION_POLICY_ID = 21;
const CORBA::PolicyType REQUEST_PROCESSING_POLICY_ID = 22;

enum ThreadPolicyValue {
    ORB_CTRL_MODEL,
    SINGLE_THREAD_MODEL
};

interface ThreadPolicy : CORBA::Policy {
    readonly attribute ThreadPolicyValue value;
};

enum LifespanPolicyValue {
    TRANSIENT,
    PERSISTENT
};

interface LifespanPolicy : CORBA::Policy {
    readonly attribute LifespanPolicyValue value;
};

enum IdUniquenessPolicyValue {
    UNIQUE_ID,
    MULTIPLE_ID
};

interface IdUniquenessPolicy : CORBA::Policy {
    readonly attribute IdUniquenessPolicyValue value;
};

enum IdAssignmentPolicyValue {
    USER_ID,
    SYSTEM_ID
};

interface IdAssignmentPolicy : CORBA::Policy {
    readonly attribute IdAssignmentPolicyValue value;
};

enum ImplicitActivationPolicyValue {
```

```
        IMPLICIT_ACTIVATION,
        NO_IMPLICIT_ACTIVATION
};

interface ImplicitActivationPolicy : CORBA::Policy {
        readonly attribute ImplicitActivationPolicyValue value;
};

enum ServantRetentionPolicyValue {
        RETAIN,
        NON_RETAIN
};

interface ServantRetentionPolicy : CORBA::Policy {
        readonly attribute ServantRetentionPolicyValue value;
};

enum RequestProcessingPolicyValue {
        USE_ACTIVE_OBJECT_MAP_ONLY,
        USE_DEFAULT_SERVANT,
        USE_SERVANT_MANAGER
};

interface RequestProcessingPolicy : CORBA::Policy {
        readonly attribute RequestProcessingPolicyValue value;
};

// POAManager interface

interface POAManager {
        exception AdapterInactive{};

        enum State {HOLDING, ACTIVE, DISCARDING, INACTIVE};

        void activate()
                raises(AdapterInactive);
        void hold_requests(
                in boolean wait_for_completion)
                raises(AdapterInactive);
        void discard_requests(
                in boolean wait_for_completion)
                raises(AdapterInactive);
        void deactivate(
                in boolean etherealize_objects,
                in boolean wait_for_completion)
                raises(AdapterInactive);
        State get_state();
};

// AdapterActivator interface
```

```
    interface AdapterActivator {
#       pragma version AdapterActivator 2.3
        boolean unknown_adapter(
            in POA parent,
            in string name);
        };

    // ServantManager interface

    interface ServantManager{ };

    interface ServantActivator : ServantManager {
#       pragma version ServantActivator 2.3
        Servant incarnate (
            in ObjectId         oid,
            in POA              adapter)
        raises (ForwardRequest);

        void etherealize (
            in ObjectId         oid,
            in POA              adapter,
            in Servant          serv,
            in boolean          cleanup_in_progress,
            in boolean          remaining_activations);
        };

    interface ServantLocator : ServantManager {
#       pragma version ServantLocator 2.3
        native Cookie;
        Servant preinvoke(
            in ObjectId         oid,
            in POA              adapter,
            in CORBA::Identifier operation,
            out Cookie          the_cookie)
        raises (ForwardRequest);

        void postinvoke(
            in ObjectId         oid,
            in POA              adapter,
            in CORBA::Identifier operation,
            in Cookie           the_cookie,
            in Servant          the_servant
        );
    };

    // POA interface

    interface POA {
#       pragma version POA 2.3
        exception AdapterAlreadyExists {};
        exception AdapterNonExistent {};
```

```
exception InvalidPolicy {unsigned short index;};
exception NoServant {};
exception ObjectAlreadyActive {};
exception ObjectNotActive {};
exception ServantAlreadyActive {};
exception ServantNotActive {};
exception WrongAdapter {};
exception WrongPolicy {};

// POA creation and destruction

POA create_POA(
    in string adapter_name,
    in POAManager a_POAManager,
    in CORBA::PolicyList policies)
raises (AdapterAlreadyExists, InvalidPolicy);

POA find_POA(
    in string adapter_name,
    in boolean activate_it)
raises (AdapterNonExistent);

void destroy(
    in boolean etherealize_objects,
    in boolean wait_for_completion);

// Factories for Policy objects

ThreadPolicy create_thread_policy(
    in ThreadPolicyValue value);
LifespanPolicy create_lifespan_policy(
    in LifespanPolicyValue value);
IdUniquenessPolicy create_id_uniqueness_policy(
    in IdUniquenessPolicyValue value);
IdAssignmentPolicy create_id_assignment_policy(
    in IdAssignmentPolicyValue value);
ImplicitActivationPolicy create_implicit_activation_policy(
    in ImplicitActivationPolicyValue value);
ServantRetentionPolicy create_servant_retention_policy(
    in ServantRetentionPolicyValue value);
RequestProcessingPolicy create_request_processing_policy(
    in RequestProcessingPolicyValue value);

// POA attributes

readonly attribute string the_name;
readonly attribute POA the_parent;
readonly attribute POAList the_children;
readonly attribute POAManager the_POAManager;
attribute AdapterActivator the_activator;
```

```
// Servant Manager registration:

ServantManager get_servant_manager()
raises (WrongPolicy);

void set_servant_manager(
    in ServantManager imgr)
raises (WrongPolicy);

// operations for the USE_DEFAULT_SERVANT policy

Servant get_servant()
raises (NoServant, WrongPolicy);

void set_servant(in Servant p_servant)
raises (WrongPolicy);

// object activation and deactivation

ObjectId activate_object(
    in Servant p_servant)
raises (ServantAlreadyActive, WrongPolicy);

void activate_object_with_id(
    in ObjectId id,
    in Servant p_servant)
raises (ServantAlreadyActive, ObjectAlreadyActive, WrongPolicy);

void deactivate_object(
    in ObjectId oid)
raises (ObjectNotActive, WrongPolicy);

// reference creation operations

Object create_reference (
    in CORBA::RepositoryId intf)
raises (WrongPolicy);

Object create_reference_with_id (
    in ObjectId oid,
    in CORBA::RepositoryId intf)
raises (WrongPolicy);

// Identity mapping operations:

ObjectId servant_to_id(
    in Servant p_servant)
raises (ServantNotActive, WrongPolicy);

Object servant_to_reference(
    in Servant p_servant)
```

```
            raises (ServantNotActive, WrongPolicy);

            Servant reference_to_servant(
                in Object reference)
            raises(ObjectNotActive, Wrongpolicy);

            ObjectId reference_to_id(
                in Object reference)
            raises (WrongAdapter, WrongPolicy);

            Servant id_to_servant(
                in ObjectId oid)
            raises (ObjectNotActive, WrongPolicy);

            Object id_to_reference(in ObjectId oid)
                raises (ObjectNotActive, WrongPolicy);
        };

        // Current interface

        interface Current : CORBA::Current {
#           pragma version Current 2.3
            exception NoContext { };

            POA get_POA()
            raises (NoContext);

            ObjectId get_object_id()
                raises (NoContext);
        };
};
```

11.5 UML Description of PortableServer

The following diagrams were generated by an automated tool and then annotated with the cardinalities of the associations. They are intended to be an aid in comprehension to those who enjoy such representations. They are not normative.

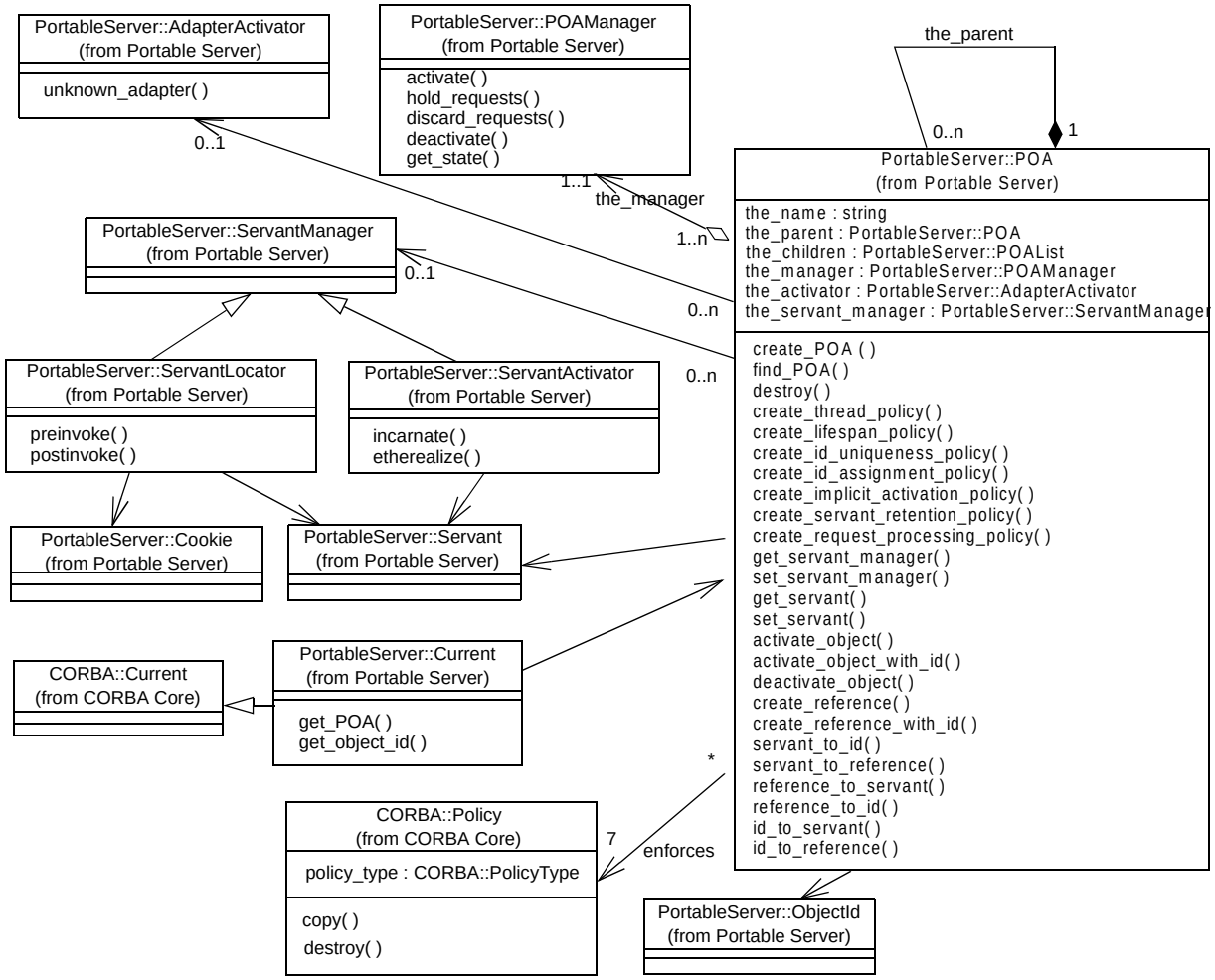


Figure 11-4 UML for main part of PortableServer

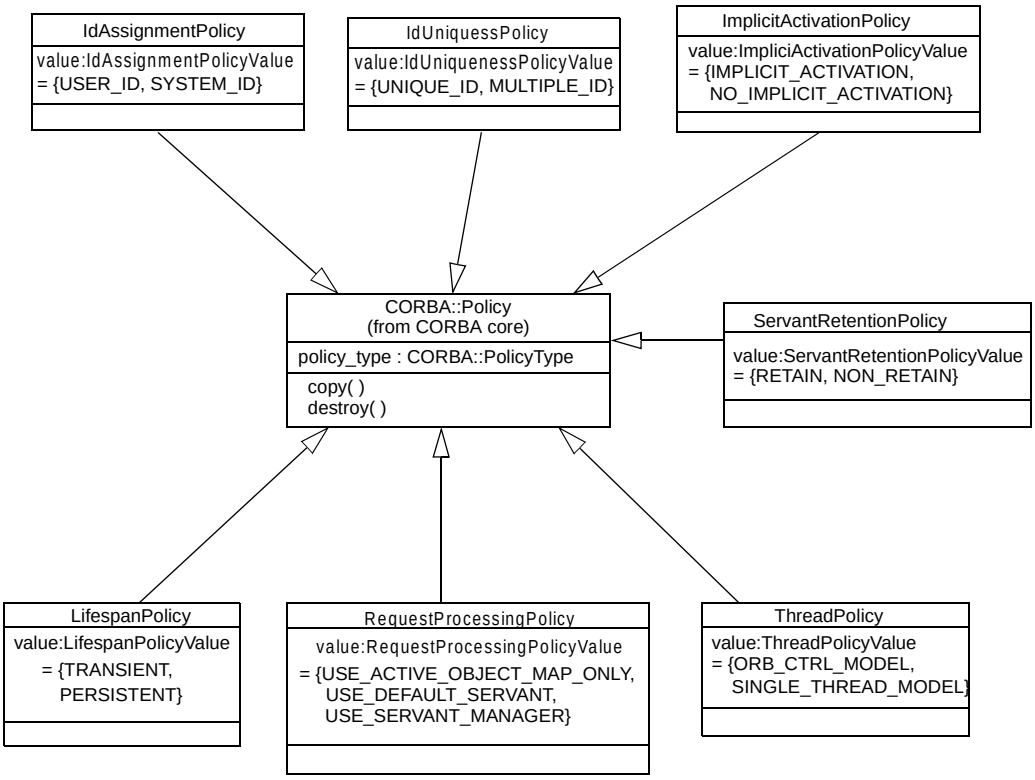


Figure 11-5 UML for PortableServer policies

11.6 Usage Scenarios

This section illustrates how different capabilities of the POA may be used in applications.

Note – In some of the following C++ examples, PortableServer names are not explicitly scoped. It is assumed that all the examples have the C++ statement
`using namespace PortableServer;`

11.6.1 Getting the root POA

All server applications must obtain a reference to the root POA, either to use it directly to manage objects, or to create new POA objects. The following example demonstrates how the application server can obtain a reference to the root POA.

```
// C++
CORBA::ORB_ptr orb = CORBA::ORB_init(argc, argv);
CORBA::Object_ptr pfobj =
orb->resolve_initial_references("RootPOA");
```

```
PortableServer::POA_ptr rootPOA;
rootPOA = PortableServer::POA::narrow(pfobj);
```

11.6.2 Creating a POA

For a variety of reasons, a server application might want to create a new POA. The POA is created as a child of an existing POA. In this example, it is created as a child of the root POA.

```
// C++
CORBA::PolicyList policies(2);
policies.length(2);
policies[0] = rootPOA->create_thread_policy(
PortableServer::ThreadPolicy::ORB_CTRL_MODEL);
policies[1] = rootPOA->create_lifespan_policy(
PortableServer::LifespanPolicy::TRANSIENT);
PortableServer::POA_ptr poa =
rootPOA->create_POA("my_little_poa",
PortableServer::POAManager::_nil(), policies);
```

11.6.3 Explicit Activation with POA-assigned Object Ids

By specifying the **SYSTEM_ID** policy on a POA, objects may be explicitly activated through the POA without providing a user-specified identity value. Using this approach, objects are activated by performing the **activate_object** operation on the POA with the object in question. For this operation, the POA allocates, assigns, and returns a unique identity value for the object.

Generally this capability is most useful for transient objects, where the Object Id needs to be valid only as long as the servant is active in the server. The Object Ids can remain completely hidden and no servant manager need be provided. When this is the case, the identity and lifetime of the servant and the abstract object are essentially equivalent. When POA-assigned Object Ids are used with persistent objects or objects that are activated on demand, the application must be able to associate the generated Object Id value with its corresponding object state.

This example illustrates a simple implementation of transient objects using POA-assigned Object Ids. It presumes a POA that has the **SYSTEM_ID**, **USE_SERVANT_MANAGER**, and **RETAIN** policies.

Assume this interface:

```
// IDL
    interface Foo {
        long doit();
    };
```

This might result in the generation of the following skeleton:

```
class POA_Foo : public ServantBase
{
   public:
      ...
   virtual CORBA::Long doit() = 0;
}
```

Derive your implementation:

```
class MyFooServant : public POA_Foo
{
   public:
      MyFooServant(POA_ptr poa, Long value)
      : my_poa(POA::_duplicate(poa)), my_value(value) {}
      ~MyFooServant() {CORBA::release(my_poa);}
      virtual POA_ptr _default_POA()
         {return POA::_duplicate(my_poa);}
      virtual Long doit() {return my_value;}
   protected:
      POA_ptr my_poa;
      Long my_value;
};
```

Now, somewhere in the program during initialization, probably in **`main()`**:

```
   MyFooServant* afoo = new MyFooServant(poa,27);
   PortableServer::ObjectId_var oid =
      poa->activate_object(afoo);
   Foo_var foo = afoo->_this();
   poa->the_POAManager()->activate();
   orb->run();
```

This object is activated with a generated Object Id.

11.6.4 Explicit Activation with User-assigned Object Ids

An object may be explicitly activated by a server using a user-assigned identity. This may be done for several reasons. For example, a programmer may know that certain objects are commonly used, or act as initial points of contact through which clients access other objects (for example, factories). The server could be implemented to create and explicitly activate these objects during initialization, avoiding the need for a servant manager.

If an implementation has a reasonably small number of servants, the server may be designed to keep them all active continuously (as long as the server is executing). If this is the case, the implementation need not provide a servant manager. When the server initializes, it could create all available servants, loading their state and identities from some persistent store. The POA supports an explicit activation operation, **activate_object_with_id**, that associates a servant with an Object Id. This operation would be used to activate all of the existing objects managed by the server during server initialization. Assuming the POA has the **USE_SERVANT_MANAGER** policy

and no servant manager is associated with a POA, any request received by the POA for an Object Id value not present in the Active Object Map will result in an **OBJECT_ADAPTER** exception.

In simple cases of well-known, long-lived objects, it may be sufficient to activate them with well-known Object Id values during server initialization, before activating the POA. This approach ensures that the objects are always available when the POA is active, and doesn't require writing a servant manager. It has severe practical limitations for a large number of objects, though.

This example illustrates the explicit activation of an object using a user-chosen Object Id. This example presumes a POA that has the **USER_ID**, **USE_SERVANT_MANAGER**, and **RETAIN** policies.

The code is like the previous example, but replace the last portion of the example shown above with the following code:

```
// C++
MyFooServant* afoo = new MyFooServant(poa, 27);
PortableServer::ObjectId_var oid =
   PortableServer::string_to_ObjectId("myLittleFoo");
poa->activate_object_with_id(oid.in(), afoo);
Foo_var foo = afoo->_this();
```

11.6.5 Creating References before Activation

It is sometimes useful to create references for objects before activating them. This example extends the previous example to illustrate this option:

```
// C++
PortableServer::ObjectId_var oid =
PortableServer::string_to_ObjectId("myLittleFoo");
CORBA::Object_var obj = poa->create_reference_with_id(
      oid.in(), "IDL:Foo:1.0");
Foo_var foo = Foo::_narrow(obj);

// ...later...
MyFooServant* afoo = new MyFooServant(poa, 27);
poa->activate_object_with_id(oid.in(), afoo);
```

11.6.6 Servant Manager Definition and Creation

Servant managers are object implementations, and are required to satisfy all of the requirements of object implementations necessary for their intended function. Because servant managers are local objects, and their use is limited to a single narrow role, some simplifications in their implementation are possible. Note that these simplifications are suggestions, not normative requirements. They are intended as examples of ways to reduce the programming effort required to define servant managers.

A servant manager implementation must provide the following things:

- implementation code for either
  - **incarnate( )** and **etherealize( )**, or
  - **preinvoke( )** and **postinvoke( )**
- implementation code for the servant operations, as for all servants

The first two are obvious; their content is dictated by the requirements of the implementation that the servant manager is managing. For the third point, the default servant manager on the root POA already supplies this implementation code. User-written servant managers will have to provide this themselves.

Since servant managers are objects, they themselves must be activated. It is expected that most servant managers can be activated on the root POA with its default set of policies (see “POA Creation” on page 11-6). It is for this reason that the root POA has the **IMPLICIT_ACTIVATION** policy so that a servant manager can easily be activated. Users may choose to activate a servant manager on other POAs.

The following is an example servant manager to activate objects on demand. This example presumes a POA that has the **USER_ID**, **USE_SERVANT_MANAGER**, and **RETAIN** policies.

Since RETAIN is in effect, the type of servant manager used is a ServantActivator. The ORB supplies a servant activator skeleton class in a library:

```
// C++
namespace POA_PortableServer
{
   class ServantActivator : public virtual ServantManager
   {
      public:
         virtual ~ServantActivator();
         virtual Servant incarnate(
            const ObjectId& POA_ptr poa) = 0;
         virtual void etherealize(
            const ObjectId&, POA_ptr poa,
            Servant, Boolean remaining_activations) = 0;
   };
};
```

A ServantActivator servant manager might then look like:

```
// C++
class MyFooServantActivator : public
   POA_PortableServer::ServantActivator
{
   public:
      // ...
      Servant incarnate(
         const ObjectId& oid, POA_ptr poa)
      {
```

```
            String_var s = PortbleServer::ObjectId_to_string
                           (oid);
            if (strcmp(s, "myLittleFoo") == 0) {
               return new MyFooServant(poa, 27);
            else {
               throw CORBA::OBJECT_NOT_EXIST();
            }
         }

         void etherealize(
                     const ObjectId& oid,
                     POA_ptr poa,
                     Servant servant,
                     Boolean remaining_activations)
         {
            if (remaining_activations == 0)
               delete servant;
         }
         // ...
};
```

11.6.7 Object Activation on Demand

The precondition for this scenario is the existence of a client with a reference for an object with which no servant is associated at the time the client makes a request on the reference. It is the responsibility of the ORB, in collaboration with the POA and the server application to find or create an appropriate servant and perform the requested operation on it. Such an object is said to be *incarnated* (or *incarnation*) when it has an active servant. Note that the client had to obtain the reference in question previously from some source. From the client's perspective, the abstract object exists as long as it holds a reference, until it receives an OBJECT_NOT_EXIST system exception in a reply from an attempted request on the object. Incarnation state does not imply existence or non-existence of the abstract object.

Note – This specification does not address the issues of communication or server process activation, as they are immaterial to the POA interface and operation. It is assumed that the ORB activates the server if necessary, and can deliver the request to the appropriate POA.

To support object activation on demand, the server application must register a servant manager with the appropriate POA. Upon receiving the request, if the POA consults the Active Object Map and discovers that there is no active servant associated with the target Object Id, the POA invokes the **incarnate** operation on the servant manager.

Note – An implication that this model has for GIOP is that the object key in the request message must encapsulate the Object Id value. In addition, it may encapsulate other values as necessitated by the ORB implementation. For example, the server must be able to determine to which POA the request should be directed. It could assign a different communication endpoint to each POA so that the POA identity is implicit in the request, or it could use a single endpoint for the entire server and encapsulate POA identities in object key values. Note that this is not a concrete requirement; the object key may not actually contain any of those values. Whatever the concrete information is, the ORB and POA must be able to use it to find the servant manager, invoke activate if necessary (which requires the actual Object Id value), and/or find the active servant in some map.

The **incarnate** invocation passes the Object Id value to the servant manager. At this point, the servant manager may take any action necessary to produce a servant that it considers to be a valid incarnation of the object in question. The operation returns the servant to the POA, which invokes the operation on it. The **incarnate** operation may alternatively raise an OBJECT_NOT_EXIST system exception that will be returned to the invoking client. In this way, the user-supplied implementation is responsible for determining object existence and non-existence.

After activation, the POA maintains the association of the servant and the Object Id in the Active Object Map. (This example presumes the **RETAIN** and **USE_SERVANT_MANAGER** policies.)

As an obvious example of transparent activation, the Object Id value could contain a key for a record in a database that contains the object's state. The servant manager would retrieve the state from the database, construct a servant of the appropriate implementation class (assuming an object-oriented programming language), initialize it with the state from the database, and return it to the POA.

The example servant manager in the last section ("Servant Manager Definition and Creation" on page 11-52) could be used for this scenario. Recall that the POA would have the **USER_ID**, **USE_SERVANT_MANAGER**, and **RETAIN** policies.

Given such a ServantActivator, all that remains is initialization code such as the following.

```
PortableServer::ObjectId_var oid =
   PortableServer::string_to_ObjectId("myLittleFoo");
CORBA::Object_var obj = poa->create_reference_with_id(
   oid, "IDL:foo:1.0");
MyFooServantActivator* fooIM = new MyFooServantActivator;
ServantActivator_var IMref = fooIM->_this();
poa->set_servant_manager(IMref);
poa->the_POAmanager()->activate();
orb->run();
```

11.6.8 Persistent Objects with POA-assigned Ids

It is possible to access the Object Id value assigned to an object by the POA, with the **POA::reference_to_id** operation. If the reference is for an object managed by the POA that is the operation's target, the operation will return the Object Id value, whether it was assigned by the POA or the user. By doing this, an implementation may provide a servant manager that associates the POA-allocated Object Id values with persistently stored state. It may also pass the POA-allocated Object Id values to POA operations such as **activate_object_with_id** and **create_reference_with_id**.

A POA with the **PERSISTENT** policy may be destroyed and later reinstantiated in the same or a different process. A POA with both the **SYSTEM_ID** and **PERSISTENT** policies generates Object Id values that are unique across all instantiations of the same POA.

11.6.9 Multiple Object Ids Mapping to a Single Servant

Each POA is created with a policy that indicates whether or not servants are allowed to support multiple object identities simultaneously. If a POA allows multiple identities per servant, the POA's treatment of the servants is affected in the following ways:

- Servants of the type may be explicitly activated multiple times with different identity values without raising an exception.
- A servant cannot be mapped onto or converted to an individual object reference using that POA, since the identity is potentially ambiguous.

11.6.10 One Servant for all Objects

By using the **USE_DEFAULT_SERVANT** policy, the developer can create a POA that will use a single servant to implement all of its objects. This approach is useful when there is very little data associated with each object, so little that the data can be encoded in the Object Id.

The following example illustrates this approach by using a single servant to incarnate all CORBA objects that export a given interface in the context of a server. This example presumes a POA that has the **USER_ID**, **NON_RETAIN,** and **USE_DEFAULT_SERVANT** policies.

Two interfaces are defined in IDL. The **FileDescriptor** interface is supported by objects that will encapsulate access to operations in a file associated with a file system. Global operations in a file system, such as the ones necessary to create **FileDescriptor** objects, are supported by objects that export the **FileSystem** interface.

```
// IDL
interface FileDescriptor {
    typedef sequence<octet> DataBuffer;

    long write (in DataBuffer buffer);
    DataBuffer read (
```

```
            in long num_bytes);
        void destroy ();
    };

    interface FileSystem {
        ...
        FileDescriptor open (
            in string file_name,
            in long flags);
        ...
    };
```

Implementation of these two IDL interfaces may inherit from static skeleton classes generated by an IDL to C++ compiler as follows:

```
// C++
class FileDescriptorImpl : public POA_FileDescriptor
{
   public:
      FileDescriptorImpl(POA_ptr poa);
      ~FileDescriptorImpl();
      POA_ptr _default_POA();
      CORBA::Long write(
         const FileDescriptor::DataBuffer& buffer);
      FileDescriptor::DataBuffer* read(
         CORBA::Long num_bytes);
      void destroy();
   private:
      POA_ptr my_poa;
};

class FileSystemImpl : public POA_FileSystem
{
   public:
      FileSystemImpl(POA_ptr poa);
      ~FileSystemImpl();
      POA_ptr _default_POA();
      FileDescriptor_ptr open(
         const char* file_name, CORBA::Long flags);
   private:
      POA_ptr my_poa;
      FileDescriptorImpl* fd_servant;
};
```

A single servant may be used to serve all requests issued to all **FileDescriptor** objects created by a **FileSystem** object. The following fragment of code illustrates the steps to perform when a **FileSystem** servant is created.

```
// C++
FileSystemImpl::FileSystemImpl(POA_ptr poa)
   : my_poa(POA::_duplicate(poa))
```

```
{
   fd_servant = new FileDescriptorImpl(poa);
   poa->set_servant(fd_servant);
};
```

The following fragment of code illustrates how **FileDescriptor** objects are created as a result of invoking an operation (**open**) exported by a **FileSystem** object. First, a local file descriptor is created using the appropriate operating system call. Then a CORBA object reference is created and returned to the client. The value of the local file descriptor will be used to distinguish the new **FileDescriptor** object from other **FileDescriptor** objects. Note that FileDescriptor objects in the example are transient, since they use the value of their file descriptors for their ObjectIds, and of course the file descriptors are only valid for the life of a process.

```
// C++
FileDescriptor_ptr
FileSystemImpl::open(
   const char* file_name, CORBA::Long flags)
{
   int fd = ::open(file_name, flags);
   ostrstream ostr;
   ostr << fd;
   PortableServer::ObjectId_var oid =
   PortableServer::string_to_ObjectId(ostr.str());
   Object_var obj = my_poa->create_reference_with_id(
      oid.in(),"IDL:FileDescriptor:1.0");
   return FileDescriptor::_narrow(obj);
};
```

Any request issued to a **FileDescriptor** object is handled by the same servant. In the context of a method invocation, the servant determines which particular object is being incarnated by invoking an operation that returns a reference to the target object and, after that, invoking **POA::reference_to_id**. In C++, the operation used to obtain a reference to the target object is **_this()**. Typically, the **ObjectId** value associated with the reference will be used to retrieve the state of the target object. However, in this example, such a step is not required since the only thing that is needed is the value for the local file descriptor and that value coincides with the **ObjectId** value associated with the reference.

Implementation of the **read** operation is rather simple. The servant determines which object it is incarnating, obtains the local file descriptor matching its identity, performs the appropriate operating system call, and returns the result in a **DataBuffer** sequence.

```
// C++
FileDescriptor::DataBuffer*
FileDescriptorImpl::read(CORBA::Long num_bytes)
{
   FileDescriptor_var me = _this();
   PortableServer::ObjectId_var oid =
      my_poa->reference_to_id(me.in());
   CORBA::String_var s =
```

```
        PortableServer::ObjectId_to_string(oid.in());
    istrstream is(s);
    int fd;
    is >> fd;
    CORBA::Octet* buffer = DataBuffer::alloc_buf(num_bytes);
    int len = ::read(fd, buffer, num_bytes);
    DataBuffer* result = new DataBuffer(len, len, buffer, 1);
    return result;
};
```

Using a single servant per interface is useful in at least two situations.

- In one case, it may be appropriate for encapsulating access to legacy APIs that are not object-oriented (system calls in the Unix environment, as we have shown in the example).
- In another case, this technique is useful in handling scalability issues related to the number of CORBA objects that can be associated with a server. In the example above, there may be a million **FileDescriptor** objects in the same server and this would only require one entry in the ORB. Although there are operating system limitations in this respect (a Unix server is not able to open so many local file descriptors) the important point to take into account is that usage of CORBA doesn't introduce scalability problems but provides mechanisms to handle them.

11.6.11 Single Servant, Many Objects and Types, Using DSI

The ability to associate a single DSI servant with many CORBA objects is rather powerful in some scenarios. It can be the basis for development of gateways to legacy systems or software that mediates with external hardware, for example.

Usage of the DSI is illustrated in the following example. This example presumes a POA that supports the **USER_ID**, **USE_DEFAULT_SERVANT** and **RETAIN** policies.

A single servant will be used to incarnate a huge number of CORBA objects, each of them representing a separate entry in a Database. There may be several types of entries in the Database, representing different entity types in the Database model. Each type of entry in the Database is associated with a separate interface which comprises operations supported by the Database on entries of that type. All these interfaces inherit from the **DatabaseEntry** interface. Finally, an object supporting the **DatabaseAgent** interface supports basic operations in the database such as creating a new entry, destroying an existing entry, etc.

```
// IDL
    interface DatabaseEntry {
        readonly attribute string name;
    };


    interface Employee : DatabaseEntry {
        attribute long id;
```

```
        attribute long salary;
    };
...

    interface DatabaseAgent {
        DatabaseEntry create_entry (
            in string key,
            in CORBA::Identifier entry_type,
            in NVPairSequence initial_attribute_values
        );

    void destroy_entry (
        in string key);
        ...
};
```

Implementation of the **DatabaseEntry** interface may inherit from the standard dynamic skeleton class as follows:

```
// C++
class DatabaseEntryImpl :
   public PortableServer::DynamicImplementation
{
   public:
      DatabaseEntryImpl (DatabaseAccessPoint db);
      virtual void invoke (ServerRequest_ptr request);
      ~DatabaseEntryImpl ();

      virtual POA_ptr _default_POA()
      {
         return poa;
      }
};
```

On the other hand, implementation of the **DatabaseAgent** interface may inherit from a static skeleton class generated by an IDL to C++ compiler as follows:

```
// C++
class DatabaseAgentImpl :
   public DatabaseAgentImplBase
{
   protected:
      DatabaseAccessPoint mydb;
      DatabaseEntryImpl * common_servant;
   public:
      DatabaseAgentImpl ();
      virtual DatabaseEntry_ptr create_entry (
         const char * key,
         const char * entry_type,
         const NVPairSequence& initial_attribute_values
      );
```

```
      virtual void destroy_entry (const char * key);
      ~DatabaseAgentImpl ();
};
```

A single servant may be used to serve all requests issued to all **DatabaseEntry** objects created by a **DatabaseAgent** object. The following fragment of code illustrates the steps to perform when a **DatabaseAgent** servant is created. First, access to the database is initialized. As a result, some kind of descriptor (a DatabaseAccessPoint local object) used to operate on the database is obtained. Finally, a servant will be created and associated with the POA.

```
// C++
void DatabaseAgentImpl::DatabaseAgentImpl ()
{
   mydb = ...;
   common_servant = new DatabaseEntryImpl(mydb);
   poa->set_servant(common_servant);
};
```

The code used to create **DatabaseEntry** objects representing entries in the database is similar to the one used for creating **FileDescriptor** objects in the example of the previous section. In this case, a new entry is created in the database and the key associated with that entry will be used to represent the identity for the corresponding **DatabaseEntry** object. All requests issued to a **DatabaseEntry** object are handled by the same servant because references to this type of object are associated with a common POA created with the **USE_DEFAULT_SERVANT** policy.

```
// C++
DatabaseEntry_ptr DatabaseAgentImpl::create_entry (
   const char * key,
   const char * entry_type,
   const NVPairSequence& initial_attribute_values)

   // creates a new entry in the database:
   mydb->new_entry (key, ...);

   // creates a reference to the CORBA object used to
   // encapsulate access to the new entry in the database.
   // There is an interface for each entry type:
   CORBA::Object_ptr obj = poa->create_reference_with_id(
      string_to_ObjectId (key),
      identifierToRepositoryId (entry_type),
   );

   DatabaseEntry_ptr entry = DatabaseEntry::_narrow (obj);
      CORBA::release (obj);
   return entry;
};
```

Any request issued to a **DatabaseEntry** object is handled by the same servant. In the context of a method invocation, the servant determines which particular object it is incarnating, obtains the database key matching its identity, invokes the appropriate operation in the database and returns the result as an output parameter in the **ServerRequest** object.

Sometimes, a program may need to determine the type of an entry in the database in order to invoke operations on the entry. If that is the case, the servant may obtain the type of an entry based on the interface supported by the **DatabaseEntry** object encapsulating access to that entry. This interface may be obtained by means of invoking the **get_interface** operation exported by the reference to the **DatabaseEntry** object.

```
// C++
void DatabaseEntryImpl::invoke (ServerRequest_ptr request)
{
   CORBA::Object_ptr current_obj = _this ();

   // The servant determines the key associated with
   // the database entry represented by current_obj:
   PortableServer::ObjectId oid =
      poa->reference_to_id (current_obj);
   char * key = ObjectId_to_string (oid);

   // The servant handles the incoming CORBA request. This
   // typically involves the following steps:
   // 1. mapping the CORBA request into a database request
   //    using the key obtained previously
   // 2. constructing output parameters to the CORBA request
   //    from the response to the database request
      ...
};
```

Note that in this example, we may have a billion **DatabaseEntry** objects in a server requiring only a single entry in map tables supported by the POA (that is, the ORB at the server). No permanent storage is required for references to DatabaseEntry objects at the server. Actually, references to DatabaseEntry objects will only occupy space:

- at clients, as long as those references are used; or
- at the server, only while a request is being served.

Scalability problems can be handled using this technique. There are many scenarios where this scalability causes no penalty in terms of performance (basically, when there is no need to restore the state of an object, each time a request to it is being served).

Interoperability Overview 12

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.

Contents

This chapter contains the following sections.

12.1 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.

12.1.1 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” that communicate using the IIOP. This approach works for both stand-alone ORBs, and networked ones that 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.

12.1.2 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 to which those systems conform to the CORBA Object Model.

12.1.3 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, 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.

12.1.4 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 build complete programs. The IIOP and other similar mappings to different transports, are concrete realizations of the abstract GIOP definitions, as shown in Figure 12-1 on page 12-4.

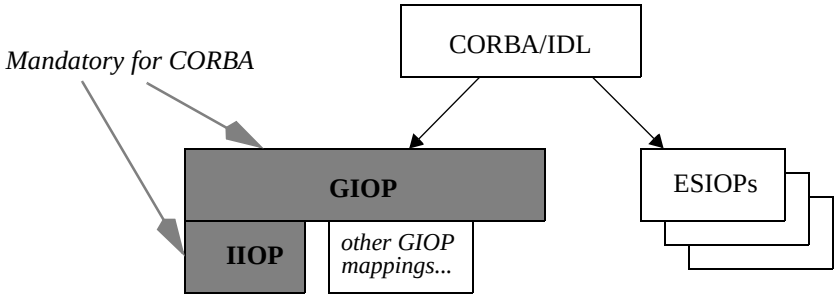


Figure 12-1 Inter-ORB Protocol Relationships.

12.1.5 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.

12.2 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 13.2, “ORBs and ORB Services,” on page 13-3). 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 the Dynamic Skeleton Interface chapter.

- 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 in the ORB Interface chapter of this book. The Relationship Service is described in *CORBAservices: Common Object Service Specifications*; refer to the CosObjectIdentity Module section of that chapter.

12.3 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.

12.3.1 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 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.

12.3.2 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.

12.3.3 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 other arbitrary 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.

12.3.4 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 described in the Interface Repository chapter of this book.
- An Internet Inter-ORB Protocol (IIOP) (explained in the Building Inter-ORB Brdiges chapter) 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 an interoperability-compliant system. However, any implementation that chooses to use the other protocols defined by the CORBA interoperability specifications must adhere to those specifications to be compliant with CORBA interoperability.

Figure 12-2 on page 12-7 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.

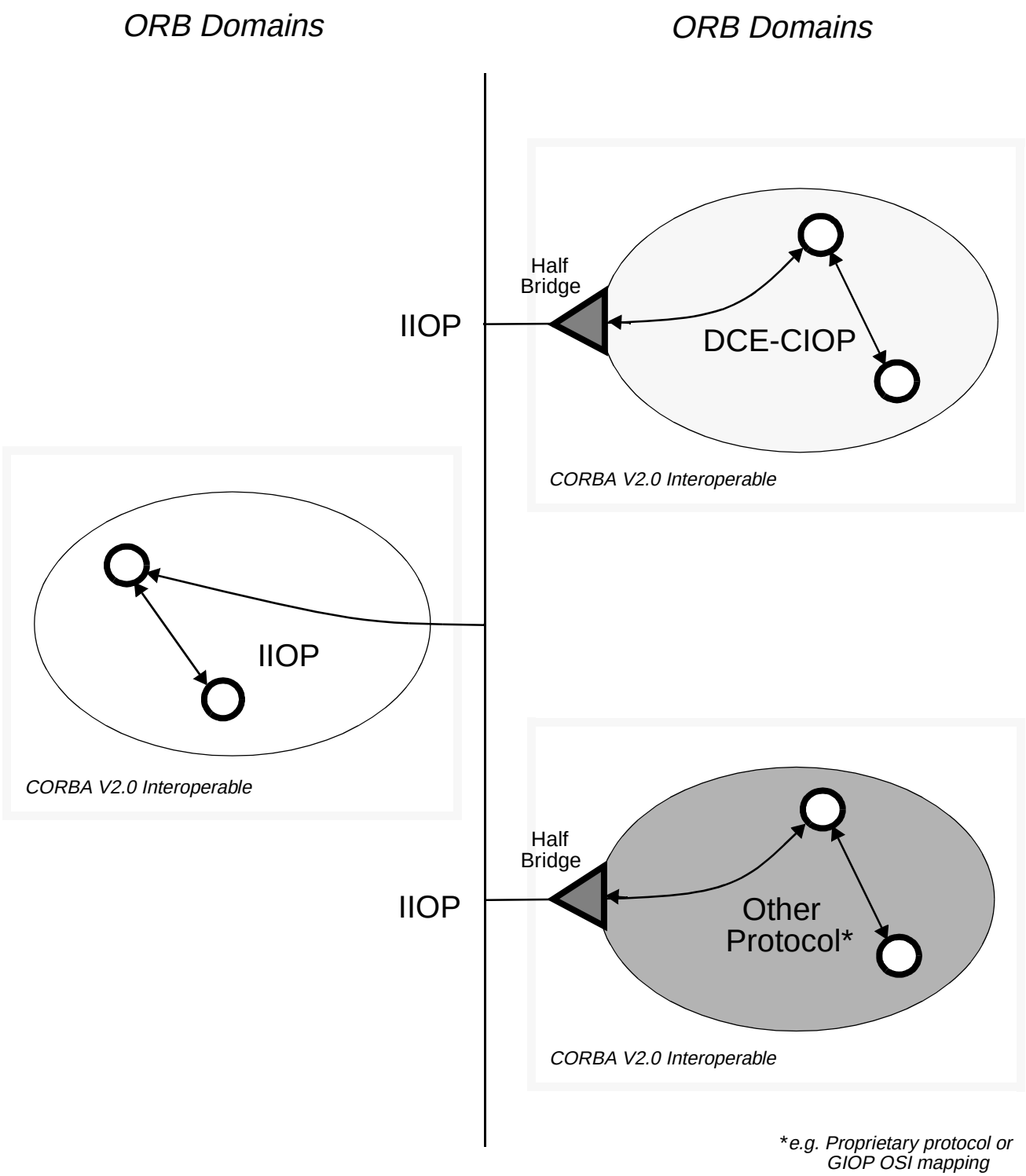


Figure 12-2 Examples of CORBA Interoperability Compliance

12.4 Motivating Factors

This section explains the factors that motivated the creation of interoperability specifications.

12.4.1 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 protocol, 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 of 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.

12.4.2 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 is 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 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, management of the state in an ORB (object reference location and translation information, interface repositories, per-object data) might also be done by creating sub-ORBs.

12.4.3 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 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.

12.5 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 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 not be 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.

12.5.1 Non-Goals

The following were taken into account, but were not goals:

- Support for security
- Support for future ORB Services

ORB Interoperability Architecture 13

The ORB Interoperability Architecture chapter has been updated based on CORE changes from ptc/98-09-04 and the Objects by Value documents (orbos/98-01-18 and ptc/98-07-06).

Contents

This chapter contains the following sections.

13.1 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.

13.1.1 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.

13.1.2 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.

13.2 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*.

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 be 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.

13.2.1 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.

13.2.2 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.

1. 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.
2. 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.
3. The server object performs the requested operation.
4. 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 an 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.

13.2.3 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 (e.g., 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 way 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.

13.3 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.

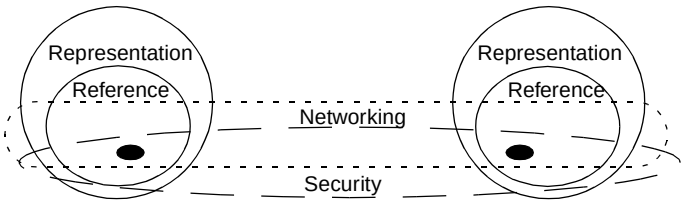


Figure 13-1 Different Kinds of Domains can Coexist.

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 (e.g., management domains, naming domains, language domains, and technology domains).

13.3.1 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 to and set up by their administrators.

13.3.2 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* 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*. Variances from this model could easily compromise some aspects of interoperability.

13.4 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.

13.4.1 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* 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).

13.4.2 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.

13.4.3 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.

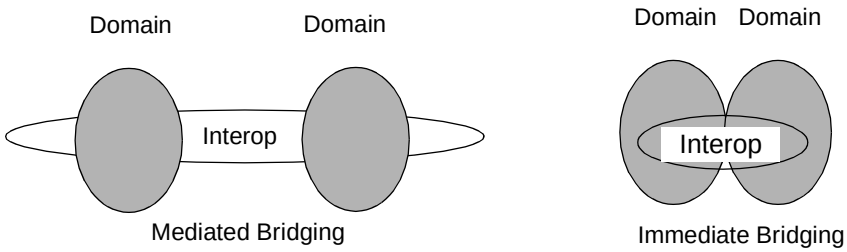


Figure 13-2 Two bridging techniques, different uses of an intermediate form agreed on between the two domains.

13.4.3.1 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 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), to intermediate bridges to the common form.

13.4.3.2 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.

13.4.3.3 *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.

13.4.3.4 *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.

13.4.4 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.

13.4.5 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.

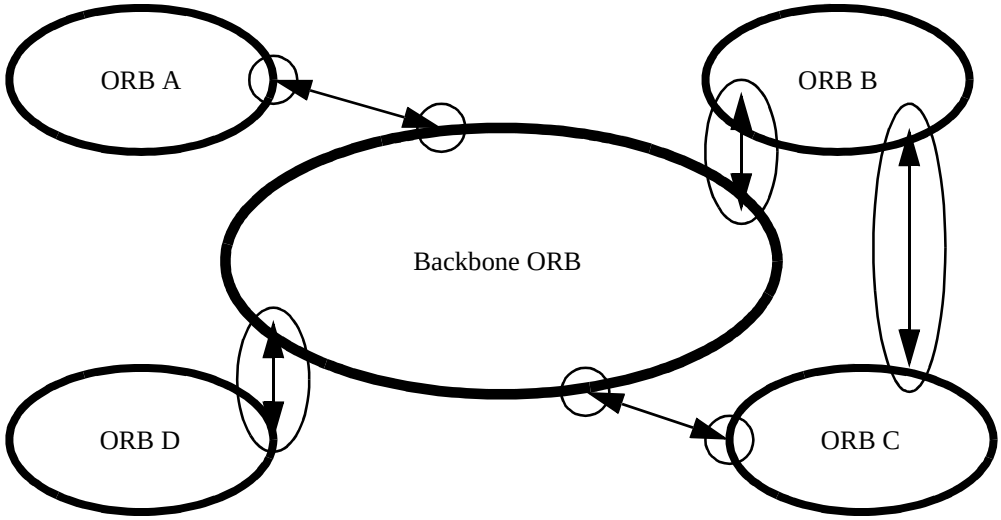


Figure 13-3 An ORB chosen as a backbone will connect other ORBs through bridges, both full-bridges and half-bridges.

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.

13.5 Object Addressing

The Object Model (see Chapter 1, Requests) 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.

13.5.1 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.

13.5.2 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*.

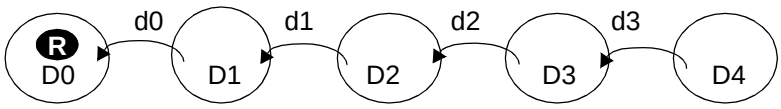


Figure 13-4 Reference encapsulation adds domain information during bridging.

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)*.

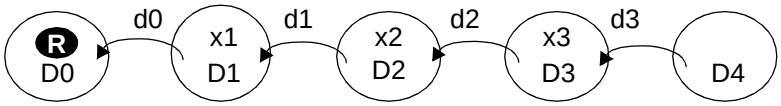


Figure 13-5 Domain Reference Translation *substitutes domain references* during bridging.

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*, 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 and Dynamic Invocation Interface)
- 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.

13.6 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 the *General Inter-ORB Protocol* chapter, *Object References* section.

13.6.1 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 "Selection of ORB Services" on page 13-4, several different ORB Services might be involved in an invocation. Providing information about those services in a standardized way could in many cases reduce or eliminate negotiation overhead in selecting them.

13.6.2 Interoperable Object References: IORs

To provide the information above, an "Interoperable Object Reference," (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*. Each profile supports one or more protocols and encapsulates all the basic information the protocols it supports need to identify an object. Any single profile holds enough information to drive a complete invocation using any of the protocols it supports; the content and structure of those profile entries are wholly specified by these protocols. 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 the OMG. The ones defined here are for the IIOP (see Section 15.7.3, "IIOP IOR Profile Components," on page 15-51) and for use in "multiple component profiles." Profile tags in the range **0x80000000** through **0xffffffff** are reserved for future use, and are not currently available for assignment.

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.

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). A Null TypeID is the only mechanism that can be used to represent the type **CORBA::Object**. Type IDs may only be “Null” in any message, requiring the client to use existing knowledge or to consult the object, to determine interface types supported. The type ID is a Repository ID identifying the interface type, and is provided to allow ORBs to preserve strong typing. 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, if provided by the server, indicates the most derived type that the server wishes to publish, at the time the reference is generated. The object’s actual most derived type may later change to a more derived type. Therefore, the type ID in the IOR can only be interpreted by the client as a hint that the object supports at least the indicated interface. The client can succeed in narrowing the reference to the indicated interface, or to one of its base interfaces, based solely on the type ID in the IOR, but must not fail to narrow the reference without consulting the object via the “_is_a” or “_get_interface” pseudo-operations.

13.6.2.1 *The* **TAG_INTERNET_IOP** *Profile*

The **TAG_INTERNET_IOP** tag identifies profiles that support the Internet Inter-ORB Protocol. The **ProfileBody** of this profile, described in detail in Section 15.7.2, “IIOP IOR Profiles,” on page 15-49, contains a CDR encapsulation of a structure containing addressing and object identification information used by IIOP. Version 1.1 of the **TAG_INTERNET_IOP** profile also includes a **sequence<TaggedComponent>** that can contain additional information supporting optional IIOP features, ORB services such as security, and future protocol extensions.

Protocols other than IIOP (such as ESIOPs and other GIOPs) can share profile information (such as object identity or security information) with IIOP by encoding their additional profile information as components in the **TAG_INTERNET_IOP** profile. All **TAG_INTERNET_IOP** profiles support IIOP, regardless of whether they also support additional protocols. Interoperable ORBs are not required to create or understand any other profile, nor are they required to create or understand any of the components defined for other protocols that might share the **TAG_INTERNET_IOP** profile with IIOP.

13.6.2.2 *The* **TAG_MULTIPLE_COMPONENTS** *Profile*

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 namespace 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.

13.6.2.3 IOR Components

TaggedComponents contained in **TAG_INTERNET_IOP** and **TAG_MULTIPLE_COMPONENTS** profiles are identified by unique numeric tags using a namespace distinct form that is used for profile tags. Component tags are assigned by the OMG.

Specifications of components must include the following information:

- *Component ID:* The compound tag that is obtained from OMG.
- *Structure and encoding:* The syntax of the component data and the encoding rules. If the component value is encoded as a CDR encapsulation, the IDL type that is encapsulated and the GIOP version which is used for encoding the value, if different than GIOP 1.0, must be specified as part of the component definition.
- *Semantics:* How the component data is intended to be used.
- *Protocols:* The protocol for which the component is defined, and whether it is intended that the component be usable by other protocols.
- *At most once:* whether more than one instance of this component can be included in a profile.

Specification of protocols must describe how the components affect the protocol. The following should be specified in any protocol definition for each **TaggedComponent** that the protocol uses:

- *Mandatory presence:* Whether inclusion of the component in profiles supporting the protocol is required (MANDATORY PRESENCE) or not required (OPTIONAL PRESENCE).
- *Droppable:* For optional presence component, whether component, if present, must be retained or may be dropped.

13.6.3 Standard IOR Components

The following are standard IOR components that can be included in **TAG_INTERNET_IOP** and **TAG_MULTIPLE_COMPONENTS** profiles, and may apply to IIOP, other GIOPs, ESIOPs, or other protocols. An ORB must not drop these components from an existing IOR.

```
module IOP {
    const ComponentId    TAG_ORB_TYPE = 0;
    const ComponentId    TAG_CODE_SETS = 1;
    const ComponentId    TAG_POLICIES = 2;
    const ComponentId    TAG_ALTERNATE_IIOP_ADDRESS = 3;

    const ComponentId    TAG_ASSOCIATION_OPTIONS = 13;
```

```
    const ComponentId    TAG_SEC_NAME = 14;
    const ComponentId    TAG_SPKM_1_SEC_MECH = 15;
    const ComponentId    TAG_SPKM_2_SEC_MECH = 16;
    const ComponentId    TAG_KerberosV5_SEC_MECH = 17;
    const ComponentId    TAG_CSI_ECMA_Secret_SEC_MECH = 18;
    const ComponentId    TAG_CSI_ECMA_Hybrid_SEC_MECH = 19;
    const ComponentId    TAG_SSL_SEC_TRANS = 20;
    const ComponentId    TAG_CSI_ECMA_Public_SEC_MECH = 21;
    const ComponentId    TAG_ GENERIC_SEC_MECH = 22;
    const ComponentId    TAG_JAVA_CODEBASE = 25;
};
```

The following additional components that can be used by other protocols are specified in the DCE ESIOP chapter of this document and *CORBAServices*, Security Service, in the Security Service for DCE ESIOP section:

```
    const ComponentId    TAG_COMPLETE_OBJECT_KEY = 5;
    const ComponentId    TAG_ENDPOINT_ID_POSITION = 6;
    const ComponentId    TAG_LOCATION_POLICY = 12;
    const ComponentId    TAG_DCE_STRING_BINDING = 100;
    const ComponentId    TAG_DCE_BINDING_NAME = 101;
    const ComponentId    TAG_DCE_NO_PIPES = 102;
    const ComponentId    TAG_DCE_SEC_MECH = 103; // Security Service
```

13.6.3.1 **TAG_ORB_TYPE** *Component*

It is often useful in the real world to be able to identify the particular kind of ORB an object reference is coming from, to work around problems with that particular ORB, or exploit shared efficiencies.

The **TAG_ORB_TYPE** component has an associated value of type **unsigned long**, encoded as a CDR encapsulation, designating an ORB type ID allocated by the OMG for the ORB type of the originating ORB. Anyone may register any ORB types by submitting a short (one-paragraph) description of the ORB type to the OMG, and will receive a new ORB type ID in return. A list of ORB type descriptions and values will be made available on the OMG web server.

The **TAG_ORB_TYPE** component can appear at most once in any IOR profile. For profiles supporting IIOP 1.1 or greater, it is optionally present and may not be dropped.

13.6.3.2 **TAG_ALTERNATE_IIOP_ADDRESS** *Component*

In cases where the same object key is used for more than one internet location, the following standard IOR Component is defined for support in IIOP version 1.2.

The **TAG_ALTERNATE_IIOP_ADDRESS** component has an associated value of type

```
struct {
    string HostID,
    short Port
};
```

encoded as a CDR encapsulation.

Zero or more instances of the **TAG_ALTERNATE_IIOP_ADDRESS** component type may be included in a version 1.2 **TAG_INTERNET_IOP** Profile. Each of these alternative addresses may be used by the client orb, in addition to the host and port address expressed in the body of the Profile. In cases where one or more **TAG_ALTERNATE_IIOP_ADDRESS** components are present in a **TAG_INTERNET_IOP** Profile, no order of use is prescribed by Version 1.2 of IIOP.

13.6.3.3 Other Components

The following standard components are specified in various OMG specifications:

- **TAG_CODE_SETS** (See Section 13.7.2.4, “CodeSet Component of IOR Multi-Component Profile,” on page 13-33.)
- **TAG_POLICIES** (See CORBA Messaging specification - currently orbos/98-05-05, will be incorporated into CORBA 3.0).
- **TAG_SEC_NAME** (See Section 15.10.2 Mechanism Tags, Security chapter - CORBAServices).
- **TAG_ASSOCIATION_OPTIONS** (See Section 15.10.3 Tag Association Options, Security chapter - CORBAServices).
- **TAG_SSL_SEC_TRANS** (See Section 15.10.2 Mechanism Tags, Security chapter - CORBAServices).
- **TAG_GENERIC_SEC_MECH** and all other tags with names in the form **TAG_*_SEC_MECH** (See Section 15.10.2 Mechanism Tags, Security chapter - CORBAServices).
- **TAG_JAVA_CODEBASE** (See the *Java to IDL Language Mapping*, Section 1.4.9.3, “Codebase Transmission,” on page 1-33).
- **TAG_COMPLETE_OBJECT_KEY** (See Section 16.5.4, “Complete Object Key Component,” on page 16-19).
- **TAG_ENDPOINT_ID_POSITION** (See Section 16.5.5, “Endpoint ID Position Component,” on page 16-20).
- **TAG_LOCATION_POLICY** (See Section 16.5.6, “Location Policy Component,” on page 16-20).
- **TAG_DCE_STRING_BINDING** (See Section 16.5.1, “DCE-CIOP String Binding Component,” on page 16-17).
- **TAG_DCE_BINDING_NAME** (See Section 16.5.2, “DCE-CIOP Binding Name Component,” on page 16-18).
- **TAG_DCE_NO_PIPES** (See Section 16.5.3, “DCE-CIOP No Pipes Component,” on page 16-19).

13.6.4 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.

13.6.5 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.

13.6.6 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:

```
(99)          <oref>  ::= <prefix> <hex_Octets>
(100)         <prefix>  ::= “IOR:”
(101)      <hex_Octets>  ::= <hex_Octet> {<hex_Octet>}*
(102)      <hex_Octet>  ::= <hexDigit> <hexDigit>
(103)      <hexDigit>  ::= <digit> | <a> | <b> | <c> | <d> | <e> | <f>
(104)         <digit>  ::= “0” | “1” | “2” | “3” | “4” | “5” |
                       |   “6” | “7” | “8” | “9”
(105)          <a>  ::= “a” | “A”
(106)          <b>  ::= “b” | “B”
(107)          <c>  ::= “c” | “C”
(108)          <d>  ::= “d” | “D”
(109)          <e>  ::= “e” | “E”
(110)          <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, as specified in GIOP 1.0. (See Section 15.3, “CDR Transfer Syntax,” on page 15-5 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.

13.6.7 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;
    const ServiceId         CodeSets = 1;
    const ServiceId         ChainBypassCheck = 2;
    const ServiceId         ChainBypassInfo = 3;
    const ServiceId         LogicalThreadId = 4;
    const ServiceId         BI_DIR_IIOP = 5;
    const ServiceId         SendingContextRunTime = 6;
    const ServiceId         INVOCATION_POLICIES = 7;
    const ServiceId         FORWARDED_IDENTITY = 8;
    const ServiceId         UnknownExceptionInfo = 9;
```

};

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 15.3.3, “Encapsulation,” on page 15-13). 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).

Each Object Service requiring implicit service-specific context to be passed through GIOP will be allocated a unique service context ID value by the 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 high-order 20 bits of service-context ID contain a 20-bit vendor service context codeset ID (VSCID); the low-order 12 bits contain the rest of the service context ID. A vendor (or group of vendors) who wish to define a specific set of system exception minor codes should obtain a unique VSCID from the OMG, and then define a specific set of service context IDs using the VSCID for the high-order bits.

The VSCID of zero is reserved for use for OMG-defined standard service context IDs (i.e., service context IDs in the range 0-4095 are reserved as OMG standard service contexts).

The **ServiceId**s currently defined are:

- **TransactionService** identifies a CDR encapsulation of the **CosTSInteroperation::PropogationContext** defined in *CORBAservices: Common Object Services Specifications*.
- **CodeSets** identifies a CDR encapsulation of the **CONV_FRAME::CodeSetContext** defined in Section 13.7.2.5, “GIOP Code Set Service Context,” on page 13-34.
- DCOM-CORBA Interworking uses three service contexts as defined in "DCOM-CORBA Interworking" in the “Interoperability with non-CORBA Systems”chapter. They are:
  - **ChainBypassCheck**, which carries a CDR encapsulation of the **struct CosBridging::ChainBypassCheck**. This is carried only in a **Request** message as described in Section 20.9.1, “CORBA Chain Bypass,” on page 20-19.
  - **ChainBypassInfo**, which carries a CDR encapsulation of the **struct CosBridging::ChainBypassInfo**. This is carried only in a **Reply** message as described in Section 20.9.1, “CORBA Chain Bypass,” on page 20-19.

  - **LogicalThreadId**, which carries a CDR encapsulation of the **struct CosBridging::LogicalThreadId** as described in Section 20.10, “Thread Identification,” on page 20-21.

- **BI_DIR_IIOP** identifies a CDR encapsulation of the **IIOP::BiDirIIOPServiceContext** defined in Section 15.8, “Bi-Directional GIOP,” on page 15-52.
- **SendingContextRunTime** identifies a CDR encapsulation of the IOR of the **SendingContext::RunTime** object (see Section 5.6, “Access to the Sending Context Run Time,” on page 5-15).
- **UnknownExceptionInfo** identifies a CDR encapsulation of a marshaled instance of a **java.lang.throwable** or one of its subclasses as described in *Java to IDL Language Mapping*, Section 1.4.8.1, “Mapping of UnknownExceptionInfo Service Context,” on page 1-32.
- The **profile_data** for the **TAG_INTERNET_IOP** profile is a CDR encapsulation of the **IIOP::ProfileBody_1_1** type, described in Section 15.7.2, “IIOP IOR Profiles,” on page 15-49.
- The **profile_data** for the **TAG_MULTIPLE_COMPONENTS** profile is a CDR encapsulation of the **MultipleComponentProfile** type, which is a sequence of **TaggedComponent** structures, described in Section 13.6, “An Information Model for Object References,” on page 13-15.
- The **component_data member** identifies a CDR encapsulation of a **BindingNameComponent** structure, described in Section 16.5.2.1, “BindingNameComponent,” on page 16-18.

Note – For more information on INVOCATION_POLICIES refer to the Asynchronous Messaging specification - orbos/98-05-05. For information on FORWARDED_IDENTITY refer to the Firewall specification - orbos/98-05-04.

Service context IDs are associated with a specific version of GIOP, but will always be allocated in the OMG service context range. This allows any ORB to recognize when it is receiving a standard service context, even if it has been defined in a version of GIOP that it does not support.

The following are the rules for processing a received service context:

- The service context is in the OMG defined range:
  - If it is valid for the supported GIOP version, then it must be processed correctly according to the rules associated with it for that GIOP version level.
  - If it is not valid for the GIOP version, then it may be ignored by the receiving ORB, however it must be passed on through a bridge. No exception shall be raised.
- The service context is not in the OMG-defined range:

- The receiving ORB may choose to ignore it, process it if it "understands" it, or raise a system exception, however it must be passed on through a bridge. If a system exception is raised, it shall be BAD_PARAM with an OMG standard minor code of 1.

The association of service contexts with GIOP versions, (along with some other supported features tied to GIOP minor version), is shown in Table 13-1.

Table 13-1 Feature Support Tied to Minor GIOP Version Number

Feature	**Version 1.0**	**Version 1.1**	**Version 1.2**
Transaction Service Context	yes	yes	yes
Codeset Negotiation Service Context		yes	yes
DCOM Bridging Service Contexts: **ChainBypassCheck** **ChainBypassInfo** **LogicalThreadId**			yes
Object by Value Service Context: **SendingContextRunTime**			yes
Bi-Directional IIOP Service Context: **BI_DIR_IIOP**			yes
Java Language Throwable Service Context: **UnknownExceptionInfo**			yes
IOR components in IIOP profile		yes	yes
TAG_ORB_TYPE		yes	yes
TAG_CODE_SETS		yes	yes
TAG_ALTERNATE_IIOP_ADDRESS			yes
TAG_ASSOCIATION_OPTION		yes	yes
TAG_SEC_NAME		yes	yes
TAG_SSL_SEC_TRANS		yes	yes
TAG_GENERIC_SEC_MECH		yes	yes
TAG_*_SEC_MECH		yes	yes
TAG_JAVA_CODEBASE			yes
IOR component nn			yes
Extended IDL data types		yes	yes
Bi-Directional GIOP Features			yes

13.7 Code Set Conversion

13.7.1 Character Processing Terminology

This section introduces a few terms and explains a few concepts to help understand the character processing portions of this document.

13.7.1.1 Character Set

A finite set of different characters used for the representation, organization, or control of data. In this specification, the term “character set” is used without any relationship to code representation or associated encoding. Examples of character sets are the English alphabet, Kanji or sets of ideographic characters, corporate character sets (commonly used in Japan), and the characters needed to write certain European languages.

13.7.1.2 Coded Character Set, or Code Set

A set of unambiguous rules that establishes a character set and the one-to-one relationship between each character of the set and its bit representation or numeric value. In this specification, the term “code set” is used as an abbreviation for the term “coded character set.” Examples include ASCII, ISO 8859-1, JIS X0208 (which includes Roman characters, Japanese hiragana, Greek characters, Japanese kanji, etc.) and Unicode.

13.7.1.3 Code Set Classifications

Some language environments distinguish between byte-oriented and “wide characters.” The byte-oriented characters are encoded in one or more 8-bit bytes. A typical single-byte encoding is ASCII as used for western European languages like English. A typical multi-byte encoding which uses from one to three 8-bit bytes for each character is eucJP (Extended UNIX Code - Japan, packed format) as used for Japanese workstations.

Wide characters are a fixed 16 or 32 bits long, and are used for languages like Chinese, Japanese, etc., where the number of combinations offered by 8 bits is insufficient and a fixed-width encoding is needed. A typical example is Unicode (a “universal” character set defined by the The Unicode Consortium, which uses an encoding scheme identical to ISO 10646 UCS-2, or 2-byte Universal Character Set encoding). An extended encoding scheme for Unicode characters is UTF-16 (UCS Transformation Format, 16-bit representations).

The C language has data types **`char`** for byte-oriented characters and **`wchar_t`** for wide characters. The language definition for C states that the sizes for these characters are implementation-dependent. Some environments do not distinguish between byte-

oriented and wide characters (e.g., Ada and Smalltalk). Here again, the size of a character is implementation-dependent. The following table illustrates code set classifications as used in this document.

Table 13-2 Code Set Classification

Orientation	Code Element Encoding	Code Set Examples	C Data Type
byte-oriented	single-byte	ASCII, ISO 8859-1 (Latin-1), EBCDIC, ...	`char`
	multi-byte	UTF-8, eucJP, Shift-JIS, JIS, Big5, ...	`char[]`
non-byte-oriented	fixed-length	ISO 10646 UCS-2 (Unicode), ISO 10646 UCS-4, UTF-16, ...	`wchar_t`

13.7.1.4 Narrow and Wide Characters

Some language environments distinguish between “narrow” and “wide” characters. Typically the narrow characters are considered to be 8-bit long and are used for western European languages like English, while the wide characters are 16-bit or 32-bit long and are used for languages like Chinese, Japanese, etc., where the number of combinations offered by 8 bits are insufficient. However, as noted above there are common encoding schemes in which Asian characters are encoded using multi-byte code sets and it is incorrect to assume that Asian characters are always encoded as “wide” characters.

Within this specification, the general terms “narrow character” and “wide character” are only used in discussing OMG IDL.

13.7.1.5 Char Data and Wchar Data

The phrase “**char** data” in this specification refers to data whose IDL types have been specified as **char** or **string**. Likewise “**wchar** data” refers to data whose IDL types have been specified as **wchar** or **wstring**.

13.7.1.6 Byte-Oriented Code Set

An encoding of characters where the numeric code corresponding to a character code element can occupy one or more bytes. A byte as used in this specification is synonymous with octet, which occupies 8 bits.

13.7.1.7 Multi-Byte Character Strings

A character string represented in a byte-oriented encoding where each character can occupy one or more bytes is called a multi-byte character string. Typically, wide characters are converted to this form from a (fixed-width) process code set before

transmitting the characters outside the process (see below about process code sets). Care must be taken to correctly process the component bytes of a character's multi-byte representation.

13.7.1.8 Non-Byte-Oriented Code Set

An encoding of characters where the numeric code corresponding to a character code element can occupy fixed 16 or 32 bits.

13.7.1.9 Char Transmission Code Set (TCS-C) and Wchar Transmission Code Set (TCS-W)

These two terms refer to code sets that are used for transmission between ORBs after negotiation is completed. As the names imply, the first one is used for **char** data and the second one for **wchar** data. Each TCS can be byte-oriented or non-byte oriented.

13.7.1.10 Process Code Set and File Code Set

Processes generally represent international characters in an internal fixed-width format which allows for efficient representation and manipulation. This internal format is called a "process code set." The process code set is irrelevant outside the process, and hence to the interoperation between CORBA clients and servers through their respective ORBs.

When a process needs to write international character information out to a file, or communicate with another process (possibly over a network), it typically uses a different encoding called a "file code set." In this specification, unless otherwise indicated, all references to a program's code set refer to the file code set, not the process code set. Even when a client and server are located physically on the same machine, it is possible for them to use different file code sets.

13.7.1.11 Native Code Set

A native code set is the code set which a client or a server uses to communicate with its ORB. There might be separate native code sets for **char** and **wchar** data.

13.7.1.12 Transmission Code Set

A transmission code set is the commonly agreed upon encoding used for character data transfer between a client's ORB and a server's ORB. There are two transmission code sets established per session between a client and its server, one for **char** data (TCS-C) and the other for **wchar** data (TCS-W). Figure 13-6 illustrates these relationships:

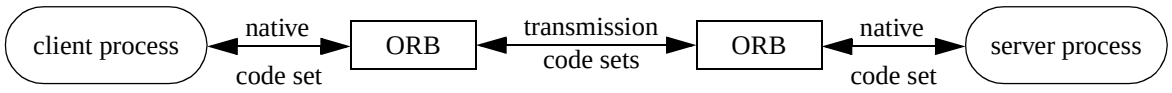


Figure 13-6 Transmission Code Sets

The intent is for TCS-C to be byte-oriented and TCS-W to be non-byte-oriented. However, this specification does allow both types of characters to be transmitted using the same transmission code set. That is, the selection of a transmission code set is orthogonal to the wideness or narrowness of the characters, although a given code set may be better suited for either narrow or wide characters.

13.7.1.13 Conversion Code Set (CCS)

With respect to a particular ORB's native code set, the set of other or target code sets for which an ORB can convert all code points or character encodings between the native code set and that target code set. For each code set in this CCS, the ORB maintains appropriate translation or conversion procedures and advertises the ability to use that code set for transmitted data in addition to the native code set.

13.7.2 Code Set Conversion Framework

13.7.2.1 Requirements

The file code set that an application uses is often determined by the platform on which it runs. In Japan, for example, Japanese EUC is used on Unix systems, while Shift-JIS is used on PCs. Code set conversion is therefore required to enable interoperability across these platforms. This proposal defines a framework for the automatic conversion of code sets in such situations. The requirements of this framework are:

1. Backward compatibility. In previous CORBA specifications, IDL type **char** was limited to ISO 8859-1. The conversion framework should be compatible with existing clients and servers that use ISO 8859-1 as the code set for **char**.

2. Automatic code set conversion. To facilitate development of CORBA clients and servers, the ORB should perform any necessary code set conversions automatically and efficiently. The IDL type **octet** can be used if necessary to prevent conversions.

3. Locale support. An internationalized application determines the code set in use by examining the LOCALE string (usually found in the LANG environment variable), which may be changed dynamically at run time by the user. Example LOCALE

strings are fr_FR.ISO8859-1 (French, used in France with the ISO 8859-1 code set) and ja_JP.ujis (Japanese, used in Japan with the EUC code set and X11R5 conventions for LOCALE). The conversion framework should allow applications to use the LOCALE mechanism to indicate supported code sets, and thus select the correct code set from the registry.

4. CMIR and SMIR support. The conversion framework should be flexible enough to allow conversion to be performed either on the client or server side. For example, if a client is running in a memory-constrained environment, then it is desirable for code set converters to reside in the server and for a Server Makes It Right (SMIR) conversion method to be used. On the other hand, if many servers are executed on one server machine, then converters should be placed in each client to reduce the load on the server machine. In this case, the conversion method used is Client Makes It Right (CMIR).

13.7.2.2 Overview of the Conversion Framework

Both the client and server indicate a native code set indirectly by specifying a locale. The exact method for doing this is language-specific, such as the XPG4 C/C++ function **`setlocale`**. The client and server use their native code set to communicate with their ORB. (Note that these native code sets are in general different from process code sets and hence conversions may be required at the client and server ends.)

The conversion framework is illustrated in Figure 13-7. The server-side ORB stores a server's code set information in a component of the IOR multiple-component profile structure (see Section 13.6.2, "Interoperable Object References: IORs," on page 13-15)[2]. The code sets actually used for transmission are carried in the service context field of an IOP (Inter-ORB Protocol) request header (see Section 13.6.7, "Object Service Context," on page 13-22 and Section 13.7.2.5, "GIOP Code Set Service Context," on page 13-34). Recall that there are two code sets (TCS-C and TCS-W) negotiated for each session.

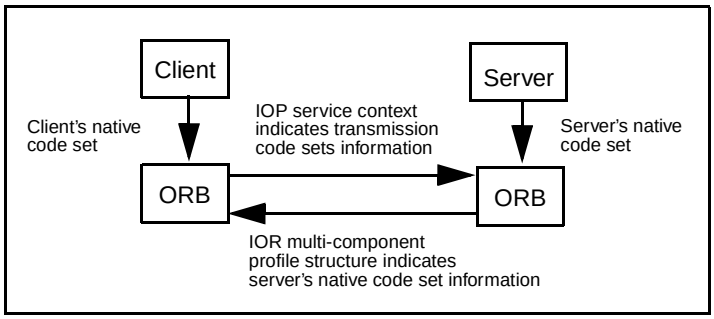


Figure 13-7 Code Set Conversion Framework Overview

2. Version 1.1 of the IIOP profile body can also be used to specify the server's code set information, as this version introduces an extra field that is a sequence of tagged components.

If the native code sets used by a client and server are the same, then no conversion is performed. If the native code sets are different and the client-side ORB has an appropriate converter, then the CMIR conversion method is used. In this case, the server's native code set is used as the transmission code set. If the native code sets are different and the client-side ORB does not have an appropriate converter but the server-side ORB does have one, then the SMIR conversion method is used. In this case, the client's native code set is used as the transmission code set.

The conversion framework allows clients and servers to specify a native **char** code set and a native **wchar** code set, which determine the local encodings of IDL types **char** and **wchar**, respectively. The conversion process outlined above is executed independently for the **char** code set and the **wchar** code set. In other words, the algorithm that is used to select a transmission code set is run twice, once for **char** data and once for **wchar** data.

The rationale for selecting two transmission code sets rather than one (which is typically inferred from the locale of a process) is to allow efficient data transmission without any conversions when the client and server have identical representations for **char** and/or **wchar** data. For example, when a Windows NT client talks to a Windows NT server and they both use Unicode for wide character data, it becomes possible to transmit wide character data from one to the other without any conversions. Of course, this becomes possible only for those wide character representations that are well-defined, not for any proprietary ones. If a single transmission code set was mandated, it might require unnecessary conversions. (For example, choosing Unicode as the transmission code set would force conversion of all byte-oriented character data to Unicode.)

13.7.2.3 ORB Databases and Code Set Converters

The conversion framework requires an ORB to be able to determine the native code set for a locale and to convert between code sets as necessary. While the details of exactly how these tasks are accomplished are implementation-dependent, the following databases and code set converters might be used:

- Locale database. This database defines a native code set for a process. This code set could be byte-oriented or non-byte-oriented and could be changed programmatically while the process is running. However, for a given session between a client and a server, it is fixed once the code set information is negotiated at the session's setup time.
- Environment variables or configuration files. Since the locale database can only indicate one code set while the ORB needs to know two code sets, one for **char** data and one for **wchar** data, an implementation can use environment variables or configuration files to contain this information on native code sets.
- Converter database. This database defines, for each code set, the code sets to which it can be converted. From this database, a set of "conversion code sets" (CCS) can be determined for a client and server. For example, if a server's native code set is eucJP, and if the server-side ORB has eucJP-to-JIS and eucJP-to-SJIS bilateral converters, then the server's conversion code sets are JIS and SJIS.

- Code set converters. The ORB has converters which are registered in the converter database.

13.7.2.4 *CodeSet Component of IOR Multi-Component Profile*

The code set component of the IOR multi-component profile structure contains:

- server's native **char** code set and conversion code sets, and
- server's native **wchar** code set and conversion code sets.

Both **char** and **wchar** conversion code sets are listed in order of preference. The code set component is identified by the following tag:

```
const IOP::ComponentID TAG_CODE_SETS = 1;
```

This tag has been assigned by OMG (See "Standard IOR Components" on page 13-18.). The following IDL structure defines the representation of code set information within the component:

```
module CONV_FRAME {                          // IDL
        typedef unsigned long CodeSetId;
        struct CodeSetComponent {
            CodeSetId                native_code_set;
            sequence<CodeSetId>      conversion_code_sets;
        };
        struct CodeSetComponentInfo {

            CodeSetComponent         ForCharData;
            CodeSetComponent         ForWcharData;
        };
};
```

Code sets are identified by a 32-bit integer id from the OSF Character and Code Set Registry (See "Character and Code Set Registry" on page 13-40 for further information). Data within the code set component is represented as a structure of type **CodeSetComponentInfo**, and is encoded as a CDR encapsulation. In other words, the **char** code set information comes first, then the **wchar** information, represented as structures of type **CodeSetComponent**.

A null value should be used in the **native_code_set** field if the server desires to indicate no native code set (possibly with the identification of suitable conversion code sets).

If the code set component is not present in a multi-component profile structure, then the default **char** code set is ISO 8859-1 for backward compatibility. However, there is no default **wchar** code set. If a server supports interfaces that use wide character data but does not specify the **wchar** code sets that it supports, client-side ORBs will raise exception INV_OBJREF.

13.7.2.5 GIOP Code Set Service Context

The code set GIOP service context contains:

- **char** transmission code set, and
- **wchar** transmission code set

in the form of a code set service. This service is identified by:

```
const IOP::ServiceID  CodeSets = 1;
```

The following IDL structure defines the representation of code set service information:

```
module CONV_FRAME {                                  // IDL
        typedef unsigned long CodeSetId;
        struct CodeSetContext {
            CodeSetId               char_data;
            CodeSetId               wchar_data;
        };
};
```

Code sets are identified by a 32-bit integer id from the OSF Character and Code Set Registry (See “Character and Code Set Registry” on page 13-40 for further information).

Note – A server’s **char** and **wchar** Code set components are usually different, but under some special circumstances they can be the same. That is, one could use the same code set for both **char** data and **wchar** data. Likewise the **CodesetId**s in the service context don’t have to be different.

13.7.2.6 Code Set Negotiation

The client-side ORB determines a server’s native and conversion code sets from the code set component in an IOR multi-component profile structure, and it determines a client’s native and conversion code sets from the locale setting (and/or environment variables/configuration files) and the converters that are available on the client. From this information, the client-side ORB chooses **char** and **wchar** transmission code sets (TCS-C and TCS-W). For both requests and replies, the **char** TCS-C determines the encoding of **char** and **string** data, and the **wchar** TCS-W determines the encoding of **wchar** and **wstring** data.

Code set negotiation is not performed on a per-request basis, but only when a client initially connects to a server. All text data communicated on a connection are encoded as defined by the TCSs selected when the connection is established.

Figure 13-8 illustrates, there are two channels for character data flowing between the client and the server. The first, TCS-C, is used for **char** data and the second, TCS-W, is used for **wchar** data. Also note that two native code sets, one for each type of data,

could be used by the client and server to talk to their respective ORBs (as noted earlier, the selection of the particular native code set used at any particular point is done via **`setlocale`** or some other implementation-dependent method).

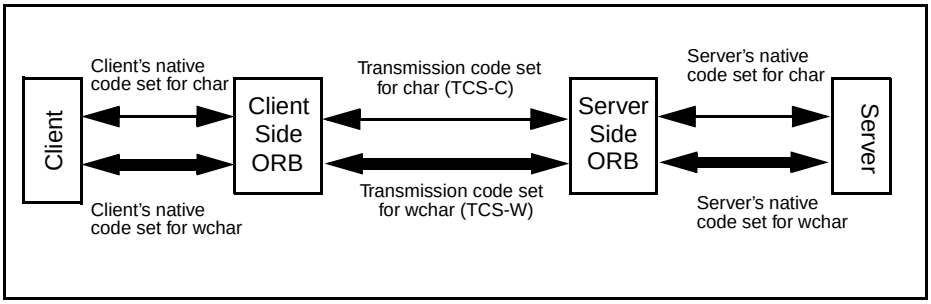


Figure 13-8 Transmission Code Set Use

Let us look at an example. Assume that the code set information for a client and server is as shown in the table below. (Note that this example concerns only **char** code sets and is applicable only for data described as **char**s in the IDL.)

	Client	**Server**
Native code set:	SJIS	eucJP
Conversion code sets:	eucJP, JIS	SJIS, JIS

The client-side ORB first compares the native code sets of the client and server. If they are identical, then the transmission and native code sets are the same and no conversion is required. In this example, they are different, so code set conversion is necessary. Next, the client-side ORB checks to see if the server's native code set, eucJP, is one of the conversion code sets supported by the client. It is, so eucJP is selected as the transmission code set, with the client (i.e., its ORB) performing conversion to and from its native code set, SJIS, to eucJP. Note that the client may first have to convert all its data described as **char**s (and possibly **wchar_t**s) from process codes to SJIS first.

Now let us look at the general algorithm for determining a transmission code set and where conversions are performed. First, we introduce the following abbreviations:

- CNCS - Client Native Code Set;
- CCCS - Client Conversion Code Sets;
- SNCS - Server Native Code Set;
- SCCS - Server Conversion Code Sets; and
- TCS - Transmission Code Set.

The algorithm is as follows:

```
if (CNCS==SNCS)
        TCS  =  CNCS;                   // no conversion required
else {
        if (elementOf(SNCS,CCCS))
```

```
        TCS  =  SNCS;   // client converts to server's native code set
    else if (elementOf(CNCS,SCCS))
        TCS  =  CNCS;   // server converts from client's native code set
    else if (intersection(CCCS,SCCS)  !=  emptySet) {
        TCS  =  oneOf(intersection(CCCS,SCCS));
        // client chooses TCS, from intersection(CCCS,SCCS), that is
        // most preferable to server;
        // client converts from CNCS to TCS and server
        // from TCS to SNCS
    else if (compatible(CNCS,SNCS))
        TCS = fallbackCS;       // fallbacks are UTF-8 (for char data) and
                                // UTF-16 (for wchar data)
    else
        raise CODESET_INCOMPATIBLE exception;
}
```

The algorithm first checks to see if the client and server native code sets are the same. If they are, then the native code set is used for transmission and no conversion is required. If the native code sets are not the same, then the conversion code sets are examined to see if

1. the client can convert from its native code set to the server's native code set,
2. the server can convert from the client's native code set to its native code set, or
3. transmission through an intermediate conversion code set is possible.

If the third option is selected and there is more than one possible intermediate conversion code set (i.e., the intersection of CCCS and SCCS contains more than one code set), then the one most preferable to the server is selected.[3]

If none of these conversions is possible, then the fallback code set (UTF-8 for **char** data and UTF-16 for **wchar** data— see below) is used. However, before selecting the fallback code set, a compatibility test is performed. This test looks at the character sets encoded by the client and server native code sets. If they are different (e.g., Korean and French), then meaningful communication between the client and server is not possible and a CODESET_INCOMPATIBLE exception is raised. This test is similar to the DCE compatibility test and is intended to catch those cases where conversion from the client native code set to the fallback, and the fallback to the server native code set would result in massive data loss. (See Section 13.9, "Relevant OSFM Registry Interfaces," on page 13-40 for the relevant OSF registry interfaces that could be used for determining compatibility.)

3.Recall that server conversion code sets are listed in order of preference.

A DATA_CONVERSION exception is raised when a client or server attempts to transmit a character that does not map into the negotiated transmission code set. For example, not all characters in Taiwan Chinese map into Unicode. When an attempt is made to transmit one of these characters via Unicode, an ORB is required to raise a DATA_CONVERSION exception.

In summary, the fallback code set is UTF-8 for **char** data (identified in the Registry as 0x05010001, “X/Open UTF-8; UCS Transformation Format 8 (UTF-8)"), and UTF-16 for **wchar** data (identified in the Registry as 0x00010109, "ISO/IEC 10646-1:1993; UTF-16, UCS Transformation Format 16-bit form"). As mentioned above the fallback code set is meaningful only when the client and server character sets are compatible, and the fallback code set is distinguished from a default code set used for backward compatibility.

If a server’s native **char** code set is not specified in the IOR multi-component profile, then it is considered to be ISO 8859-1 for backward compatibility. However, a server that supports interfaces that use wide character data is required to specify its native **wchar** code set; if one is not specified, then the client-side ORB raises exception INV_OBJREF.

Similarly, if no **char** transmission code set is specified in the code set service context, then the **char** transmission code set is considered to be ISO 8859-1 for backward compatibility. If a client transmits wide character data and does not specify its **wchar** transmission code set in the service context, then the server-side ORB raises exception BAD_PARAM.

To guarantee “out-of-the-box” interoperability, clients and servers must be able to convert between their native **char** code set and UTF-8 and their native **wchar** code set (if specified) and Unicode. Note that this does not require that all server native code sets be mappable to Unicode, but only those that are exported as native in the IOR. The server may have other native code sets that aren’t mappable to Unicode and those can be exported as SCCSs (but not SNCSs). This is done to guarantee out-of-the-box interoperability and to reduce the number of code set converters that a CORBA-compliant ORB must provide.

ORB implementations are strongly encouraged to use widely-used code sets for each regional market. For example, in the Japanese marketplace, all ORB implementations should support Japanese EUC, JIS and Shift JIS to be compatible with existing business practices.

13.7.3 Mapping to Generic Character Environments

Certain language environments do not distinguish between byte-oriented and wide characters. In such environments both **char** and **wchar** are mapped to the same “generic” character representation of the language. **String** and **wstring** are likewise mapped to generic strings in such environments. Examples of language environments that provide generic character support are Smalltalk and Ada.

Even while using languages that do distinguish between wide and byte-oriented characters (e.g., C and C++), it is possible to mimic some generic behavior by the use of suitable macros and support libraries. For example, developers of Windows NT and

Windows 95 applications can write portable code between NT (which uses Unicode strings) and Windows 95 (which uses byte-oriented character strings) by using a set of macros for declaring and manipulating characters and character strings. Appendix A in this chapter shows how to map wide and byte-oriented characters to these generic macros.

Another way to achieve generic manipulation of characters and strings is by treating them as abstract data types (ADTs). For example, if strings were treated as abstract data types and the programmers are required to create, destroy, and manipulate strings only through the operations in the ADT interface, then it becomes possible to write code that is representation-independent. This approach has an advantage over the macro-based approach in that it provides portability between byte-oriented and wide character environments even without recompilation (at runtime the string function calls are bound to the appropriate byte-oriented/wide library). Another way of looking at it is that the macro-based genericity gives compile-time flexibility, while ADT-based genericity gives runtime flexibility.

Yet another way to achieve generic manipulation of character data is through the ANSI C++ Strings library defined as a template that can be parameterized by **`char`**, **`wchar_t`**, or other integer types.

Given that there can be several ways of treating characters and character strings in a generic way, this standard cannot, and therefore does not, specify the mapping of **char**, **wchar, string**, and **wstring** to all of them. It only specifies the following normative requirements which are applicable to generic character environments:

- **wchar** must be mapped to the generic character type in a generic character environment.
- **wstring** must be mapped to a string of such generic characters in a generic character environment.
- The language binding files (i.e., stubs) generated for these generic environments must ensure that the generic type representation is converted to the appropriate code sets (i.e., CNCS on the client side and SNCS on the server side) before character data is given to the ORB runtime for transmission.

13.7.3.1 Describing Generic Interfaces

To describe generic interfaces in IDL we recommend using **wchar** and **wstring**. These can be mapped to generic character types in environments where they do exist and to wide characters where they do not. Either way interoperation between generic and non-generic character type environments is achieved because of the code set conversion framework.

13.7.3.2 Interoperation

Let us consider an example to see how a generic environment can interoperate with a non-generic environment. Let us say there is an IDL interface with both **char** and **wchar** parameters on the operations, and let us say the client of the interface is in a generic environment while the server is in a non-generic environment (for example the client is written in Smalltalk and the server is written in C++).

Assume that the server's (byte-oriented) native **char** code set (SNCS) is eucJP and the client's native **char** code set (CNCS) is SJIS. Further assume that the code set negotiation led to the decision to use eucJP as the **char** TCS-C and Unicode as the **wchar** TCS-W.

As per the above normative requirements for mapping to a generic environment, the client's Smalltalk stubs are responsible for converting all **char** data (however they are represented inside Smalltalk) to SJIS and all **wchar** data to the client's **wchar** code set before passing the data to the client-side ORB. Note that this conversion could be an identity mapping if the internal representation of narrow and wide characters is the same as that of the native code set(s). The client-side ORB now converts all **char** data from SJIS to eucJP and all **wchar** data from the client's **wchar** code set to Unicode, and then transmits to the server side.

The server side ORB and stubs convert the eucJP data and Unicode data into C++'s internal representation for **char**s and **wchar**s as dictated by the IDL operation signatures. Notice that when the data arrives at the server side it does not look any different from data arriving from a non-generic environment (e.g., that is just like the server itself). In other words, the mappings to generic character environments do not affect the code set conversion framework.

13.8 Example of Generic Environment Mapping

This Appendix shows how **char**, **wchar**, **string**, and **wchar** can be mapped to the generic C/C++ macros of the Windows environment. This is merely to illustrate one possibility. This section is not normative and is applicable only in generic environments. See Section 13.7.3, "Mapping to Generic Character Environments," on page 13-37.

13.8.1 Generic Mappings

Char and **string** are mapped to C/C++ **`char`** and **`char*`** as per the standard C/C++ mappings. **wchar** is mapped to the **`TCHAR`** macro which expands to either **`char`** or **`wchar_t`** depending on whether **`_UNICODE`** is defined. **wstring** is mapped to pointers to **`TCHAR`** as well as to the string class **`CORBA::Wstring_var`**. Literal strings in IDL are mapped to the **`_TEXT`** macro as in **`_TEXT(<literal>)`**.

13.8.2 Interoperation and Generic Mappings

We now illustrate how the interoperation works with the above generic mapping. Consider an IDL interface operation with a **wstring** parameter, a client for the operation which is compiled and run on a Windows 95 machine, and a server for the operation which is compiled and run on a Windows NT machine. Assume that the locale (and/or the environment variables for CNCS for **wchar** representation) on the Windows 95 client indicates the client's native code set to be SJIS, and that the corresponding server's native code set is Unicode. The code set negotiation in this case will probably choose Unicode as the TCS-W.

Both the client and server sides will be compiled with **_UNICODE** defined. The IDL type **wstring** will be represented as a string of **wchar_t** on the client. However, since the client's locale or environment indicates that the CNCS for wide characters is SJIS, the client side ORB will get the **wstring** parameter encoded as a SJIS multi-byte string (since that is the client's native code set), which it will then convert to Unicode before transmitting to the server. On the server side the ORB has no conversions to do since the TCS-W matches the server's native code set for wide characters.

We therefore notice that the code set conversion framework handles the necessary translations between byte-oriented and wide forms.

13.9 Relevant OSFM Registry Interfaces

13.9.1 Character and Code Set Registry

The OSF character and code set registry is defined in *OSF Character and Code Set Registry* (see References in the Preface) and current registry contents may be obtained directly from the Open Software Foundation (obtain via anonymous ftp to ftp.opengroup.org:/pub/code_set_registry). This registry contains two parts: character sets and code sets. For each listed code set, the set of character sets encoded by this code set is shown.

Each 32-bit code set value consists of a high-order 16-bit organization number and a 16-bit identification of the code set within that organization. As the numbering of organizations starts with 0x0001, a code set null value (0x00000000) may be used to indicate an unknown code set.

When associating character sets and code sets, OSF uses the concept of "fuzzy equality," meaning that a code set is shown as encoding a particular character set if the code set can encode "most" of the characters.

"Compatibility" is determined with respect to two code sets by examining their entries in the registry, paying special attention to the character sets encoded by each code set. For each of the two code sets, an attempt is made to see if there is at least one (fuzzy-defined) character set in common, and if such a character set is found, then the assumption is made that these code sets are "compatible." Obviously, applications which exploit parts of a character set not properly encoded in this scheme will suffer information loss when communicating with another application in this "fuzzy" scheme.

The ORB is responsible for accessing the OSF registry and determining "compatibility" based on the information returned.

OSF members and other organizations can request additions to both the character set and code set registries by email to cs-registry@opengroup.org; in particular, one range of the code set registry (**0xf5000000** through **0xffffffff**) is reserved for organizations to use in identifying sets which are not registered with the OSF (although such use would not facilitate interoperability without registration).

13.9.2 Access Routines

The following routines are for accessing the OSF character and code set registry. These routines map a code set string name to code set id and vice versa. They also help in determining character set compatibility. These routine interfaces, their semantics and their actual implementation are not normative (i.e., ORB vendors do not have to bundle the OSF registry implementation with their products for compliance).

The following routines are adopted from *RPC Runtime Support For I18N Characters - Functional Specification* (see References in the Preface).

13.9.2.1 dce_cs_loc_to_rgy

Maps a local system-specific string name for a code set to a numeric code set value specified in the code set registry.

Synopsis

```
void dce_cs_loc_to_rgy(
   idl_char *local_code_set_name,
   unsigned32 *rgy_code_set_value,
   unsigned16 *rgy_char_sets_number,
   unsigned16 **rgy_char_sets_value,
   error_status_t *status);
```

Parameters

Input

local_code_set_name - A string that specifies the name that the local host's locale environment uses to refer to the code set. The string is a maximum of 32 bytes: 31 data bytes plus a terminating NULL character.

Output

rgy_code_set_value 0 - The registered integer value that uniquely identifies the code set specified by local_code_set_name.

rgy_char_sets_number - The number of character sets that the specified code set encodes. Specifying NULL prevents this routine from returning this parameter.

rgy_char_sets_value - A pointer to an array of registered integer values that uniquely identify the character set(s) that the specified code set encodes. Specifying NULL prevents this routine from returning this parameter. The routine dynamically allocates this value.

status - Returns the status code from this routine. This status code indicates whether the routine completed successfully or, if not, why not.

The possible status codes and their meanings are as follows:

- dce_cs_c_ok – Code set registry access operation succeeded.
- dce_cs_c_cannot_allocate_memory – Cannot allocate memory for code set info.
- dce_cs_c_unknown – No code set value was not found in the registry which corresponds to the code set name specified.
- dce_cs_c_notfound – No local code set name was found in the registry which corresponds to the name specified.

Description

The dce_cs_loc_to_rgy() routine maps operating system-specific names for character/code set encodings to their unique identifiers in the code set registry.

The dce_cs_loc_to_rgy() routine takes as input a string that holds the host-specific “local name” of a code set and returns the corresponding integer value that uniquely identifies that code set, as registered in the host's code set registry. If the integer value does not exist in the registry, the routine returns the status dce_cs_c_unknown.

The routine also returns the number of character sets that the code set encodes and the registered integer values that uniquely identify those character sets. Specifying NULL in the rgy_char_sets_number and rgy_char_sets_value[] parameters prevents the routine from performing the additional search for these values. Applications that want only to obtain a code set value from the code set registry can specify NULL for these parameters in order to improve the routine's performance. If the value is returned from the routine, application developers should free the array after it is used, since the array is dynamically allocated.

13.9.2.2 dce_cs_rgy_to_loc

Maps a numeric code set value contained in the code set registry to the local system-specific name for a code set.

Synopsis

```
void dce_cs_rgy_to_loc(
        unsigned32 *rgy_code_set_value,
        idl_char **local_code_set_name,
        unsigned16 *rgy_char_sets_number,
        unsigned16 **rgy_char_sets_value,
        error_status_t *status);
```

Parameters

Input

rgy_code_set_value - The registered hexadecimal value that uniquely identifies the code set.

Output

local_code_set_name - A string that specifies the name that the local host's locale environment uses to refer to the code set. The string is a maximum of 32 bytes: 31 data bytes and a terminating NULL character.

rgy_char_sets_number - The number of character sets that the specified code set encodes. Specifying NULL in this parameter prevents the routine from returning this value.

rgy_char_sets_value - A pointer to an array of registered integer values that uniquely identify the character set(s) that the specified code set encodes. Specifying NULL in this parameter prevents the routine from returning this value. The routine dynamically allocates this value.

status - Returns the status code from this routine. This status code indicates whether the routine completed successfully or, if not, why not.

The possible status codes and their meanings are as follows:

- dce_cs_c_ok – Code set registry access operation succeeded.
- dce_cs_c_cannot_allocate_memory – Cannot allocate memory for code set info.
- dce_cs_c_unknown – The requested code set value was not found in the code set registry.
- dce_cs_c_notfound – No local code set name was found in the registry which corresponds to the specific code set registry ID value. This implies that the code set is not supported in the local system environment.

Description

The dce_cs_rgy_to_loc() routine maps a unique identifier for a code set in the code set registry to the operating system-specific string name for the code set, if it exists in the code set registry.

The dce_cs_rgy_to_loc() routine takes as input a registered integer value of a code set and returns a string that holds the operating system-specific, or local name, of the code set.

If the code set identifier does not exist in the registry, the routine returns the status dce_cs_c_unknown and returns an undefined string.

The routine also returns the number of character sets that the code set encodes and the registered integer values that uniquely identify those character sets. Specifying NULL in the rgy_char_sets_number and rgy_char_sets_value[] parameters prevents the routine from performing the additional search for these values. Applications that want only to obtain a local code set name from the code set registry can specify NULL for

these parameters in order to improve the routine's performance. If the value is returned from the routine, application developers should free the rgy_char_sets_value array after it is used.

13.9.2.3 *rpc_cs_char_set_compat_check*

Evaluates character set compatibility between a client and a server.

Synopsis

```
void rpc_cs_char_set_compat_check(
        unsigned32 client_rgy_code_set_value,
        unsigned32 server_rgy_code_set_value,
        error_status_t *status);
```

Parameters

Input

client_rgy_code_set_value - The registered hexadecimal value that uniquely identifies the code set that the client is using as its local code set.

server_rgy_code_set_value - The registered hexadecimal value that uniquely identifies the code set that the server is using as its local code set.

Output

status - Returns the status code from this routine. This status code indicates whether the routine completed successfully or, if not, why not.

The possible status codes and their meanings are as follows:

- rpc_s_ok – Successful status.
- rpc_s_ss_no_compat_charsets – No compatible code set found. The client and server do not have a common encoding that both could recognize and convert.
- The routine can also return status codes from the dce_cs_rgy_to_loc() routine.

Description

The rpc_cs_char_set_compat_check() routine provides a method for determining character set compatibility between a client and a server; if the server's character set is incompatible with that of the client, then connecting to that server is most likely not acceptable, since massive data loss would result from such a connection.

The routine takes the registered integer values that represent the code sets that the client and server are currently using and calls the code set registry to obtain the registered values that represent the character set(s) that the specified code sets support. If both client and server support just one character set, the routine compares client and server registered character set values to determine whether or not the sets are compatible. If they are not, the routine returns the status message rpc_s_ss_no_compat_charsets.

If the client and server support multiple character sets, the routine determines whether at least two of the sets are compatible. If two or more sets match, the routine considers the character sets compatible, and returns a success status code to the caller.

13.9.2.4 *rpc_rgy_get_max_bytes*

Gets the maximum number of bytes that a code set uses to encode one character from the code set registry on a host

Synopsis

```
void rpc_rgy_get_max_bytes(
   unsigned32 rgy_code_set_value,
   unsigned16 *rgy_max_bytes,
   error_status_t *status);
```

Parameters

Input

rgy_code_set_value - The registered hexadecimal value that uniquely identifies the code set.

Output

rgy_max_bytes - The registered decimal value that indicates the number of bytes this code set uses to encode one character.

status - Returns the status code from this routine. This status code indicates whether the routine completed successfully or, if not, why not.

The possible status codes and their meanings are as follows:

- rpc_s_ok – Operation succeeded.
- dce_cs_c_cannot_allocate_memory – Cannot allocate memory for code set info.
- dce_cs_c_unknown – No code set value was not found in the registry which corresponds to the code set value specified.
- dce_cs_c_notfound – No local code set name was found in the registry which corresponds to the value specified.

Description

The rpc_rgy_get_max_bytes() routine reads the code set registry on the local host. It takes the specified registered code set value, uses it as an index into the registry, and returns the decimal value that indicates the number of bytes that the code set uses to encode one character.

This information can be used for buffer sizing as part of the procedure to determine whether additional storage needs to be allocated for conversion between local and network code sets.

Building Inter-ORB Bridges 14

Contents

This chapter contains the following sections.

14.1 Introduction

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.

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 13.6.2, “Interoperable Object References: IORs,” on page 13-15.

14.2 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 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 has 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 be built similarly without needing information about ORB internals. Only in-line bridges (using either standard or nonstandard mediation mechanisms) need potentially proprietary information about ORB internals.

---

1. Special initialization supporting object referencing domains (e.g., two protocols) to be exposed to application programmers to support construction of this style bridge.

14.2.1 In-line Bridging

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.)

In this approach, the required bridging functionality can be provided by a combination of software components at various levels:

- As additional or alternative services provided by the underlying ORBs
- As additional or alternative stub and skeleton code.

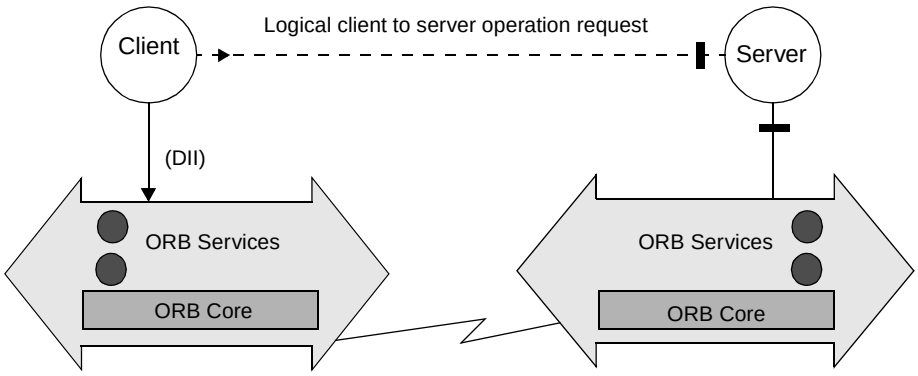


Figure 14-1 In-Line bridges are built using *ORB internal APIs*.

14.2.2 Request-level Bridging

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.

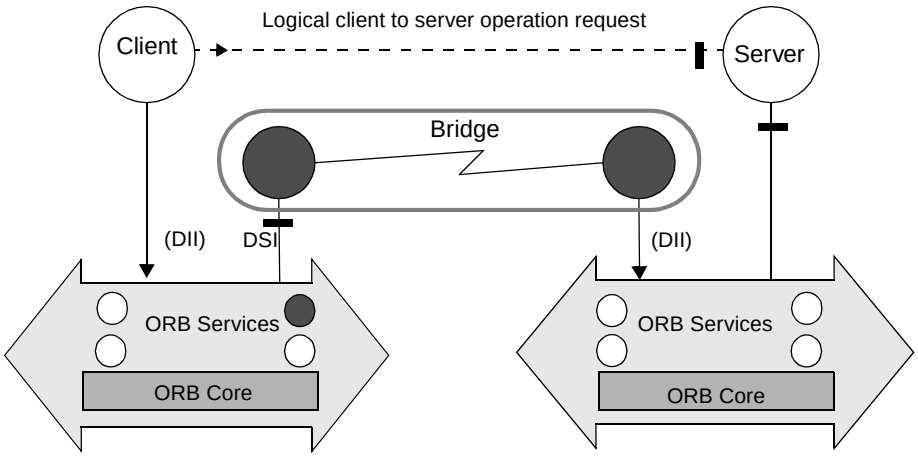


Figure 14-2 Request-Level bridges are built using *public ORB APIs*.

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.

14.2.3 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.

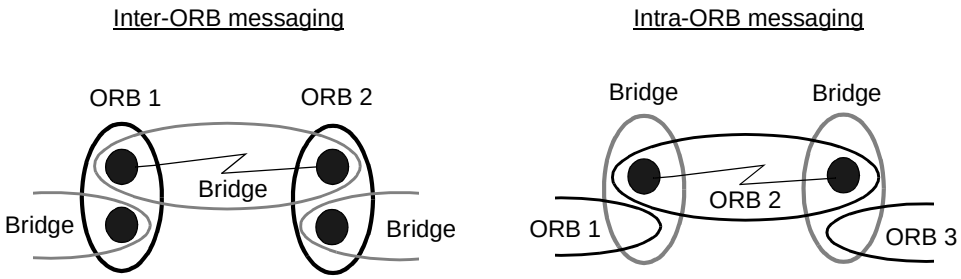


Figure 14-3 When the two ORBs are collocated in a bridge execution environment, network communications will be purely intra-ORB. If the ORBs are not collocated, such communications must go between ORBs.

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.

14.3 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 Section 13.5.2, “Handling of Referencing Between Domains,” on page 13-13.

- 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 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-compliant ORBs, type domains by definition do not need to be bridged.

14.4 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 using generic bridges.

14.5 Building Generic Request-Level Bridges

The CORBA Core specifications 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 the General Inter-ORB Protocol chapter).
- 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 14-2, 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 ORB-specific proxy object references, and bridge-internal interfaces are used to create the analogous data structure for the mediating protocol.

14.6 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.

14.7 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: Common Object Services Specification*.) 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 Section 4.7, “Obtaining Initial Object References,” on page 4-18 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 15

The General Inter-ORB Protocol chapter has been updated based on CORE changes from ptc/98-09-04, the Object by Value documents (orbos/98-01-18 and ptc/98-07-06), bidirectional GIOP changes (interop/98-07-01) and the results of Interop RTF 2.4 (interop/99-03-01) have been incorporated. Please note that all changes since the base 2.2 version of this chapter are marked with changebars.

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.

Contents

This chapter contains the following sections.

15.1 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.

15.2 GIOP 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 IIOP are shown in Section 15.10, “OMG IDL,” on page 15-56. Fragments of the specification are used throughout this chapter as necessary.

15.2.1 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.

15.2.2 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: Common Object Service Specifications*). This mechanism is designed to support any service that requires service related context to be implicitly passed with requests.

15.2.3 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 15-43). 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. In GIOP versions 1.0 and 1.1, the server side receives requests and sends replies. The server side of a connection may not send object requests. This restriction, which was included to make GIOP 1.0 and 1.1 much simpler and avoid certain race conditions, has been relaxed for GIOP version 1.2, as specified in the BiDirectional GIOP specification, see Section 15.8, "Bi-Directional GIOP," on page 15-52.
- ***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.

15.3 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 Section 15.3.1.1, “Alignment,” on page 15-5.

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 Section 15.4, “GIOP Message Formats,” on page 15-29.

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 be marshaled subsequently 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.

15.3.1 Primitive Types

Primitive data types are specified for both big-endian and little-endian orderings. The message formats (see Section 15.4, “GIOP Message Formats,” on page 15-29) 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 Section 15.3.3, “Encapsulation,” on page 15-13. 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.

15.3.1.1 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 15-1 gives alignment boundaries for CDR/OMG-IDL primitive types.

Table 15-1 Alignment requirements for OMG IDL primitive data types

TYPE	OCTET ALIGNMENT
char	1
wchar	1, 2, or 4, depending on code set
octet	1
short	2
unsigned short	2
long	4
unsigned long	4
long long	8
unsigned long long	8
float	4
double	8
long 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 a GIOP message header (see Section 15.4.1, “GIOP Message Header,” on page 15-29) and at the beginning of an encapsulation, even if the encapsulation itself is nested in another encapsulation. (See Section 15.3.3, “Encapsulation,” on page 15-13).

15.3.1.2 Integer Data Types

Figure 15-1 on page 15-7 illustrates the representations for OMG IDL integer data types, including the following data types:

- **short**
- **unsigned short**
- **long**
- **unsigned long**
- **long long**

- **unsigned long long**

The figure illustrates bit ordering and size. Signed types (**short**, **long**, and **long long**) are represented as two's complement numbers; unsigned versions of these types are represented as unsigned binary numbers.

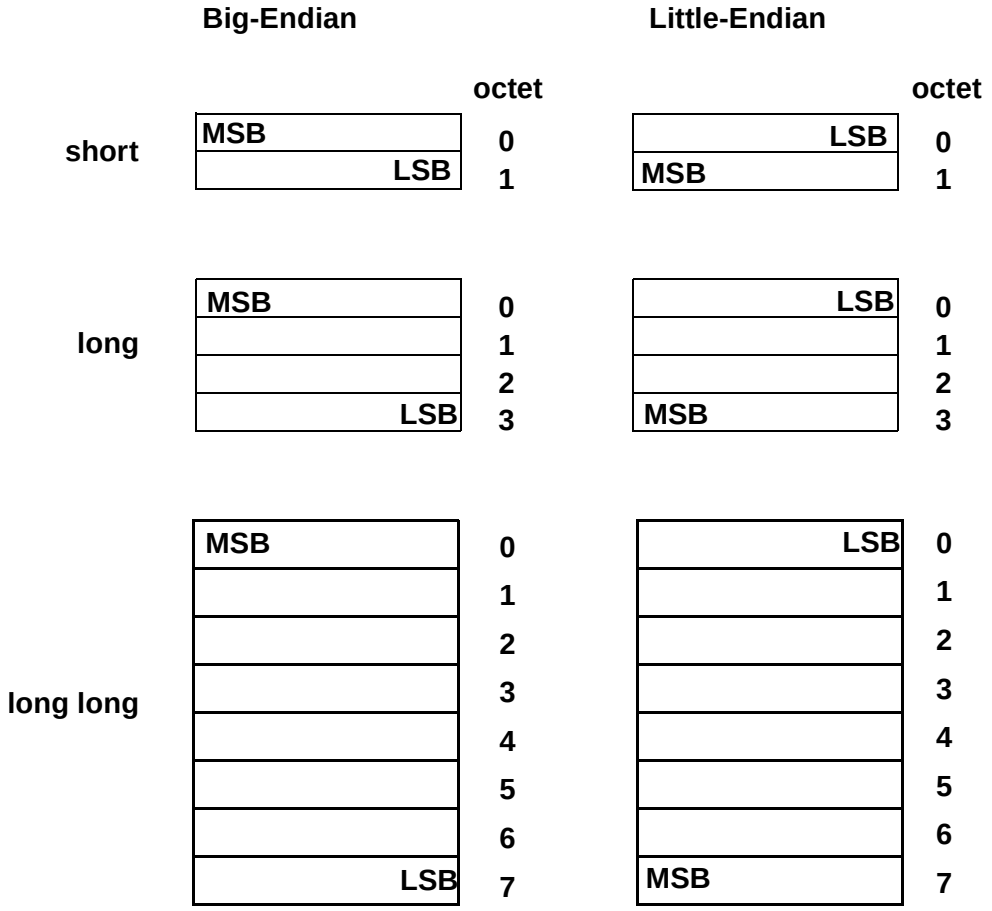


Figure 15-1 Sizes and bit ordering in big-endian and little-endian encodings of OMG IDL integer data types, both signed and unsigned.

15.3.1.3 Floating Point Data Types

Figure 15-2 on page 15-9 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

1. "IEEE Standard for Binary Floating-Point Arithmetic," ANSI/IEEE Standard 754-1985, Institute of Electrical and Electronics Engineers, August 1985.

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:

$$-1^{sign} \times 2^{(exponent - 127)} \times (1 + fraction)$$

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:

$$-1^{sign} \times 2^{(exponent - 1023)} \times (1 + fraction)$$

For double-extended floating-point values the exponent is 15 bits long, comprising e1 and e2 in the figure, where the 7 bits in e1 are the most significant. The fractional mantissa (f1 through f14) is 112 bits long, with f1 being the most significant. The value of a **long double** is determined by:

$$-1^{sign} \times 2^{(exponent - 16383)} \times (1 + fraction)$$

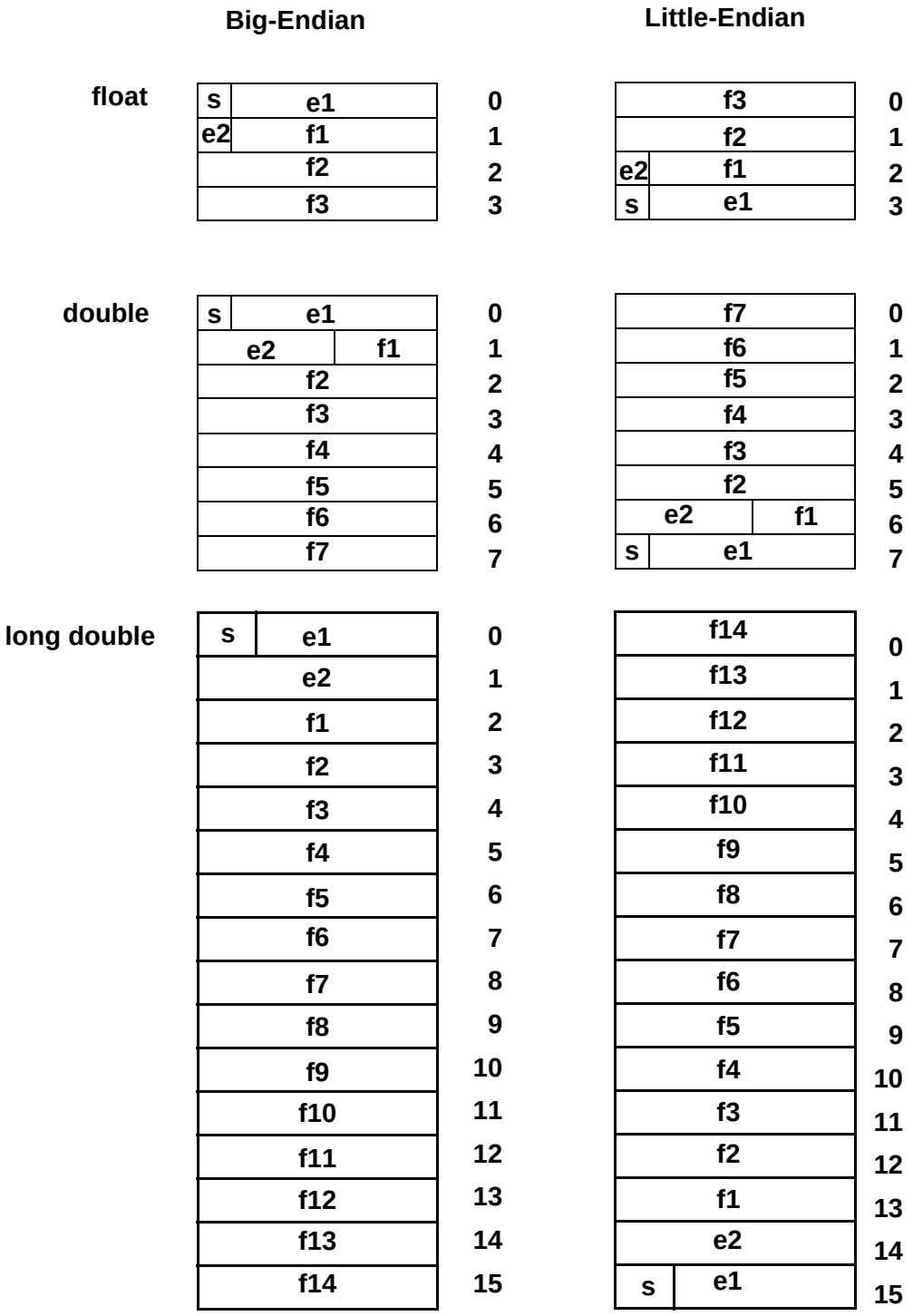


Figure 15-2 Sizes and bit ordering in big-endian and little-endian representations of OMG IDL single, double precision, and double extended floating point numbers.

15.3.1.4 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.

15.3.1.5 Boolean

Boolean values are encoded as single octets, where **TRUE** is the value 1, and **FALSE** as 0.

15.3.1.6 Character Types

An IDL character is represented as a single octet; the code set used for transmission of character data (e.g., TCS-C) between a particular client and server ORBs is determined via the process described in Section 13.7, “Code Set Conversion,” on page 13-27. In the case of multi-byte encodings of characters, a single instance of the **char** type may only hold one octet of any multi-byte character encoding.

Note – Full representation of multi-byte characters will require the use of an array of IDL **char** variables.

For GIOP version 1.1, the transfer syntax for an IDL wide character depends on whether the transmission code set (TCS-W, which is determined via the process described in Section 13.7, “Code Set Conversion,” on page 13-27) is byte-oriented or non-byte-oriented:

- Byte-oriented (e.g., SJIS). Each wide character is represented as one or more octets, as defined by the selected TCS-W.
- Non-byte-oriented (e.g., Unicode UTF-16). Each wide character is represented as one or more codepoints. A codepoint is the same as “Coded-Character data element,” or “CC data element” in ISO terminology. Each codepoint is encoded using a fixed number of bits as determined by the selected TCS-W. The OSF Character and Code Set Registry may be examined using the interfaces in Section 13.9, “Relevant OSFM Registry Interfaces,” on page 13-40 to determine the maximum length (max_bytes) of any character codepoint. For example, if the TCS-W is ISO 10646 UCS-2 (Universal Character Set containing 2 bytes), then wide characters are represented as **unsigned shorts**. For ISO 10646 UCS-4, they are represented as **unsigned longs**.

For GIOP version 1.2, **wchar** is encoded as an unsigned binary octet value, followed by the elements of the octet sequence representing the encoded value of the **wchar**. The initial octet contains a count of the number of elements in the sequence, and the elements of the sequence of octets represent the **wchar**, using the negotiated wide character encoding.

Note – The GIOP 1.2 encoding of **wchar** is similar to the encoding of an octet sequence, except for its use of a single octet to encode the value of the length.

For GIOP versions prior to 1.2, interoperability for wchar is limited to the use of two-octet fixed-length encoding.

Wchar values in encapsulations are assumed to be encoded using GIOP version 1.2 CDR.

15.3.2 OMG IDL Constructed Types

Constructed types are built from OMG IDL's data types using facilities defined by the OMG IDL language.

15.3.2.1 Alignment

Constructed types 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.

15.3.2.2 Struct

The components of a structure are encoded in the order of their declaration in the structure. Each component is encoded as defined for its data type.

15.3.2.3 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.

15.3.2.4 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.

15.3.2.5 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.

15.3.2.6 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 take ascending numeric values, in order of declaration from left to right.

15.3.2.7 Strings and Wide Strings

A string is encoded as an **unsigned long** indicating the length of the string in octets, followed by the string value in single- or multi-byte form represented as a sequence of octets. Both the string length and contents include a terminating null.

For GIOP version 1.1 and 1.2, when encoding a string, always encode the length as the total number of bytes used by the encoding string, regardless of whether the encoding is byte-oriented or not.

For GIOP version 1.1, a wide string is encoded as an **unsigned long** indicating the length of the string in octets or unsigned integers (determined by the transfer syntax for **wchar**) followed by the individual wide characters. Both the string length and contents include a terminating null. The terminating null character for a **wstring** is also a wide character.

For GIOP version 1.2, when encoding a **wstring**, always encode the length as the total number of octets used by the encoded value, regardless of whether the encoding is byte-oriented or not. For GIOP version 1.2 a **wstring** is not terminated by a **NUL** character. In particular, in GIOP version 1.2 a length of 0 is legal for **wstring**.

Note – For GIOP versions prior to 1.2, interoperability for wstring is limited to the use of two-octet fixed-length encoding.

Wstring values in encapsulations are assumed to be encoded using GIOP version 1.2 CDR.

15.3.2.8 Fixed-Point Decimal Type

The IDL **fixed** type has no alignment restrictions, and is represented as shown in Table 15-4 on page 15-13. Each **octet** contains (up to) two decimal digits. If the **fixed** type has an odd number of decimal digits, then the representation begins with the first (most significant) digit — d0 in the figure. Otherwise, this first half-octet is all zero, and the first digit is in the second half-octet — d1 in the figure. The sign configuration, in the last half-octet of the representation, is 0xD for negative numbers and 0xC for positive and zero values.

Decimal digits are encoded as hexadecimal values in each half-octet as follows:

Decimal Digit	Half-Octet Value
0	0x0
1	0x1
2	0x2
...	...
9	0x9

Figure 15-3 Decimal Digit Encoding for Fixed Type

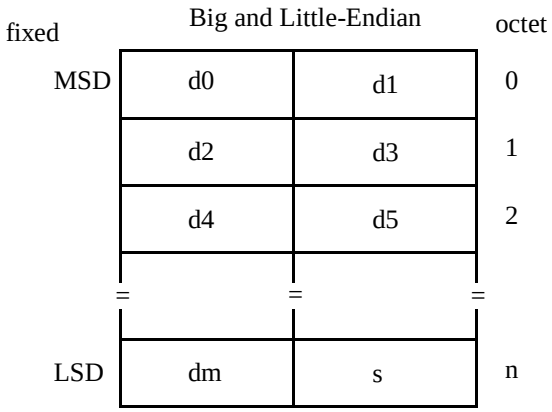


Figure 15-4 IDL Fixed Type Representation

15.3.3 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 Section 15.3.5.1, “TypeCode,” on page 15-22), the IIOP protocol profile inside an IOR (see Section 15.3.6, “Object References,” on page 15-28), and in service-specific context (see Section 13.6.7, “Object Service Context,” on page 13-22). In addition, some ORBs may choose to use an encapsulation to hold the **object_key** (see Section 15.7.2, “IIOP IOR Profiles,” on page 15-49), 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 Section 15.4.1, "GIOP Message Header," on page 15-29). 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 Section 15.3.2.5, "Sequence," on page 15-12). 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]

15.3.4 Value Types

Value types are built from OMG IDL's value type definitions. Their representation and encoding is defined in this section.

Value types may be used to transmit and encode complex state. The general approach is to support the transmission of the data (state) and type information encoded as **RepositoryID**s.

The loading (and possible transmission) of code is outside of the scope of the GIOP definition, but enough information is carried to support it, via the CodeBase object.

The format makes a provision for the support of custom marshaling (i.e., the encoding and transmission of state using application-defined code). Consistency between custom encoders and decoders is not ensured by the protocol

The encoding supports all of the features of value types as well as supporting the "chunking" of value types. It does so in a compact way.

At a high level the format can be described as the linearization of a graph. The graph is the depth-first exploration of the transitive closure that starts at the top-level value object and follows its "reference to value objects" fields (an ordinary remote reference is just written as an IOR). It is a recursive encoding similar to the one used for TypeCodes. An indirection is used to point to a value that has already been encoded.

---

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.

The data members are written beginning with the highest possible base type to the most derived type in the order of their declaration.

15.3.4.1 *Partial Type Information and Versioning*

The format provides support for partial type information and versioning issues in the receiving context. However the encoding has been designed so that this information is only required when “advanced features” such as truncation are used.

The presence (or absence) of type information and codebase URL information is indicated by flags within the <value_tag>, which is a **long** in the range between **0x7fffff00** and **0x7fffffff** inclusive. The last octet of this tag is interpreted as follows:

- The least significant bit (<value_tag> & **0x00000001**) is the value **1** if a <codebase_URL> is present. If this bit is **0**, no <codebase_URL> follows in the encoding. The <codebase_URL> is a blank-separated list of one or more URLs.
- The second and third least significant bits (<value_tag> & **0x00000006**) are:
  - the value **0** if no type information is present in the encoding. This indicates the actual parameter is the same type as the formal argument.
  - the value **2** if only a single repository id is present in the encoding, which indicates the most derived type of the actual parameter (which may be either the same type as the formal argument or one of its derived types).
  - the value **6** if the partial type information list of repository ids is present in the encoding as a list of repository ids.

When a list of **RepositoryIDs** is present, the encoding is a **long** specifying the number of **RepositoryIDs**, followed by the **RepositoryIDs**. The first **RepositoryID** is the id for the most derived type of the value. If this type has any base types, the sending context is responsible for listing the **RepositoryIDs** for all the base types to which it is safe to truncate the value passed. These truncatable base types are listed in order, going up the derivation hierarchy. The sending context may choose to (but need not) terminate the list at any point after it has sent a **RepositoryID** for a type well-known to the receiving context. A well-known type is any of the following:

- a type that is a formal parameter, result of the method call, or exception, for which this GIOP message is being marshaled
- a base type of a well-known type
- a member type of a well-known type
- an element type of a well known type

For value types that have an RMI: **RepositoryId**, ORBs must include at least the most derived **RepositoryId**, in the value type encoding.

For value types marshaled as abstract interfaces (see Section 15.3.7, “Abstract Interfaces,” on page 15-29), **RepositoryId** information must be included in the value type encoding.

If the receiving context needs more typing information than is contained in a GIOP message that contains a codebase URL information, it can go back to the sending context and perform a lookup based on that **RepositoryID** to retrieve more typing information (e.g., the type graph).

CORBA **RepositoryIDs** may contain standard version identification (major and minor version numbers or a hash code information). The ORB run time may use this information to check whether the version of the value being transmitted is compatible with the version expected. In the event of a version mismatch, the ORB may apply product-specific truncation/conversion rules (with the help of a local interface repository or the **SendingContext::RunTime** service). For example, the Java serialization model of truncation/conversion across versions can be supported. See the JDK 1.1 documentation for a detailed specification of this model.

15.3.4.2 Example

The following examples demonstrate legal combinations of truncatability, actual parameter types and GIOP encodings. This is not intended to be an exhaustive list of legal possibilities.

The following example uses valuetypes **animal** and **horse**, where **horse** is derived from **animal**. The actual parameters passed to the specified operations are **an_animal** of runtime type **animal** and **a_horse** of runtime type **horse**.

The following combinations of truncatability, actual parameter types and GIOP encodings are legal.

1. If there is a single operation:

op1(in animal a);

a) If the type **horse** cannot be truncated to **animal** (i.e., **horse** is declared):

valuetype horse: animal ...

then the encoding is as shown in Table 15-2 below:

Table 15-2

Actual Invocation	Legal Encoding
op1(a_horse)	2 horse
	6 1 horse

Note that if the type horse is not available to the receiver, then the receiver throws a demarshaling exception.

b). If the type **horse** can be truncated to **animal** (i.e., **horse** is declared):

valuetype horse: truncatable animal ...

then the encoding is as shown in Table 15-3 below

Table 15-3

Actual Invocation	Legal Encoding
op1(a_horse)	6 2 horse animal

Note that if the type horse is not available to the receiver, then the receiver tries to truncate to animal.

c) Regardless of the truncation relationships, when the exact type of the formal argument is sent, then the encoding is as shown in Table 15-4 below:

Table 15-4

Actual Invocation	Legal Encoding
op1(an_animal)	0
	2 animal
	6 1 animal

2. Given the additional operation:

op2(in horse h);

(i.e., the sender knows that both types **horse** and **animal** and their derivation relationship are known to the receiver)

a). If the type horse cannot be truncated to animal (i.e., horse is declared):

valuetype horse: animal ...

then the encoding is as shown in Table 15-5 below:

Table 15-5

Actual Invocation	Legal Encoding
op2(a_horse)	2 horse
	6 1 horse

Note that the demarshaling exception of case 1 will not occur, since horse is available to the receiver.

b). If the type horse can be truncated to animal (i.e., horse is declared):

valuetype horse: truncatable animal ...

then the encoding is as shown in Table 15-6 below:

Table 15-6

Actual Invocation	Legal Encoding
op2 (a_horse)	2 horse
	6 1 horse
	6 2 horse animal

Note that truncation will not occur, since horse is available to the receiver.

15.3.4.3 Scope of the Indirections

The special value **0xffffffff** introduces an indirection (i.e., it directs the decoder to go somewhere else in the marshaling buffer to find what it is looking for). This can be codebase URL information which has already been encoded, a **RepositoryID** which has already been encoded, a list of repository IDs which has already been encoded, or another value object which is shared in a graph. **0xffffffff** is always followed by a **long** indicating where to go in the buffer.

The encoding used for indirection is the same as that used for recursive TypeCodes (i.e., a **0xffffffff** indirection marker followed by a **long** offset (in units of **octets**) from the beginning of the long offset). As an example, this means that an offset of negative four (-4) is illegal, because it is self-indirecting to its indirection marker. Indirections may refer to any preceding location in the GIOP message, including previous fragments if fragmentation is used. This includes any previously marshaled parameters. Non-negative offsets are reserved for future use. Indirections may not cross encapsulation boundaries.

15.3.4.4 Other Encoding Information

A “new” value is coded as a value header followed by the value’s state. The header contains a tag and codebase URL information if appropriate, followed by the **RepositoryID** and an octet flag of bits. Because the same **RepositoryID** (and codebase URL information) could be repeated many times in a single request when sending a complex graph, they are encoded as a regular string the first time they appear, and use an indirection for later occurrences.

15.3.4.5 Fragmentation

It is anticipated that value types may be rather large, particularly when a graph is being transmitted. Hence the encoding supports the breaking up of the serialization into an arbitrary number of chunks in order to facilitate incremental processing.

Values with truncatable base types need a length indication in case the receiver needs to truncate them to a base type. Value types that are custom marshaled also need a length indication so that the ORB run time can know exactly where they end in the stream without relying on user-defined code. This allows the ORB to maintain consistency and

ensure the integrity of the GIOP stream when the user-written custom marshaling and demarshaling does not marshal the entire value state. For simplicity of encoding, we use a length indication for all values whether or not they have a truncatable base type or use custom marshaling.

If limited space is available for marshaling, it may be necessary for the ORB to send the contents of a marshaling buffer containing a partially marshaled value as a GIOP fragment. At that point in the marshaling, the length of the entire value being marshaled may not be known. Calculating this length may require processing as costly as marshaling the entire value. It is therefore desirable to allow the value to be encoded as multiple chunks, each with its own length. This allows the portion of a value that occupies a marshaling buffer to be sent as a chunk of known length with no need for additional length calculation processing.

The data may be split into multiple chunks at arbitrary points except within primitive CDR types, arrays of primitive types, strings, and wstrings. It is never necessary to end a chunk within one of these types as the length of these types is known before starting to marshal them so they can be added to the length of the currently open chunk. It is the responsibility of the CDR stream to hide the chunking from the marshaling code.

The presence (or absence) of chunking is indicated by flags within the <value_tag>. The fourth least significant bit (<value_tag> & **0x00000008**) is the value 1 if a chunked encoding is used for the value's state. The chunked encoding is required for custom marshaling and truncation. If this bit is 0, the state is encoded as <**octets**>.

Each chunk is preceded by a positive long which specifies the number of octets in the chunk.

A chunked value is terminated by an end tag which is a non-positive long so the start of the next value can be differentiated from the start of another chunk. In the case of values which contain other values (e.g., a linked list) the "nested" value is started without there being an end tag. The absolute value of an end tag (when it finally appears) indicates the nesting level of the value being terminated. A single end tag can be used to terminate multiple nested values. The detailed rules are as follows:

- End tags, chunk size tags, and value tags are encoded using non-overlapping ranges so that the unmarshaling code can tell after reading each chunk whether:
  - another chunk follows (positive tag).
  - one or multiple value types are ending at a given point in the stream (negative tag).
  - a nested value follows (special large positive tag).
- The end tag is a negative long whose value is the negation of the absolute nesting depth of the value type ending at this point in the CDR stream. Any value types that have not already been ended and whose nesting depth is greater than the depth indicated by the end tag are also implicitly ended. The tag value **0** is reserved for future use (e.g., supporting a nesting depth of more than **2^31**). The outermost value type will always be terminated by an end tag with a value of **-1**.

  The following example describes how end tags may be used. Consider a valuetype declaration that contains two member values:

```
// IDL
valuetype simpleNode { ... };
valuetype node truncatable simpleNode {
    public node node1;
    public node node2;
};
```

When an instance of type '**node**' is passed as a parameter of type '**simpleNode**', a chunked encoding is used. In all cases, the outermost value is terminated with an end tag with a value of **-1**. The nested value '**node1**' is terminated with an end tag with a value of **-2** since only the second-level value, '**node1**,' ends at that point. Since the nested value '**node2**' coterminates with the outermost value, either of the following end tag layouts is legal:

- A single end tag with a value of **-1** marks the termination of the outermost value, implying the termination of the nested value, '**node2**'as well. This is the most compact marshaling.
- An end tag with a value of **-2** marks the termination of the nested value, '**node2**.' This is then followed by an end tag with a value of **-1** to mark the termination of the outermost value.

Because data members are encoded in their declaration order, declaring a value type data member of a value type last is likely to result in more compact encoding on the wire because it maximizes the number of values ending at the same place and so allows a single end tag to be used for multiple values. The canonical example for that is a linked list.

- Chunks are never nested. When a value is nested within another value, the outer value's chunk ends at the place in the stream where the inner value starts. If the outer value has non-value data to be marshaled following the inner value, the end tag for the inner value is followed by a continuation chunk for the remainder of the outer value. For the purposes of chunking, values encoded as indirections or null are treated as non-value data.
- Regardless of the above rules, any value nested within a chunked value is always chunked. Furthermore, any such nested value that is truncatable must encode its type information as a list of **RepositoryIDs** (see Section 15.3.4.1, "Partial Type Information and Versioning," on page 15-15).

Truncating a value type in the receiving context may require keeping track of unused nested values (only during unmarshaling) in case further indirection tags point back to them. These values can be held in their "raw" GIOP form, as fully unmarshaled value objects, or in any other product-specific form.

Value types that are custom marshaled are encoded as chunks in order to let the ORB run-time know exactly where they end in the stream without relying on user-defined code.

15.3.4.6 Notation

The on-the-wire format is described by a BNF grammar with conventions similar to the ones used to define IDL syntax. ***The terminals of the grammar are to be interpreted differently.*** We are describing a protocol format. Although the terminals have the same names as IDL tokens they represent either:

- constant tags, or
- the GIOP CDR encoding of the corresponding IDL construct.

For example, **long** is a shorthand for the GIOP encoding of the IDL **long** data type - with all the GIOP alignment rules. Similarly **struct** is a shorthand for the GIOP CDR encoding of a **struct**.

A **(type) constant** means that an instance of the given type having the given value is encoded according to the rules for that type. So that **(long) 0** means that a CDR encoding for a long having the value **0** appears at that location.

15.3.4.7 The Format

```
(1)          <value>  ::= <value_tag> [  <codebase_URL> ]
                          [ <type_info> ] <state>
                      |  <value_ref>
(2)      <value_ref>  ::= <indirection_tag>  <indirection> | <null_tag>
(3)      <value_tag>  ::= long// 0x7fffff00 <= value_tag <= 0x7fffffff
(4)      <type_info>  ::= <rep_ids> | <repository_id>
(5)          <state>  ::= <octets> |<value_data>+ [ <end_tag> ]
(6)     <value_data>  ::= <value_chunk> | <value>
(7)        <rep_ids>  ::= long <repository_id>+
                      |  <indirection_tag> <indirection>
(8)  <repository_id>  ::= string | <indirection_tag> <indirection>
(9)    <value_chunk>  ::= <chunk_size_tag> <octets>
(10)        <null_tag>  ::= (long) 0
(11)  <indirection_tag>  ::= (long) 0xffffffff
(12)   <codebase_URL>  ::= string | <indirection_tag> <indirection>
(13)  <chunk_size_tag>  ::= long
                          //  0 < chunk_size_tag < 2^31-256 (0x7fffff00)
(14)      <end_tag>  ::= long     // -2^31 < end_tag < 0
(15)   <indirection>  ::= long     // -2^31 < indirection < 0
(16)        <octets>  ::= octet | octet <octets>
```

The concatenated octets of consecutive value chunks within a value encode state members for the value according to the following grammar:

```
(1)   <state members>  ::= <state_member>
                       |  <state_member> <state members>
(2)    <state_member>  ::= <value_ref>
                          // All legal IDL types should be here
                       |  octet
```

| **boolean**
| **char**
| **short**
| **unsigned short**
| **long**
| **unsigned long**
| **float**
| **wchar**
| **wstring**
| **string**
| **struct**
| **union**
| **sequence**
| **array**
| **Object**
| **any**

15.3.5 Pseudo-Object Types

CORBA defines some kinds of entities that are neither primitive types (integral or floating point) nor constructed ones.

15.3.5.1 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 15-7 on page 15-24. The *integer value* column of 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.
- Typecodes with *simple parameter lists* are encoded as the **TCKind** enum value followed by the parameter value(s), encoded as indicated in Table 15-7. A “simple” parameter list has a fixed number of fixed length entries, or a single parameter which has its length encoded first.

- All other typecodes have *complex parameter lists*, which are encoded as the **TCKind** enum value followed by a CDR encapsulation octet sequence (see Section 15.3.3, “Encapsulation,” on page 15-13) containing the encapsulated, marshaled parameters. The order of these parameters is shown in the fourth column of Table 15-7.

The third column of Table 15-7 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 15-27. This indirection does not need to be exposed to application programmers.

TypeCode Parameter Notation

TypeCode parameters are specified in the fourth column of Table 15-7. The ordering and meaning of parameters is a superset of those given in Section 10.7, “TypeCodes,” on page 10-48; 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**, and **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, before the next tuple is encoded (first, then second, etc.).

Note that the tuples identifying **struct**, union, **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]

Encoded Identifiers and Names

The Repository ID parameters in **tk_objref**, **tk_struct**, **tk_union**, **tk_enum**, **tk_alias**, **tk_except**, **tk_native**, **tk_value**, **tk_value_box** and **tk_abstract_interface** TypeCodes are Interface Repository **RepositoryId** values, whose format is described in the specification of the Interface Repository. For GIOP 1.2 onwards, repositoryID values are mandatory. For GIOP 1.0 and 1.1, **RepositoryId**

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 15-7 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)}
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)}

Table 15-7 TypeCode enum values, parameter list types, and parameters

TCKind	Integer Value	Type	Parameters
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)}
tk_longlong	23	empty	– none –
tk_ulonglong	24	empty	– none –
tk_longdouble	25	empty	– none –
tk_wchar	26	empty	– none –
tk_wstring	27	simple	ulong(max length or zero if unbounded)
tk_fixed	28	simple	ushort(digits), short(scale)
tk_value	29	complex	string (repository ID), string (name, may be empty), short(ValueModifier), TypeCode (of concrete base)[4], ulong (count), {string (member name), TypeCode (member type), short(Visibility)}
tk_value_box	30	complex	string (repository ID), string(name), TypeCode

Table 15-7 TypeCode enum values, parameter list types, and parameters

TCKind	Integer Value	Type	Parameters
tk_native	31	complex	string (repository ID), string(name)
tk_abstract_interface	**32**	**complex**	**string(RepositoryId), string(name)**
– none –	0xffffffff	simple	long (indirection[5])

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. Should be **tk_null** if there is no concrete base.

5. See “Indirection: Recursive and Repeated TypeCodes” on page 15-27.

values are required for **tk_objref** and **tk_except** TypeCodes; for **tk_struct**, **tk_union**, **tk_enum**, and **tk_alias** TypeCodes **RepositoryIds** are optional and encoded as empty strings if omitted.

The name parameters in **tk_objref, tk_struct, tk_union, tk_enum, tk_alias, tk_value, tk_value_box, tk_abstract_interface, tk_native** and **tk_except** TypeCodes and the member name parameters in **tk_struct, tk_union, tk_enum**, **tk_value** 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.

The discriminant value used in the actual typecode parameter associated with the default member position in the list, may be any valid value of the discriminant type, and has no semantic significance (i.e., it should be ignored and is only included for syntactic completeness of union type code marshaling).

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 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 the first constraint identified above. Because of the second constraint, the value of the offset will always be negative.

Fragmentation support in GIOP versions 1.1 and 1.2 introduces the possibility of a header for a **Fragment** Message being marshaled between the target of an indirection and the start of the encapsulation containing the indirection. The octets occupied by any such headers are not included in the calculation of the offset value.

The encoding of such an indirection is as a TypeCode with a “**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. (This means that an offset of negative four (-4) is illegal because it will be self-indirecting.)

15.3.5.2 Any

Any values are encoded as a TypeCode (encoded as described above) followed by the encoded value.

15.3.5.3 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 serving to identify callers (and potential callers). This specification does not 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.

15.3.5.4 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.

15.3.5.5 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.

If an ORB receives a non-standard system exception that it does not support, or a user exception that is not defined as part of the operation's definition, the exception shall be mapped to UNKNOWN.

15.3.6 Object References

Object references are encoded in OMG IDL (as described in Section 13.5, “Object Addressing,” on page 13-12). 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 15-49).

In general, GIOP profiles include at least these three elements:

1. The version number of the transport-specific protocol specification that the server supports.
2. The address of an endpoint for the transport protocol being used.
3. 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.

15.3.7 Abstract Interfaces

Abstract interfaces are encoded as a union with a **boolean** discriminator. The **union** has an *object reference* (see Section 15.3.6, “Object References,” on page 15-28) if the discriminator is **TRUE**, and a *value type* (see Section 15.3.4, “Value Types,” on page 15-14) if the discriminator is **FALSE**. The encoding of value types marshaled as abstract interfaces always includes **RepositoryId** information. If there is no indication whether a nil abstract interface represents a nil object reference or a null valuetype, it shall be encoded as a null valuetype.

15.4 GIOP Message Formats

GIOP has restriction on client and server roles with respect to initiating and receiving messages. For the purpose of GIOP versions 1.0 and 1.1, a client is the agent that opens a connection (see more details in Section 15.5.1, “Connection Management,” on page 15-44) and originates requests. Likewise, for GIOP versions 1.0 and 1.1, a server is an agent that accepts connections and receives requests.When Bidirectional GIOP is in use for GIOP protocol version 1.2, either side may originate messages, as specified in Section 15.8, “Bi-Directional GIOP,” on page 15-52.

GIOP message types are summarized in Table 15-8, 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 15-8 GIOP Message Types and Originators

Message Type	Originator	Value	GIOP Versions
Request	Client	0	1.0, 1.1, 1.2
Reply	Server	1	1.0, 1.1, 1.2
CancelRequest	Client	2	1.0, 1.1, 1.2
LocateRequest	Client	3	1.0, 1.1, 1.2
LocateReply	Server	4	1.0, 1.1, 1.2
CloseConnection	Server	5	1.0, 1.1, 1.2
MessageError	Both	6	1.0, 1.1, 1.2
Fragment	Both	7	1.1, 1.2

15.4.1 GIOP Message Header

All GIOP messages begin with the following header, defined in OMG IDL:

```
module GIOP { // IDL extended for version 1.1 and 1.2
    struct Version {
        octet       major;
```

```
        octet       minor;
    };

#ifndef GIOP_1_1
    // GIOP 1.0
    enum MsgType_1_0 { // Renamed from MsgType
        Request, Reply, CancelRequest,
        LocateRequest, LocateReply,
        CloseConnection, MessageError
    };

#else
    // GIOP 1.1
    enum MsgType_1_1 {
        Request, Reply, CancelRequest,
        LocateRequest, LocateReply,
        CloseConnection, MessageError,
        Fragment            // GIOP 1.1 addition
    };
#endif  // GIOP_1_1

    // GIOP 1.0
    struct MessageHeader_1_0 { // Renamed from MessageHeader
        char            magic [4];
        Version         GIOP_version;
        boolean         byte_order;
        octet           message_type;
        unsigned long   message_size;
    };

    // GIOP 1.1
    struct MessageHeader_1_1 {
        char            magic [4];
        Version         GIOP_version;
        octet           flags;          // GIOP 1.1 change
        octet           message_type;
        unsigned long   message_size;
    };

    // GIOP 1.2
    typedef MessageHeader_1_1 MessageHeader_1_2;
};
```

The message header clearly identifies GIOP messages and their byte-ordering. The header is independent of byte ordering except for the field encoding message size.

- **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 Section 15.3.6, “Object References,” on page 15-28) though it has the same structure. The major GIOP version number of this specification is one (1); the minor versions are zero (0), one (1), and two (2).

A server implementation supporting a minor GIOP protocol version 1.n (with n > 0 and n < 3), must also be able to process GIOP messages having minor protocol version 1.m, with m less than n. A GIOP server which receives a request having a greater minor version number than it supports, should respond with an error message having the highest minor version number that that server supports, and then close the connection.

A client should not send a GIOP message having a higher minor version number than that published by the server in the tag Internet IIOP Profile body of an IOR.

- **byte_order** (in GIOP 1.0 only) 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.
- flags (in GIOP 1.1 and 1.2) is an 8-bit octet. The least significant bit 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. The byte order for fragment messages must match the byte order of the initial message that the fragment extends.

  The second least significant bit indicates whether or not more framents follow. A value of **FALSE** (0) indicates this message is the last fragment, and **TRUE** (1) indicates more fragments follow this message.

  The most significant 6 bits are reserved. These 6 bits must have value 0 for GIOP version 1.1 and 1.2.
- **message_type** indicates the type of the message, according to Table 15-8; these correspond to enum values of type **MsgType**.
- **message_size** contains the number of octets in the message following the message header, encoded using the byte order specified in the byte order bit (the least significant bit) in the **flags** field (or using the byte_order field in GIOP 1.0). It refers to the size of the message body, not including the 12-byte message header. 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.

  For GIOP version 1.2, if the second least significant bit of **Flags** is 1, the sum of the **message_size** value and 12 must be evenly divisible by 8.

15.4.2 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

15.4.2.1 Request Header

The request header is specified as follows:

```
module GIOP { // IDL extended for version 1.1 and 1.2

    // GIOP 1.0
    struct RequestHeader_1_0 { // Renamed from RequestHeader
        IOP::ServiceContextList     service_context;
        unsigned long               request_id;
        boolean                     response_expected;
        sequence <octet>            object_key;
        string                      operation;
        Principal                   requesting_principal;
    };

    // GIOP 1.1
    struct RequestHeader_1_1 {
        IOP::ServiceContextList     service_context;
        unsigned long               request_id;
        boolean                     response_expected;
        octet                       reserved[3]; // Added in GIOP 1.1
        sequence <octet>            object_key;
        string                      operation;
        Principal                   requesting_principal;
    };

    // GIOP 1.2
    typedef short               AddressingDisposition;
    const short                 KeyAddr = 0;
    const short                 ProfileAddr = 1;
    const short                 ReferenceAddr = 2;

    struct IORAddressingInfo {
        unsigned long               selected_profile_index;
        IOP::IOR                    ior;
    };
```

```
    union TargetAddress switch (AddressingDisposition) {
        case KeyAddr:           sequence <octet> object_key;
        case ProfileAddr:       IOP::TaggedProfile profile;
        case ReferenceAddr:     IORAddressingInfo ior;
    };

    struct RequestHeader_1_2 {
        unsigned long                   request_id;
        octet                           response_flags;
        octet                           reserved[3];
        TargetAddress                   target;
        string                          operation;
        IOP::ServiceContextList         service_context;
        // Principal not in GIOP 1.2
    };
};
```

The members have the following definitions:

- **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 15-38).

- The lowest order bit of **response_flags** is set to **1** if a reply message is expected for this request. If the operation is not defined as **oneway**, and the request is not invoked via the DII with the **INV_NO_RESPONSE** flag set, **response_flags** must be set to **\x03**.

  If the operation is defined as oneway, or the request is invoked via the DII with the **INV_NO_RESPONSE** flag set, **response_flags** may be set to **x00 or \x01**. Asking for a reply gives the client ORB an opportunity to receive **LOCATION_FORWARD** responses and replies that might indicate system exceptions. When this flag is set to **\x01** for a oneway operation, receipt of a reply does not imply that the operation has necessarily completed.

- **reserved** is always set to **0** in GIOP 1.1. These three octets are reserved for future use.

- For GIOP 1.0 and 1.1, **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.

- For GIOP 1.2, **target** identifies the object which is the target of the invocation. The possible values of the union are:
  - **KeyAddr** 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.

- **ProfileAddr** is the transport-specific GIOP profile selected for the target's IOR by the client ORB.
- **IORAddressingInfo** is the full IOR of the target object. The **selected_profile_index** indicates the transport-specific GIOP profile that was selected by the client ORB.

- **operation** is the IDL identifier naming, within the context of the interface (not a fully qualified scoped name), 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 (Section 4.3, "Object Reference Operations," on page 4-8) and that correspond to GIOP request messages, the operation names are **_interface**, **_is_a**, **_non_existent**, and **_get_domain_managers**.

  For GIOP 1.2 and later versions, only the operation name **_non_existent** shall be used.

  The correct operation name to use for GIOP 1.0 and 1.1 is **_non_existent**. Due to a typographical error in CORBA 2.0, 2.1, and 2.2, some legacy implementations of GIOP 1.0 and 1.1 respond to the operation name **_not_existent**. For maximum interoperability with such legacy implementations, new implementations of GIOP 1.0 and 1.1 may wish to respond to both operation names, **_non_existent** and **_not_existent**.

- **service_context** contains ORB service data being passed from the client to the server, encoded as described in Section 13.6.7, "Object Service Context," on page 13-22.

- **requesting_principal** contains a value identifying the requesting principal. It is provided to support the **BOA::get_principal** operation. The usage of the **requesting_principal** field is deprecated for GIOP versions 1.0 and 1.1. The field is not present in the request header for GIOP version 1.2.

15.4.2.2 Request Body

In GIOP versions 1.0 and 1.1, request bodies are marshaled into the CDR encapsulation of the containing Message immediately following the Request Header. In GIOP version 1.2, the Request Body is always aligned on an 8-octet boundary. The fact that GIOP specifies the maximum alignment for any primitive type is 8 guarantees that the Request Body will not require remarshaling if the Message or Request header are modified. The data for 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 Section 15.3.5.4, “Context,” on page 15-28. 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 long p);
```

would be equivalent to this structure:

```
struct example_body {
    short           m;      // leftmost in or inout parameter
    long            p;      // ... to the rightmost
};
```

15.4.3 Reply Message

Reply messages are sent in response to **Request** messages if and only if the response expected flag in the request is set to **TRUE**. Replies include inout and out parameters, operation results, and may include exception values. In addition, Reply messages may provide object location information. In GIOP versions 1.0 and 1.1, replies flow only from server to client.

Reply messages have three elements, encoded in this order:

- A GIOP message header
- A ReplyHeader structure
- The reply body

15.4.3.1 Reply Header

The reply header is defined as follows:

```
module GIOP {                               // IDL extended for 1.2

#ifndef GIOP_1_2
    // GIOP 1.0 and 1.1
    enum ReplyStatusType_1_0 { // Renamed from ReplyStatusType
        NO_EXCEPTION,
        USER_EXCEPTION,
        SYSTEM_EXCEPTION,
        LOCATION_FORWARD
    };

    // GIOP 1.0
    struct ReplyHeader_1_0 { // Renamed from ReplyHeader
        IOP::ServiceContextList     service_context;
        unsigned long               request_id;
        ReplyStatusType_1_0         reply_status;
```

```
    };

    // GIOP 1.1
    typedef ReplyHeader_1_0 ReplyHeader_1_1;
    // Same Header contents for 1.0 and 1.1

#else
    // GIOP 1.2
    enum ReplyStatusType_1_2 {
        NO_EXCEPTION,
        USER_EXCEPTION,
        SYSTEM_EXCEPTION,
        LOCATION_FORWARD,
        LOCATION_FORWARD_PERM,// new value for 1.2
        NEEDS_ADDRESSING_MODE // new value for 1.2
    };

    struct ReplyHeader_1_2 {
        unsigned long                   request_id;
        ReplyStatusType_1_2             reply_status;
        IOP:ServiceContextList          service_context;
    };
#endif // GIOP_1_2
};
```

The members have the following definitions:

- **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
  - directs the client to reissue the request to an object at some other location, or
  - directs the client to supply more addressing information.
- **service_context** contains ORB service data being passed from the server to the client, encoded as described in Section 15.2.3, “GIOP Message Transfer,” on page 15-4.

15.4.3.2 Reply Body

In GIOP version 1.0 and 1.1, reply bodies are marshaled into the CDR encapsulation of the containing Message immediately following the Reply Header. In GIOP version 1.2, the Reply Body is always aligned on an 8-octet boundary. The fact that GIOP specifies the maximum alignment for any primitive type is 8 guarantees that the ReplyBody will not require remarshaling if the Message or the Reply Header are modified. The data for the reply body is determined by the value of **reply_status**. There are the following 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 Section 15.3.5.5, "Exception," on page 15-28. (Only the user-defined exceptions listed in the operation's OMG IDL definition may be raised.)

  When a GIOP Reply message contains a `**reply_status**' value of SYSTEM_EXCEPTION, the body of the Reply message conforms to the following structure:

```
module GIOP {                                   // IDL
    struct SystemExceptionReplyBody {
        string          exception_id;
         unsigned long  minor_code_value;
        unsigned long   completion_status;
        };
};
```

  The high-order 20 bits of **minor_code_value** contain a 20-bit "Vendor Minor Codeset ID"(**VMCID**); the low-order 12 bits contain a minor code. A vendor (or group of vendors) wishing to define a specific set of system exception minor codes should obtain a unique **VMCID** from the OMG, and then define up to 4096 minor codes for each system exception. Any vendor may use the special **VMCID** of zero (0) without previous reservation, but minor code assignments in this codeset may conflict with other vendor's assignments, and use of the zero **VMCID** is officially deprecated.

Note – OMG standard minor codes are identified with the 20 bit **VMCID \x4f4d0**. This appears as the characters 'O' followed by the character 'M' on the wire, which is defined as a 32-bit constant called **OMGVMCID \x4f4d0000** (see Section 3.17.2, "Standard Minor Exception Codes," on page 3-58) so that allocated minor code numbers can be or-ed with it to obtain the **minor_code_value**.

- If the **reply_status** value is **LOCATION_FORWARD**, then the body contains an object reference (IOR) encoded as described in "Object References" on page 15-28. 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.
- The usage of the **reply_status** value **LOCATION_FORWARD_PERM** behaves like the usage of **LOCATION_FORWARD**, but when used by a server it also provides an indication to the client that it may replace the old IOR with the new IOR. Both the old IOR and the new IOR are valid, but the new IOR is preferred for future use.

- If the **reply_status** value is **NEEDS_ADDRESSING_MODE** then the body contains a **GIOP::AddressingDisposition**. The client ORB is responsible for re-sending the original request using the requested addressing mode. The 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 15-35 would be equivalent to the following structure:

```
struct example_reply {
    double          return_value;       // return value
    string          str;
    long            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 Section 15.6, "Object Location," on page 15-46.

15.4.4 CancelRequest Message

CancelRequest messages may be sent, in GIOP versions 1.0 and 1.1, only 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**

15.4.4.1 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.

15.4.5 LocateRequest Message

LocateRequest messages may be sent from a client to a server 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, and if not,
- 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**

15.4.5.1 LocateRequest Header.

The **LocateRequest** header is defined as follows:

```
module GIOP {                                   // IDL extended for version 1.2

    // GIOP 1.0
        struct LocateRequestHeader_1_0 {
                                                // Renamed LocationRequestHeader
            unsigned long           request_id;
            sequence <octet>        object_key;
        };

    // GIOP 1.1
        typedef LocateRequestHeader_1_0 LocateRequestHeader_1_1;
        // Same Header contents for 1.0 and 1.1

    // GIOP 1.2
        struct LocateRequestHeader_1_2 {
            unsigned long           request_id;
            TargetAddress           target;
        };
};
```

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 Section 15.4.2, “Request Message,” on page 15-32 for the applicable rules.
- For GIOP 1.0 and 1.1, **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.
- For GIOP 1.2, target identifies the object being located. The possible values of this union are:
  - **KeyAddr** 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.
  - **ProfileAddr** is the transport-specific GIOP profile selected for the target’s IOR by the client ORB.
  - **IORAddressingInfo** is the full IOR of the target object. The **selected_profile_index** indicates the transport-specific GIOP profile that was selected by the client ORB.

See Section 15.6, “Object Location,” on page 15-46 for details on the use of **LocateRequest.**

15.4.6 LocateReply Message

LocateReply messages are sent from servers to clients in response to **LocateRequest** messages. In GIOP versions 1.0 and 1.1 the **LocateReply** message is only sent from the server to the client.

A **LocateReply** message has three elements, encoded in this order:

1. A GIOP message header
2. A **LocateReplyHeader**
3. The locate reply body

15.4.6.1 Locate Reply Header

The locate reply header is defined as follows:

```
module GIOP {                          // IDL extended for GIOP 1.2
#ifndef GIOP_1_2
    // GIOP 1.0 and 1.1
    enum LocateStatusType_1_0 {// Renamed from LocateStatusType
        UNKNOWN_OBJECT,
        OBJECT_HERE,
        OBJECT_FORWARD
    };
```

```
    // GIOP 1.0
    struct LocateReplyHeader_1_0 {// Renamed from LocateReplyHeader
        unsigned long           request_id;
        LocateStatusType_1_0    locate_status;
    };

    // GIOP 1.1
    typedef LocateReplyHeader_1_0 LocateReplyHeader_1_1;
    // same Header contents for 1.0 and 1.1

#else
    // GIOP 1.2
    enum LocateStatusType_1_2 {
        UNKNOWN_OBJECT,
        OBJECT_HERE,
        OBJECT_FORWARD,
        OBJECT_FORWARD_PERM,        // new value for GIOP 1.2
        LOC_SYSTEM_EXCEPTION,       // new value for GIOP 1.2
        LOC_NEEDS_ADDRESSING_MODE   // new value for GIOP 1.2
    };

    struct LocateReplyHeader_1_2 {
        unsigned long           request_id;
        LocateStatusType_1_2    locate_status;
    };
#endif // GIOP_1_2
};
```

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** and **OBJECT_FORWARD_PERM -** a **LocateReply** body exists.
  - **LOC_SYSTEM_EXCEPTION** - a **LocateReply** body exists.
  - **LOC_NEEDS_ADDRESSING_MODE** - a **LocateReply** body exists.

15.4.6.2 LocateReply Body

The body is empty, except for the following cases:

- If the **LocateStatus** value is **OBJECT_FORWARD** or **OBJECT_FORWARD_PERM**, the body contains an object reference (IOR) that may be used as the target for requests to the object specified in the

LocateRequest message. The usage of **OBJECT_FORWARD_PERM** behaves like the usage of **OBJECT_FORWARD**, but when used by the server it also provides an indication to the client that it may replace the old IOR with the new IOR. When using **OBJECT_FORWARD_PERM**, both the old IOR and the new IOR are valid, but the new IOR is preferred for future use.

- If the **LocateStatus** value is **LOC_SYSTEM_EXCEPTION**, the body contains a marshaled **GIOP::SystemExceptionReplyBody**.
- If the **LocateStatus** value is **LOC_NEEDS_ADDRESSING_MODE** then the body contains a **GIOP::AddressingDisposition**. The client ORB is responsible for re-sending the **LocateRequest** using the requested addressing mode.

15.4.7 CloseConnection Message

CloseConnection messages are sent only by servers in GIOP protocol versions 1.0 and 1.1. 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 are awaiting replies will never be processed, and may safely be reissued (on another connection). In GIOP version 1.2 both sides of the connection may send the **CloseConnection** message.

The **CloseConnection** message consists only of the GIOP message header, identifying the message type.

For details on the usage of **CloseConnection** messages, see Section 15.5.1, “Connection Management,” on page 15-44.

15.4.8 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 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.

15.4.9 Fragment Message

This message is added in GIOP 1.1.

The **Fragment** message is sent following a previous request or response message that has the more fragments bit set to **TRUE** in the **flags** field.

All of the GIOP messages begin with a GIOP header. One of the fields of this header is the **message_size** field, a 32-bit unsigned number giving the number of bytes in the message following the header. Unfortunately, when actually constructing a GIOP **Request** or **Reply** message, it is sometimes impractical or undesirable to ascertain the

total size of the message at the stage of message construction where the message header has to be written. GIOP 1.1 provides an alternative indication of the size of the message, for use in those cases.

In GIOP 1.1, a **Request** or **Reply** message can be broken into multiple fragments. In GIOP 1.2, a **Request**, **Reply**, **LocateRequest**, or **LocateReply** message can be broken into multiple fragment. The first fragment is a regular message (e.g., **Request** or **Reply**) with the **more** fragments bit in the **flags** field set to **TRUE**. This initial fragment can be followed by one or more messages using the fragment messages. The last fragment shall have the more fragment bit in the flag field set to **FALSE**.

A **CancelRequest** message may be sent by the client before the final fragment of the message being sent. In this case, the server should assume no more fragments will follow.

Note – A GIOP client which fragments the header of a **Request** message before sending the request ID, may not send a **CancelRequest** message pertaining to that request ID and may not send another **Request** message until after the request ID is sent.

A primitive data type of 8 bytes or smaller should never be broken across two fragments.

For GIOP version 1.2, the total length (including the message header) of a fragment other than the final fragment of a fragmented message are required to be a multiple of 8 bytes in length, allowing bridges to defragment and/or refragment messages without having to remarshal the encoded data to insert or remove padding.

For GIOP version 1.2, a fragment header is included in the message, immediately after the GIOP message header and before the fragment data. The request ID, in the fragment header, has the same value as that used in the original message associated with the fragment.

```
module GIOP {//IDL extension for GIOP 1.2
        // GIOP 1.2
        struct FragmentHeader_1_2 {
            unsigned long request_id;
        };
};
```

15.5 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. (See Section 15.5.1, "Connection Management," on page 15-44 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.

15.5.1 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.

In GIOP protocol versions 1.0 and 1.1, connections are not symmetrical. Only clients can send **Request**, **LocateRequest**, and **CancelRequest** messages over a connection, in GIOP 1.0 and 1.1. In all GIOP versions, a server can send **Reply**, **LocateReply**, and **CloseConnection** messages over a connection; however, in GIOP 1.2 the client can send them as well. Either client or server can send **MessageError** messages, in GIOP 1.0 and 1.1.

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.

15.5.1.1 Connection Closure

Connections can be closed in two ways: orderly shutdown, or abortive disconnect.

For GIOP versions 1.0, and 1.1:

- Orderly shutdown is initiated by servers sending a **CloseConnection** message, or by clients just closing down a connection.
- Orderly shutdown may be initiated by the client at any time.
- 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 detects connection closure without receiving a **CloseConnection** message, it must assume an abortive disconnect has occurred, and treat the condition as an error.

For GIOP Version 1.2:

- Orderly shutdown is initiated by either the originating client ORB (connection initiator) or by the server ORB (connection responder) sending a **CloseConnection** message
- If the ORB sending the **CloseConnection** is a server, or bidirectional GIOP is in use, the sending ORB must not currently be processing any Requests from the other side.
- The ORB which sends the **CloseConnection** must not send any messages after the **CloseConnection**.
- If either ORB detects connection closure without receiving a **CloseConnection** message, it must assume an abortive disconnect has occurred, and treat the condition as an error.
- If bidirectional GIOP is in use, the conditions of Section 15.8, “Bi-Directional GIOP,” on page 15-52 apply.

For all uses of **CloseConnection** (for GIOP versions 1.0, 1.1, and 1.2):

- If there are any pending non-oneway requests which were initiated on a connection by the ORB shutting down that connection, the connection-peer ORB should consider them as canceled.
- If an ORB receives a **CloseConnection** message from its connection-peer ORB, it should assume that any outstanding messages (i.e., without replies) were received after the connection-peer ORB sent the CloseConnection message, were not processed, and may be safely resent on a new connection.
- After issuing a **CloseConnection** message, the issuing ORB 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 as part of the mapping to that protocol's connection mechanisms, to guarantee that both ends of the connection understand the disposition of any outstanding GIOP requests.

15.5.1.2 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.

15.5.2 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.

15.6 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 **LocateRequest** 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 be prepared to accept **LocateReply** messages.

A client shall not make any assumptions about the longevity of object addresses returned by **LOCATION_FORWARD** (**OBJECT_FORWARD**) mechanisms. Once a connection based on location-forwarding information is closed, a client can attempt to reuse the forwarding information it has, but, if that fails, it shall restart the location process using the original address specified in the initial object reference.

For GIOP version 1.2, the usage of **LOCATION_FORWARD_PERM** (**OBJECT_FORWARD_PERM**) behaves like the usage of **LOCATION_FORWARD** (**OBJECT_FORWARD**), but when used by the server it also provides an indication to the client that it may replace the old IOR with the new IOR. When using **LOCATION_FORWARD_PERM** (**OBJECT_FORWARD_PERM**), both the old IOR and the new IOR are valid, but the new IOR is preferred for future use.

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.

15.7 Internet Inter-ORB Protocol (IIOP)

The baseline transport specified for GIOP is TCP/IP[4]. 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).

IIOP 1.0 is based on GIOP 1.0.

IIOP 1.1 can be based on either GIOP 1.0 or 1.1. An IIOP 1.1 client must support GIOP 1.1, and may also support GIOP 1.0. An IIOP 1.1 server must support processing both GIOP 1.0 and GIOP 1.1 messages.

IIOP 1.2 can be based on any of the GIOP minor versions 1.0, 1.1, or 1.2. An IIOP 1.2 client must support GIOP 1.2, and may also support lesser GIOP minor versions. An IIOP 1.2 server must also support processing messages with all lesser GIOP versions.

15.7.1 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 15-49. 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 an 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. In GIOP 1.2, the client may send the **CloseConnection** message, and if BiDirectional GIOP is in use, the client may also send **Reply** and **LocateReply** messages.

After receiving a **CloseConnection** message, an ORB must close the TCP/IP connection. After sending a **CloseConnection**, an ORB may close the TCP/IP connection immediately, or may delay closing the connection until it receives an indication that the other side has closed the connection. For maximum interoperability with ORBs using TCP implementations which do not properly implement orderly shutdown, an ORB may wish to only shutdown the sending side of the connection, and then read any incoming data until it receives an indication that the other side has also shutdown, at which point the TCP connection can be closed completely.

---

4. Postel, J., “Transmission Control Protocol – DARPA Internet Program Protocol Specification,” RFC-793, Information Sciences Institute, September 1981

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.

15.7.2 IIOP IOR Profiles

IIOP profiles, identifying individual objects accessible through the Internet Inter-ORB Protocol, have the following form:

```
module IIOP {  // IDL extended for version 1.1 and 1.2
        struct Version {
            octet           major;
            octet           minor;
        };

        struct ProfileBody_1_0 {// renamed from ProfileBody
            Version             iiop_version;
            string              host;
            unsigned short      port;
            sequence <octet>    object_key;
        };

        struct ProfileBody_1_1 {// also used for 1.2
            Version             iiop_version;
            string              host;
            unsigned short      port;
            sequence <octet>    object_key;

        // Added in 1.1 unchanged for 1.2
            sequence <IOP::TaggedComponent> components;
        };
};
```

IIOP Profile version number:

- Indicates the IIOP protocol version.
- Major number can stay the same if the new changes are backward compatible.
- Clients with lower minor version can attempt to invoke objects with higher minor version number by using only the information defined in the lower minor version protocol (ignore the extra information).

Profiles supporting only IIOP version 1.0 use the **ProfileBody_1_0** structure, while those supporting IIOP version 1.1 or 1.2 use the **ProfileBody_1_1** structure. An instance of one of these structure types is marshaled into an encapsulation octet stream.

This encapsulation (a **sequence <octet>**) becomes the **profile_data** member of the **IOP::TaggedProfile** structure representing the IIOP profile in an IOR, and the tag has the value **TAG_INTERNET_IOP** (as defined earlier).

The version number published in the Tag Internet IIOP Profile body signals the highest GIOP minor version number that the server supports at the time of publication of the IOR.

If the major revision number is 1, and the minor revision number is greater than 0, then the length of the encapsulated profile may exceed the total size of components defined in this specification for profiles with minor revision number 0. ORBs that support only revision 1.0 IIOP profiles must ignore any data in the profile that occurs after the **object_key**. If the revision of the profile is 1.0, there shall be no extra data in the profile (i.e., the length of the encapsulated profile must agree with the total size of components defined for version 1.0).

For Version 1.2 of IIOP, no order of use is prescribed in the case where more than one TAG Internet IOP Profile is present in an IOR.

The members of **IIOP::ProfileBody_1_0 and IOP::ProfileBody_1_1** 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 1; the minor versions defined to date are 0, 1, and 2. Compliant ORBs must generate version 1.1 profiles, and must accept any profile with a major version of 1, regardless of the minor version number. If the minor version number is 0, the encapsulation is fully described by the **ProfileBody_1_0** structure. If the minor version number is 1 or 2, the encapsulation is fully described by the **ProfileBody_1_1** structure. If the minor version number is greater than 1, then the length of the encapsulated profile may exceed the total size of components defined in this specification for profiles with minor version number 1. ORBs that support only version 1.1 or 1.2 IIOP profiles must ignore, but preserve, any data in the profile that occurs after the **components** member.

Note – As of version 1.2 of GIOP and IIOP and minor versions beyond, the minor version in the **TAG_INTERNET_IOP** profile signals the highest minor revision of GIOP supported by the server at the time of publication of the IOR.

- **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.
- **components** is a sequence of **TaggedComponent**, which contains additional information that may be used in making invocations on the object described by this profile. **TaggedComponent**s that apply to IIOP 1.2 are described below in "IIOP IOR Profile Components" on page 15-51. Other components may be included to support enhanced versions of IIOP, to support ORB services such as security, and to support other GIOPs, ESIOPs, and proprietary protocols. If an implementation puts a non-standard component in an IOR, it cannot be assured that any or all non-standard components will remain in the IOR.

  The relationship between the IIOP protocol version and component support conformance requirements is as follows:
  - Each IIOP version specifies a set of standard components and the conformance rules for that version. These rules specify which components are mandatory presence, which are optional presence, and which can be dropped. A conformant implementation has to conform to these rules, and is not required to conform to more than these rules.
  - New components can be added, but they do not become part of the versions conformance rules.
  - When there is a need to specify conformance rules which include the new components, there will be a need to create a new IIOP version.

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; therefore, individual agents will need to assign previously unused ports as part of their installation procedures. IIOP supports such multiple agents per host.

15.7.3 IIOP IOR Profile Components

The following components are part of the IIOP 1.1 and 1.2 conformance. All these components are optional presence in the IIOP profile and cannot be dropped from an IIOP 1.1 or 1.2 IOR.

- **TAG_ORB_TYPE**
- **TAG_CODE_SETS**
- **TAG_SEC_NAME**
- **TAG_ASSOCIATION_OPTIONS**
- **TAG_GENERIC_SEC_MECH**
- **TAG_SSL_SEC_TRANS**
- **TAG_SPKM_1_SEC_MECH**

- **TAG_SPKM_2_SEC_MECH**
- **TAG_KerberosV5_SEC_MECH**
- **TAG_CSI_ECMA_Secret_SEC_MECH**
- **TAG_CSI_ECMA_Hybrid_SEC_MECH**
- **TAG_CSI_ECMA_Public_SEC_MECH**
- **TAG_INTERNET_IOP**
- **TAG_MULTIPLE_COMPONENTS**
- **TAG_JAVA_CODEBASE**

The following components are part of the IIOP 1.2 conformance. All these components are optional presence in the IIOP profile and cannot be dropped from an IIOP 1.2 IOR.

- **TAG_ALTERNATE_IIOP_ADDRESS**
- **TAG_POLICIES**
- **TAG_DCE_STRING_BINDING**
- **TAG_DCE_BINDING_NAME**
- **TAG_DCE_NO_PIPES**
- **TAG_DCE_MECH**
- **TAG_COMPLETE_OBJECT_KEY**
- **TAG_ENDPOINT_ID_POSITION**
- **TAG_LOCATION_POLICY**

15.8 Bi-Directional GIOP

The specification of GIOP connection management, in GIOP minor versions 1.0 and 1.1, states that connections are not symmetrical. For example, only clients that initialize connections can send requests, and only servers that accept connections can receive them.

This GIOP 1.0 and 1.1 restriction gives rise to significant difficulties when operating across firewalls. It is common for firewalls not to allow incoming connections, except to certain well-known and carefully configured hosts, such as dedicated HTTP or FTP servers. For most CORBA-over-the-internet applications it is not practicable to require that all potential client firewalls install GIOP proxies to allow incoming connections, or that any entities receiving callbacks will require prior configuration of the firewall proxy.

An applet, for example, downloaded to a host inside such a firewall will be restricted in that it cannot receive requests from outside the firewall on any object it creates, as no host outside the firewall will be able to connect to the applet through the client's firewall, even though the applet in question would typically only expect callbacks from the server it initially registered with.

In order to circumvent this unnecessary restriction, GIOP minor protocol version 1.2 specifies that the asymmetry stipulation above be relaxed in cases where the client and the server agree on it. In these cases, the client (the applet in the above case) would still initiate the connection to the server, but any requests from the server on any objects exported by the client to the server via this connection will be sent back to the client on this same connection.

The mechanism by which the client and server agree on this capability is as follows:

The client creates an object for exporting to a server.

The client exports the IOR as a parameter of a GIOP Request on the server object. If the ORB policy permits bi-directional use of a connection, a **Request** message should contain an **IOP::ServiceContext** structure in its **Request** header, which indicates that this GIOP connection is bi-directional. The service context may provide additional information that the server may need to invoke the callback object. To determine whether an ORB may support bi-directional GIOP a new POA policy has been defined (Section 15.9, “Bi-directional GIOP policy,” on page 15-55).

Each mapping of GIOP to a particular transport should define a transport-specific bi-directional service context, and have an **IOP::ServiceId** allocated by the OMG. It is recommended that names for this service context follows the pattern *BiDir<protocolname>*, where <protocol name> identifies a mapping of GIOP to a transport protocol (e.g., for IIOP the name is **BiDirIIOP**). The service context for bi-directional IIOP is defined in Section 15.8.1, “Bi-Directional IIOP,” on page 15-54.

The server receives the **Request**. If it recognizes the service context and supports bi-directional connections, it may send invocations on this object back along the connection.

The server may not wish to support bi-directionality either due to lack of support for it, or because it has been configured that way. In this case, it may fall back to initiating a connection to the object in the usual way.

If a GIOP connection is used bi-directionally, the client should attempt to keep the connection alive as long as is necessary to complete its object's service to the server. If the client initiates a new connection it is not foreseen here that the server can use that connection for requests on the object exported previously.

A server talking to a client on a bi-directional GIOP connection can use any message type traditionally used by clients only, so it can use **Request**, **LocateRequest**, **CancelRequest**, **MessageError**, and **Fragment** (for GIOP 1.1). Similarly the client can use message types traditionally used only by servers: **Reply**, **LocateReply**, **MessageError**, **CloseConnection**, and **Fragment**.

CloseConnection messages are a special case however. Either ORB may send a **CloseConnection** message, but the conditions in Section 15.5.1, “Connection Management,” on page 15-44 apply.

Bi-directional GIOP connections modify the behavior of Request IDs. In the GIOP specification, Section 15.5.1, “Connection Management,” on page 15-44, it is noted that “Request IDs must unambiguously associate replies with requests within the scope and lifetime of a connection”. With bi-directional IIOP, the Request ID unambiguously

associates replies with requests per connection *and per direction*, so the same Request ID can be used for a **Request** going from client-to-server and for a **Request** going from server-to-client, simultaneously.

It should be noted that a single-threaded ORB needs to perform event checking on the connection, in case a **Request** from the other endpoint arrives in the window between it sending its own **Request** and receiving the corresponding reply; otherwise a client and server could send **Request**s simultaneously, resulting in deadlock. If the client cannot support event checking, it must not indicate that bi-directionality is supported. If the server cannot support event checking, it must not make callbacks along the same connection even if the connection indicates it is supported.

A server making a callback to a client cannot specify its own bi-directional service context – only the client can announce the connection's bi-directionality.

It is possible for a client to masquerade for a callback object, by pretending that a callback object can be reached over an existing connection to the client. If the server has doubts in the integrity of the client, it is recommended that bi-directional GIOP not be used.

15.8.1 Bi-Directional IIOP

The **IOP::ServiceContext** used to support bi-directional IIOP contains a **BiDirIIOPServiceContext** structure as defined below:

```
// IDL
module IIOP {

    struct ListenPoint {
        string   host;
        unsigned short port;
    };

    typedef sequence<ListenPoint> ListenPointList;

    struct BiDirIIOPServiceContext {
        ListenPointList listen_points;
    };
};
```

The data encapsulated in the **BiDirIIOPServiceContext** structure which is identified by the ServiceId **BI_DIR_IIOP** as defined in Section 13.6.7, “Object Service Context,” on page 13-22, allows the ORB, which intends to open a new connection in order to invoke on an object, to look up its list of active client-initiated connections and examine the structures associated with them, if any. If a **host** and **port** pair in a **listen_points** list matches a host and port which the ORB intends to open a connection to, rather than open a new connection to that **listen_point**, the server may re-use any of the connections that were initiated by the client on which the listen point data was received.

The **host** element of the structure should contain whatever values the client may use in the IORs it creates. The rules for **host** and **port** are identical to the rules for the IIOP IOR **ProfileBody_1_1 host** and **port** elements; see Section 15.7.2, “IIOP IOR Profiles,” on page 15-49. Note that if the server wishes to make a callback connection to the client in the standard way, it must use the values from the client object's IOR, not the values from this **BiDirIIOPServiceContext** structure; these values are only to be used for bi-directional GIOP support.

The **BI_DIR_IIOP** service context may be sent by a client at any point in a connection's lifetime. The **listen_points** specified therein must supplement any **listen_points** already sent on the connection, rather than replacing the existing points. Typically, when the same client has multiple connections to the same server, the **listen_points** will be identical. However, if they differ, they supplement each other (i.e., any of the listen points received on any of the connections may be used).

If a client supports a secure connection mechanism, such as SECIOP or IIOP/SSL, and sends a **BI_DIR_IIOP** service context over an insecure connection, the **host** and **port** endpoints listed in the **BI_DIR_IIOP** should not contain the details of the secure connection mechanism if insecure callbacks to the client's secure objects would be a violation of the client's security policy.

If a client has not set up any mechanism for traditional-style callbacks using a listening socket, then the **port** entry in its IOR must be set to the outgoing connection's local port (as retrieved using the getsockname() sockets API call). The **port** in the **BI_DIR_IIOP** structure must match this value. This will allow multiple clients, all running in restrictive security modes (such as Java applets) on the same host, all of them connecting to one server, to each receive callbacks on their correct connection.

15.8.1.1 IIOP/SSL considerations

Bi-directional IIOP can operate over IIOP/SSL (*see CORBAservices Chapter 15*) without defining any additions to the IIOP/SSL or the bi-directional GIOP mechanisms. However, if the client wants to authenticate the server when the client receives a callback this cannot be done unless the client has already authenticated the server. This has to be performed during the initial SSL handshake. It is not possible to do this at any point after the initial handshake without establishing a new SSL connection (which defeats the purpose of the bi-directional connections).

15.9 Bi-directional GIOP policy

In GIOP protocol versions 1.0 and 1.1, there are strict rules on which side of a connection can issue what type of messages (for example version 1.0 and 1.1 clients can not issue GIOP reply messages). However, as documented above, it is sensible to relax this restriction if the ORB supports this functionality and policies dictate that bi-directional connection are allowed. To indicate a bi-directional policy, the following is defined.

```
// Self contained module for Bi-directional GIOP policy

module BiDirPolicy {

    typedef unsigned short BidirectionalPolicyValue;
    const BidirectionalPolicyValue NORMAL = 0;
    const BidirectionalPolicyValue BOTH = 1;

    const CORBA::PolicyType BIDIRECTIONAL_POLICY_TYPE = 37;

    interface BidirectionalPolicy : CORBA::Policy {
        readonly attribute BidirectionalPolicyValue value;
    };
};
```

A **BidirectionalPolicyValue** of **NORMAL** states that the usual GIOP restrictions of who can send what GIOP messages apply (i.e., bi-directional connections are not allowed). A value of **BOTH** indicates that there is a relaxation in what party can issue what GIOP messages (i.e., bi-directional connections are supported). The default value of a **BidirectionalPolicy** is **NORMAL**.

In the absence of a **BidirectionalPolicy** being passed in the **PortableServer::POA::create_POA** operation, a **POA** will assume a policy value of **NORMAL**.

A client and a server **ORB** must each have a **BidirectionalPolicy** with a value of **BOTH** for bi-directional communication to take place.

To create a **BidirectionalPolicy**, the **ORB::create_policy** operation is used.

15.10 OMG IDL

This section contains the OMG IDL for the GIOP and IIOP modules.

15.10.1 GIOP Module

```
module GIOP {      // IDL extended for version 1.1 and 1.2

    struct Version {
        octet       major;
        octet       minor;
    };

#ifndef GIOP_1_1
    // GIOP 1.0
    enum MsgType_1_0{    // rename from MsgType
        Request, Reply, CancelRequest,
        LocateRequest, LocateReply,
            CloseConnection, MessageError
        };
```

```
#else
    // GIOP 1.1
    enum MsgType_1_1{
        Request, Reply, CancelRequest,
        LocateRequest, LocateReply,
        CloseConnection, MessageError,
        Fragment   // GIOP 1.1 addition
    };
#endif

    // GIOP 1.0
    struct MessageHeader_1_0 {// Renamed from MessageHeader
        char                    magic [4];
        Version                 GIOP_version;
        boolean                 byte_order;
        octet                   message_type;
        unsigned long           message_size;
    };

    // GIOP 1.1
    struct MessageHeader_1_1 {
        char                    magic [4];
        Version                 GIOP_version;
        octet                   flags;          // GIOP 1.1 change
        octet                   message_type;
        unsigned long           message_size;
    };

    // GIOP 1.2
    typedef MessageHeader_1_1 MessageHeader_1_2;

    // GIOP 1.0
    struct RequestHeader _1_0 {
        IOP::ServiceContextList service_context;
        unsigned long           request_id;
        boolean                 response_expected;
        sequence <octet>        object_key;
        string                  operation;
        Principal               requesting_principal;
    };

    // GIOP 1.1
    struct RequestHeader_1_1 {
        IOP::ServiceContextList service_context;
        unsigned long           request_id;
        boolean                 response_expected;
        octet                   reserved[3]; // Added in GIOP 1.1
        sequence <octet>        object_key;
        string                  operation;
        Principal               requesting_principal;
    };
```

```
    // GIOP 1.2
    typedef short               AddressingDisposition;
    const short                 KeyAddr = 0;
    const short                 ProfileAddr = 1;
    const short                 ReferenceAddr = 2;

    struct IORAddressingInfo {
        unsigned long           selected_profile_index;
        IOP::IOR                ior;
    };

    union TargetAddress switch (AddressingDisposition) {
        case KeyAddr:           sequence <octet> object_key;
        case ProfileAddr:       IOP::TaggedProfile profile;
        case ReferenceAddr:     IORAddressingInfo ior;
    };

    struct RequestHeader_1_2 {
        unsigned long           request_id;
        octet                   response_flags;
        octet                   reserved[3];
        TargetAddress           target;
        string                  operation;
        // Principal not in GIOP 1.2
        IOP::ServiceContextList service_context;    // 1.2 change
    };

#ifndef GIOP_1_2
    // GIOP 1.0 and 1.1
    enum ReplyStatusType_1_0 {// Renamed from ReplyStatusType
        NO_EXCEPTION,
        USER_EXCEPTION,
        SYSTEM_EXCEPTION,
        LOCATION_FORWARD
    };

    // GIOP 1.0
    struct ReplyHeader_1_0 {// Renamed from ReplyHeader
        IOP::ServiceContextList service_context;
        unsigned long           request_id;
        ReplyStatusType_1_0     reply_status;
    };

    // GIOP 1.1
    typedef ReplyHeader_1_0 ReplyHeader_1_1;
    // Same Header contents for 1.0 and 1.1

#else
    // GIOP 1.2
    enum ReplyStatusType_1_2 {
        NO_EXCEPTION,
```

```
        USER_EXCEPTION,
        SYSTEM_EXCEPTION,
        LOCATION_FORWARD,
        LOCATION_FORWARD_PERM,      // new value for 1.2
        NEEDS_ADDRESSING_MODE       // new value for 1.2
    };

    struct ReplyHeader_1_2 {
        unsigned long               request_id;
        ReplyStatusType_1_2         reply_status;
        IOP:ServiceContextList      service_context;    // 1.2 change
    };

#endif // GIOP_1_2
        struct SystemExceptionReplyBody {
            string exception_id;
            unsigned long minor_code_value;
            unsigned long completion_status;
    };

            struct CancelRequestHeader {
            unsigned long               request_id;
    };

    // GIOP 1.0
        struct LocateRequestHeader_1_0 {
                                    // Renamed LocationRequestHeader
            unsigned long           request_id;
            sequence <octet>        object_key;
    };

    // GIOP 1.1
        typedef LocateRequestHeader_1_0 LocateRequestHeader_1_1;
        // Same Header contents for 1.0 and 1.1

    // GIOP 1.2
        struct LocateRequestHeader_1_2 {
            unsigned long           request_id;
            TargetAddress           target;
    };

#ifndef GIOP_1_2
    // GIOP 1.0 and 1.1
        enum LocateStatusType_1_0 {// Renamed from LocateStatusType
            UNKNOWN_OBJECT,
            OBJECT_HERE,
            OBJECT_FORWARD
    };

    // GIOP 1.0
        struct LocateReplyHeader_1_0 {
```

```
                                                    // Renamed from LocateReplyHeader
            unsigned long               request_id;
            LocateStatusType_1_0        locate_status;
        };

        // GIOP 1.1
        typedef LocateReplyHeader_1_0 LocateReplyHeader_1_1;
                        // same Header contents for 1.0 and 1.1

    #else
        // GIOP 1.2
        enum LocateStatusType_1_2 {
            UNKNOWN_OBJECT,
            OBJECT_HERE,
            OBJECT_FORWARD,
            OBJECT_FORWARD_PERM,        // new value for GIOP 1.2
            LOC_SYSTEM_EXCEPTION,       // new value for GIOP 1.2
            LOC_NEEDS_ADDRESSING_MODE   // new value for GIOP 1.2
        };

        struct LocateReplyHeader_1_2 {
            unsigned long               request_id;
            LocateStatusType_1_2        locate_status;
        };
        #endif // GIOP_1_2

        // GIOP 1.2
        struct FragmentHeader_1_2 {
            unsigned long               request_id;
        };
    };
```

15.10.2 IIOP Module

```
        module IIOP {  // IDL extended for version 1.1 and 1.2
        struct Version {
            octet           major;
            octet           minor;
        };

        struct ProfileBody_1_0 {// renamed from ProfileBody
            Version             iiop_version;
            string              host;
            unsigned short      port;
            sequence <octet>    object_key;
        };

        struct ProfileBody_1_1 {// also used for 1.2
            Version             iiop_version;
            string              host;
```

```
        unsigned short      port;
        sequence <octet>  object_key;

    // Added in 1.1 unchanged for 1.2
        sequence <IOP::TaggedComponent> components;
    };

    struct ListenPoint {
        string   host;
        unsigned short port;
    };

    typedef sequence<ListenPoint> ListenPointList;

    struct BiDirIIOPServiceContext {// BI_DIR_IIOP Service Context
        ListenPointList listen_points;
    };
};
```

15.10.3 BiDirPolicy Module

```
// Self contained module for Bi-directional GIOP policy

module BiDirPolicy {

    typedef unsigned short BidirectionalPolicyValue;
    const BidirectionalPolicyValue NORMAL = 0;
    const BidirectionalPolicyValue BOTH = 1;

    const CORBA::PolicyType BIDIRECTIONAL_POLICY_TYPE = 37;

    interface BidirectionalPolicy : CORBA::Policy {
        readonly attribute BidirectionalPolicyValue value;
    };
};
```

The DCE ESIOP 16

This chapter specifies an Environment-Specific Inter-ORB Protocol (ESIOP) for the OSF DCE environment, the DCE Common Inter-ORB Protocol (DCE-CIOP).

Contents

This chapter contains the following sections.

16.1 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.

16.2 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 16.7, “OMG IDL for the DCE CIOP Module,” on page 16-25. Fragments are used throughout this chapter as necessary.

16.2.1 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.

---

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 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.

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.

16.2.2 DCE-CIOP Data Representation

DCE-CIOP messages represent OMG IDL types by using the CDR transfer syntax, which is defined in Section 15.2.1, “Common Data Representation (CDR),” on page 15-3. 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 16-1.

Table 16-1 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
wchar	byte, unsigned short, or unsigned long, depending on transmission code set
octet	byte
short	short
unsigned short	unsigned short
long	long
unsigned long	unsigned long

Table 16-1 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
long long	hyper
unsigned long long	unsigned hyper
float	float[1]
double	double[2]
long double	long double[3]
boolean	byte[4]

1. Restricted to IEEE format.

2. Restricted to IEEE format.

3. Restricted to IEEE format.

4. 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.

A wide string is encoded as a unidimensional conformant array of octets in DCE 1.1 NDR. This consists of an unsigned long of four octets, specifying the number of octets in the array, followed by that number of octets, with no null terminator.

The NDR representation of OMG IDL fixed-point type, **fixed**, will be proposed as an addition to the set of DCE IDL types.

As new data types are added to OMG IDL, NDR can be used as a model for their CDR representations.

16.2.3 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.

16.2.4 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 either **TAG_INTERNET_IOP** or **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.
- In the **TAG_MULTIPLE_COMPONENTS** profile, to identify the target object when request messages are sent to the server. In the **TAG_INTENET_IOP** profile, the **object_key** profile member is used instead.
- 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.

16.3 DCE-CIOP Message Transport

DCE-CIOP defines two DCE-RPC interfaces for the transport of messages between client ORBs and server ORBs[3]. 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[4].

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.

16.3.1 Pipe-based Interface

The **dce_ciop_pipe** interface is defined by the DCE IDL specification shown below:

```
[/* DCE IDL */
uuid(d7d99f66-97ee-11cf-b1a0-0800090b5d3e),/* 2nd revision
*/
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.

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

---

3. Previous DCE-CIOP revisions used different DCE RPC interface UUIDs and had incompatible message formats. These previous revisions are deprecated, but implementations can continue to support them in conjunction with the current interface UUIDs and message formats.

4. A future DCE-CIOP revision may eliminate the array-based interface and require support of the pipe-based interface.

perform the RPC. See “DCE-CIOP String Binding Component” on page 16-17, “DCE-CIOP Binding Name Component” on page 16-18, and “DCE-CIOP Object Location” on page 16-21 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 16-1 illustrates the layering of DCE-CIOP messages on the DCE-RPC protocol as NDR pipes:

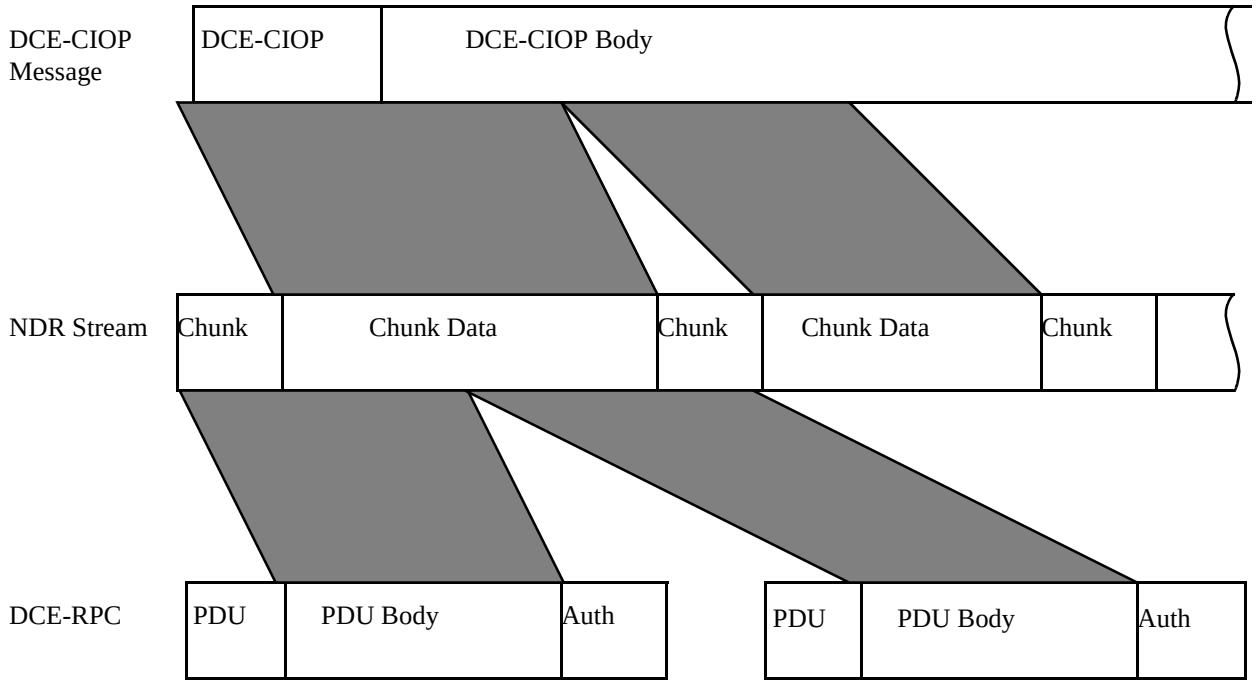


Figure 16-1 Pipe-based Interface Protocol Layering

The DCE-RPC protocol data unit (PDU) bodies, after any appropriate authentication is performed, 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.

16.3.1.1 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 Section 16.4.1, "DCE_CIOP Invoke Request Message," on page 16-11 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 Section 16.4.2, "DCE-CIOP Invoke Response Message," on page 16-12 for a description of the response format.

16.3.1.2 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 Section 16.4.3, "DCE-CIOP Locate Request Message," on page 16-14 and Section 16.4.4, "DCE-CIOP Locate Response Message," on page 16-15 for descriptions of their formats. Use of the **locate RPC** is described in detail in Section 16.6, "DCE-CIOP Object Location," on page 16-21.

16.3.2 Array-based Interface

The **dce_ciop_array** interface is defined by the DCE IDL specification shown below:

```
[ /* DCE IDL */
uuid(09f9ffb8-97ef-11cf-9c96-0800090b5d3e),/* 2nd revision
*/
version(1.0)
]
interface dce_ciop_array
{
    typedef struct {
        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 16-2below:

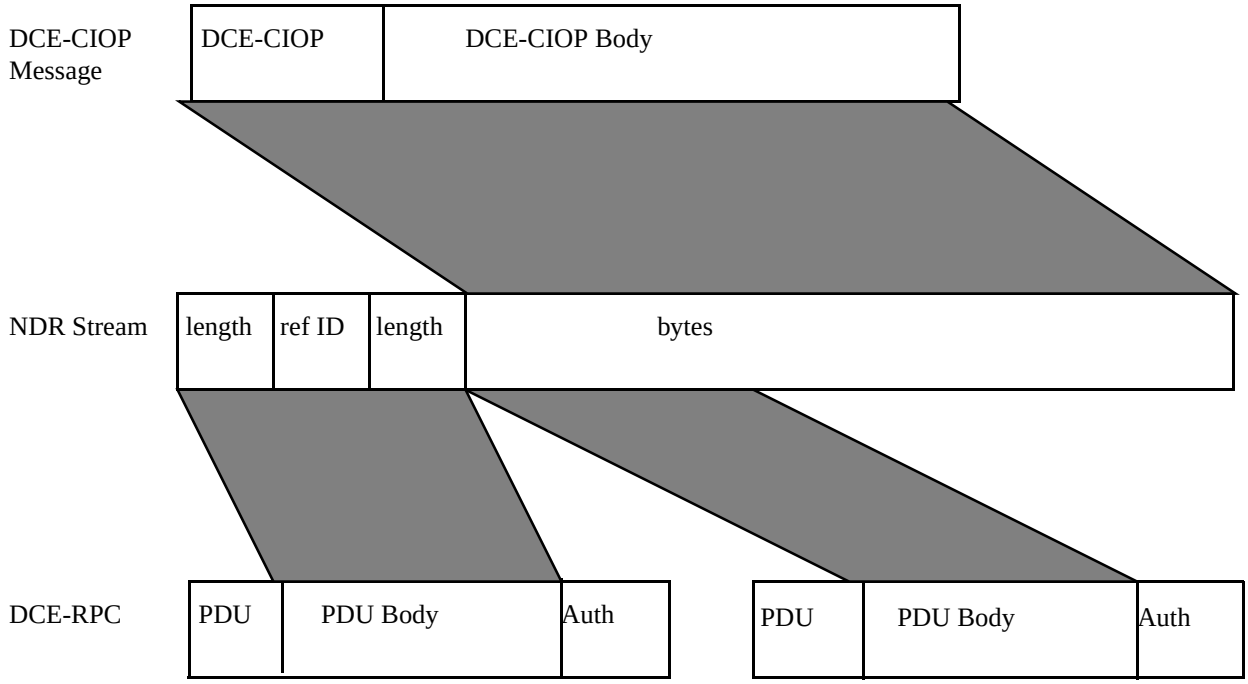


Figure 16-2 Array-based Interface Protocol Layering

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.

16.3.2.1 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.

16.3.2.2 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.

16.4 DCE-CIOP Message Formats

This 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 Section 16.3, "DCE-CIOP Message Transport," on page 16-5.

16.4.1 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.

16.4.1.1 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 operation;
        CORBA::Principal principal;

        // 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[5]. Its value is obtained from the **object_key** field of the **TAG_INTERNET_IOP** profile or the **TAG_COMPLETE_OBJECT_KEY** component of the **TAG_MULTIPLE_COMPONENTS** profile.
- **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.

16.4.1.2 Invoke request body

The invoke request body contains 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 Section 15.3.5.4, "Context," on page 15-28[6]. 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.

16.4.2 DCE-CIOP Invoke Response Message

Invoke response messages are returned from servers to clients as the **`response_message`** parameter of an **`invoke`** RPC.

---

5. Previous revisions of DCE-CIOP included an endpoint_id member, obtained from an optional TAG_ENDPOINT_ID component, as part of the object identity. The endpoint ID, if used, is now contained within the object key, and its position is specified by the optional TAG_ENDPOINT_ID_POSITION component.
6. Previous revisions of DCE-CIOP encoded the Context in the InvokeRequestHeader. It has been moved to the body for consistency with GIOP.

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.

16.4.2.1 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::IOR 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.

16.4.2.2 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 IOR containing a **TAG_INTERNET_IOP** or **TAG_MULTIPLE_COMPONENTS** profile whose components can be used to communicate with the object specified in the invoke request message[7]. This profile must provide at least one new DCE-CIOP binding component. The client ORB is responsible for resending 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 16-21 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[8].

16.4.3 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 Section 16.6, "DCE-CIOP Object Location," on page 16-21.

Locate request messages contain a fixed-format header, but no body.

16.4.3.1 Locate Request Header

DCE-CIOP locate request headers have the following format:

```
module DCE_CIOP {                        // IDL
    struct LocateRequestHeader {
        boolean byte_order;
```

---

7. Previous revisions of DCE-CIOP returned a MultipleComponentProfile structure. An IOR is now returned to allow either a TAG_INTERNET_IOP or a TAG_MULTIPLE_COMPONENTS profile to be used.
8. An exponential back-off algorithm is recommended, but not required.

```
            sequence <octet> object_key;
            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. Its value is obtained from the **object_key** field of the **TAG_INTERNET_IOP** profile or the **TAG_COMPLETE_OBJECT_KEY** component of the **TAG_MULTIPLE_COMPONENTS** profile.
- **operation** contains the name of the CORBA operation being invoked. It is encoded as in the invoke request header.

16.4.4 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.

16.4.4.1 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::IOR 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.

16.4.4.2 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 IOR containing a **TAG_INTERNET_IOP** or **TAG_MULTIPLE_COMPONENTS** profile whose components 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[9].

16.5 DCE-CIOP Object References

The information necessary to invoke operations on objects using DCE-CIOP is encoded in an IOR in a profile identified either by **TAG_INTERNET_IOP** or by **TAG_MULTIPLE_COMPONENTS**. The **profile_data** for the **TAG_INTERNET_IOP** profile is a CDR encapsulation of the **IIOP::ProfileBody_1_1** type, described in Section 15.7.2, “IIOP IOR Profiles,” on page 15-49. The **profile_data** for the **TAG_MULTIPLE_COMPONENTS** profile is a CDR encapsulation of the **MultipleComponentProfile** type, which is a sequence of **TaggedComponent** structures, described in Section 13.6, “An Information Model for Object References,” on page 13-15.

DCE-CIOP defines a number of IOR components that can be included in either profile. Each is identified by a unique tag, and the encoding and semantics of the associated **component_data** are specified.

Either IOR profile 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

9. An exponential back-off algorithm is recommended, but not required.

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_INTERNET_IOP** or by **TAG_MULTIPLE_COMPONENTS** that do not support DCE-CIOP.

16.5.1 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 Section 16.3, "DCE-CIOP Message Transport," on page 16-5.

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.

16.5.2 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.

16.5.2.1 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.

16.5.3 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 16-5, 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.

16.5.4 Complete Object Key Component

An IOR profile supporting DCE-CIOP must include an object key that identifies the object the IOR represents. The object key is an opaque sequence of octets used as the **object_key** member in invoke and locate request message headers. In a **TAG_INTERNET_IOP** profile, the **object_key** member of the **IIOP::ProfileBody_1_1** structure is used. In a **TAG_MULTIPLE_COMPONENTS** profile supporting DCE-CIOP[10], a single **TAG_COMPLETE_OBJECT_KEY** component must be included to identify the object.

The **TAG_COMPLETE_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. This component should never be included in a **TAG_INTERNET_IOP** profile.

```
module IOP {                                                    // IDL
    const ComponentId TAG_COMPLETE_OBJECT_KEY = 5;
};
```

The sequence of octets comprising 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.

10. Previous DCE-CIOP revisions used a different component.

16.5.5 Endpoint ID Position Component

An optional endpoint ID position 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 position component can be included in a profile.

```
module IOP {                                                    // IDL
        const ComponentId TAG_ENDPOINT_ID_POSITION = 6;

        struct EndpointIdPositionComponent {
            unsigned short begin;
            unsigned short end;
        };
};
```

An endpoint ID position component, identified by **TAG_ENDPOINT_ID_POSITION**, indicates the portion of the profile's object key that identifies the endpoint at which operations on an object can be invoked. The **component_data** is a CDR encapsulation of an **EndpointIdPositionComponent** structure. The **begin** member of this structure specifies the index in the object key of the first octet of the endpoint ID. The **end** member specifies the index of the last octet of the endpoint ID. An index value of zero specifies the first octet of the object key. The value of **end** must be greater than the value of **begin**, but less than the total number of octets in the object key. The endpoint ID is made up of the octets located between these two indices inclusively.

The endpoint ID should be unique within the domain of interoperability. A binary or stringified UUID is recommended.

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 16-24.

16.5.6 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;

    // IDL does not support octet constants
    #define LOCATE_NEVER = 0
    #define LOCATE_OBJECT = 1
    #define LOCATE_OPERATION = 2
```

```
        #define 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 16-21 for more details.

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.

16.6 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.

16.6.1 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 Section 16.5.6, "Location Policy Component," on page 16-20. 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.

16.6.2 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.

16.6.3 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_INTERNET_IOP** or **TAG_MULTIPLE_COMPONENTS** from the IOR. Make this the *original* profile and the *current* profile. If no profiles with either tag are available, operations cannot be invoked using DCE-CIOP with this IOR.

2. Get a binding handle to try from the *current* profile. See Section 16.5.1, "DCE-CIOP String Binding Component," on page 16-17 and Section 16.5.2, "DCE-CIOP Binding Name Component," on page 16-18. 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 object key from the *current* 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 IOR profile returned in the response message body the *current* profile and go to step 2.
   - If the RPC returns **LOCATE_UNKNOWN_OBJECT**, and the *original* profile was used, 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 profile 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.

16.6.4 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 position components can be used by the client ORB, if present in the IOR profile, to optimize this algorithm.

16.6.4.1 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.

16.6.4.2 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.

16.6.4.3 Original Endpoint ID

If a component with **TAG_ENDPOINT_ID_POSITION** 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. The client only needs to perform the location algorithm once per endpoint.

An endpoint ID position component should never be combined in the same profile with a location policy of **LOCATE_OPERATION** or **LOCATE_ALWAYS**.

16.7 OMG IDL for the DCE CIOP Module

This section shows the DCE_CIOP module and DCE_CIOP additions to the IOP module.

```
module DCE_CIOP {
    struct InvokeRequestHeader {
        boolean byte_order;
        IOP::ServiceContextList service_context;
        sequence <octet> object_key;
        string operation;
        CORBA::Principal principal;

        // in and inout parameters follow
    };

    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::IOR follows
    };
    struct LocateRequestHeader {
        boolean byte_order;
        sequence <octet> object_key;
        string operation;

        // no body follows
    };

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::IOR follows
};

const IOP::ComponentId TAG_DCE_STRING_BINDING = 100;

const IOP::ComponentId TAG_DCE_BINDING_NAME = 101;

struct BindingNameComponent {
        unsigned long entry_name_syntax;
        string entry_name;
        string object_uuid;
    };

    const IOP::ComponentId TAG_DCE_NO_PIPES = 102;
};

module IOP {
    const ComponentId TAG_COMPLETE_OBJECT_KEY = 5;

    const ComponentId TAG_ENDPOINT_ID_POSITION = 6;

    struct EndpointIdPositionComponent {
        unsigned short begin;
        unsigned short end;
    };

    const ComponentId TAG_LOCATION_POLICY = 12;

    // IDL does not support octet constants
    #define LOCATE_NEVER 0
    #define LOCATE_OBJECT 1
    #define LOCATE_OPERATION 2
    #define LOCATE_ALWAYS 3
};
```

16.8 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 17

The OMG document used to update this chapter was orbos/97-09-07.

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.

Contents

This chapter contains the following sections.

17.1 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.

17.1.1 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 a 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.

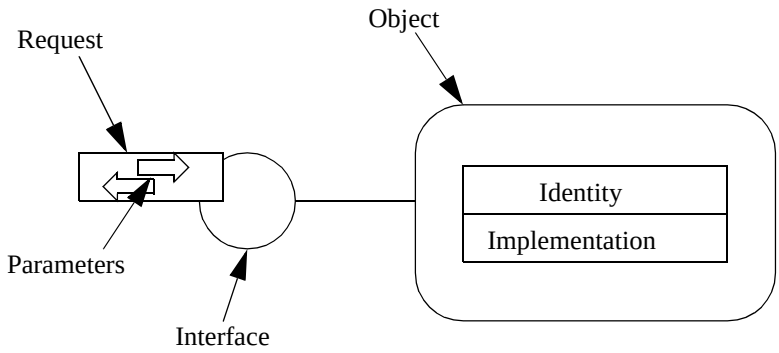


Figure 17-1 Interworking Object Model

17.2 Interworking Object Model

17.2.1 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.

17.2.2 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 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. Automation, for example, does not support user-defined constructed types such as structs or unions.

Thus, use of and interoperating with objects exposing Automation interfaces is considerably different from other COM objects. Although Automation is implemented through COM, for the purposes of this document, Automation and COM are considered to be distinct object models. Interworking between CORBA and Automation will be described separately from interworking with the basic COM model.

17.2.3 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 17-2. 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.

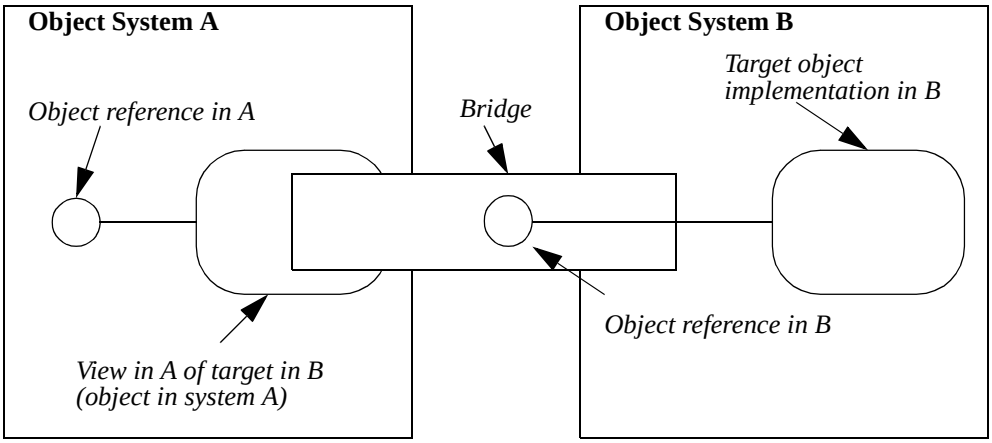


Figure 17-2 B/A Interworking Model

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.

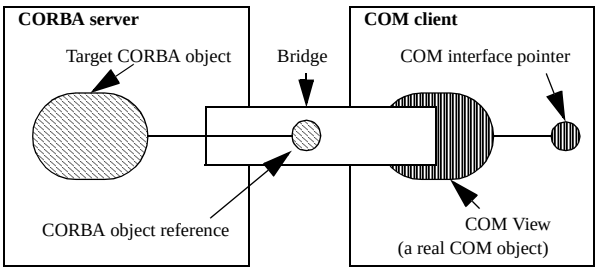
The client treats the View as though it is the real object in system A, and makes the request in the vernacular request form of system A. The request is translated into the vernacular of object system B, and delivered to the target object. The net effect is that a request made on an interface in A is 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.

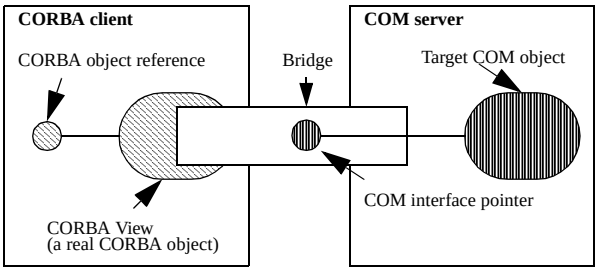
Figure 17-3 shows the interworking mappings discussed in the Interworking chapters. They represent the following:

- 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

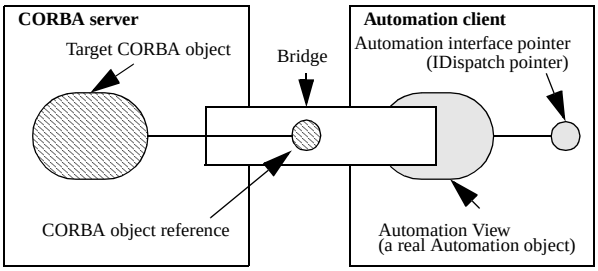
a)



b)



c)



d)

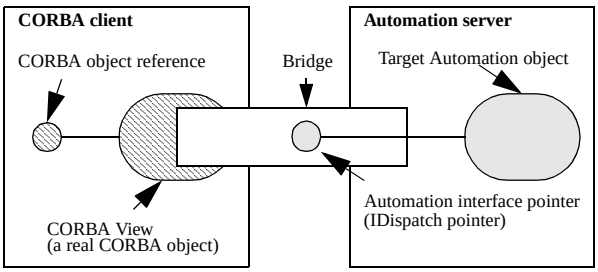


Figure 17-3 Interworking Mapping

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.

Likewise, Figure 17-3 does not define any particular location of the bridge. The bridge is conceptually between the two object models. The implementation of the bridge may be located on the client or the server or on an intermediate machine.

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.

17.3 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, 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.

17.4 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.

17.4.1 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 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.

17.4.2 CORBA/Automation

There is a limited fit between 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 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 Automation controllers. Currently 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 Automation client development environments.

17.4.3 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.

17.4.4 Automation/CORBA

The Automation interface model and type system are designed for dynamic scripting. The type system is a reduced set of the COM type system designed such that custom marshaling and demarshaling code is not necessary for invoking operations on 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 Automation 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.

17.5 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.

Automation objects typically provide all Automation operations in a single “flattened” IDispatch interface. While an analogous mechanism to QueryInterface could be supported in Automation as a standard method, it is not the current use model for OLE Automation services.[2]

17.5.1 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.

---

1. This is established in the CORBA specification, Chapter 1, Interfaces Section, 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 Query-Interface for the Z interface. Many Automation controllers, however, do not use the dual interface mechanism.

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 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 17.5.2, “Detailed Mapping Rules,” on page 17-13 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.

17.5.1.1 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 17.4, “Interface Mapping,” on page 17-8, is sufficiently isomorphic to allow composite COM interfaces to be uniformly mapped into composite OMG IDL interfaces with no loss of capability.

17.5.1.2 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 to sort the operations based on the byte-by-byte comparison of the ISO-Latin-1 encoding values of their respective names (e.g., operation ‘A’ comes before operation ‘B’).

17.5.1.3 Automation/CORBA

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.

17.5.2 Detailed Mapping Rules

17.5.2.1 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 current ordering is to sort the operations based on the byte-by-byte comparison of the ISO-Latin-1 encoding values of their respective names (e.g., operation ‘A’ comes before operation ‘B.’
- Similarly, the resulting mapping of attributes within an interface are ordered based upon the ISO-Latin-1 encoding of attribute name. If the attribute is not read-only, the get <attribute name> method immediately precedes the set <attribute name> method.

17.5.2.2 Ordering Rules for the CORBA->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 idname. The names are put in ascending order based upon the byte-by-byte comparison of ISO-Latin-1 encoding values of the interface names (e.g., interface ‘AZ’ comes before interface ‘BA’).
- 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 operations are put in ascending order based upon the ISO-Latin-1 encoding values of the operation names.

- Similarly, the mapping of an OMG IDL interface's attributes are ordered in the resulting mapping based upon the byte-by-byte comparison of the ISO-Latin-1 encoding of the attribute name. For non-read-only attributes, the [propget] method immediately precedes the [propput] method.
- For OMG IDL interfaces that multiply inherit from parent interfaces, the new interface is mapped as deriving from the mapping of its first parent.
  - Then for each subsequent parent interface, the new interface will repeat the mapping of all operations and attributes of that parent excluding any operations or attributes that have already been mapped (i.e., these operations/attributes are grouped per interface and each group is internally ordered using the rules described above.
  - After all the parent interfaces are mapped, the new operations and attributes that were introduced in the new interface are then mapped using the ordering rules for operations and attributes.

17.5.3 Example of Applying Ordering Rules

Consider the OMG IDL description shown in Figure 17-4.

```
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();
};
```

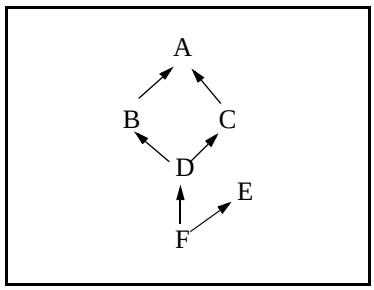
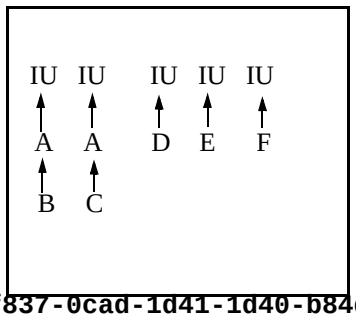


Figure 17-4 OMG IDL Description with Multiple Inheritance

Following the rules in "Detailed Mapping Rules" on page 17-13 the interface description would map to the Microsoft MIDL definition shown in Figure 17-5 and would map to the ODL definition shown in Figure 17-6.

```
[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();
```




```
};
[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 17-5 MIDL Description

```
[uuid(7fc56270-e7a7-0fa8-1dd9-35b72eacbe29),
oleautomation, dual]
interface DA : IDispatch {
    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);
};
```

// Microsoft ODL

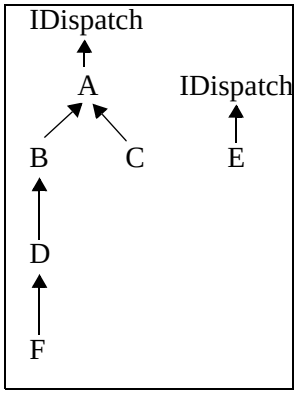


Figure 17-6 Example: ODL Mapping for Multiple Inheritance

17.5.4 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.

17.5.4.1 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).

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.

---

4. Rivest, R. “The MD5 Message-Digest Algorithm,” RFC 1321, MIT and RSA Data Security, Inc., April 1992.

17.5.4.2 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”
...
};
```

17.6 Object Identity, Binding, and Life Cycle

The interworking model illustrated in Figure 17-2 on page 17-5 and Figure 17-3 on page 17-7 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?

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}.

17.6.1 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.

17.6.1.1 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.

17.6.1.2 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 Unknown pointer. COM specifies an invariant that two interface references refer to the same object if QueryInterface (IID IUnknown) returns the same pointer when applied to both interfaces.[6] Individual COM class types may establish a strong notion of persistent identity (for example, through the use of Monikers), but this is not the responsibility of the COM model itself.
- 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 a particular class’s implementation, or the user’s imagination.

17.6.2 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.

17.6.2.1 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. COM objects are reference-counted and as such are susceptible to certain problems: most notably, circular reference counts (where two objects hold references to each other and thus neither can die) and dangling servers (where a client has crashed without releasing its references).

Detecting circular reference counts is not handled by COM and is currently left up to the application code. To help detect dangling servers, COM has added support in the infrastructure for client machines to ping server machines. If the ping is not received by the server within a negotiated time period, the client will be assumed dead and its references released.

---

6. This invariant appears to be true in DCOM as well as COM. A combination of IPID and OXID is used to create a unique identity for remote IUnknown pointers.

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 they become idle. This behavior allows resources (such as memory and networking addresses) to be released from active use for long-lived (but generally idle) services. The advantage of this model is that it does not require pinging or maintaining reference counts. The disadvantage is that it requires the application to explicitly decide when an object has been made obsolete and its references should become invalid. Activation and deactivation in COM can, to some degree, be accomplished using Monikers (persistent interface references). However, unlike CORBA, the client must be programmed to explicitly use this alternate form of binding to allow the server the opportunity to pacify its state.

In both the COM and CORBA lifecycle models, it is possible for a client to have an invalid reference to a server object. This can occur in COM because a server has crashed, or in CORBA because the target of the reference was explicitly destroyed. Thus, in both models, applications should be written to check for error codes indicating invalid references.

17.6.2.2 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 specification defines an interface, ICORBAFactory, which allows interworking solutions to provide alternate location and registration mechanisms[7] 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 17.7.3, "ICORBAFactory Interface," on page 17-24.

7. For example, using ICORBAFactory, an interworking solution can provide an active object registry that is distributed, federated, and fault-tolerant.

17.6.2.3 Binding COM Objects to CORBA Views

As described in Section 17.6.1, “Object Identity Issues,” on page 17-19, 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 in Section 17.7.1, “SimpleFactory Interface,” on page 17-23) 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.

17.6.2.4 COM View of CORBA Life Cycle

The SimpleFactory interface (Section 17.7.1, “SimpleFactory Interface,” on page 17-23) 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 specification 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 17.7.2, “IMonikerProvider Interface and Moniker Use,” on page 17-23.

17.6.2.5 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.

17.7 Interworking Interfaces

17.7.1 SimpleFactory Interface

Although a general instance factory interface can be defined in either COM or CORBA, it is the common practice in COM to have factories which support only the IClassFactory of ICoassfactory2 interfaces. These interfaces only support parameterless object constructors (i.e., the **CreateInstance()** operation takes no parameters). To allow CORBA objects to be created under this factory model in COM, the SimpleFactory interface is defined. The SimpleFactory interface is supported by all 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. CORBA objects that can be created with no initial state should provide factories that implement the SimpleFactory interface.

17.7.2 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).

17.7.3 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] LPWSTR factoryName,
         [out, retval] IUknown ** val);
      HRESULT GetObject([in] LPWSTR 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 Automation clients. DICORBAFactory is an Automation Dual Interface. (For an explanation of Automation Dual interfaces, see the *Mapping: Automation and CORBA* chapter.)

```
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 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.

17.7.4 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 GetUniqueId([out] LPSTR *id
```

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 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 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 19.8.4, "Mapping for anys," on page 19-24) 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.

17.7.5 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 *val);
   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]
      VARIANT BOOL *val);
   HRESULT IsNil([out, retval] VARIANT BOOL *val);
   HRESULT IsEquivalent([in] IDispatch* obj,[out,retval]
         VARIANT BOOL * val);
   HRESULT NonExistent([out,retval] VARIANT BOOL *val);
   HRESULT Hash([out, retval] long *val);
};
```

The UUID for DICORBAObject is:

```
   {204F6244-3AEC-11cf-BBFC-444553540000}
```

The UUID for DCORBAObject is:

```
   {7271ff40-21f6-11d1-9d47-00a024a73e4f}
```

17.7.6 ICORBAObject2

ICORBAObject 2 is the direct mapping following the COM mapping rules for the CORBA::Object interface.

17.7.7 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] VARIANT *val);
   HRESULT ResolveInitialReference ([in] BSTR name,
                  [out retval] IDispatch * val);
   HRESULT GetCORBAObject ([in] IDispatch* obj,
                  [out, retval] DICORBAObject ** val);
}
```

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.

The UUID for DIORBObject is:

{204F6246-3AEC-11cf-BBFC-444553540000}

The UUID for DORBObject is:

{adff0da0-21f6-11d1-9d47-00a024a73e4f}

17.7.8 Naming Conventions for View Components

17.7.8.1 Naming the COM View Interface

The default name for 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

17.7.8.2 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.

17.7.8.3 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

17.7.8.4 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 will be:

DIOuterModule_MyModule_MyInterface

17.7.8.5 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 will 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 will be:

“OuterModule.MyModule.MyInterface”

17.7.8.6 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
```

17.8 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.

17.8.1 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-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.

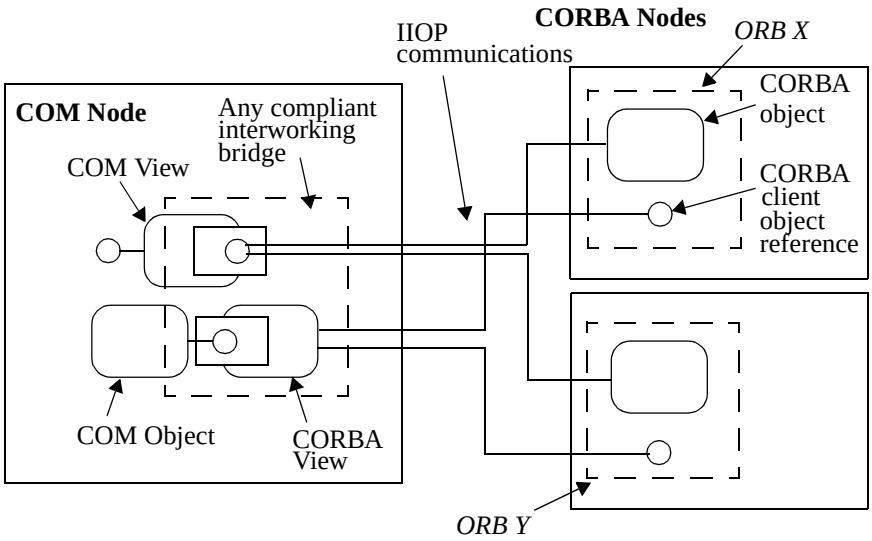


Figure 17-7 Bridge Locality

Figure 17-7 illustrates the required locality for interworking components. All of the transformations between CORBA interfaces and COM interfaces described in this specification 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-compliant networking ORB that shares the same view of interfaces with the interworking solution.

17.8.2 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*. Figure 17-7 illustrates the required distribution architecture. The following statements articulate the responsibilities of compliant solutions.

- All externalized CORBA object references will follow *CORBA* 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*.
- 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.

17.9 Interworking Targets

This specification is targeted specifically at interworking between the following systems and versions:

- CORBA as described in *CORBA: 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.

17.10 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 the Preface, Section 0.5, Definition of CORBA Compliance.

17.10.1 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

17.10.1.1 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 Automation interfaces and which also parses 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.

17.10.1.2 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

Automation interfaces. Another example would be a generic bridging component that interpretively maps 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.

17.10.1.3 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 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.

17.10.2 Compliance Points

The intent of this specification is to allow the construction of implementations that fit in the design space described in Section 17.2, “Interworking Object Model,” on page 17-3, 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 17.7.5, “ICORBAObject Interface,” on page 17-27 and Section 17.7.7, “IORBObject Interface,” on page 17-28, 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 17.7.5, “ICORBAObject Interface,” on page 17-27 and Section 17.7.7, “IORBObject Interface,” on page 17-28, 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 17.7.4, “IForeignObject Interface,” on page 17-26. 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 17.7.2, “IMonikerProvider Interface and Moniker Use,” on page 17-23.
- All COM interfaces specified in this specification 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 Internet Inter-ORB Protocol (IIOP), and use the interoperability architecture described in CORBA in the manner prescribed in Section 17.8, “Distribution,” on page 17-32. 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 17.7.3, “ICORBAFactory Interface,” on page 17-24.
- Interworking solutions that expose Automation Views of CORBA objects are required to provide the DICORBAFactory object as defined in Section 17.7.3, “ICORBAFactory Interface,” on page 17-24.
- 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, as described under Section 17.6.2, “Binding and Life Cycle,” on page 17-20.

Mapping: COM and CORBA 18

The OMG document used to update this chapter was orbos/97-09-07.

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 fully implemented by automated interworking tools.

Contents

This chapter contains the following sections.

18.1 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 marshaling” 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 18.2.10, “Interface Mapping,” on page 18-11.

18.2 CORBA to COM Data Type Mapping

18.2.1 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 18-1.

Table 18-1 OMG IDL to MIDL Intrinsic Data Type Mappings

OMG IDL	Microsoft IDL	Microsoft ODL	Description
short	short	short	Signed integer with a range of $-2^{15}...2^{15} - 1$
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
octet	byte	unsigned char	8-bit opaque data type, guaranteed to not undergo any conversion during transfer between systems.

Note – midl and mktyplib disagree about the size of boolean when used in an ODL specification. To avoid this ambiguity, we make the mapping explicit and use the VARIANT BOOL type instead of the built-in boolean type.

18.2.2 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 float F = ...;
    const char C = ...;
    const boolean B = ...;
    const string STR = "...";
```

map to 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 float F = ...;
    const char C = ...;
    const boolean B = ...;
    const string STR = "...";
```

18.2.3 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
interface MyInft {
        enum A_or_B_or_C {A, B, C};
};
```

CORBA enumerators are mapped to COM enumerations directly according to 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
uuid(...),
interface IMyIntf {
      typedef [v1_enum]
      enum tagA or B or C {MyIntf A = 0,
                           MyInft B,
                           MyIntf C }
```

```
                                    MyIntf A or B or C;
};
```

A maximum of 2^{32} identifiers may be specified in an enumeration in CORBA. Enumerators in Microsoft IDL and ODL will only support 2^{16} identifiers, and therefore, truncation may result.

18.2.4 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 18-2 illustrates how to map the string data types in OMG IDL to their corresponding data types in both Microsoft IDL and ODL.

Table 18-2 OMG IDL to Microsoft IDL/ODL String Mappings

OMG IDL	Microsoft IDL	Microsoft ODL	Description
string	LPSTR [string,unique] char *	LPSTR	Null-terminated 8-bit character string
wstring	LPWSTR [string,unique] wchar t *	LPWSTR	Null-terminated Unicode string

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 strings do not have a maximum length specified.

18.2.4.1 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.

18.2.4.2 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.

18.2.5 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;
```

18.2.6 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=0,
        dShort,
        dLong,
        dFloat,
        dDouble
    };

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

is mapped into encapsulated unions in Microsoft IDL as follows:

```
// Microsoft IDL
typedef enum [v1 enum]
    {
    dchar=0,
    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
```

18.2.7 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.

18.2.7.1 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.

18.2.7.2 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 reserved;
   unsigned long cbLengthUsed;
   [length_is(cbLengthUsed)] U Value[N];
   } BOUNDED_SEQUENCE_OF_N;
```

Note – The maximum size of the bounded sequence is declared in the declaration of the array and therefore a [size is ()] attribute is not needed.

18.2.8 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.

18.2.9 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);
            }
```

In most cases, a COM client can use the **_get_value()** or **_put_value()** method to set and get the value of a CORBA **any**. However, the data types supported by a VARIANT are too restrictive to support all values representable 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.

18.2.10 Interface Mapping

18.2.10.1 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 17.5.4, “Mapping Interface Identity,” on page 17-16.

18.2.10.2 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 18-3 lists the HRESULT constants and their values.

Table 18-3 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

Table 18-3 Standard Exception to SCODE Mapping *(Continued)*

ITF_E_IMP_LIMIT_NO	0x40203
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

Table 18-3 Standard Exception to SCODE Mapping *(Continued)*

ITF_E_BAD_OPERATION_MAYBE	0x4220C
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

Table 18-3 Standard Exception to SCODE Mapping *(Continued)*

ITF_E_BAD_CONTEXT_YES	0x41216
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
ITF_E_OBJ_NOT_EXIST_NO	0X40219
ITF_E_OBJ_NOT_EXIST_MAYBE	0X41219
ITF_E_OBJ_NOT_EXIST_YES	0X42219
ITF_E_TRANSACTION_REQUIRED_NO	0x40220
ITF_E_TRANSACTION_REQUIRED_MAYBE	0x41220
ITF_E_TRANSACTION_REQUIRED_YES	0x42220
ITF_E_TRANSACTION_ROLLEDBACK_NO	0x40221
ITF_E_TRANSACTION_ROLLEDBACK_MAYBE	0x41221
ITF_E_TRANSACTION_ROLLEDBACK_YES	0x42221
ITF_E_INVALID_TRANSACTION_NO	0x40222
ITF_E_INVALID_TRANSACTION_MAYBE	0x41222
ITF_E_INVALID_TRANSACTION_YES	0x42222

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 18-4.

Table 18-4 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
                  ->InterfaceSupportsError-
Info(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(...);
      }
}
```

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. If an interface raises a user exception, a structure is constructed whose name is the interface name [fully scoped] followed by "Exceptions." For example, if an operation in MyModule::MyInterface raises a user exception, then there will be a structure created named MyModule_MyInterfaceExceptions.

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;
        I<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[1]. The following example illustrates this point.

```
// Microsoft IDL
interface IBANKAccount
      {
    HRESULT Withdraw(          [in] float fAmount,
                               [out] float pfNewBalance,
                               [out] BANk_AccountExceptions
                                  ** pException);
      };
```

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.

---

1. Vendors that map the MIDL definition directly to C++ should map the exception struct parameter as defaulting to a NULL pointer.

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 Bank_InsufFunds
          {
          float balance;
          };

struct Bank_InvalidAmount
          {
          float amount;
          };

struct BANK_Account_NotAuthorized
          {
          };

interface IBANK_AccountUserExceptions : IUnknown
          {
        HRESULT _get_InsufFunds( [out]  BANK_InsufFunds
                              * exceptionBody );
        HRESULT _get_InvalidAmount( [out]
BANK_InvalidAmount
                              * exceptionBody );
        HRESULT _get_NotAuthorized( [out]

BANK_Account_NotAuthorized
             * exceptionBody );
         };
```

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

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 18-5.

Table 18-5 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 (COM_ERROR or COM_ERROREX) user exception (defined under Section 18.3.10.2, “Mapping for COM Errors,” on page 18-45), 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 (and COM_ERROREX) 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.

18.2.10.3 Mapping for Nested Types

OMG IDL and Microsoft MIDL/ODL do not agree on the scoping level of types declared within interfaces. Microsoft, for example, considers all types in a MIDL or ODL file to be declared at global scope. OMG IDL considers a type to be scoped within its enclosing module or interface. This means that to prevent accidental name collisions, types declared within OMG IDL modules and OMG IDL interfaces must be fully qualified in Microsoft IDL or ODL.

The OMG IDL construct:

```
Module BANK{
        interface ATM {
            enum type {CHECKS, CASH};
            Struct DepositRecord {
                string account;
                float amount;
                type kind;
            };
            void deposit (in DepositRecord val);
        };
```

Must be mapped in Microsoft MIDL as:

```
[uuid(...), object]
interface IBANK ATM : IUnknown {
       typedef [v1 enum] enum
          {BANK ATM CHECKS,
          BANK ATM CASH} BANK ATM type;
       typedef struct {
          LPSTR account;
          BANK ATM type kind;
       } BANK ATM DepositRecord;
       HRESULT  deposit (in BANK ATM DepositRecord *val);
};
```

and to Microsoft ODL as:

```
[uuid(...)]
library BANK {
...
[uuid(...), object]
interface IBANK ATM : IUnknown {
      typedef enum { BANK ATM CHECKS,
                     {BANK ATM CASH} BANK ATM type;
      typedef struct {
         LPSTR struct;
         float amount;
         BANK ATM type kind;
      } BANK ATM DepositRecord;
      HRESULT deposit (in BANK ATM DepositRecord *val);
};
```

18.2.10.4 *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 (shown below) illustrates 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::Open Account** 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 Account(Type);
        void Transfer(          in Account Account1,
                                in Account Account2,
                                in float      Account)
                                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 Teller: IUnknown
    {
    HRESULT  OpenAccount(
                  [in]   float    StartingBalance,
                  [in]   BANK_AccountTypes AccountType,
                  [out] IBANK_Account   ** ppiNewAccount );
    HRESULT  Transfer(
                  [in] IBANK_Account * Account1,
                  [in] IBANK_Account * Account2,
                  [in] float Amount,
                  [out] BANK_TellerExceptions
                     ** ppException);
    };
```

and to the following statements in Microsoft ODL

```
// Microsoft ODL
[ uuid(682d22fb-78ac-0000-0c03-4d0000000000) odl ]
interface IBANK_Teller: IUnknown
    {
    HRESULT OpenAccount(
                  [in] float              StartingBalance,
                  [in] BANK_AccountTypes  AccountType,
                  [out, retval] IBANK_Account
                     ** ppiNewAccount );
    HRESULT Transfer(
                  [in] IBANK_Account * Account1,
                  [in] IBANK_Account  * Account2,
                  [in] float   Amount,
                  [out]BANK_TellerExceptions
                     ** ppException);
  };
```

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 Section 18.2.10.2, "Mapping for exception types," on page 18-11 for further details.

18.2.10.5 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.

18.2.10.6 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 IBANK_Customer : IUnknown
    {
    [propget]  HRESULT  Profile(
                    [out]  BANK_CustomerData * val);
    [propput] HRESULT  Profile(
                    [in]  BANK_CustomerData * val);
    };
```

Note – The attribute is actually mapped as two different operations in both Microsoft IDL and ODL. The **`IBANK_Customer::get_profile`** operation (in Microsoft IDL) and the [**`propget`**] Profile operation (in Microsoft ODL) are used to retrieve the value of the attribute. The **`IBANK_Customer::put_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 *val);
    };
```

Note that only a single operation was defined since the attribute was defined to be read-only.

18.2.10.7 Indirection Levels for Operation Parameters

- For integral types (such as long, enum, char,...) these are passed by value as [in] parameters and by reference as out parameters.
- string/wstring parameters are passed as LPSTR/LPWSTR as an in parameter and LPSTR*/LPWSTR* as an out parameter.
- composite types (such as unions, structures, exceptions) are passed by reference for both [in] and [out] parameters.
- optional parameters are passed using double indirection (e.g., IntfException ** val).

18.2.11 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* (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.
- Operations are sorted in ascending order based upon the ISO Latin-1 encoding values of the respective operation names.

- The resulting mapping of attributes within an interface are ordered based upon the attribute name. The attributes are similarly sorted in ascending order based upon the ISO-Latin-1 encoding values of the respective attribute names. If the attribute is not readonly, the get_<attribute name> method immediately precedes the set_<attribute name> method.

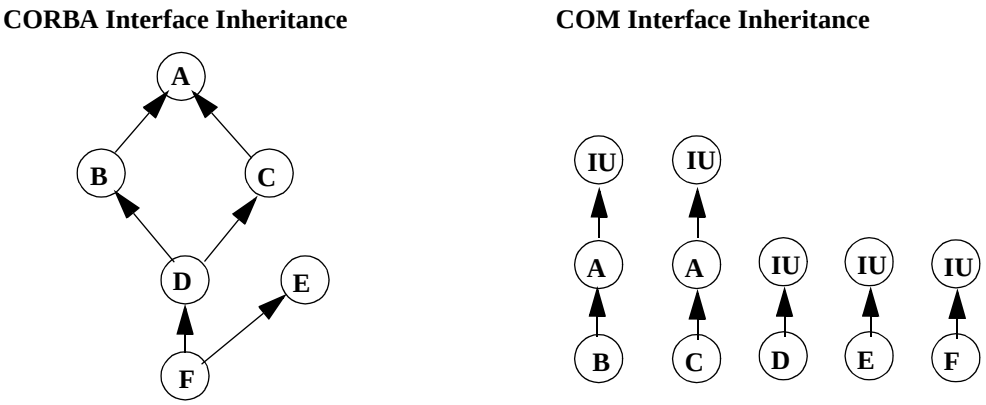


Figure 18-1 CORBA Interface Inheritance to COM Interface Inheritance Mapping

```
//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.

18.2.12 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.

18.2.12.1 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 *pException-
Body);
   HRESULT _get_BadKind( [out] TypeCodeBadKind  * pExcep-
tionBody );
};

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
   } CORBA_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] CORBA_TypeCodeExceptions** ppUserExceptions);
      HRESULT kind(
```

```
        [out] TCKind                * pRetVal,
        [out] CORBA_TypeCodeExceptions **ppUserExceptions);
    HRESULT id(
        [out] RepositoryId          * pszRetVal,
        [out] CORBA_TypeCodeExceptions **ppUserExceptions);
    HRESULT name(
        [out] Identifier            * pszRetVal,
        [out] CORBA_TypeCodeExceptions **ppUserExceptions);
    HRESULT member_count(
        [out] unsigned long         * pulRetVal,
        [out] CORBA_TypeCodeExceptions **ppUserExceptions);
    HRESULT member_name(
        [in]  unsigned long           ulIndex,
        [out] Identifier            * pszRetVal,
        [out] CORBA_TypeCodeExceptions ** ppUserExceptions;)
    HRESULT member_type(
        [in]  unsigned long           ulIndex,
        [out] ICORBA_TypeCode      ** ppRetVal,
        [out] CORBA_TypeCodeExceptions ** ppUserExceptions;)
    HRESULT member_label(
        [in]  unsigned long           ulIndex,
        [out] ICORBA_Any           ** ppRetVal,
        [out] CORBA_TypeCodeExceptions ** ppUserExceptions;)
    HRESULT discriminator_type(
        [out] ICORBA_TypeCode      ** ppRetVal,
        [out] CORBA_TypeCodeExceptions ** ppUserExceptions;)
    HRESULT default_index(
        [out] long                  * plRetVal,
        [out] CORBA_TypeCodeExceptions **ppUserExceptions);
    HRESULT length(
        [out] unsigned long         * pulRetVal,
        [out] CORBA_TypeCodeExceptions   **ppUserExceptions);
    HRESULT content_type(
        [out] ICORBA_TypeCode         ** ppRetVal,
        [out] CORBA_TypeCodeExceptions **ppUserExceptions);
    HRESULT param_count(
        [out] long                      * plRetVal,
        [out] CORBA_TypeCodeExceptions **ppUserExceptions);
    HRESULT parameter(
        [in]  long                    lIndex,
        [out] ICORBA_Any           ** ppRetVal,
        [out] CORBA_TypeCodeExceptions  ** ppUserExceptions
    );
}
```

Note – Use of the methods **`param_count()`** and **`parameter()`** is deprecated.

18.2.12.2 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.

```
// 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
                              ** pValues );
      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.

18.2.12.3 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.

18.2.13 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 18-6 defines the mapping between equivalent CORBA and COM interface description concepts. Where there is no equivalent, the field is left blank.

Table 18-6 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.

18.3 COM to CORBA Data Type Mapping

18.3.1 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 18-7.

Table 18-7 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 $-2^{31}...2^{31}-1$

Table 18-7 Microsoft IDL and ODL to OMG IDL Intrinsic Data Type Mappings *(Continued)*

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

18.3.2 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 float F = ...;
const double D = ...;
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 float F = ...;
const double D = ...;
const char C = ...;
const boolean B = ...;
const string STR = “...”;
```

18.3.3 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 in increasing order according to their tagged values.

The Microsoft IDL or ODL specification:

```
// 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  {B, C, A};
```

In this manner, the precedence relationship is maintained in the OMG system such that B is less than C is less than A.

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

18.3.4 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 18-8 illustrates how to map the string data types in OMG IDL to their corresponding data types in both Microsoft IDL and ODL.

Table 18-8 Microsoft IDL/ODL to OMG IDL String Mappings

Microsoft IDL	Microsoft ODL	OMG IDL	Description
LPSTR [string,unique] char *	LPSTR,	string	Null-terminated 8-bit character string
BSTR	BSTR	wstring	Null-terminated 16-bit character string
LPWSTR [string,unique] char *	LPWSTR	wstring	Null-terminated Unicode string

If a COM Server returns a BSTR containing embedded nulls to a CORBA client, a E_DATA_CONVERSION exception will be raised.

18.3.4.1 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.

18.3.4.2 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 whose number of valid characters can vary at run-time.

18.3.4.3 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] LPWSTR
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.

18.3.4.4 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] wchar t(*
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.

18.3.5 Mapping for Structure Types

Support for structures in Microsoft IDL and ODL maps bidirectionally to OMG IDL. Each structure member 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;
    };
```

18.3.6 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.

18.3.6.1 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];
};
```

18.3.6.2 *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;
```

18.3.7 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];
```

18.3.7.1 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;
```

18.3.7.2 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.

18.3.8 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 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 18-9, along with a description of how to use them to represent the OMG IDL **any** data type.

Table 18-9 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*.

Table 18-9 Valid OLE VARIANT Data Types *(Continued)*

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 *bstrVal*. 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.
VT_ERROR	An SCODE was specified. The type of 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*.

Table 18-9 Valid OLE VARIANT Data Types *(Continued)*

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 18-9 Valid OLE VARIANT Data Types *(Continued)*

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.

18.3.9 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.

18.3.10 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.

18.3.10.1 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.

18.3.10.2 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 18-10 lists the mapping from COM FACILITY_NULL exceptions to CORBA standard exceptions.

*Table 18-10*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

*Table 18-10*Mapping from COM FACILITY_NULL Error Codes to CORBA Standard (System) Exceptions *(Continued)*

E_FAIL	UNKNOWN
E_ACCESSDENIED	NO_PERMISSION
E_UNEXPECTED	UNKNOWN
E_ABORT	UNKNOWN
E_POINTER	BAD_PARAM
E_HANDLE	BAD_PARAM

Table 18-11 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 18-11 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

Table 18-11 Mapping from COM FACILITY_RPC Error Codes to CORBA Standard (System) Exceptions *(Continued)*

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_ERROREX
{
        long hresult;
        Any info;
};
```

The **COM_ERROREX** extension is designed to allow exposure of exceptions passed using the per-thread ErrorObject. The Any contained in the **COM_ERROREX** is defined to hold a CORBA object reference which supports the OMG IDL mapping for the IErrorInfo interface.

18.3.10.3 Mapping of Nested Data Types

Microsoft MIDL (and ODL) consider all definitions to be at global (or library) scope regardless of position in the file. This can lead to name collisions in datatypes across interfaces. Operations or types later in the file can refer to a datatype without fully qualifying the name even if the type is nested within another interface.

For purposes of mapping MIDL/ODL to OMG IDL, we treat nested datatypes as if they had been prepended with the name of the scoping level. Thus:

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

is mapped as if it were defined as:

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

18.3.10.4 Mapping of Names

Microsoft MIDL and ODL support prefixing types/names with leading underscores. When mapping from Microsoft MIDL or ODL to OMG IDL, the leading underscores are removed.

Note – This simple rule is not sufficient to avoid all name collisions (such as MIDL types that clash with OMG IDL reserved names or situations where two operation names differ only in the leading underscore). However, this rule will cover many common cases and leads to a more natural mapping than prepending a character before the underscore.

18.3.10.5 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),
COM_ERROREX);
        HRESULT permute(inout short value);

        HRESULT tryPermute(inout short value, out long newValue)
        };
```

18.3.10.6 *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.

18.3.11 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;
    ...
    };
```

18.3.12 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.

18.3.12.1 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.[2]

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.

```
module CORBA                                                    // PIDL
    {
    interface Composite
            {
            Object query_interface(in RepositoryId whichOne);
    };
```

2. This mapping fails in some cases, for example, if operation names are the same.

```
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);
        };
```

18.3.12.2 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 18-12*CORBA Interface Repository to OLE Type Library Mappings

CORBA	COM
TypeCode	TYPEDESC
Repository	

*Table 18-12*CORBA Interface Repository to OLE Type Library Mappings *(Continued)*

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: Automation and CORBA 19

The OMG document used to update this chapter was orbos/97-09-07.

This chapter describes the bidirectional data type and interface mapping between 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 the Interworking Architecture chapter, 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.

Contents

This chapter contains the following sections.

19.1 Mapping CORBA Objects to Automation

19.1.1 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 19-1 (the call to the Automation View could be a call in the same process).

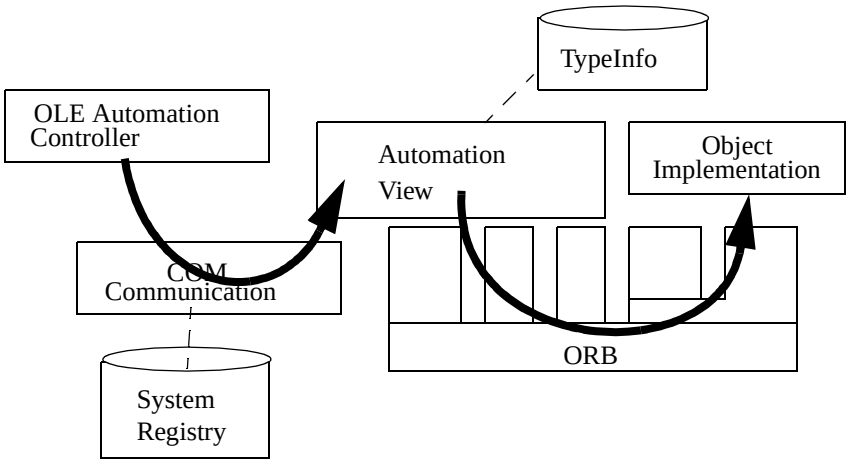


Figure 19-1 CORBA Object Architectural Overview

The Automation View is an 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.

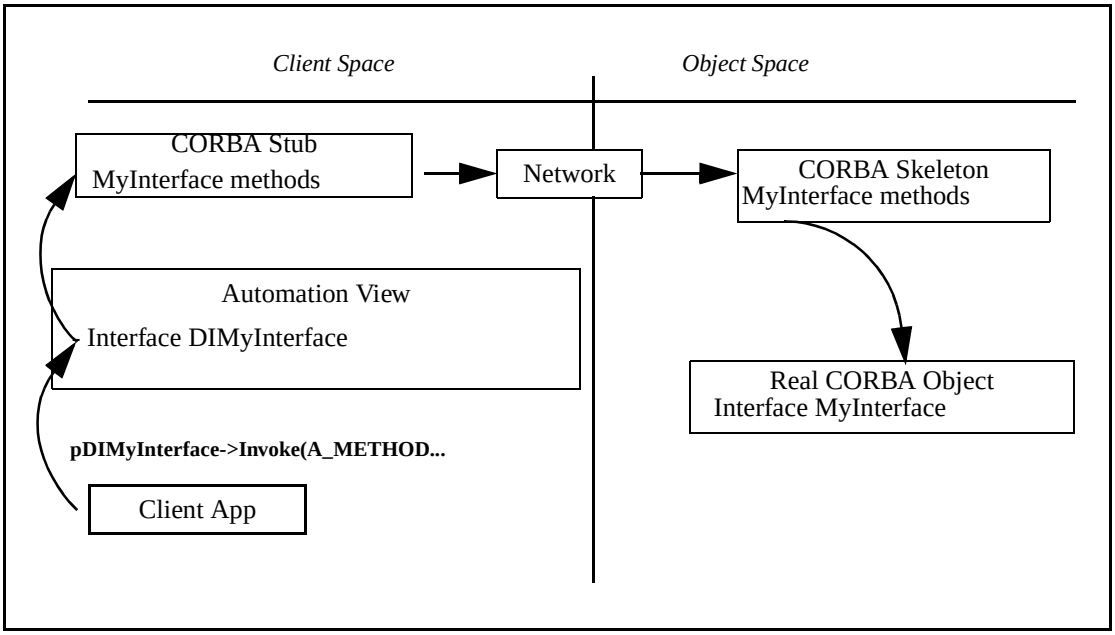


Figure 19-2 Methods of the Automation View Interface Delegate to the CORBA Stub

19.1.2 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 VARIANTS containing an Automation Safearray.

19.2 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 **`DIMyModule_account`** is an 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, **`DIMyModule_account`** contains the methods of IDispatch as well as separate vtable-entries for its operations and property get/set methods.

19.2.1 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 17.5.2, “Detailed Mapping Rules,” on page 17-13.

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

OMG IDL **in**, **out**, and **inout** parameters map to ODL **`[in]`**, **`[out]`**, and **`[in,out]`** parameters, respectively. Section 19.3, “Mapping for Basic Data Types,” on page 19-9, 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 19.8.9, “Mapping CORBA Exceptions to Automation Exceptions,” on page 19-30 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.

19.2.2 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 * val);
   [propput] HRESULT balance( [in] float balance);
   [propget] HRESULT owner(   [retval,out] BSTR * val);
};

[odl, dual, uuid(ffe752b2-a73f-2a28-1de4-21754778ab4b)]
interface DIMyModule_checkingAccount: IMyModule_account {
   HRESULT orderChequeBook(
         [optional, out] VARIANT * excep_OBJ,
         [retval,out] short * val);
         [propget] HRESULT overdraftLimit (
         [retval,out] short * val);
};
```

19.2.3 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 19-3.

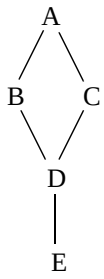


Figure 19-3 A CORBA Interface Hierarchy Using Multiple Inheritance

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 19-4.

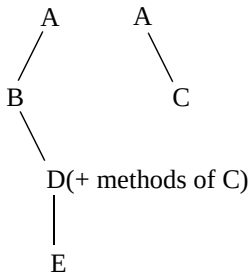


Figure 19-4 The Mapped Automation Hierarchy Splits at the Point of Multiple Inheritance

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 after operations and both should be ordered in ascending order based upon the operation/attribute names. The ordering is based on a byte-by-byte comparison of the ISO-Latin-1 encoding values of the operation names going from first character to last. 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 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 19-3 on page 19-7.

```
// 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.

19.3 Mapping for Basic Data Types

19.3.1 Basic Automation Types

Table 19-1 lists the basic data types supported by 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 19-1 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 a return type for a function, or in a function parameter list to indicate no parameters.
BSTR	Length-prefixed string. Prefix is an integer.

Type	Description
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 19-2.

Table 19-2 OMG CORBA to 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

19.3.2 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.

19.3.2.1 Converting 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.

19.3.2.2 Demoting 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.

19.3.2.3 Demoting 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.

19.3.2.4 Converting 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.

19.3.3 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.

When mapping a fixed length string, the Automation view is required to raise the exception DISP_E_OVERFLOW if a BSTR is longer than the maximum size.

19.4 IDL to ODL Mapping

19.4.1 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 *val);
   HRESULT charReturn ([optional,out] VARIANT * excep_OBJ,
               [retval,out] short *val);
```

```
HRESULT doubleReturn ([optional,out] VARIANT * excep_OBJ,
            [retval,out] double *val);
HRESULT floatReturn ([optional,out] VARIANT * excep_OBJ,
            [retval,out] float *val);
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 * val);
HRESULT longReturn ([optional,out] VARIANT * excep_OBJ,
            [retval,out] long *val);
HRESULT octetReturn ([optional,out] VARIANT * excep_OBJ,
            [retval,out] short *val);
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 * val);
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 *val);
HRESULT shortReturn ([optional,out] VARIANT * excep_OBJ,
            [retval,out] short *val);
HRESULT stringReturn ([optional,out] VARIANT * excep_OBJ,
            [retval,out] BSTR *val);
```

```
    HRESULT stringnReturn ([optional,out] VARIANT *
            exep_OBJ,
               [retval,out] BSTR *val);
    HRESULT ulongReturn ([optional,out] VARIANT * excep_OBJ,
               [retval,out] long *val);
    HRESULT ushortReturn ([optional,out] VARIANT * excep_OBJ,
               [retval,out] long *val);
    [propget] HRESULT boolTest([retval,out] short *val);
    [propput] HRESULT boolTest([in] short boolTest);
    [propget] HRESULT charTest([retval,out] short *val);
    [propput] HRESULT charTest([in] short charTest);
    [propget] HRESULT doubleTest([retval,out] double *val);
    [propput] HRESULT doubleTest([in] double doubleTest);
    [propget] HRESULT floatTest([retval,out] float *val);
    [propput] HRESULT floatTest([in] float floatTest);
    [propget] HRESULT longTest([retval,out] long *val);
    [propput] HRESULT longTest([in] long longTest);
    [propget] HRESULT octetTest([retval,out] short *val);
    [propput] HRESULT octetTest([in] short octetTest);
    [propget] HRESULT readonlyShortTest([retval,out] short
*val);
    [propget] HRESULT shortTest([retval,out] short *val);
    [propput] HRESULT shortTest([in] short shortTest);
    [propget] HRESULT stringTest([retval,out] BSTR *val);
    [propput] HRESULT stringTest([in] BSTR stringTest);
    [propget] HRESULT stringnTest([retval,out] BSTR *val);
    [propput] HRESULT stringnTest([in] BSTR stringnTest);
    [propget] HRESULT ulongTest([retval,out] long *val);
    [propput] HRESULT ulongTest([in] long ulongTest);
    [propget] HRESULT ushortTest([retval,out] long *val);
    [propput] HRESULT ushortTest([in] long ushortTest);
}
```

19.5 Mapping for Object References

19.5.1 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 *val);
   [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 ** val);

   [propget] HRESULT simpleTest([retval, out]
            DIMyModule_Simple **val);
   [propput] HRESULT simpleTest([in] DIMyModule_Simple
               *simpleTest);

}
```

19.5.2 Object Reference Parameters and IForeignObject

As described in the Interworking Architecture chapter, 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 19-5, Automation Views expose IForeignObject, as required of all Views.

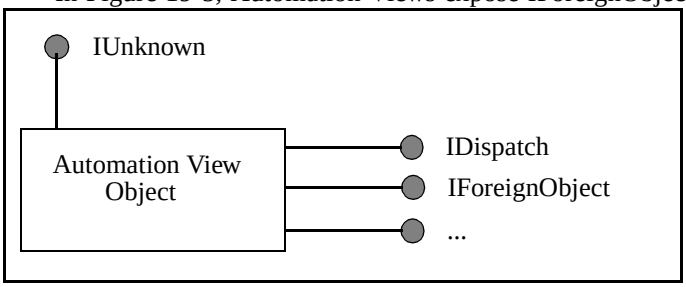


Figure 19-5 Partial Picture of the Automation View

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.

19.6 Mapping for Enumerated Types

CORBA enums map to Automation enums. Consider the following example

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

which maps to the following ODL:

```
// ODL
typedef enum {MyModule_red, MyModule_green, MyModule_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, 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 17.7.8, "Naming Conventions for View Components," on page 17-30.)

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.

19.7 Mapping for Arrays and Sequences

OMG IDL Arrays and Sequences are mapped as a VARIANT containing an Automation SAFEARRAY. SAFEARRAYs are one- or multi-dimensional arrays whose elements are of any of the basic Automation types. The following ODL syntax describes an array parameter:

```
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.

For bounded Sequence, Safearray will grow dynamically up to the specified bounded size and maintain information on its current length. Unbounded OMG IDL sequences are mapped to VARIANTS containing a Safearray 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 the VARIANT containing the Safearray as a parameter, has the intelligence to handle the parameter properly. When the VARIANT is 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 the VARIANT containing the Safearray is an **`out`** parameter, 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 VARIANTs containing multidimensional Safearrays. The order of dimensions in the OMG IDL array from left to right corresponds to ascending order of dimensions in the Safearray. If the number of dimensions of an input SAFEARRAY does not match the CORBA type, the Automation view will generate the HRESULT DISP_E_TYPEMISMATCH.

19.8 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 ([retval,out]
              BSTR *val);
    HRESULT ([in] IDispatch *pDispatch,
              [out, retval] IDispatch **val);
}
```

The UUID for DIForeignComplexType is:

{A8B553C0-3B72-11cf-BBFC-444553540000}

This interface can also be implemented as a generic (nondual) Automation Interface, in which case it is named **DForeignComplexType** and its UUID is:

{E977F900-3B75-11cf-BBFC-444553540000}

The direct use of the INSTANCE repositoryID () is deprecated. The approved way to retrieve the repositoryId is through the **DIObjectInfo::unique id ()** method.

The direct use of the INSTANCE clone () method is deprecated. The approved way to clone the data referred to by a reference is to use the **DIObjectInfo::clone ()** method.

19.8.1 Mapping for Structure Types

CORBA structures are mapped to a Pseudo-Struct, which is a 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;
};
```

19.8.2 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
{
   [hidden] HRESULT repositoryID ([out) BSTR * val);
}
```

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}

To support OMG IDL described unions that support multiple case labels per union branch, the DICORBAUnion2 interface is defined in a way to provide two additional accessors.

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

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

The uuid for the DICORBAUnion2 interface is:
{1a2face0-2199-11d1-9d47-00a024a73e4f}

The uuid for the DCORBAUnion2 interface is:
{5d4b8bc0-2199-11d1-9d47-00a024a73e4f}

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 DICORBAUnion2.

```
interface A;                              // IDL

interface U
{
        // Switch discriminant
        readonly attribute long UNION_d;

        attribute long l;
        attribute float f;
        attribute A obj;
};
```

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

19.8.3 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_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
   } CORBATCKind;

[odl, dual, uuid(...)]
interface DICORBATypeCode: DIForeignComplexType {
   [propget] HRESULT kind([retval,out] TCKind * val);

   // for tk_objref, tk_struct, tk_union, tk_alias,
            tk_except
   [propget] HRESULT id([retval,out] BSTR *val);
   [propget] HRESULT name([retval,out] BSTR * val);

//tk_struct,tk_union,tk_enum,tk_except
                  [propget] HRESULT
member_count([retval,out]
      long * val);
   HRESULT member_name([in] long index,[retval,out]
      BSTR * val);
   HRESULT member_type([in] long index,
   [retval,out] DICORBATypeCode ** val),

// tk_union
   HRESULT member_label([in] long index,[retval,out]
         VARIANT * val);
   [propget] HRESULT discriminator_type([retval,out]
         IDispatch ** val);
   [propget] HRESULT default_index([retval,out]
         long * val);

// tk_string, tk_array, tk_sequence
   [propget] HRESULT length([retval,out] long * val);

// tk_sequence, tk_array, tk_alias
   [propget] HRESULT content_type([retval,out]
```

```
            IDispatch ** val);
}
```

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 CORBA_TCKind 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 19.8, “Mapping for CORBA Complex Types,” on page 19-19 for a description of the DIForeignComplexType interface.

19.8.4 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 * val);
   [propput] HRESULT value([in] VARIANT val);
   [propget] HRESULT typeCode([retval,out]
      DICORBATypeCode ** val);
}
```

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 19.8, “Mapping for CORBA Complex Types,” on page 19-19 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.

19.8.5 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”)
```

19.8.6 Mapping for Constants

The notion of a constant does not exist in 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.

19.8.7 Getting Initial CORBA Object References

The DICORBAFactory interface, described in Section 17.7.3, “ICORBAFactory Interface,” on page 17-24, 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 17.7.3, “ICORBAFactory Interface,” on page 17-24.

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 Automation bridging object directly with the Windows Registry. The administration and assignment of ProgIds for direct registration should follow the naming rules described in the *Interworking Architecture* chapter.

19.8.8 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 17.7.3, “ICORBAFactory Interface,” on page 17-24 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 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.

19.8.8.1 ITypeFactory Interface

The DICORBAFactoryEx 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] LPWSTR typeName, [out] VARIANT
         *val);
   HRESULT CreateTypeById( [in] RepositoryId repositoryID,
                           [out] VARIANT *val);
}
```

The UUID for ITypeFactory is:

{A8B553C6-3B72-11cf-BBFC-444553540000}

The methods on ITypeFactory provide the behaviors previously described for the corresponding DICORBAFactoryEx methods.

19.8.8.2 DIObjectInfo Interface

The DIObjectInfo interface provides helper functions for retrieving information about a composite data type (such as a union, structure, exception, …), which is held as an IDispatch pointer.

```
// ODL
[odl, dual, uuid(...)]
interface DIObjectInfo: DICORBAFactoryEx
{
   HRESULT type_name([in] IDispatch *target,
                     [out, optional] VARIANT *except_obj,
                     [out, retval] BSTR *typeName);
   HRESULT scoped_name( [in] IDispatch *target,
                        [out, optional] VARIANT *except_obj,
                        [out, retval] BSTR *repositoryId);
   HRESULT unique_id([in] IDispatch *target,
                     [out, optional] VARIANT *except_obj,
                     [out, retval] BSTR *repositoryId);
}
```

The UUID for DIObjectInfo is:
{6dd1b940-21a0-11d1-9d47-00a024a73e4f}

This interface can also be implemented as generic (nondual) Automation Interface, in which case it is named DObjectInfo and its UUID is:

{8fbbf980-21a0-11d1-9d47-00a024a73e4f}

The Automation object having the ProgId "CORBA.Factory" exposes DIObjectInfo.

19.8.9 Mapping CORBA Exceptions to Automation Exceptions

19.8.9.1 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.

19.8.9.2 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 *val);
   [propget] HRESULT EX_repositoryID([retval,out] BSTR *val);
```

```
};
```

The **EX_Id()** method will return the name of the exception. For CORBA exceptions, this will be the unscoped name of the exception. Additional accessors are available on the DIObjectInfo interface for returning the scoped name and repository id for CORBA exceptions.

Note – Renaming **EX_RepositoryId** to **EX_Id** does break backwards compatibility, but should simplify the use of exceptions from VB.

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 exceptions 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.

19.8.9.3 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);
}
```

19.8.9.4 *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 19-3 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 19-4 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.

19.8.9.5 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 *val);
   [propget] HRESULT EX_completionStatus([retval,out] long
*val);
```

```
}
```

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 =
```

These values may be specified as an enum in the typelibrary information:

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

19.8.9.6 *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 19-5. 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 19-5 on page 19-35. 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 19-5 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 19-6 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 18-3 on page 18-12.

Table 19-7 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.

19.8.10 Conventions for Naming Components of the Automation View

The conventions for naming components of the Automation View are detailed in Section 17.7.8, "Naming Conventions for View Components," on page 17-30.

19.8.11 Naming Conventions for Pseudo-Structs, Pseudo-Unions, and Pseudo-Exceptions

The formulas used to name components of the Automation View (see Section 17.7.8, "Naming Conventions for View Components," on page 17-30) 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.

19.8.12 Automation View Interface as a Dispatch Interface (Nondual)

In addition to implementing the Automation View Interface as an Automation Dual Interface, it is also acceptable to map it as a generic Dispatch Interface.

Note – All views which expose the dual interface must respond to QueryInterface for both the dual interface IID as well as for the non-dual interface IID.

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] <type1> arg1, [out] <type2> arg2,
   [retval, out] <return type> *val)
```

would be declared in ODL in a dispatch interface in the following form:

```
<return type> aMethod([in] <type1> arg1, [out] <type2> arg2)
```

Using the example from Section 19.2, “Mapping for Interfaces,” on page 19-3:

```
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.

```
[uuid(e268443e-43d9-3dab-1dbe-f303bbe9642f), oleautomation]
dispinterface Daccount: IUnknown {// ODL
   properties:
      [id(0)] float balance;
      [id(i), readonly] BSTR owner;
   methods:
      [id(2)] void makeLodgement([in] float amount,
                                 [out] float *balance,
                                 [out, optional]VARIANT OBJ);
      [id(3)] void makeWithdrawal ([in] float amount,
                           [out] float *balance,
                           [out,optional]VARIANT *excep OBJ);
   };
```

The dispatch interface is Daccount. In the example used for mapping object references in Section 19.5, “Mapping for Object References,” on page 19-15, the reference to the Simple interface in the OMG IDL would map to a reference to **`DMyModule_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 17.7.8, “Naming Conventions for View Components,” on page 17-30 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.

19.8.13 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.

19.8.14 DII and DSI

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.

19.9 Mapping 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.

19.9.1 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, COM proxies and stubs, and 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.

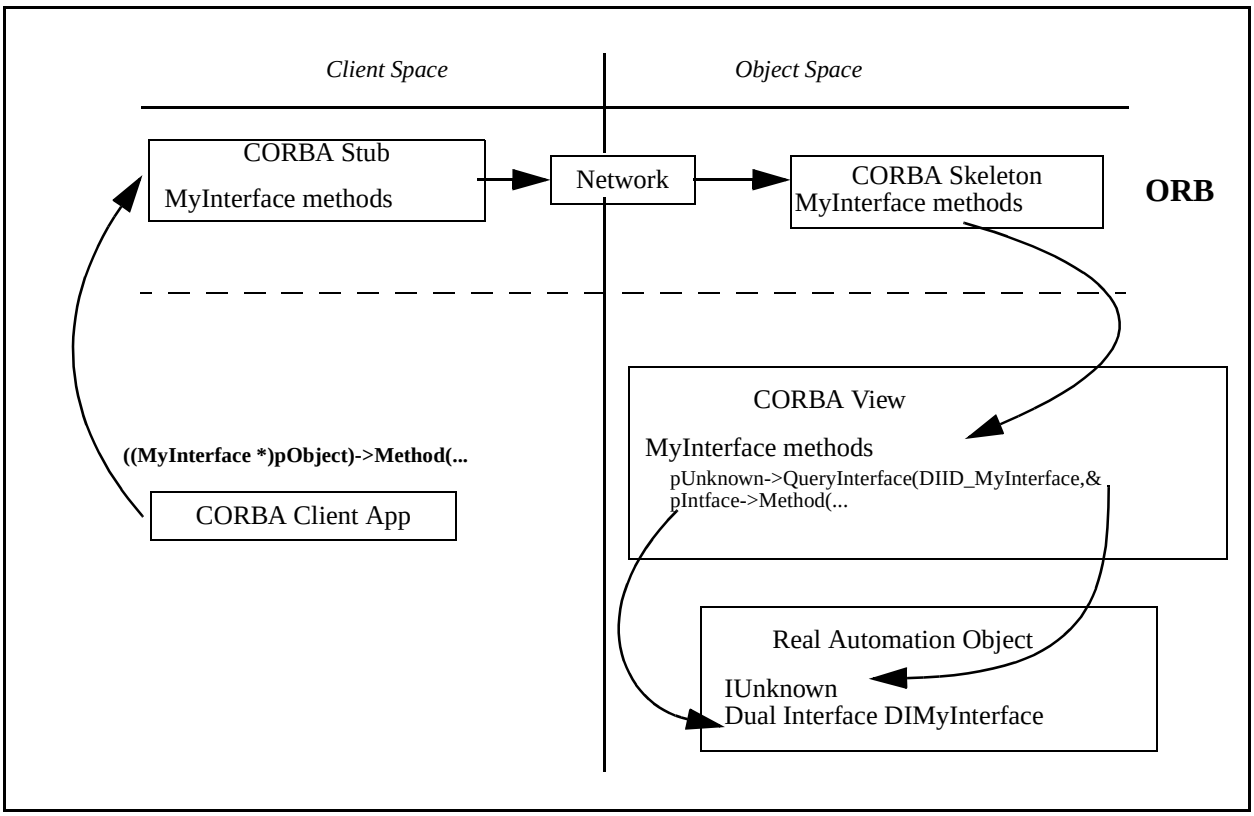


Figure 19-6 The CORBA View: a CORBA Object, which is a Client of a COM Object

19.9.2 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.

19.9.3 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.

19.9.4 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.

19.9.5 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 ();
};
```

19.9.6 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 19.3, “Mapping for Basic Data Types,” on page 19-9 explains the mapping for basic types.

19.9.7 Mapping for Automation Basic Data Types

19.9.7.1 Basic automation types

The basic data types allowed by Automation as parameters and return values are detailed in Section 19.3, “Mapping for Basic Data Types,” on page 19-9.

The formal mapping of CORBA types to Automation types is shown in Table 19-8.

Table 19-8 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

Note – The mapping of BSTR to WString breaks backwards compatibility where BSTR was mapped to string.

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.

19.9.8 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.

19.9.9 Special Cases of Data Type Conversion

19.9.9.1 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.

19.9.9.2 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.

19.9.9.3 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.

19.9.10 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] VARIANT BOOL boolTest);
   [propget] HRESULT boolTest([retval,out] short *val);
   [propput] HRESULT doubleTest([in] double doubleTest);
   [propget] HRESULT doubleTest([retval,out] double *val);
   [propput] HRESULT floatTest([in] float floatTest);
   [propget] HRESULT floatTest([retval,out] float *val);
   [propput] HRESULT longTest([in] long longTest);
   [propget] HRESULT longTest([retval,out] long *val);
   [propput] HRESULT shortTest([in] short shortTest);
   [propget] HRESULT shortTest([retval,out] short *val);
   [propput] HRESULT stringTest([in] BSTR stringTest);
   [propget] HRESULT stringTest([retval,out] BSTR *val);
   [propput] HRESULT dateTest([in] DATE stringTest);
   [propget] HRESULT dateTest([retval,out] DATE *val);
   [propput] HRESULT currencyTest([in] CURRENCY stringTest);
   [propget] HRESULT currencyTest([retval,out] CURRENCY
      *val);
   [propget] HRESULT readonlyShortTest([retval,out] short
      *val);
   HRESULT setAll(   [in] VARIANT BOOL 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 * val);
   HRESULT getAll(   [out] VARIANT BOOL *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 * val);
    HRESULT setAndIncrement([in,out] VARIANT BOOL *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 *val);
    HRESULT boolReturn(     [retval,out] VARIANT BOOL *val);
    HRESULT doubleReturn(   [retval,out] double *val);
    HRESULT floatReturn(    [retval,out] float *val);
    HRESULT longReturn(     [retval,out] long *val);
    HRESULT shortReturn(    [retval,out] short *val);
    HRESULT stringReturn(   [retval,out] BSTR *val);
    HRESULT octetReturn(    [retval,out] DATE *val);
    HRESULT currencyReturn( [retval,out] CURRENCY *val);
}
```

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
```

19.9.11 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 * val);
   [propput] HRESULT shortTest([in] short sshortTest);
}

[odl, dual, uuid(...)]
interface DIMyModule_ObjRefTest: IDispatch
{
```

```
   [propget] HRESULT simpleTest([retval, out]
      DIMyModule_Simple ** val);
   [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 **val);
}
```

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);
};
```

19.9.12 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
        enum MyModule_color {red, green, blue};
        interface       foo: COM::CORBA_View {
            long op1(in MyModule_color col);
        };
};
```

Note – An ODL enumeration is mapped to OMG IDL such that the enumerators in the enumeration are ordered according to the ascending order of the value of the enumerators. Because OMG IDL does not support explicitly tagged enumerators, the CORBA view of an automation/dual object must maintain the mapping of the values of the enumeration.

19.9.13 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.

19.9.13.1 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.

19.9.14 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 done according to “Mapping for Enumerated Types” on page 19-17.

19.9.15 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 “Mapping for Automation Basic Data Types” on page 19-42.

19.9.16 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 exactly the same as for COM to CORBA (see Mapping for COM Errors under Section 18.2.10, “Interface Mapping,” on page 18-11. 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 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 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.

19.10 Older Automation Controllers

This section provides some solutions that vendors might implement to support existing and older Automation controllers. These solutions are suggestions; they are strictly optional.

19.10.1 Mapping for OMG IDL Arrays and Sequences to Collections

Some 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 * val);
   [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.

19.11 Example Mappings

19.11.1 Mapping the OMG Naming Service to 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 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)
```

19.11.2 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 18.3, “COM to CORBA Data Type Mapping,” on page 18-33 in the Mapping Com and CORBA chapter. 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 19-9.

Table 19-9 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";
};
```

19.11.3 Mapping an OMG Object Service to Automation

This section provides an example of mapping an OMG-defined interface to an equivalent Automation interface. This example is based on the OMG Naming Service and follows the mapping rules from the *Mapping: Automation and CORBA* chapter. The Naming Service is defined by two interfaces and some associated types, which are scoped in the *OMG IDL CosNaming* module.

Table 19-10 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 establish 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 SAFEAR-
RAY(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();
   };
}
```

Interoperability with non-CORBA Systems 20

The OMG documents used to create this chapter were orbos/97-09-06 and orbos/97-09-19.

Contents

This chapter contains the following sections.

20.1 Introduction

The primary goal of this specification is to allow effective access to CORBA servers through DCOM and the reverse. To reduce the total cost of ownership of CORBA applications that are built with COM or Automation clients for CORBA servers, COM or Automation clients on machines with no ORB or interworking mechanism should be

able to act as clients to CORBA servers through DCOM. In addition, a CORBA client could, through a CORBA view, access a DCOM server that is not co-located with the view with no additional interworking support on the DCOM server's machine. These scenarios help to reduce installation and maintenance costs through the lifetime of applications which span multiple object systems.

Note – This specification refers to COM/CORBA Part A and COM/CORBA Part B. The Interworking Architecture, Mapping: COM and CORBA, and Mapping Automation and CORBA chapters comprise the COM/CORBA Part A and this specification comprises the COM/CORBA Part B.

Converting a COM or Automation client to contact a server through DCOM is relatively easy and requires no application changes to the server. Thus, applications that use existing Part A compliant solutions could, today, have remote DCOM clients access the COM or Automation views of the CORBA servers and CORBA clients could access (through a view) DCOM or DCOM Automation servers. However, allowing CORBA access to CORBA views that are not co-located with the COM or Automation servers or allowing DCOM access to remote views of CORBA servers introduces a number of issues in terms of performance and scalability that will be discussed below.

20.1.1 COM/CORBA Part A

The COM/CORBA Part A specifications (see the Interworking Architecture chapter, Mapping: COM and CORBA chapter, and Mapping Automation and CORBA chapter) address most of the requirements of this Part B specification. The basic architecture and approach is sound. And, in general DCOM requires few changes to existing COM programs. With appropriate changes in the COM Registry, legacy COM client and server applications can operate unchanged in a DCOM environment. However, due to limitations of DCOM and DCOM Automation, a number of performance and scalability issues arise when interworking with CORBA using only the COM/CORBA Part A specification. The primary purpose of this specification is to address these issues; in particular this specification focuses on addressing the issues related to using native DCOM and DCOM Automation clients with CORBA servers. Note that readers are expected to be familiar with the terminology used in the other COM/CORBA specifications.

20.2 Conformance Issues

This specification, as a whole, is optional and is not required for COM/CORBA interworking compliance.

Solutions which choose to implement this specification must, in order to be conformant, implement the DCOM extent and all defined interfaces. There are no optional compliance points. Solutions which conform to this specification may label themselves as supporting *Advanced DCOM Interworking*.

20.2.1 Performance Issues

When accessing DCOM views of CORBA servers through DCOM (i.e., the DCOM client and DCOM view are not co-located), major performance issues arise for two primary reasons:

1. Pseudo objects are specific to CORBA and are thus not available in DCOM.

2. Automation does not support complex types such as structs and unions.

The COM/CORBA Part A specification maps CORBA pseudo objects into regular COM and Automation objects since there is no equivalent to pseudo objects in COM or Automation. In the Automation mapping, structs and unions are also mapped to objects since there is no Automation equivalent construct (essentially structs and unions are also handled as pseudo objects). When these pseudo objects are passed to a remote DCOM client that uses standard DCOM marshaling, all access to all members require a remote call. For example, a DCOM Automation client accessing the members of a structure would make one remote call for each get or set of a structure member. This, of course, introduces a significant performance bottleneck.

20.2.2 Scalability Issues

A scalability issue known as *proxy explosion* arises when passing object references among clients and servers across object systems. For example, an object reference is received from a CORBA server and is encapsulated in a DCOM view. This view is passed to a different DCOM server. This server then attempts to pass the object to a CORBA server. Without prior knowledge that the object was originally a CORBA object, a CORBA view would be built for what appeared to be a DCOM object (but which was really a view). This means that when the CORBA server attempts to call an operation on this object, it passes through a chain of views until the request is delivered to the real implementation instead of the call being direct CORBA to CORBA. In order to resolve the proxy explosion view chain problem, an efficient mechanism must be provided for interworking solutions to determine whether any object is a view or a native object and, if the object is a view, what is the original object behind the view. The problem or proxy explosion is not specific to COM/CORBA interworking. Instead, it can occur between CORBA and any other system where bidirectional interworking is supported.

The COM/CORBA Part A specification defines a mechanism to help avoid proxy chains using **`IForeignObject::GetForeignReference`**. However, calling this operation remotely on each object reference to avoid proxy chains would have introduced a significant performance problem.

20.2.3 CORBA Clients for DCOM Servers

In cases where CORBA clients need to access DCOM servers, the performance issues that occur in the other direction are not applicable since native DCOM servers do not have pseudo objects (since there is no such concept in COM or Automation) and since native Automation servers do not use structures or unions (since these constructs do not exist). However, the scalability issue remains.

20.3 Locality of the Bridge

The COM/CORBA Part A specification states that the interworking be performed local to the COM or Automation client or server since, at the time, COM objects could only communicate within the same machine. Thus, the possibility for the location of the view was limited to those in Figure 20-1.

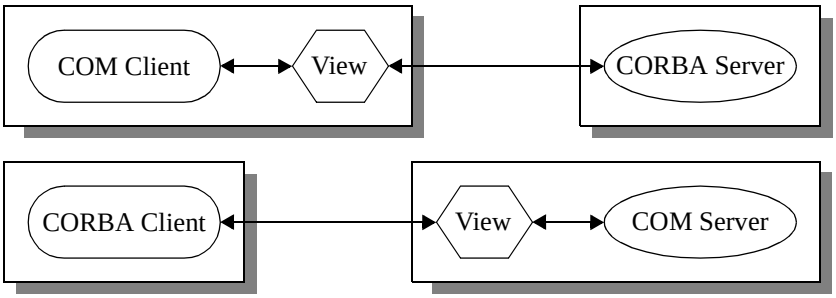


Figure 20-1 COM/CORBA Part A Configurations

The addition of support for DCOM removes the requirement that the interworking occur in the COM environment. The use of DCOM adds four possibilities for the location of the view (see Figure 20-2 on page 20-5). Note that the communications between the view and the CORBA server or client is still performed as per existing OMG specifications.

The performance issues described above relate, in particular, to the first and third configuration shown in Figure 20-2. The scalability issues can affect any of these configurations provided that objects are being passed through multiple different bridges or through an intermediate object system.

.

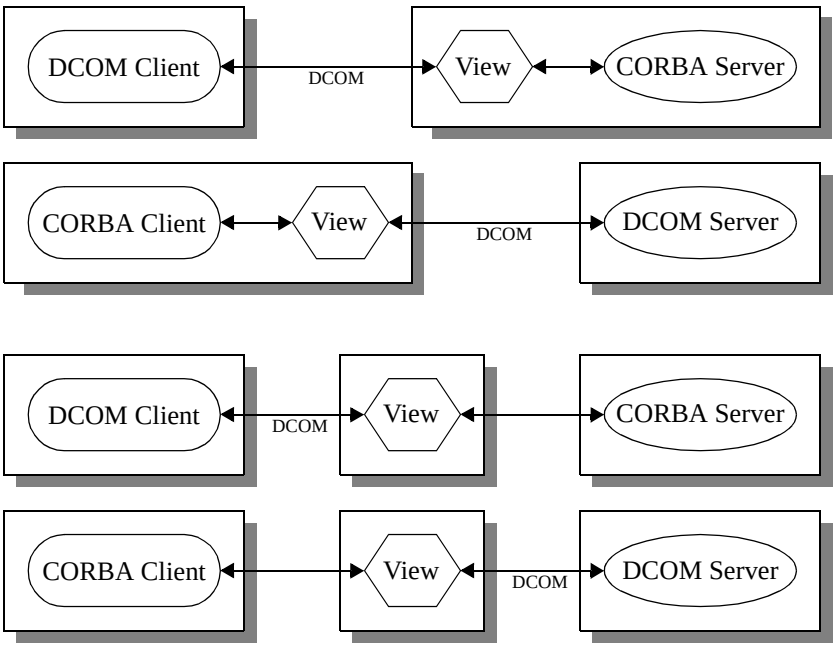


Figure 20-2 COM/CORBA Part B Additional Configurations

20.4 Extent Definition

The ideal solution to the performance issues would be to have DCOM able to pass CORBA pseudo objects with similar semantics to CORBA, and to have Automation support structs and unions natively. However, this is not likely to occur and cannot be implemented adequately using DCOM standard marshaling. In addition, since Part B is not required to be implemented, COM/CORBA Part A compliant solutions must still interoperate with solutions that also support the COM/CORBA Part B extensions. Thus, another mechanism needs to be defined in order to avoid the performance and scalability problems while still maintaining compatibility.

To handle all of these cases using a standard mechanism, a category of DCOM objects called *DCOM value objects* was defined. DCOM value objects are DCOM objects which have little or no behavior other than accessors for their underlying data. Proxies for DCOM value objects act as local caches for the information in the original object. The Automation and COM views of CORBA pseudo objects, as well as the Automation views of CORBA structs and unions, are all DCOM value objects.

Note – CORBA objects-by-value will be able to be viewed in DCOM as DCOM value objects.

When DCOM value objects are passed across DCOM systems the data of the DCOM value object, called the *value data*, is also passed. Systems that support DCOM value objects can use the passed data to improve performance. However, when a DCOM value object is passed to a system that does not support it as a value object, then the

DCOM value object is accessed remotely just as any other DCOM object would be. There are two types of DCOM value objects to support these semantics: 1) a *primary DCOM value object* which is the real (i.e., original) instance of the value object, and 2) *local DCOM value objects* which are the local proxies for the primary value object and caches for the values data of the primary value object. Note that the local DCOM value objects are essentially DCOM proxies with some methods (the ones that access the value data) implemented locally.

To implement DCOM value objects while still providing compatibility with systems that do not support DCOM value objects, the value data needs to be passed as, essentially "out-of-band" data. DCOM allows out-of-band data to be passed with requests in DCOM extents. DCOM extents are a standard DCOM feature used to pass additional data with a request. On the receiving end, if a handler is not available for the extent, it is ignored. Extents are similar to CORBA service contexts except that they are not propagated through a chain of calls.

DCOM value objects are passed in a DCOM extent. Receivers that recognize the extent can take advantage of the data it provides. Receivers that do not recognize the extend safely ignore it. This occurs with *no changes* at all to the standard marshaling packet. This allows DCOM developers to use standard DCOM tools and services instead of entirely custom special purpose solutions.

20.4.1 Marshaling Constraints

The layout of the marshaling packet is significant in matching marshaled data to a proxy on the receiving side. If the receiving side supports DCOM value objects for all passed value data, then the unmarshal process is simple: the first subset of value data goes to the first proxy (local DCOM value object) created by standard marshaling and so on. DCOM, however, allows for proxies in any given client process to be provided by different vendors. Thus, no assurance can be made that all DCOM value objects marshaled into the extent can have their value data unmarshaled on the receiving end. Thus, the value data in the extent is organized in a tree structure in order to be able to skip information that cannot be decoded.

20.4.2 Marshaling Key

The interface ID corresponds to the interface used to encapsulate the unmarshaled data. It must provide accessors for all the members that are being marshaled, in the order that they are marshaled. This interface ID may be different than the interface ID actually marshaled in the call, since it reflects the content of an object rather than the interface through which it is used at the time of the call. For instance, a class encapsulating a structure may be marshaled as an **`IUnknown`**, which will be the class ID in the standard marshaling packet, but this is of no help in unmarshaling the structure. Thus, this identifier is used to describe the marshaled members.

If the object is standard marshaled, the unmarshal class ID field should be **`CLSID_NULL`**. However, if an interface pointer is custom marshaled, its marshaling data does not contain a standard OBJREF which could be used by the proxy to recover the marshaled data (since nothing can be presumed about the way that the proxy will

be communicating with its server). In that case, the object's proxy will be different than the regular proxy for this interface, so the correct custom marshaler must be loaded if correct unmarshaling is to be achieved.

20.4.3 Extent Format

The marshaling format of the extent is best described using the following C++ structures. Note that the size of the extent is encoded in the extent header maintained by DCOM, so it does not have to be repeated here.

```
struct DVO_EXTENT          // DCOM value object extent
{
   HRESULT     statusCode;  // Status of marshaling
   DVO_IFACE   interfaces[];// Marshaled interfaces
};

struct DVO_IFACE            // value data container
                            // for 1 interface
{
   unsigned long  dataLen;  // Total length of packet data
   IID         remotedIID;  // Remoted interface
   CLSID       unmarshalCLSID;// Unmarshal class
   unsigned shortcImpl;    // Count of Implementations
   DVO_IMPLDATAimplData[]; // Marshaled implementations
};

struct DVO_IMPLDATA     // Marshaled implementation
{
   unsigned long  dataLen; // Length of data
   IID         iid;     // Implementation interface
   DVO_BLOB    data;    // Value data
   DVO_IFACE   interfaces[];// Recursive DVO interface
};

struct DVO_BLOB         // Opaque type containing
         // marshaled members
{
   unsigned long  dataLen; // Length of value data
   byte           data[];  // Value data
};
```

20.4.3.1 DVO_EXTENT

This structure contains the entire DCOM value object information for a given DCOM call. The size and ID of the extent are specified in the ORPC_EXTENT (DCOM defined) structure. The **`statusCode`** is used to pass error information, which cannot be returned normally between the client and server extent. The interfaces array, **`interfaces`**, contains the value data for each DCOM value object for the DCOM call.

The DCOM value object extent will be identified with the following GUID:

{106454c0-14b2-11d1-8a22-006097cc044d}

20.4.3.2 DVO_IFACE

This structure contains value data for a single DCOM value object. The **dataLen** member makes it easy to skip this structure; in doing so, one automatically skips any recursively marshaled interfaces. The **remotedIID** member identifies the most derived interface of the DCOM value object itself. The member **unmarshalCLSID** indicates the unmarshal class used in custom marshaling, if any.

The **cImpl** member indicates how many interface DCOM value object interfaces are marshaled. Normally, this member has a value of 1, but it may be necessary to send value data for more than one interface.

The **implData** array contains the blocks of marshaled value data.

20.4.3.3 DVO_IMPLDATA

This structure contains the value data of a DCOM value object. The value data corresponds to the DCOM value object identified by the **iid** member of the **DVO_IMPLDATA** structure. The value data is written to the **data** blob. If any marshaled data is itself a DCOM value object, its marshaling data will be added as an entry in the **interfaces** array.

20.4.3.4 DVO_BLOB

This contains the actual value data for the DCOM value object. The data has been marshaled using standard DCOM (NDR) marshaling.

20.5 Request/Reply Extent Semantics

Clients, which support the extent, add the extent to outgoing requests that have DCOM value objects which should have their value data transmitted. The **statusCode** member of the extent should be **NO_ERROR**. Even when the outgoing request does not contain any DCOM value objects, the client must still add the extent (consisting of just the **statusCode** member in this case) if it supports the extent at all.

Servers, which support the extent, can retrieve the information from the extent during unmarshaling to get the value data for the local DCOM value objects (DCOM proxies). If the unmarshaling of the data within the extent fails with an error, this error is returned in a corresponding reply extent containing the error which occurred. If the unmarshaling is successful, the request is processed and an extent is added to the reply. Any out parameter or return DCOM value objects are included in the reply extent. The **statusCode** member should be **NO_ERROR**. Even when the outgoing request does not contain any DCOM value objects, the callee must still add the extent (consisting of just the **statusCode** member in this case).

If the receiver of a DCOM value object passes a reference to the object to another client/server, the object reference of the primary DCOM value object should be marshaled in the request, not the object reference for the local DCOM value object.

20.6 Consistency

If the client supports the DCOM value object semantics for a given object reference, then an in-process copy of the value data is created using the data from the extent, and all read accesses are performed with no network calls.

When all clients and servers support the DCOM value object semantics, changes made to a local copy of the object can then be passed to other clients or servers. However, since the implementation of this specification is optional, it cannot be assumed that all clients and servers support this feature.

If the client of a DCOM value object does not support the extent, or the appropriate support for a given DCOM value object to be unmarshaled locally, then all reads or writes to members of the object are transmitted over the network to the server, which originally provided the object reference.

In cases where the receiver modifies the local copy of the object, these changes must be propagated back to the server to maintain consistency between systems that support the DCOM value object and those that do not.

The interfaces used to manage consistency were designed so that applications on homogenous networks (where every interworking solution supports Part B) can disable the synchronization used to maintain consistency. Applications running on heterogeneous networks can control the synchronization behavior to best suit the needs of the application.

In cases where the receiver modifies the local DCOM value object, these changes must be propagated back to the server to maintain consistency between systems that support DCOM value objects and those that do not. To maintain consistency, three additional DCOM interfaces are defined:

```
[
   object,
   pointer_default(unique),
   uuid(c9362b80-14bd-11d1-8a22-006097cc044d)
]
interface IValueObject : IUnknown
{
   HRESULT GetValue( [out] unsigned long        *length,
                     [out, size_is(,*length)] byte**data);

   HRESULT PutValue( [in] unsigned long         length,
                     [in, size_is(length)] byte *data);
};

typedef enum tagSyncronizeMode
{
```

```
    kNeverSync,
    kSyncOnSend,
    kSyncOnChange
} SyncronizeMode;

[
    object,
    pointer_default(unique),
    uuid(c82fb800-14bd-11d1-8a22-006097cc044d)
]
interface ISynchronize : IUnknown
{
    HRESULT get_Mode([out, retval] SyncronizeMode *mode);
    HRESULT put_Mode([in] SyncronizeMode mode);
    HRESULT SyncNow();
    HRESULT ReCopy();

};

[
    odl,
    dual,
    oleautomation,
    uuid(c8c84e80-14bd-11d1-8a22-006097cc044d)
]
interface DISynchronize : IDispatch
{
    [propget] HRESULT Mode([out, retval] SyncronizeMode
*mode);
    [propput] HRESULT Mode([in] SyncronizeMode mode);
    HRESULT SyncNow();
    HRESULT ReCopy();
};
```

20.6.1 IValueObject

This interface is implemented on the primary DCOM value object. The purpose of this interface is to allow batch updates of the value data of the object. The data contained within the data array for the **`GetValue`** and **`PutValue`** methods is a **`DVO_IFACE`** marshaled according to “Extent Definition” on page 20-5.

Local DCOM value objects that are not primary DCOM value objects are not required to support this interface.

20.6.2 ISynchronize and DISynchronize

These interfaces are implemented on local DCOM value objects (**`ISynchronize`** is found on COM proxies, **`DISynchronize`** is found on Automation proxies). If the interface is available, it means that this is a local DCOM value object, not a regular object or a primary DCOM value object.

20.6.2.1 Mode Property

The property **`Mode`** is used to control when synchronization is done. A value of **`kNeverSync`** means that the local and the primary value objects are never synchronized.

A value of **`kSyncOnSend`** means that, if the local value object has been changed, the primary value object will be synchronized with the local value object when the local value object is sent to another client/server which cannot be reliably determined to support the required DCOM value object. Implementations can choose to synchronize using either batch synchronization through a call to **`IValueObject`**, or through calls for each changed member through the regular remote interface.

A value of **`kSyncOnChange`** means that, as a member is changed, the update of the member should be propagated to the primary value object as a regular remote call.

20.6.2.2 SyncNow Method

The **`SyncNow`** method can be called by application code to force the changes to the local value object to be propagated to the primary value object. Implementations can choose to synchronize using either batch synchronization through a call to **`IValueObject`**, or through calls for each changed member through the regular remote interface.

20.6.2.3 ReCopy Method

The **`ReCopy`** method can be called by application code to retrieve the current value of the primary value object and update the local value object.

20.7 DCOM Value Objects

20.7.1 Passing Automation Compound Types as DCOM Value Objects

Compound types such as structures and unions are encapsulated in Automation classes so they may be used by Automation applications. These are DCOM value objects. When a DCOM value object representing a compound type is passed to a remote client, its interface pointer is marshaled using standard marshaling (as with any DCOM value object), and its value data is forwarded simultaneously using the extent described in“Extent Definition” on page 20-5.

20.7.2 Passing CORBA-Defined Pseudo-Objects as DCOM Value Objects

To handle the DCOM views of CORBA pseudo objects as DCOM value objects, the memory representation of these data types must be defined. The following sections detail the value data which will be passed in the extent.

20.7.3 IForeignObject

Supporting **IForeignObject**'s as a DCOM value object is required to avoid proxy explosion. The marshaled data for value objects of type **IForeignObject** is described in Section 20.8.2, "COM Chain Avoidance," on page 20-17.

20.7.4 DIForeignComplexType

The value data for DCOM value objects of type **DIForeignComplexType** can be represented by the following structure (note that this also includes the state for **DIObjectInfo**):

```
struct FOREIGN_COMPLEX
{
   LPSTR       name;          // Name of type
   LPSTR       scopedName;    // Scoped name (if available)
   LPSTR       repositoryId;  // Repository ID of type
};
```

20.7.5 DIForeignException

The value data for DCOM value objects of type **DIForeignException** can be represented by the following structure:

```
struct FOREIGN_EXCEPTION
{
   FOREIGN_COMPLEX   base;
   long              majorCode;
};
```

20.7.6 DISystemException

The value data for DCOM value objects of type **DISystemException** can be represented by the following structure:

```
struct CORBA_SYSTEM_EXCEPTION
{
   FOREIGN_EXCEPTION base;
   long              minorCode;
   long              completionStatus;
};
```

20.7.7 DICORBAUserException

The value data for **DICORBAUserException** is identical to that of **DIForeignException**. Value objects deriving from **DICORBAUserException** are passed as DCOM value objects according to the previously defined format. The value data of exception members must be preceded by the value data of **DIForeignException**.

20.7.8 DICORBAStruct

The value data for **DICORBAStruct** is identical to that of **DIForeignComplexType**. Value objects deriving from **DICORBAStruct** are passed as DCOM value objects according to the previously defined format. The value data of struct members must be preceded by the value data of **DIForeignComplexType**.

20.7.9 DICORBAUnion

The value data for **DICORBAUnion** is identical to that of **DIForeignComplexType**. Value objects deriving from **DICORBAUnion** are passed as DCOM value objects according to the previously defined format. The value data of a union must be preceded by the value data of **DIForeignComplexType**. The value data for the union itself is the discriminant followed by the selected member, if any.

20.7.10 DICORBATypeCode and ICORBATypeCode

The value data for type code DCOM value objects can be represented by the following struct:

```
struct CORBA_TYPECODE
{
   FOREIGN_COMPLEX   base;
   TCKind            kind; // TypeCode kind

   union TypeSpecific switch(kind)
   {
      case tk_objref:
         LPSTR       id;
         LPSTR       name;

      case tk_struct:
      case tk_except:
         LPSTR       id;
         LPSTR       name;
         long        member_count;
         [size_is(member_count,)] LPSTR   *member_names;
         [size_is(member_count,)] IUnknown**member_types;

      case tk_union:
```

```
            LPSTR       id;
            LPSTR       name;
            long        member_count;
            LPSTR       member_names[];
            [size_is(member_count,)] IUnknown**member_types;
            [size_is(member_count)] VARIANT  *member_labels;
            IUnknown    *discriminator_type;
            long        default_index;

        case tk_enum:
            long        member_count;
            [size_is(member_count,)] LPSTR   *member_names;
            [size_is(member_count,)] IUnknown**member_types;

        case tk_string:
            long        length;

        case tk_array:
        case tk_sequence:
            long        length;
            IUnknown    *content_type;

        case tk_alias:
            LPSTR       id;
            LPSTR       name;
            long        length;
            IUnknown    *content_type;
    }
};
```

Note that members of type **IUnknown** will actually be **ICORBATypeCode** instances for COM and **DICORBATypeCode** instances for Automation.

20.7.11 DICORBAAny

The value data for DCOM value objects of type **DICORBAAny** can be represented by the following structure:

```
struct CORBA_ANY_AUTO
{
    FOREIGN_COMPLEXbase;
    VARIANT        value;
    DICORBATypeCode*typeCode;
};
```

20.7.12 ICORBAAny

The value data for DCOM value objects of type **ICORBAAny** can be represented by a **CORBAAnyDataUnion** as defined in COM/CORBA Part A.

20.7.13 User Exceptions In COM

In COM, all CORBA user exceptions used in an interface are represented by another interface which contains one method per user exception. The value data for one of these exception interfaces is an encapsulated DCOM union where each member of the union is one of the exception definition structures. The discriminant values are the indices of the corresponding structure retrieval method from the user exception interface.

```
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);
        };
};
```

Per the COM/CORBA Part A specification, the above IDL results in the following interface used for user exceptions:

```
struct Bank_InsufFunds { float balance; };
struct Bank_InvalidAmount { float amount; };
struct Bank_Account_NotAuthorized {};

interface IBank_AccountUserExceptions : IUnknown
{
   HRESULT get_InsufFunds([out] Bank_InsufFunds *);
   HRESULT get_InvalidAmount([out] Bank_InvalidAmount *);
   HRESULT get_NotAuthorized([out]
Bank_Account_NotAuthorized *);
};
```

When this DCOM value object is passed, the value data is marshaled as the following data structure:

```
union Bank_AccountUserExceptionsData switch(unsigned short)
{
   case 0:  Bank_InsufFunds            m0;
   case 1:  Bank_InvalidAmount         m1;
   case 2:  Bank_Account_NotAuthorized m2;
};
```

20.8 Chain Avoidance

To avoid view chaining (and thus proxy explosion), we define a general mechanism to carry chain information along with object references. This mechanism is defined in both COM and in CORBA to allow for bidirectional chain avoidance. Views in either system carry this information along with their object references. For example, the information carried in the object reference to a CORBA view of an Automation object would describe the object referred to by the view (i.e., information about the Automation object).

20.8.1 CORBA Chain Avoidance

In CORBA, the chain avoidance information is carried as an IOP profile in an object reference which is part of a chain.

```
module CosBridging
{
   typedef sequence<octet> OpaqueRef;
   typedef sequence<octet> OpaqueData;
   typedef unsigned long   ObjectSystemID;

   interface Resolver
   {
      OpaqueRef Resolve(in ObjectSystemID objSysID,
                        in unsigned long  chainDataFormat,
                        in octet          chainDataVersion,
                        in OpaqueData     chainData);
   };

   struct ResolvableRef
   {
      Resolver       resolver;
      ObjectSystemID objSysID;
      unsigned long  chainDataFormat;
      octet          chainDataVersion;
      OpaqueData     chainData;
   };

   typedef sequence<ResolvableRef> ResolvableChain;

   struct BridgingProfile
   {
      ResolvableChainchain;
   };
};
```

The content of the profile is defined as a single **`BridgingProfile`** structure. The ID for this profile will be allocated by the OMG. The profile structure contains a sequence of **`ResolvableRef`** structures, potentially one for each object system in the chain.

The **ResolvableRef** structure contains a **resolver** CORBA object reference which can be called at runtime through its **Resolve** method to return an opaque (because it is not CORBA) object reference for the specified link in the chain. The link in the chain is identified by the object system ID, **objSysID**.

Currently defined object system IDs are: 1 for CORBA, 2 for Automation, and 3 for COM. IDs in the range from 0 through 100000 are reserved for use by the OMG for future standardization.

The **ResolvableRef** structure also contains information which can be used by the **resolver** as context to find the appropriate information to return. While this chain data is opaque, it is also tagged with a format identifier so that bridges which understand the format can directly interpret the contents of **chainData** instead of making a remote call to **Resolve**. The only currently defined format tag is 0 which is currently defined as *private*; that is, **chainData** tagged as private cannot be directly interpreted and must be passed to the resolver for interpretation. All other format tags are specific to each object system. Format tags in the range of 1 to 100000 are reserved for allocation by the OMG.

The result of calling the **Resolve** method on a COM or Automation **ResolvableRef** is an NDR marshaled DCOM object reference with at least one strong reference.

20.8.2 COM Chain Avoidance

A similar approach is adopted to resolve the same chain avoidance issues in COM; however, since DCOM does not support profiles, the implementation is different. The information for chain avoidance (also used by **IForeignObject** and **IForeignObject2**) is provided as DCOM value data associated with each passed view object. This information is represented by a **ResolvableRefChain**.

```
struct OpaqueRef
{
   unsigned long              len;
   unsigned long              maxlen;
   BYTE [size_is(len)]        *data;
};

struct OpaqueData
{
   unsigned long              len;
   unsigned long              maxlen;
   BYTE [size_is(len)]        *data;
};

typedef unsigned long ObjectSystemID;

struct ResolvableRef
{
   IResolver                  resolver;
```

```
   ObjectSystemID             objSysID;
   unsigned long              chainDataFormat;
   BYTE                       chainDataVersion;
   OpaqueData                 chainData;
};

struct ResolvableRefChain
{
   unsigned long              len;
   unsigned long              maxlen;
   ResolvableRef [size_is(len,)]**data;
};

[
   object,
   pointer_default(unique),
   uuid(5473e440-20ac-11d1-8a22-006097cc044d)
]
interface IResolver : IUnknown
{
   OpaqueRef Resolve([in] ObjectSystemIDobjSysID,
                     [in] unsigned longchainDataFormat,
                     [in] BYTE         chainDataVersion,
                     [in] OpaqueData   chainData);
};

[
   object,
   pointer_default(unique),
   uuid(60674760-20ac-11d1-8a22-006097cc044d)
]
interface IForeignObject2 : IForeignObject
{
   ResolvedRefChain ChainInfo();
};
```

The use semantics of the resolver is identical to the use semantics described in Section 20.8.1, “CORBA Chain Avoidance,” on page 20-16. One format tag with value 1 is defined for a **`ResolvableRef`** with **`objSysID`** 1 (CORBA). If the format tag is 1, the **`chainDataVersion`** must be 0 and the **`chainData`** contains a (CDR marshaled) byte defining the byte ordering for the rest of the **`chainData`** (the byte value is identical to that used to encode GIOP messages) followed by a CDR marshaled object reference. If **`Resolve`** were called for this **`ResolvableRef`**, the same value as contained in the **`chainData`** would be returned by **`Resolve`** (i.e., a CDR-marshaled object reference).

In addition to this mechanism, the interface **`IForeignObject2`** is defined on COM or Automation views to return the **`ResolvableRefChain`** in cases where this information has been lost.

20.9 Chain Bypass

Using the chain avoidance technique defined in this specification, the formation of view chains can be avoided. However, there are cases where the chain avoidance information carried with the object references may have been discarded (for instance, as the object reference is passed through a firewall). In this case, chaining of views cannot be avoided without an explicit performance hit which was deemed unacceptable. However, at the point when the first invocation is performed, information about the current chain can be returned as out-of-band data. This information can then be used on subsequent invocations to bypass as many views as possible in order to avoid the performance hit of multiple view translations.

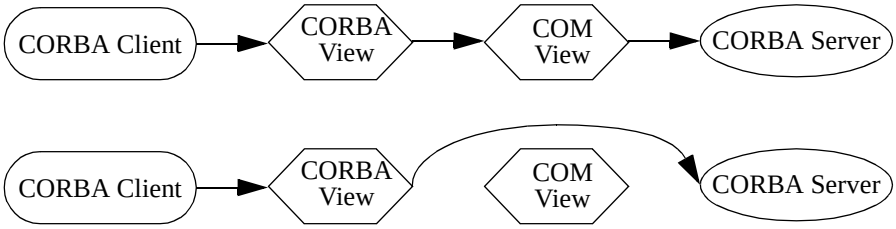


Figure 20-3 Invocation With and Without Chain Bypass

Figure 20-3 shows an example of a call that does not perform chain bypass followed by one that does. Note that chain bypass cannot eliminate all unnecessary calls since the client already has a reference to the view (not to the original object) and thus invokes an operation on the view. It is the responsibility of the view to perform the chain bypass if it so chooses -- in this case the view essentially becomes a rebindable stub.

20.9.1 CORBA Chain Bypass

For views to discover the chain information, two service contexts are defined as follows:

```
ChainBypassCheck = 2
ChainBypassInfo = 3

module CosBridging
{
   struct ResolvedRef
   {
      Resolver       resolver;
      ObjectSystemID objSysID;
      unsigned long  chainDataFormat;
      octet          chainDataVersion;
      OpaqueData     chainData;
      OpaqueRef      reference;
      };
```

```
   typedef sequence<ResolvedRef> ResolvedRefChain;

   struct ChainBypassCheck // Outgoing service context
   {
      Object   objectToCheck;
   };

   struct ChainBypassInfo // Reply service context
   {
      ResolvedRefChain  chain;
   };
};
```

The **ChainBypassCheck** service context is sent out with the first outgoing (non-oneway) request. Since the service context is propagated automatically to subsequent calls, an object is provided in the service context to avoid returning chain information for an incorrect object. For a reply service context to be generated, the object in the service context must match the object (a view) being invoked.

If a reply service context, **ChainBypassInfo**, is received with the reply message, then a view has been detected. The information in the **ResolvedRefChain** can be used to bypass intermediate views. Each **ResolvedRef** is identical to a **ResolvableRef** except that it also contains the result of the resolution -- the **reference** member contains the data that would be returned if **Resolve** were called on the included resolver. If the reference field of **ResolvedRef** is an empty sequence, then the marshaled object reference is assumed to be identical to the **chainData**.

20.9.2 COM Chain Bypass

The technique used for COM chain bypass is very similar to the technique used in CORBA. The only difference is the result of the fact that DCOM extents are not propagated into subsequent calls unlike CORBA service contexts.

```
struct ResolvedRef
{
   IResolver                  resolver;
   ObjectSystemID             objSysID;
   unsigned long              chainDataFormat;
   BYTE                       chainDataVersion;
   OpaqueData                 chainData;
   OpaqueRef                  reference;
};

struct ResolvableRefChain
{
   unsigned long              len;
   unsigned long              maxlen;
   ResolvableRef [size_is(len,)]**data;
};
```

```
struct ChainBypassCheck    // Outgoing extent body
{
};

struct ChainBypassInfo     // Reply extent body
{
   ResolvableRefChain      chain;
};
```

The **ChainBypassCheck** extent is sent out with the first outgoing request. If a reply extent, **ChainBypassInfo**, is received with the reply message, then a view has been detected. The information in the **ResolvedRefChain** can be used to bypass intermediate views. Each **ResolvedRef** is identical to a **ResolvableRef** except that it also contains the result of the resolution -- the **reference** member contains the data that would be returned if **Resolve** were called on the included resolver. If the reference field of **ResolvedRef** is an empty sequence, then the marshaled object reference is assumed to be identical to the **chainData**.

The UUID for the request and reply extents are both:

1eba96a0-20b1-11d1-8a22-006097cc044d

20.10 Thread Identification

To correlate incoming requests with previous outgoing requests, DCOM requires a *causality ID*. The identifier is essentially a logical thread ID used to determine whether an incoming request is from an existing logical thread or is a different logical thread of execution. CORBA, on the other hand, does not strictly require a logical thread ID. To maintain the logical thread ID as requests pass through both DCOM and CORBA, we define a general purpose service context, which can be used to maintain logical thread identifiers for any system a thread of execution passes through.

```
module CosBridging
{
   struct OneThreadID
   {
      ObjectSystemID objSysID;
      OpaqueData     threadID;
   };

      typedef sequence<OneThreadID> ThreadIDs;

   struct LogicalThreadID  // Service context
   {
      ThreadIDs      IDs;
   };
};
```

The logical thread ID information is propagated through a CORBA call chain using a service context (IDs to be assigned by the OMG) containing the **`LogicalThreadID`** structure.

For future use, a DCOM extent is defined to allow the same logical thread identification information to be passed through a DCOM call chain. If the OMG chooses to standardize a logical thread ID format for CORBA, this can be passed through a DCOM call chain using this extent.

```
struct OneThreadID
{
   ObjectSystemID objSysID;
   OpaqueData     threadID;
};

struct ThreadIDs
{
   unsigned long              len;
   unsigned long              maxlen;
   OneThreadID [size_is(len)] *data;
};

struct LogicalThreadID  // DCOM extent
{
   ThreadIDs      IDs;
};
```

This extent, used for passing logical thread IDs, is identified by the following UUID:

```
   f81f4e20-2234-11d1-8a22-006097cc044d
```

Interceptors 21

Contents

This chapter contains the following sections.

21.1 Introduction

This chapter defines ORB operations that allow services such as security to be inserted in the invocation path. Interceptors are not security-specific; they could be used to invoke any ORB service. These interceptors permit services internal to the ORB to be cleanly separated so that, for example, security functions can coexist with other ORB services such as transactions and replication.

Interceptors are an optional extension to the ORB to allow implementation of the Replaceable Security option defined in the Security Service specification (Chapter 15 of CORBA Services).

21.1.1 ORB Core and ORB Services

The ORB Core is defined in the CORBA architecture as “that part of the ORB which provides the basic representation of objects and the communication of requests.” ORB Services, such as the Security Services, are built on this core and extend the basic functions with additional qualities or transparencies, thereby presenting a higher-level ORB environment to the application.

The function of an ORB service is specified as a transformation of a given message (a request, reply, or derivation thereof). A client may generate an object request, which necessitates some transformation of that request by ORB services (for example, Security Services may protect the message in transit by encrypting it).

21.2 Interceptors

An interceptor is responsible for the execution of one or more ORB services. Logically, an interceptor is interposed in the invocation (and response) path(s) between a client and a target object. When several ORB services are required, several interceptors may be used.

Two types of interceptors are defined in this specification:

- **Request-level interceptors**, which execute the given request.
- **Message-level interceptors**, which send and receive messages (unstructured buffers) derived from the requests and replies.

Interceptors provide a highly flexible means of adding portable ORB Services to a CORBA-compliant object system. The flexibility derives from the capacity of a binding between client and target object to be extended and specialized to reflect the mutual requirements of client and target. The portability derives from the definition of the interceptor interface in OMG IDL.

The kinds of interceptors available are known to the ORB. Interceptors are created by the ORB as necessary during binding, as described next.

21.2.1 Generic ORB Services and Interceptors

An Interceptor implements one or more ORB services. Logically, an interceptor is interposed in the invocation (and response) path(s) between a client and target object. There are two types of interceptors:

- **Request-level interceptor**, which perform transformations on a structured request.
- **Message-level interceptors**, which perform transformations on an unstructured buffer.

Figure 21-1 shows interceptors being called during the path of an invocation.

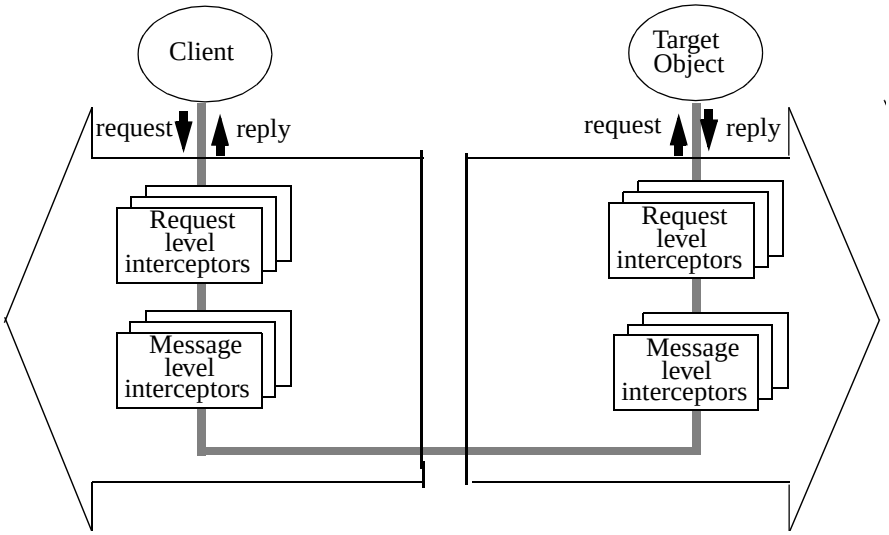


Figure 21-1 Interceptors Called During Invocation Path

21.2.2 Request-Level Interceptors

Request-level interceptors are used to implement services which may be required regardless of whether the client and target are collocated or remote. They resemble the CORBA bridge mechanism in that they receive the request as a parameter, and subsequently re-invoke it using the Dynamic Invocation Interface (DII). An example of a request-level interceptor is the Access Control interceptor, which uses information about the requesting principal and the operation in order to make an access control decision.

The ORB core invokes each request-level interceptor via the **`client_invoke`** operation (at the client) or the **`target_invoke`** operation (at the target) defined in this section. The interceptor may then perform actions, including invoking other objects, before re-invoking the (transformed) request using **`CORBA::Request::invoke`**. When the latter invocation completes, the interceptor has the opportunity to perform other actions, including recovering from errors and retrying the invocation or auditing the result if necessary, before returning.

21.2.3 Message-Level Interceptors

When remote invocation is required, the ORB will transform the request into a message, which can be sent over the network. As functions such as encryption are performed on messages, a second kind on interceptor interface is required.

The ORB code invokes each message-level interceptor via the **`send_message`** operation (when sending a message, for example, the request at the client and the reply at the target) or the **`receive_message`** operation (when receiving a message). Both have a message as an argument. The interceptor generally transforms the message and

then invokes **`send`**. Send operations return control to the caller without waiting for the operation to finish. Having completed the **`send_message`** operation, the interceptor can continue with its function or return.

21.2.4 Selecting Interceptors

An ORB that uses interceptors must know which interceptors may need to be called, and in what order they need to be called. An ORB that supports interceptors, when serving as a client, uses information in the target object reference, as well as local policy, to decide which interceptors must actually be called during the processing of a particular request sent to a particular target object.

When an interceptor is first invoked, a bind time function is used to set up interceptor binding information for future use.

21.3 Client-Target Binding

The selection of ORB Services is part of the process of establishing a binding between a client and a target object.

A binding provides the context for a client communicating with a target object via a particular object reference. The binding determines the mechanisms that will be involved in interactions such that compatible mechanisms are chosen and client and target policies are enforced. Some requirements, such as auditing or access control, may be satisfied by mechanisms in one environment, while others, such as authentication, require cooperation between client and target. Binding may also involve reserving resources in order to guarantee the particular qualities of service demanded.

Although resolution of mechanisms and policies involves negotiation between the two parties, this need not always involve interactions between the parties as information about the target can be encoded in the object reference, allowing resolution of the client and target requirements to take place in the client. The outcome of the negotiation can then be sent with the request, for example, in the GIOP service context. Where there is an issue of trust, however, the target must still check that this outcome is valid.

The binding between client and target at the application level can generally be decomposed into bindings between lower-level objects. For example, the agreement on transport protocol is an agreement between two communications endpoints, which will generally not have a one-to-one correspondence to application objects. The overall binding therefore includes a set of related sub-bindings which may be shared, and also potentially distributed among different entities at different locations.

21.3.1 Binding Model

No object representing the binding is made explicitly visible since the lifetime of such an object is not under the control of the application, an existing binding potentially being discarded, and a new one made without the application being aware of the fact.

Instead, operations that will affect how a client will interact with a target are provided on the **`Object`** interface and allow a client to determine how it will interact with the target denoted by that object reference. On the target side, the binding to the client may be accessed through the **`Current`** interface. This indirect arrangement permits a wide range of implementations that trade the communication and retention of binding information in different ways.

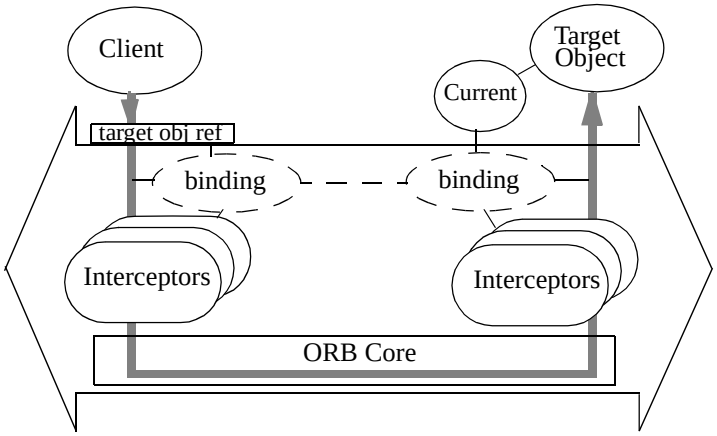


Figure 21-2 Binding Model

The action of establishing a binding is generally implicit, occurring no later than the first invocation between client and target. It may be necessary for a client to establish more than one binding to the same target object, each with different attributes (for example, different security features). In this case, the client can make a copy of the object reference using **`Object::duplicate`** and subsequently specify different attributes for that reference.

The scope of attributes associated with an object reference is that of the object reference instance (i.e., the attributes are *not* copied if the object reference is used as an argument to another operation or copied using **`Object::duplicate`**). If an object reference is an *inout* argument, the attributes will still be associated with the object reference after the call if the reference still denotes the same object, but not otherwise.

21.3.2 Establishing the Binding and Interceptors

An ORB maintains a list of interceptors, which it supports, and when these are called. Note that at the client, when handling the request, the request-level interceptors are always called before the message level ones, while at the target the message-level ones are called first.

When the ORB needs to bind an object reference, it refers to the characteristics of the target object and relates this to the types of interceptor it supports. From this it determines the appropriate type of interceptor to handle the request and creates it, passing the object reference in the call. (No separate interceptor initialization operation

is used. The **client_invoke/target_invoke** or **send_message/receive_message** calls are used both for the first invocation and for subsequent ones.)

When an interceptor is created, it performs its bind time functions. These may involve getting the policies that apply to the client and to the target. This could involve communicating with the target, for example, a secure invocation interceptor setting up a security association. Note that the ORB Core itself is unaware of service-specific policies. In addition to performing its specific functions, the interceptor must continue the request by invoking object(s) derived from the given object reference.

The interceptors themselves maintain per-binding information relevant to the function they perform. This information will be derived from:

- The policies that apply to the client and target object because of the domains to which they belong, for example the access policies, default quality of protection.
- Other static properties of the client and target object such as the security mechanisms and protocols supported.
- Dynamic attributes, associated with a particular execution context or invocation (for example, whether a request must be protected for confidentiality).

If the relevant client or target environment changes, part or all of a binding may need to be reestablished. For example, the secure invocation interceptor may detect that the invocation credentials have changed and therefore needs to establish a new security association using the new credentials. If the binding cannot be reestablished, an exception is raised to the application, indicating the cause of the problem.

Similarly, at the target, the ORB will create an instance of each interceptor needed there. A single interceptor handles both requests and replies at the client (or target), as these share context information.

21.4 Using Interceptors

When a client performs an object request, the ORB Core uses the binding information to decide which interceptors provide the required ORB Services for this client and target as described in “Establishing the Binding and Interceptors” on page 21-5.

21.4.1 Request-Level Interceptors

Request-level interceptors could be used for services such as transaction management, access control, or replication. Services at this level process the request in some way. For example, they may transform the request into one or more lower-level invocations or make checks that the request is permitted. The request-level interceptors, after performing whatever action is needed at the client (or target), reinvoke the (transformed) request using the Dynamic Invocation Interface (DII) **CORBA::Request::invoke**. The interceptor is then stacked until the invocation completes, when it has an opportunity to perform further actions, taking into account the response before returning.

Interceptors can find details of the request using the operations on the request as defined in the Dynamic Skeleton interface in CORBA 2. This allows the interceptor to find the target object[1], operation name, context, parameters, and (when complete) the result.

If the interceptor decides not to forward the request, for example, the access control interceptor determines that access is not permitted, it indicates the appropriate exception and returns.

When the interceptor resumes after an inner request is complete, it can find the result of the operation using the **`result`** operation on the Request object, and check for exceptions using the **`exception`** operation before returning.

21.4.2 Message-Level Interceptors

When remote invocation is required, the ORB will transform the request into a message that can be sent over the network. Message-level interceptors operate on messages in general without understanding how these messages relate to requests (for example, the message could be just a fragment of a request). Note that the message interceptors may achieve their purpose not by just transforming the given message, but by communicating using their own message (for example, to establish a secure association). Fragmentation and message protection are possible message-level interceptors.

`Send_message` is always used when sending a message, so it is used by the client to send a request (or part of a request), and by the target to send a reply.

When a client message-level interceptor is activated to perform a **`send_message`** operation, it transforms the message as required, and calls a **`send`** operation to pass the message on to the ORB and hence to its target. Unlike invoke operations, **`send`** operations may return to the caller without completing the operation. The interceptor can then perform other operations if required before exiting. The client interceptor may next be called either using **`send_message`** to process another outgoing message, or using **`receive_message`** to process an incoming message.

A target message-level interceptor also supports **`send_message`** and **`receive_message`** operations, though these are obviously called in a different order from the client side.

21.5 Interceptor Interfaces

Two interceptor interfaces are specified, both used only by the ORB:

---

1. It is assumed that the target object reference is available, as this is described in the C++ mapping for DSI, though not yet in the OMG IDL.

- **RequestInterceptor** for operations on request-level interceptors. Two operations are supported:
  - **`client_invoke`** for invoking a request-level interceptor at the client.
  - **`target_invoke`** for invoking a request-level interceptor at the target.
- **MessageInterceptor** for operations on message-level interceptors. Two operations are supported:
  - **`send_message`** for sending a message from the client to the target or the target to the client.
  - **`receive_message`** for receiving a message.

Request-level interceptors operate on a representation of the request itself as used in the CORBA Dynamic Invocation and Skeleton interfaces.

21.5.1 Client and Target Invoke

These invoke a request-level interceptor at the client or target. Both operations have identical parameters and return values.

```
module CORBA {
    interface RequestInterceptor: Interceptor {// PIDL
        void client_invoke (
            inout   CORBA::Request      request
        );
        void target_invoke (
            inout   CORBA::Request      request
        );
    };
};
```

Parameters

request - The request being invoked. This is defined in the Dynamic Invocation Interface. After invocation, output parameters and the associated result and exceptions may have been updated.

21.5.2 Send and Receive Message

These invoke a message-level interceptor to send and receive messages. Both operations have identical parameters and return values.

```
module CORBA {
    native Message;
    interface MessageInterceptor: Interceptor {// PIDL
        void send_message (
            in      Object              target,
            in      Message             msg
        );
        void receive_message (
```

```
            in      Object          target,
            in      Message         msg
        );
    };
};
```

Parameters

target - The target object reference.

Note – The target here may not be the same as seen by the application. For example, a replication request-level interceptor may send the request to more than one underlying object.

---

msg - The message to be handled by this interceptor.

21.6 IDL for Interceptors

```
module CORBA {
    interface Interceptor {};                                   // PIDL
    interface RequestInterceptor: Interceptor {// PIDL
        void client_invoke (
            inout   Request                 request
        );
        void target_invoke (
            inout   Request                 request
        );
    };
    interface MessageInterceptor: Interceptor {// PIDL
        void send_message (
            in      Object                  target,
            in       Message                msg
        );
        void receive_message (
            in      Object                  target,
            in      Message                 msg
        );
    };
};
```

OMG IDL Tags *A*

This appendix lists the standardized profile, service, component, and policy tags described in the CORBA documentation. 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*.

Table A-1 Profile Tags

Tag Name	Tag Value	Described in
ProfileId	TAG_INTERNET_IOP = 0	Section 13.6.2, “Interoperable Object References: IORs,” on page 13-15
ProfileId	TAG_MULTIPLE_COMPONENTS = 1	Section 13.6, “An Information Model for Object References,” on page 13-15

Table A-2 Service Tags

Tag Name	Tag Value	Described in
ServiceId	TransactionService = 0	CORBAservice - Transaction Service
ServiceId	CodeSets = 1	Section 13.7.2, “Code Set Conversion Framework,” on page 13-30
ServiceId	ChainBypassCheck = 2	Section 20.9, “Chain Bypass,” on page 20-19
ServiceId	ChainBypassInfo = 3	Section 20.9, “Chain Bypass,” on page 20-19
ServiceId	LogicalThreadId = 4	Section 20.10, “Thread Identification,” on page 20-21
ServiceId	BI_DIR_IIOP = 5	Section 15.8, “Bi-Directional GIOP,” on page 15-52

Table A-2 Service Tags

Tag Name	Tag Value	Described in
ServiceId	SendingContextRunTime = 6	Section 5.6, “Access to the Sending Context Run Time,” on page 5-15
ServiceId	UnknownExceptionInfo = 9	Java to IDL Language Mapping: Section 1.4.8, “Mapping CORBA System Exceptions to RMI Exceptions,” on page 1-31

Note – For more information on INVOCATION_POLICIES refer to the Asynchronous Messaging specification - orbos/98-05-05. For information on FORWARDED_IDENTITY refer to the Firewall specification - orbos/98-05-04.

Table A-3 Component Tags

Tag Name	Tag Value	Described in
ComponentId	TAG_ORB_TYPE = 0	Section 13.6.3.1, “TAG_ORB_TYPE Component,” on page 13-19
ComponentId	TAG_CODE_SETS = 1	Section 13.7.2, “Code Set Conversion Framework,” on page 13-30
ComponentId	TAG_ALTERNATE_IIOP_ADDRESS = 3	Section 15.7.3, “IIOP IOR Profile Components,” on page 15-51
ComponentId	TAG_COMPLETE_OBJECT_KEY = 5	Section 16.5.4, “Complete Object Key Component,” on page 16-19
ComponentId	TAG_ENDPOINT_ID_POSITION = 6	Section 16.5.5, “Endpoint ID Position Component,” on page 16-20
ComponentId	TAG_LOCATION_POLICY = 12	Section 16.5.6, “Location Policy Component,” on page 16-20
ComponentId	TAG_ASSOCIATION_OPTIONS =13	See CORBAservices - Security chapter
ComponentId	TAG_SEC_NAME = 14	See CORBAservices - Security chapter
ComponentId	TAG_SPKM_1_SEC_MECH = 15	See CORBAservices - Security chapter
ComponentId	TAG_SPKM_2_SEC_MECH = 16	See CORBAservices - Security chapter
ComponentId	TAG_KerberosV5_SEC_MECH = 17	See CORBAservices - Security chapter
ComponentId	TAG_CSI_ECMA_Secret_SEC_MECH = 18	See CORBAservices - Security chapter
ComponentId	TAG_CSI_ECMA_Hybrid_SEC_MECH = 19	See CORBAservices - Security chapter
ComponentId	TAG_SSL_SEC_TRANS = 20	See CORBAservices - Security chapter
ComponentId	TAG_CSI_ECMA_Public_SEC_MECH = 21	See CORBAservices - Security chapter
ComponentId	TAG_GENERIC_SEC_MECH = 22	See CORBAservices - Security chapter
ComponentId	TAG_JAVA_CODEBASE = 25	Java to IDL Language Mapping - Section 1.4.9.3, “Codebase Transmission,” on page 1-33
ComponentId	TAG_DCE_STRING_BINDING = 100	Section 16.5.1, “DCE-CIOP String Binding Component,” on page 16-17

Table A-3 Component Tags

Tag Name	Tag Value	Described in
ComponentId	TAG_DCE_BINDING_NAME = 101	Section 16.5.2, “DCE-CIOP Binding Name Component,” on page 16-18
ComponentId	TAG_DCE_NO_PIPES = 102	Section 16.5.3, “DCE-CIOP No Pipes Component,” on page 16-19
ComponentId	TAG_DCE_SEC_MECH = 103	See CORBAservices - Security chapter

Table A-4 Policy Type Tags

Tag Name	Tag Value	Described in
PolicyId	SecClientInvocationAccess = 1	CORBAServices: Chapter 15 - Security
PolicyId	SecTargetInvocationAccess = 2	CORBAServices: Chapter 15 - Security
PolicyId	SecApplicationAccess = 3	CORBAServices: Chapter 15 - Security
PolicyId	SecClientInvocationAudit = 4	CORBAServices: Chapter 15 - Security
PolicyId	SecTargetInvocationAudit = 5	CORBAServices: Chapter 15 - Security
PolicyId	SecApplicationAudit = 6	CORBAServices: Chapter 15 - Security
PolicyId	SecDelegation = 7	CORBAServices: Chapter 15 - Security
PolicyId	SecClientSecureInvocation = 8	CORBAServices: Chapter 15 - Security
PolicyId	SecTargetSecureInvocation = 9	CORBAServices: Chapter 15 - Security
PolicyId	SecNonRepudiation = 10	CORBAServices: Chapter 15 - Security
PolicyId	SecConstruction = 11	Section 4.10.2.2, “Construction Policy,” on page 4-32
PolicyId	SecMechanismPolicy = 12	CORBAServices: Chapter 15 - Security
PolicyId	SecInvocationCredentialsPolicy = 13	CORBAServices: Chapter 15 - Security
PolicyId	SecFeaturesPolicy = 14	CORBAServices: Chapter 15 - Security
PolicyId	SecQOPPolicy = 15	CORBAServices: Chapter 15 - Security
PolicyId	THREAD_POLICY_ID = 16	Section 11.3.7.1, “Thread Policy,” on page 11-27
PolicyId	LIFESPAN_POLICY_ID = 17	Section 11.3.7.2, “Lifespan Policy,” on page 11-28
PolicyId	ID_UNIQUENESS_POLICY_ID = 18	Section 11.3.7.3, “Object Id Uniqueness Policy,” on page 11-28
PolicyId	ID_ASSIGNMENT_POLICY_ID = 19	Section 11.3.7.4, “Id Assignment Policy,” on page 11-29
PolicyId	IMPLICIT_ACTIVATION_POLICY_ID = 20	Section 11.3.7.7, “Implicit Activation Policy,” on page 11-31
PolicyId	SERVANT_RETENTION_POLICY_ID = 21	Section 11.3.7.5, “Servant Retention Policy,” on page 11-29

Table A-4 Policy Type Tags

Tag Name	Tag Value	Described in
PolicyId	REQUEST_PROCESSING_POLICY_ID = 22	Section 11.3.7.6, “Request Processing Policy,” on page 11-29
PolicyId	BIDIRECTIONAL_POLICY_TYPE = 37	Section 15.9, “Bi-directional GIOP policy,” on page 15-55
PolicyId	SecDelegationDirectivePolicy = 38	CORBAServices: Chapter 15 - Security
PolicyId	SecEstablishTrustPolicy = 39	CORBAServices: Chapter 15 - Security

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.

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.

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.

portable object adapter — The object adapter described in Chapter 9.

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.

Index

Index

Index