91.8.1

Table of Contents

CORBA 1.0

DRAFT

List of Figures

1 Overview

1.1 Overview

As defined by the Object Management Group (OMG), the Object Request Broker (ORB) provides the mechanisms by which objects transparently make requests and receive responses. The ORB provides interoperability between applications on different machines in heterogeneous distributed environments and seemlessly interconnects multiple object systems.[1]

The Common Object Request Broker Architecture and Specification described in this document is a self-contained response to the Request For Proposals (RFP) issued by the ORB Task Force of the OMG, submitted jointly by the following companies:

- Digital Equipment Corporation
- Hewlett-Packard Company
- Hyperdesk Corporation
- NCR Corporation

1. From *Object Management Architecture Guide*, Revision 1.0, OMG TC Document 90.9.1.

- Object Design, Inc.
- SunSoft, Inc.

It completely replaces the separate proposals previously submitted. At the request of the ORB Task Force, representatives of the above companies worked to find a way to support a common submission that could have rapid commercial availability using technologies each company had already developed. The result is the Common ORB Architecture and Specification, which defines a framework for different ORB implementations to provide common ORB services and interfaces to support portable clients and implementations of objects.

This document is organized as follows:

Chapter 1: Overview
Overview of the document and status.

Chapter 2: The Object Model
The concrete object model for the Common ORB Architecture.

Chapter 3: The Common Object Request Broker Architecture
The overall ORB architecture and interface description.

Chapter 4: IDL Syntax and Semantics
The specification of the Interface Definition Language.

Chapter 5: C Language Stub Mapping
The mapping provided for the C programming language.

Chapter 6: Dynamic Invocation Interface
The dynamic request invocation interface.

Chapter 7: The Interface Repository
The interface (i.e., type) definition repository.

Chapter 8: ORB Interface
The direct interface to the ORB.

Chapter 9: The Basic Object Adapter
The Basic Object Adapter interface.

Chapter 10: Interoperability
A discussion of interoperability between ORBs.

Chapter 11: Glossary Glossary.

1.2 Unresolved Issues

This document contains material which has not been agreed to by all of the parties involved. Such issues are considered not significant enough to prevent reaching the overall agreement, and are unlikely to cause major difficulty in accomplishing the goals of OMG and the ORB. Despite these issues, the submitters of this document are satisfied with this submission and endorse it.

In some cases possible solutions to the issues have been proposed, however resolution of the issue is pending futher study by each of the submitters.

The submitters recognize that not all of the document is consistent and that some errors and omissions remain. The submitters expect to correct these problems and expand on the text where needed for explanatory purposes.

Paragraphs containing unresolved issues are highlighted in the text in the following way:

*** This material has not been resolved.

The unresolved issues are listed below along with their page number.

Page 55: It is an unresolved issue as to whether to allow IDL interface inheritance from multiple interfaces which define the same operation names, thereby causing a name conflict.

Page 62: It is unresolved as to how to map IDL union data structures to languages other than C.

Page 64: It is unresolved whether it is possible to return arbitrary values along with an exception using existing programming language exception mechanisms.

Page 68: Since IDL interface names have global scope, they must all be unique in any particular context. This may limit the use of interfaces by a client because of name clashes. It is unresolved how to handle this name scoping problem.

Page 70: The list of IDL standard exceptions is provisional. In addition, a listing of "well-known types" (such as objref_t) needs to be added to the end of this chapter.

Page 87: Datatypes are normally provided through the interface in native (eg. compiler generated) format. It is an issue, in addition to passing structures in native form as to whether it will be allowed for constructed datatypes to be provided as an exploded structure defined as a list of lists for the dynamic invocation interface. Note that however provided on the client side, the datatypes always appear in native format to the object implementation.

Page 88: The IDL datatype set does not currently support the concept of "void *" (eg. any) which is necessary for certain servers. The dynamic invocation interface currently does define this concept. The use of this construct is not resolved (resolution depends in part on the issue of datatype representation).

Page 89: The memory management definition used in the dynamic invocation mechanism is not agreed to.

Page 90: It is unresolved as to whether the context override list will be allowed in a dynamic invocation.

Page 92: Memory management support in the List routines of the dynamic request invocation is not resolved. The particular issue is whether the ORB will allocate storage for return values and out parameters according to the list allocation mechanism described here, or using the same policy as for stubs. Alternatively stated, It is still unresolved whether an ORB mechanism, such as lists, is used to manage memory in both the dynamic and static interfaces; or memory managment is always handled in a language specific manner.

Page 94: It is unresolved whether the ORB directly supports persistent context objects, or if the ORB is simply a client of context objects, which could have a variety of persistent or non-persistent implementations.

Page 104: This definition of the interface repository depends on the "any" type (see the issue about datatypes in the dynamic invocation interface).

Page 104: It is unresolved if all of the attributes (eg. Time_Created) specified for the three types of "Container" objects are to be included in the interface.

Page 104: It is unresolved as to whether the "survey" interfaces should be included.

Page 115: The exact definition of typecodes has not yet been resolved. What is required is that all the types representable in IDL be handled.

Page 133: The syntax for method signatures in an object implementation is provisional.

2 The Object Model

This chapter describes the concrete object model which underlies the Common ORB Architecture. The model is derived from the abstract object model defined by the Object Management Group[1].

2.1 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:

1. Object Management Architecture Guide 1.0, Chapter 4, OMG TC Document 90.9.1, Object Management Group, Inc., 492 Old Connecticut Path, Framingham, MA 01701, November 1990.

- 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. See Chapter 9 for more information on implementation rules for objects which are managed by the Basic Object Adapter.

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

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

2.2 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, i.e. the concepts relevant to clients.

2.2.1 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.

2.2.2 Requests

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

A *request form* is a description or pattern that can be evaluated or performed multiple times to cause the issuing of requests. Request forms are defined by particular language bindings. In the C language binding, a request form is a C function invocation naming a function created by the IDL compiler from an IDL interface definition. 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.[1]

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

A *handle* is an object name that reliably identifies a particular object. Specifically, a handle will identify the same object each time the handle is used in a request (subject to certain pragmatic limits of space and time). An *object reference* is a handle.

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

A request causes a service to be performed on behalf of the client. One outcome of performing a service is that results are returned to the client.

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.

1. Descriptions of these request forms may be found in Chapter 5 for the C-language binding for IDL and Chapter 6 for the dynamic invocation interface.

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

The following semantics hold for all requests:

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

Descriptions of the permitted values in requests and the permitted exceptions may be found in §2.2.4 on page 20, and §2.2.6.3 on page 23.

2.2.3 Object Creation and Destruction

Objects can be created. Clients create objects by issuing requests. The result of object creation is revealed to the client in the form of an object reference that identifies the new object.

Objects may also be destroyed.

2.2.4 Types

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

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

A type may have different predicates at different times. The *extension of a type* is the set of values that satisfy the type at any particular time.

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

Data types in this model are constrained as follows:

Basic types:

- 16-bit and 32-bit signed and unsigned 2's complement integers

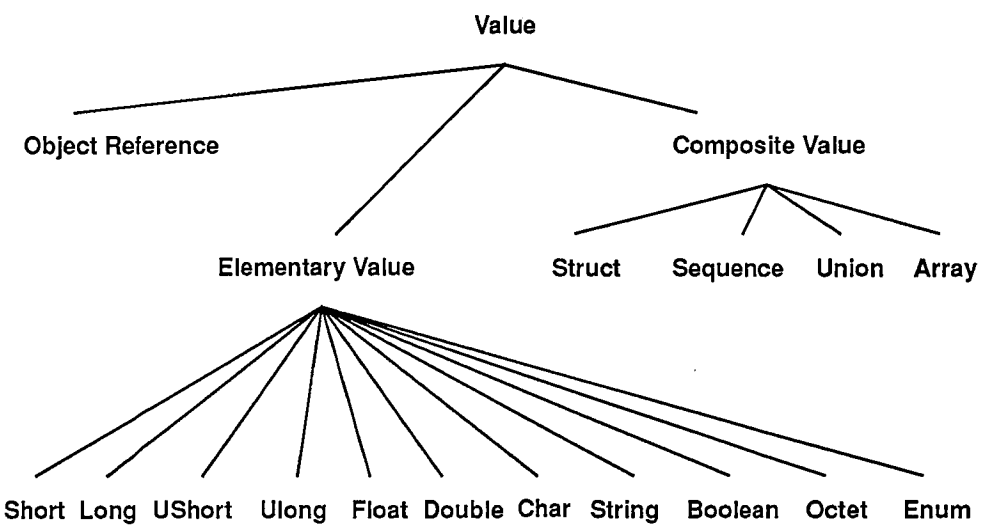
- 32-bit and 64-bit IEEE floating point numbers
- an enumerated set of characters
- a boolean type taking the values TRUE and FALSE
- an 8-bit opaque datatype, guaranteed to *not* undergo any conversion during transfer between systems
- an enumerated type which defines ordered sequences of identifiers
- a string type which consists of a variable-length array of characters; the length of the string is available at runtime

Constructed types:

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

Values in a request are constrained to values which satisfy these type constraints. The legal values are shown in FIG. 1 on page 21.

FIG. 1 Legal Values



2.2.5 Interfaces

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

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

Interfaces are specified in 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.

2.2.6 Operations

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

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

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

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

Operations are (potentially) *generic*, meaning that a single operation can be uniformly requested on objects with different implementations, 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.

An operation may be identified in a request form by an *operation name*. The operation names are determined by language bindings.

The general form for an operation signature is:

```
[oneway] <op_type_spec> <identifier> (param1, ..., paramL)
        [throw(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>** simply provides a name for the operation in the interface; the actual operation name that a programmer must refer to in a request form is dependent upon the language mapping in use
- 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 **throw** expression indicates which exceptions can be signalled to terminate a request for this operation; if such an expression is not provided, no exceptions will be signalled
- 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

2.2.6.1 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 direction[s] dictated by the mode.

2.2.6.2 Return Result

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

2.2.6.3 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 §2.2.4 on page 20.

2.2.6.4 Contexts

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

2.2.6.5 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.

2.2.7 Attributes

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

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

2.3 Object Implementation

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

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

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

2.3.1 The Execution Model: Performing Services

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

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

When a client issues a request, a method of the target object is called. The input parameters passed by the requestor are passed to the method and the output parameters and return values (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.*

2.3.2 The Construction Model

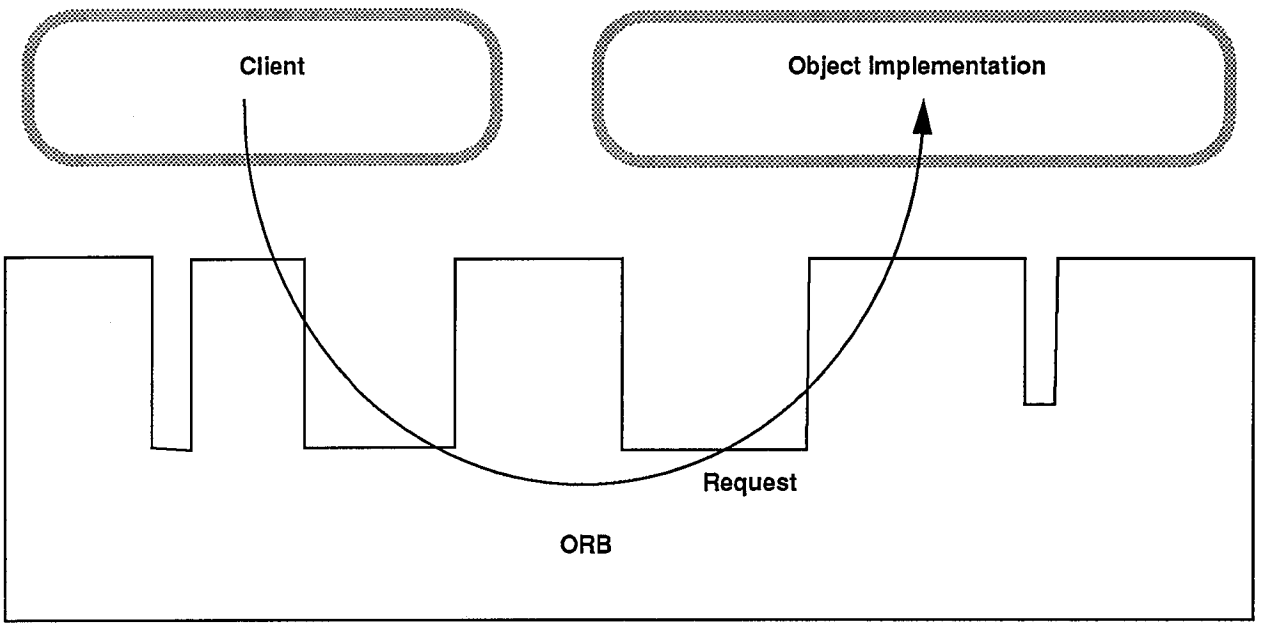
A computational object system must provide mechanisms for realizing behavior of requests. These mechanisms include definitions of object state, definitions of methods, and definitions of how the object infrastructure is to select the methods to execute and to select the cells to be made accessible to 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 that the state of an object. It also typically includes information about the intended type of the object.

3 The Common Object Request Broker Architecture

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 should become more clear.

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



3.1 The Structure of an Object Request Broker

FIG. 2 on page 28 shows a request being sent by a client to an object implementation.The Client is the entity that wishes to perform an operation on the object and the Object Implementation is the code and data that actually implements the object. The ORB is responsible for all of the mechanisms required to find the object implementation for the request, to prepare the object implementation to receive the request, and to communicate the data making up the request. The interface the client sees is completely independent of where the object is located, what programming language it is implemented in, or any other aspect which is not reflected in the object's interface.

FIG. 3 The Structure of Object Request Broker Interfaces

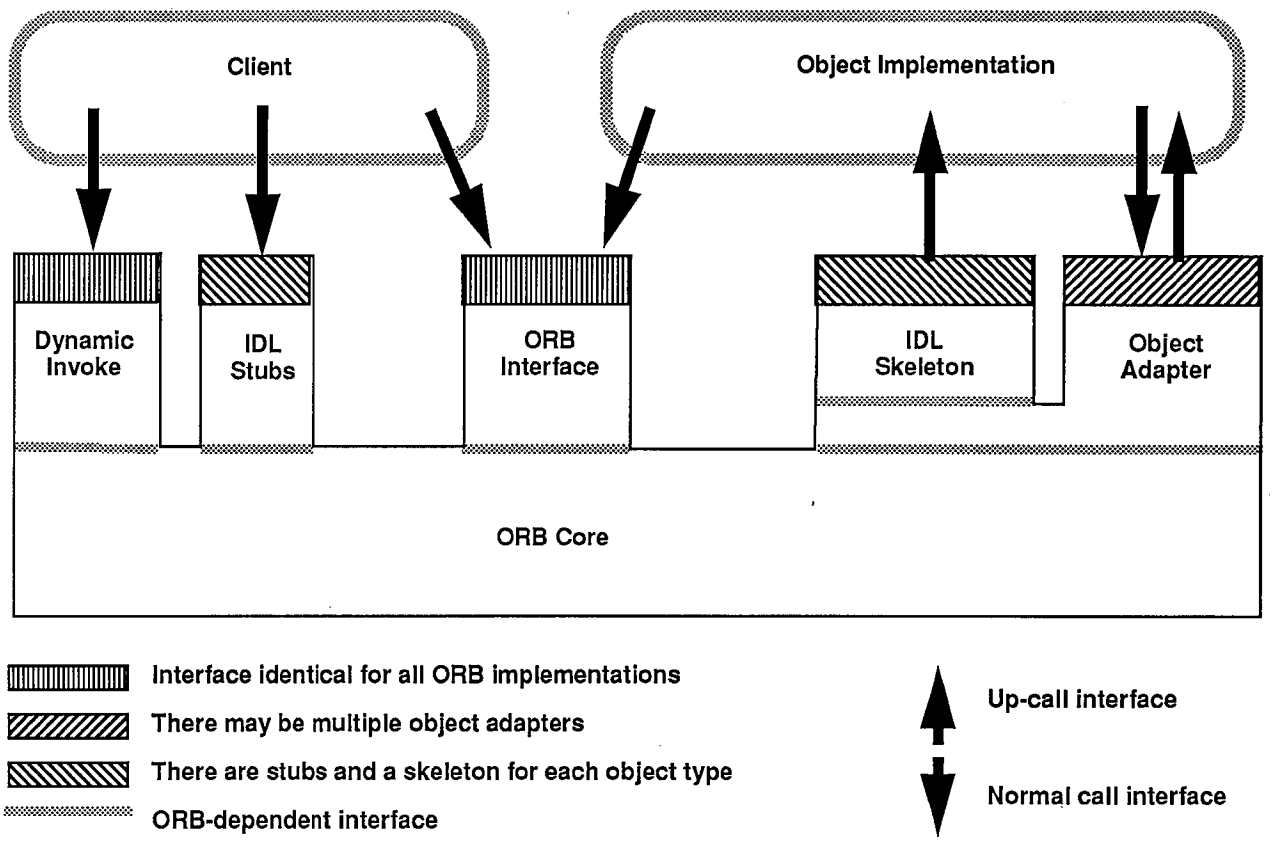


FIG. 3 on page 29 shows the structure of an individual Object Request Broker (ORB). The interfaces to the ORB are shown by striped boxes, and the arrows indicate whether the ORB is called or performs an up-call across the interface.

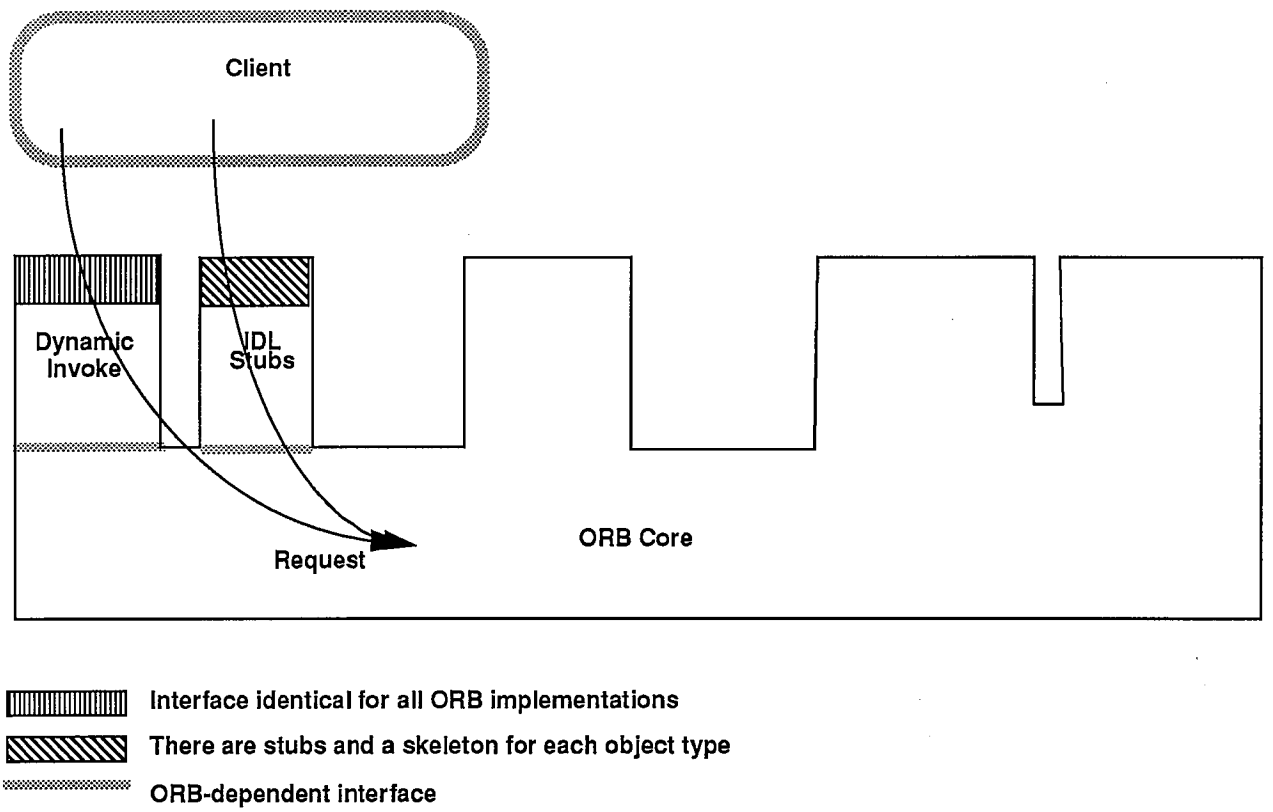
To make a request, the Client can use the Dynamic Invoke interface (the same interface independent of the interface of the target object) or an IDL stub (the specific stub depending on the interface of the target object). The Client can also directly interact with the ORB for some functions.

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

A central premise of CORBA is that there are definitions of the interfaces to objects in an interface definition language, herein called the Interface Definition Language (IDL). This

language defines the types of objects according to the operations that may be performed on them and the parameters to those operations.

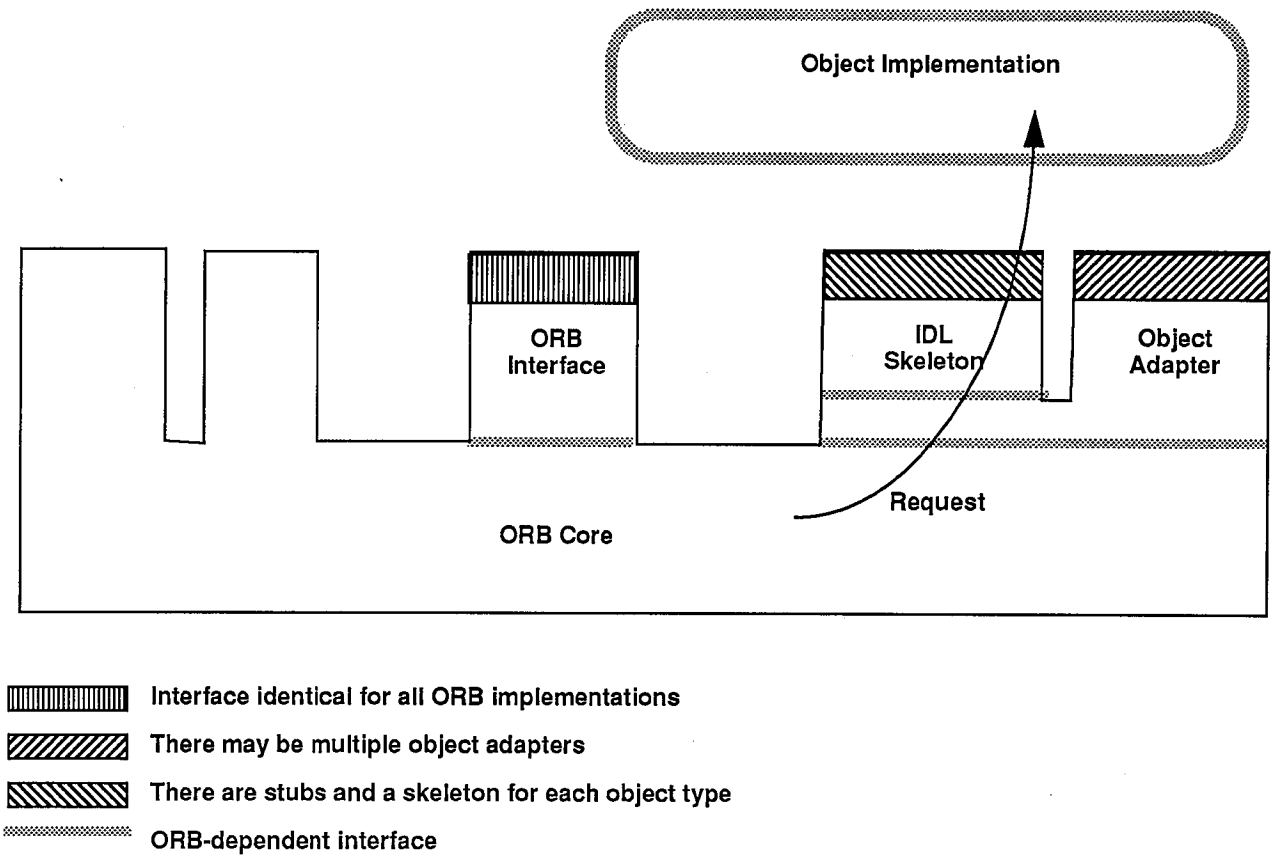
FIG. 4 A Client using the Stub or Dynamic Invocation Interface



The client performs a request by having access to an Object Reference for an object and knowing the type of the object and the desired operation to be performed. The client initiates the request be calling stub routines that are specific to the object or by constructing the call dynamically (see FIG. 4 on page 30).

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

FIG. 5 An Object Implementation Receiving a Request



The ORB locates the appropriate implementation code, transmits parameters and transfers control to the Object Implementation through a object interface-specific skeleton (see FIG. 5 on page 31). In performing the request, the object implementation may obtain some services from the ORB through the Object Adapter. When the request is complete, control and output values are returned to the client.

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

FIG. 6 Interface and Implementation Repositories

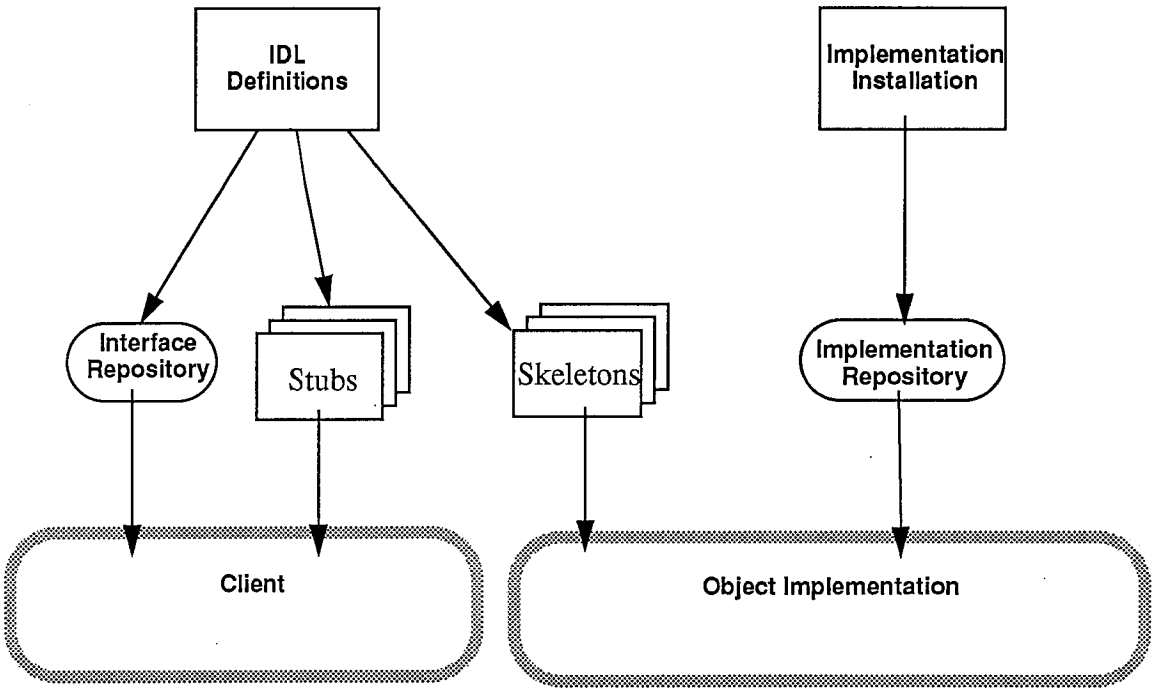


FIG. 6 on page 32 shows how interface and implementation information is made available to clients and object implementations. The object interface is defined in IDL and that definition is used to generate the client side Stubs, the object implementation side Skeletons, and is put into the Interface Repository to be used for dynamic invocation.

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

3.1.1 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. those operations that are the same for all ORB implementations,
2. those operations that are specific to particular types of objects, and
3. those operations that are specific to particular styles of object implementations.

Different ORBs may make quite different implementation choices, and, together with the IDL compiler 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 objects implemented using different ORBs. The ORBs must keep track of which objects are implemented in which ORB.

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.

3.1.2 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.

3.1.3 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 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. Object implementations are portable across any ORB that supports the desired language mapping and implements the desired Object Adapter.

3.1.4 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 References.

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

3.1.5 IDL Interface Definition Language

The IDL Interface Definition Language 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 runtime type 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.

3.1.6 Programming Language Mapping

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 language mapping to CORBA should be the same for all ORB implementations. The 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, the dynamic invocation interface, the implementation skeleton, the object adapter, and the direct ORB interface.

The 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.

3.1.7 Client Stubs

For a particular language mapping, there will be an interface to the stubs for each object type. Generally, the stubs will present access to the IDL-defined operations on an object in a way that is easy for programmers to predict once they are familiar with 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.

3.1.8 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 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 a type repository or other runtime source).

3.1.9 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.

3.1.10 Object Adapters

An object adapter is the primary way that an object implementation accesses services provided by the ORB. There are expected to be a few object adapters that will be widely available, with interfaces that are appropriate for specific kinds of objects. Services provided by the ORB through an Object Adapter often include: invocation, activation and deactivation of object implementations or object instances, creation of objects, generation of object references, association of persistent storage and access control information with objects, authentication of clients, object implementation registration.

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.

3.1.11 ORB Interface

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

3.1.12 Interface Repository

The Interface Repository is a service that provides persistent objects that represent the IDL information in a form available at runtime. 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 CORBA objects. For example, debugging information, libraries of stubs or skeletons, routines that can format or browse particular kinds of objects, etc., might be associated with the Interface Repository.

3.1.13 Implementation Repository

The Implementation Repository contains information that allows the ORB to locate and activate implementations of objects. Although most of the information in the Implementation Repository is specific to an ORB or operating environment, the Implementation Repository is the conventional place for recording such information. Ordinarily, installation of classes 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 CORBA objects. For example, debugging information, administrative control, resource allocation, security, etc., might be associated with the Implementation Repository.

3.2 Some 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.

3.2.1 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.

3.2.2 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.

3.2.3 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.

3.2.4 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.

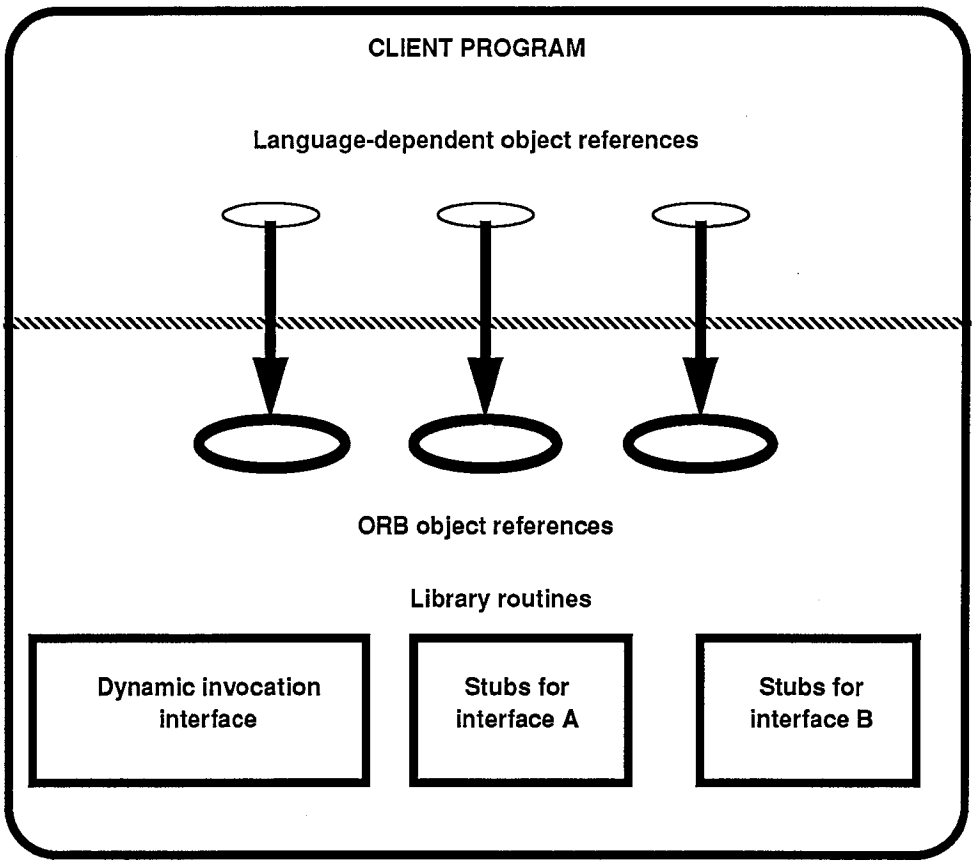
3.3 The Structure of a Client

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 error response is provided. Ordinarily, a client calls a routine in its program that performs the invocation and returns when the operation is complete.

Most clients access object-type-specific stubs as library routines in their program (see FIG. 7 on page 38). The client program thus sees routines callable in the normal way in its programming language. Most implementations will provide a language-specific data type to use to refer to objects, often an opaque pointer. The client then passes that pointer to the stub routines to initiate an invocation. The stubs have access to the actual object reference and interact with the ORB to perform the invocation.

FIG. 7 The Structure of a Typical Client



An alternative set of library code may also be available to perform invocations on objects when stubs are not available, for example when the object was not defined at compile

time. In that case, the client program must provide 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.

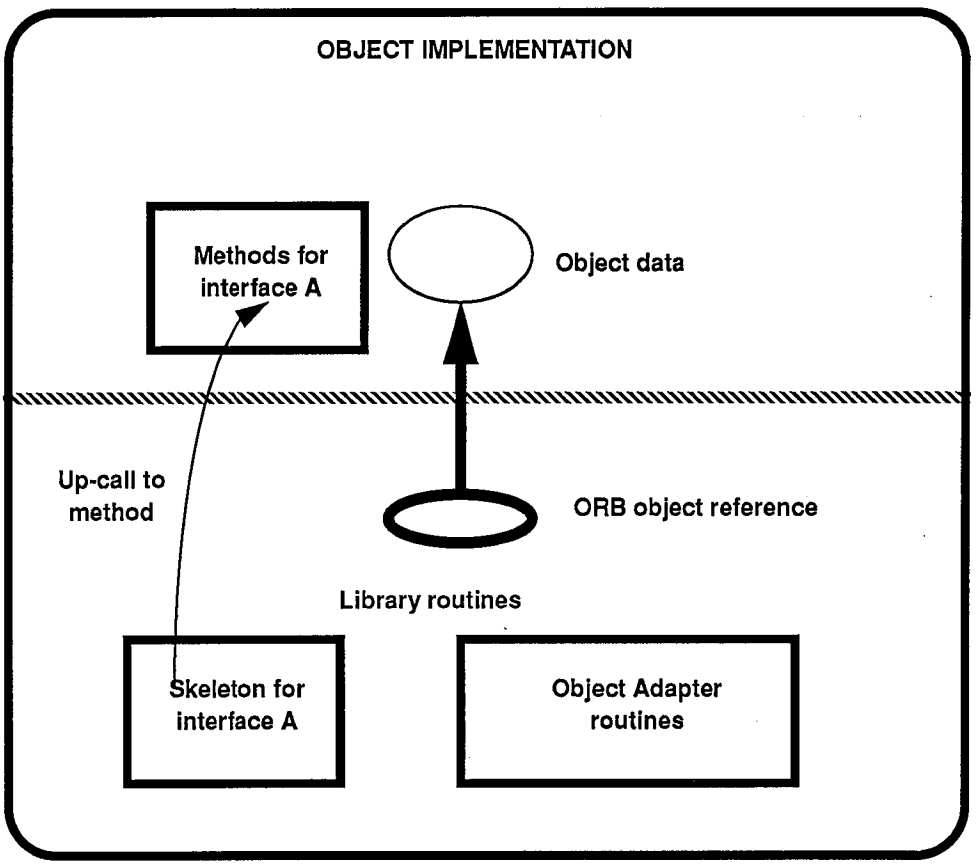
Clients most commonly obtain object references by receiving them as output parameters from invocations on other objects they have, or as input parameters on invocations to objects they implement. Object references can also be converted to data types that can be stored in files or preserved or communicated by different means and subsequently turned back into object references.

3.4 The Structure of an Object Implementation

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

The object implementation (see FIG. 8 on page 40) interacts with the ORB in a variety of ways 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 particular kinds of objects.

FIG. 8 The Structure of a Typical Object Implementation



Because of the range of possible object implementations, it is difficult to be definitive about how in general an object implementation is structured. More details will be supplied in the context of a particular 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 to be transmitted back to the client.

When a new object is created, the ORB must be notified so that the it knows where to find the implementation for that object. Usually, the implementation also registers itself as

implementing objects of a particular class, and specifies how to start up the implementation if it is not already running.

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

3.5 The Structure of an Object Adapter

An object adapter (see FIG. 9 on page 42) 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.

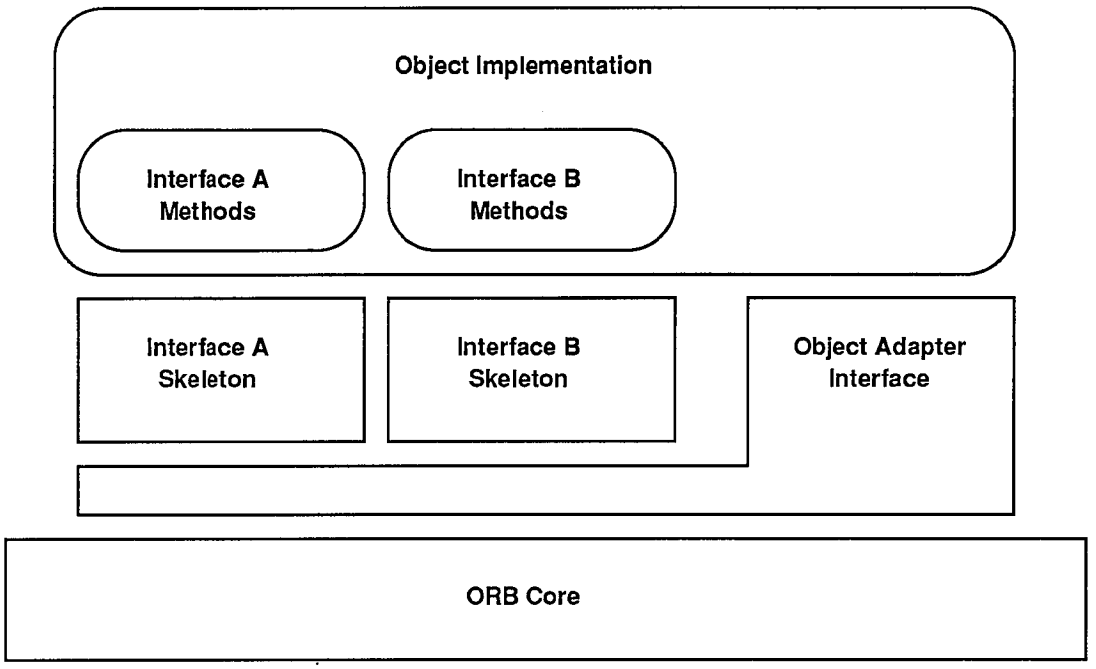
Object adapters are responsible for the following functions:

- generation and interpretation of object references
- method invocation
- security of interactions
- object and implementation activation and deactivation
- distinguishing of object references by the implementation
- 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.

As shown in FIG. 9 on page 42, the Object Adapter is implicitly involved in invocation of the methods, although the direct interface is through the skeletons. For example, the Object Adapter may be involved in activating the implementation or authenticating the request.

FIG. 9 The Structure of a Typical Object Adapter



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

3.6 Some example Object Adapters

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

3.6.1 Basic Object Adapter

This proposal defines an object adapter that can be used for most ORB objects with conventional implementations (See Chapter 9). For this object adapter, implementations are generally separate programs. It allows there to be a program started per method, a separate program for each object, or a shared program for all instances of the class. It provides a small amount of persistent storage for each object, which can be used as a name or identifier for other storage, for access control lists, or other object properties. If the implementation is not active when an invocation is performed, it will start one.

3.6.2 Library Object Adapter

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

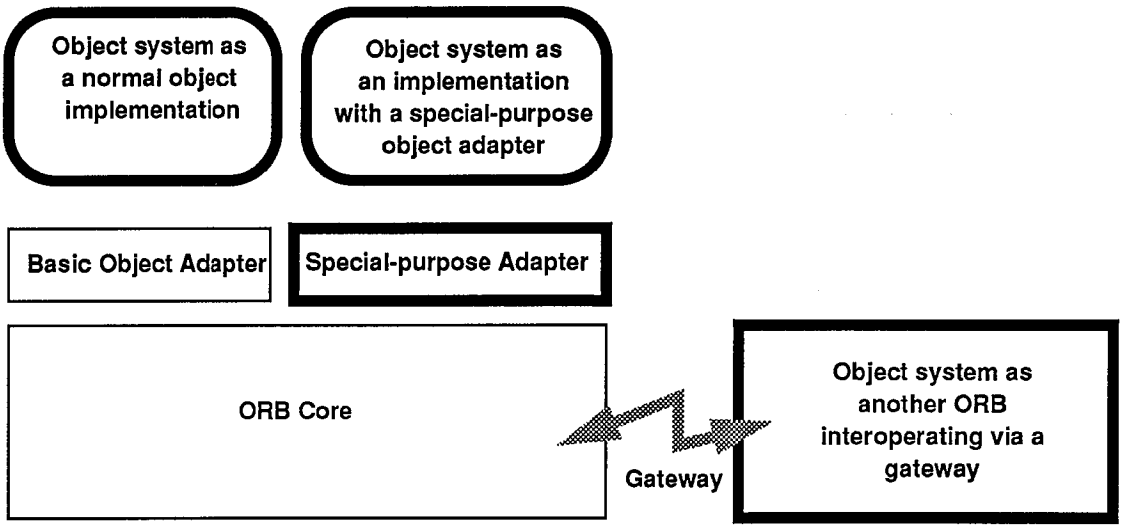
3.6.3 Object-Oriented Database Adapter

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

3.7 The Integration of Foreign Object Systems

The Common ORB Architecture is designed to allow interoperation with a wide range of object systems (see FIG. 10 on page 44). Because there are many existing object systems, a common desire will be to allow the objects in those systems to be accessible via the ORB. For those object systems that are ORBs themselves, they may be connected to other ORBs through the mechanisms described in chapter 10 on page 135.

FIG. 10 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, another solution 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 normal object implementation. A typical object adapter will 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.

4 IDL Syntax and Semantics

IDL (the Interface Definition Language) is the language used to describe the interfaces that client objects call and server objects provide. An interface definition written in IDL completely defines the interface and fully specifies each remote operation's parameters. An IDL remote interface provides the information needed to develop clients that use the interface's operations.

IDL obeys the same lexical rules as C++, although new keywords are introduced to support distribution concepts. It also provides full support for standard C++ preprocessing features. The full description of IDL's lexical conventions is presented in §4.1 on page 46. A description of IDL preprocessing is presented in §4.2 on page 49. The scope rules for identifiers in an IDL specification are described in §4.10 on page 68.

IDL grammar is a subset of ANSI C++ with additional constructs to support the remote operation invocation mechanism. 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 full grammar is presented in §4.3 on page 50.

IDL-specific pragmas may appear anywhere in a specification; the textual location of these pragmas may be semantically constrained.

This chapter describes IDL semantics and gives the syntax for IDL grammatical constructs. The description of IDL grammar uses a syntax notation that is similar to Extended Backus-Naur format (EBNF); TBL. 1 on page 46 lists the symbols used in this format and their meaning.

TBL. 1 IDL EBNF Format

Symbol	Meaning
::=	Is defined to be
|	Alternatively
<text>	Non-terminal
"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

4.1 Lexical Conventions

This section[1] presents the lexical conventions of IDL. It defines tokens in an IDL specification and describes comments, identifiers, keywords, and literals—integer, character, and floating point constants and string literals.

An IDL specification 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.

4.1.1 Tokens

There are five kinds of tokens: identifiers, keywords, literals, operators, and other separators. Blanks, horizontal and vertical tabs, newlines, formfeeds, and comments (collective,

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

"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.

4.1.2 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.

4.1.3 Identifiers

An identifier is an arbitrarily long sequence of letters and digits. The first character must be a letter; the underscore _ counts as a letter. Upper- and lower-case letters are treated as the same. All characters are significant.

4.1.4 Keywords

The following identifiers are reserved for use as keywords, and may not be used otherwise:

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

Identifiers starting with an underscore (_) should probably be avoided to avert clashes with certain language bindings.

The ASCII representation of IDL specifications uses the following characters as punctuation:

; { } : , = + - ( ) < > [ ] ' " \

In addition, the following tokens are used by the preprocessor:

! | || &&

4.1.5 Literals

4.1.5.1 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.

The type of an integer literal depends on its form, value, and suffix. If it is decimal, octal, or hexadecimal and has no suffix, it has the first of these types in which its value can be represented: long int, unsigned long int. If it is suffixed by u or U, its type is unsigned long int. If it is suffixed by l or L, its type is the first of these types in which its value can be represented: long int, unsigned long int. If it is suffixed by ul, lu, uL, Lu, Ul, lU, UL, or LU, its type is unsigned long int.

4.1.5.2 Character Literals

A character literal is one or more characters enclosed in single quotes, as in 'x'. Character literals have type char. The value of a character literal is the numerical value of the character in the machine's character set.

Certain nongraphic characters, the single quote ', the double quote ", the question mark ?, and the backslash \, may be represented according to the escape sequences shown in TBL. 2 on page 48.

TBL. 2 Escape Sequences

Description	Abbrev.	Escape
new-line	NL(LF)	\n
horizontal tab	HT	\t
vertical tab	VT	\v
backspace	BS	\b
carriage return	CR	\r
form feed	FF	\f
alert	BEL	\a
backslash	\	\\
question mark	?	\?
single quote	'	\'
double quote	"	\"
octal number	ooo	\ooo
hexadecimal number	hh	\xhh

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

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

4.1.5.3 Floating Literals

A floating literal consists of an integer part, a decimal point, a fraction part, an e or E, an optionally signed integer exponent, and an optional type suffix. 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. The type of a floating literal is double unless explicitly specified by a suffix. The suffixes f and F specify float, the suffixes l and L specify double.

4.1.5.4 String Literals

A string literal is a sequence of characters (as defined in §4.1.5.2 on page 48) surrounded by double quotes, as in "...". A string has type "sequence of char" and is initialized with the given characters.

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'.

4.2 Preprocessing

IDL preprocessing, which is based on ANSI C++ preprocessing, provides macro substitution, conditional compilation, and source file inclusion. In addition, directives are pro-

vided 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 IDL; they may appear anywhere and have effects that last (independent of the IDL scoping rules) until the end of the translation unit. The textual location of IDL-specific pragmas may be semantically constrained.

A preprocessing directive may be continued on the next line in a source file by placing a backslash character, \, immediately before the new-line at the end of the line to be continued. The preprocessor effects the continuation by deleting the backslash and the new-line 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 IDL token (§4.1.1 on page 46), 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 IDL specifications. A complete description of the preprocessing facilities may be found in *The Annotated C++ Reference Manual*, Chapter 16.

4.3 IDL Grammar

```
(1) <specification>      ::= <definition>+

(2) <definition>         ::= <type_dcl> ";"
                         |   <const_dcl> ";"
                         |   <except_dcl> ";"
                         |   <interface> ";"

(3) <interface>          ::= <interface_dcl>
                         |   <forward_dcl>

(4) <interface_dcl>      ::= <interface_header> "{" <interface_body> "}"

(5) <forward_dcl>        ::= "interface" <identifier>

(6) <interface_header>   ::= "interface" <identifier> [ <inheritance_spec> ]

(7) <interface_body>     ::= <export>*

(8) <export>             ::= <type_dcl> ";"
                         |   <const_dcl> ";"
```

```
                    |   <except_dcl> ";"
                    |   <attr_dcl> ";"
                    |   <op_dcl> ";"
(9)  <inheritance_spec>   ::=  ":" <identifier> { "," <identifier> }*
(10) <const_dcl>          ::=  "const" <const_type> <identifier> "=" <const_exp>
(11) <const_type>         ::=  <integer_type>
                    |   <char_type>
                    |   <boolean_type>
                    |   <floating_pt_type>
                    |   <string_type>
                    |   <scoped_name>
(12) <const_exp>          ::=  <or_expr>
(13) <or_expr>            ::=  <xor_expr>
                    |   <or_expr> "|" <xor_expr>
(14) <xor_expr>           ::=  <and_expr>
                    |   <xor_expr> "^" <and_expr>
(15) <and_expr>           ::=  <shift_expr>
                    |   <and_expr> "&" <shift_expr>
(16) <shift_expr>         ::=  <add_expr>
                    |   <shift_expr> ">>" <add_expr>
                    |   <shift_expr> "<<" <add_expr>
(17) <add_expr>           ::=  <mult_expr>
                    |   <add_expr> "+" <mult_expr>
                    |   <add_expr> "-" <mult_expr>
(18) <mult_expr>          ::=  <unary_expr>
                    |   <mult_expr> "*" <unary_expr>
                    |   <mult_expr> "/" <unary_expr>
                    |   <mult_expr> "%" <unary_expr>
(19) <unary_expr>         ::=  <unary_operator> <primary_expr>
                    |   <primary_expr>
(20) <unary_operator>     ::=  "-"
                    |   "+"
                    |   "~"
(21) <primary_expr>       ::=  <scoped_name>
                    |   <literal>
                    |   "(" <constant_expr> ")"
(22) <literal>            ::=  <integer_literal>
                    |   <string_literal>
                    |   <character_literal>
                    |   <floating_pt_literal>
                    |   <boolean_literal>
```

```
(23) <boolean_literal>      ::= "TRUE"
                            |   "FALSE"
(24) <scoped_name>          ::= <identifier>
                            |   "::" <identifier>
                            |   <scoped_name> "::" <identifier>
(25) <positive_int_const>   ::= <const_exp>
(26) <type_dcl>             ::= "typedef" <type_declarator>
                            |   <struct_type>
                            |   <union_type>
                            |   <enum_type>
(27) <type_declarator>      ::= <type_spec> <declarators>
(28) <type_spec>            ::= <simple_type_spec>
                            |   <constr_type_spec>
(29) <simple_type_spec>     ::= <base_type_spec>
                            |   <template_type_spec>
                            |   <scoped_name>
(30) <base_type_spec>       ::= <floating_pt_type>
                            |   <integer_type>
                            |   <char_type>
                            |   <boolean_type>
                            |   <octet_type>
(31) <constr_type_spec>     ::= <struct_type>
                            |   <union_type>
                            |   <enum_type>
(32) <template_type_spec>   ::= <sequence_type>
                            |   <string_type>
(33) <declarators>          ::= <declarator> { "," <declarator> }*
(34) <declarator>           ::= <simple_declarator>
                            |   <complex_declarator>
(35) <simple_declarator>    ::= <identifier>
(36) <complex_declarator>   ::= <array_declarator>
(37) <floating_pt_type>     ::= "float"
                            |   "double"
(38) <integer_type>         ::= <signed_int>
                            |   <unsigned_int>
(39) <signed_int>           ::= <signed_long_int>
                            |   <signed_short_int>
(40) <signed_long_int>      ::= "long"
(41) <signed_short_int>     ::= "short"
```

```
(42) <unsigned_int>          ::= <unsigned_long_int>
                               |  <unsigned_short_int>
(43) <unsigned_long_int>     ::= "unsigned" "long"
(44) <unsigned_short_int>    ::= "unsigned" "short"
(45) <char_type>             ::= "char"
(46) <boolean_type>          ::= "boolean"
(47) <octet_type>            ::= "octet"
(48) <struct_type>           ::= "struct" <identifier> "{" <member_list> "}"
(49) <member_list>           ::= <member>+
(50) <member>                ::= <type_spec> <declarators> ";"
(51) <union_type>            ::= "union" <identifier> "switch" "(" <switch_type_spec> ")"
                                    "{" <switch_body> "}"
(52) <switch_type_spec>      ::= <integer_type>
                               |  <char_type>
                               |  <boolean_type>
                               |  <enum_type>
                               |  <scoped_name>
(53) <switch_body>           ::= <case>+
(54) <case>                  ::= <case_label>+ <element_spec> ";"
(55) <case_label>            ::= "case" <const_exp> ":"
                               |  "default" ":"
(56) <element_spec>          ::= <type_spec> <declarator>
(57) <enum_type>             ::= "enum" <identifier> "{" <enumerator> { "," <enumerator> }* "}"
(58) <enumerator>            ::= <identifier>
(59) <sequence_type>         ::= "sequence" "<" <simple_type_spec> "," <template_bound> ">"
(60) <template_bound>        ::= <positive_int_constant>
                               |  "UNBOUNDED"
(61) <string_type>           ::= "string" "<" <template_bound> ">"
(62) <array_declarator>      ::= <identifier> <fixed_array_size>+
(63) <fixed_array_size>      ::= "[" <positive_int_const> "]"
(64) <attr_dcl>              ::= [ "readonly" ] "attribute" <simple_type_spec> <declarators>
(65) <except_dcl>            ::= "exception" <identifier> "{" <member_list> "}"
(66) <op_dcl>                ::= [ <op_attribute> ] <op_type_spec> <identifier> <parameter_dcls>
                                    [ <throw_expr> ] [ <context_expr> ]
(67) <op_attribute>          ::= "oneway"
(68) <op_type_spec>          ::= <simple_type_spec>
                               |  "void"
```

```
(69) <parameter_dcls>   ::= "(" <param_dcl> { "," <param_dcl> }* ")"
                        |   "(" ")"
(70) <param_dcl>        ::= <param_attribute> <simple_type_spec> <declarator>
(71) <param_attribute>  ::= "in"
                        |   "out"
                        |   "inout"
(72) <throw_expr>       ::= "throw" "(" <scoped_name> { "," <scoped_name> }* ")"
(73) <context_expr>     ::= "context" "(" <string_literal> { "," <string_literal> }* ")"
```

4.4 Interface Declaration

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

```
<specification>  ::= <definition>+
<definition>     ::= <type_dcl> ";"
                 |   <const_dcl> ";"
                 |   <except_dcl> ";"
                 |   <interface> ";"
```

See §4.5 on page 56, §4.6 on page 58, and §4.7 on page 64, respectively, for specifications of <const_dcl>, <type_dcl>, and <except_dcl>.

An interface definition satisfies the following syntax:

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

4.4.1 Interface Header

The interface header consists of two elements:

- The interface name. The name must be preceded by the keyword **interface**, and consists of an identifier that names the interface.

- An optional inheritance specification. The inheritance specification is described in §4.4.2 on page 55.

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.

4.4.2 Inheritance Specification

An interface can be derived from another interface, which is then called a **base** interface of the **derived** interface. An interface can be derived from one or more interfaces. The derived interface inherits the constants, types, attributes, exceptions, and operations of its base interfaces and, transitively, those of their base interfaces. In addition, the derived interface can declare additional constants, types, attributes, exceptions, and operations.

The syntax for inheritance is as follows:

<inheritance_spec> ::= ":" **<identifier>** { "," **<identifier>** }*

Each <identifier> in an <inheritance_spec> must denote a previously defined interface and can appear only once in the list.

All interfaces are implicitly derived from the Object interface; this interface has no operations—it exists solely to permit the use of generic interface references in IDL specifications.

A derived interface may redefine any of the type, constant and exception names which have been inherited; constraints on such redefinition, and the scope rules for such names, are described in §4.10 on page 68.

It is not currently legal to inherit from two interfaces with the same operation name.

*** It is an unresolved issue as to whether to allow IDL interface inheritance from multiple interfaces which define the same operation names, thereby causing a name conflict.

4.4.3 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 §4.5 on page 56.

- Type declarations, which specify the type definitions that the interface exports; type declaration syntax is described in §4.6 on page 58.
- Exception declarations, which specify the exception structures that the interface exports; exception declaration syntax is described in §4.7 on page 64.
- Attribute declarations, which specify the associated attributes exported by the interface; attribute declaration syntax is described in §4.8 on page 64.
- Operation declarations, which specify the operations that the interface exports and the format of each, including function 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 §4.9 on page 65.

Empty interfaces (i.e. those that contain no declarations) are permitted.

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

4.4.4 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.

4.5 Constant Declaration

This section describes the syntax for constant declarations.

4.5.1 Syntax

The syntax for a constant declaration is:

```
<const_dcl>     ::= "const" <const_type> <identifier> "=" <const_exp>
<const_type>    ::= <integer_type>
                  | <char_type>
                  | <boolean_type>
                  | <floating_pt_type>
                  | <string_type>
                  | <scoped_name>
<const_exp>     ::= <or_expr>
```

```
<or_expr>           ::= <xor_expr>
                    |   <or_expr> "|" <xor_expr>
<xor_expr>          ::= <and_expr>
                    |   <xor_expr> "^" <and_expr>
<and_expr>          ::= <shift_expr>
                    |   <and_expr> "&" <shift_expr>
<shift_expr>        ::= <add_expr>
                    |   <shift_expr> ">>" <add_expr>
                    |   <shift_expr> "<<" <add_expr>
<add_expr>          ::= <mult_expr>
                    |   <add_expr> "+" <mult_expr>
                    |   <add_expr> "-" <mult_expr>
<mult_expr>         ::= <unary_expr>
                    |   <mult_expr> "*" <unary_expr>
                    |   <mult_expr> "/" <unary_expr>
                    |   <mult_expr> "%" <unary_expr>
<unary_expr>        ::= <unary_operator> <primary_expr>
                    |   <primary_expr>
<unary_operator>    ::= "-"
                    |   "+"
                    |   "~"
<primary_expr>      ::= <scoped_name>
                    |   <literal>
                    |   "(" <constant_expr> ")"
<literal>           ::= <integer_literal>
                    |   <string_literal>
                    |   <character_literal>
                    |   <floating_pt_literal>
                    |   <boolean_literal>
<boolean_literal>   ::= "TRUE"
                    |   "FALSE"
<scoped_name>       ::= <identifier>
                    |   "::" <identifier>
                    |   <scoped_name> "::" <identifier>
<positive_int_const> ::= <const_exp>
```

4.5.2 Semantics

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

<const_exp> must evaluate to the same type as specified in <const_type>. This constraint distributes over any sub-expressions in the <const_exp>; in particular, if a <scoped_name> is

used in an expression, it must have been defined in a preceding constant declaration with the same type as specified in <const_type>.

The "~" unary operator can be applied *only* to integer types. It 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:

short	**- (value + 1)**
unsigned short	**$(2^{16} - 1)$ - value**
long	**- (value + 1)**
unsigned long	**$(2^{32} - 1)$ - value**

<positive_int_const> must evaluate to a positive integer constant.

4.6 Type Declaration

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

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

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

```
<type_spec>         ::= <simple_type_spec>
                      | <constr_type_spec>
<simple_type_spec>  ::= <base_type_spec>
                      | <template_type_spec>
                      | <scoped_name>
<base_type_spec>    ::= <floating_pt_type>
                      | <integer_type>
                      | <char_type>
                      | <boolean_type>
                      | <octet_type>
<constr_type_spec>  ::= <struct_type>
                      | <union_type>
                      | <enum_type>
```

```
<template_type_spec>    ::= <sequence_type>
                        |   <string_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, IDL type specifiers consist of simple scalar data types and type constructors. 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.

4.6.1 Basic Types

The syntax for the supported basic types is as follows:

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

Each IDL data type is mapped to a native data type via the appropriate language mapping. Conversion errors between IDL data types and the native types to which they are mapped can occur during the performance of an operation invocation. It is the responsibility of the invocation mechanism (client stub, dynamic invocation engine, and skeletons) to 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 §4.12 on page 70.

4.6.1.1 Integer Types

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

4.6.1.2 Floating-Point Types

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

It is not required that there be native equivalences for the IEEE notions of not-a-number and infinity; if attempts are made to transmit these quantities, the behavior is undefined.

NOTE *Floating-point conversion errors fall into four categories: 1) the native mantissa has more precision than the corresponding IEEE type - in this case the value should be rounded to the precision available; reporting such loss of precision is not strictly required, 2) the native number is larger than the largest value of the IEEE type - in this case (floating overflow) an exception condition should be reported to the programmer, 3) the native number is smaller than the smallest value of the IEEE type - in this case (floating underflow) the stub may silently round the value to 0, and 4) the native representation does not distinguish between -0 and +0 - in this case, the stub can silently convert -0 to 0.*

4.6.1.3 Char Type

IDL defines a **char** data type, consisting of the ISO Latin-1 printing characters and the escape characters described in §4.1.5.2 on page 48. Characters are transmitted as 8-bit quantities in their ASCII representation.

4.6.1.4 Boolean Type

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

4.6.1.5 Octet Type

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

4.6.2 Constructed Types

The constructed types are:

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

4.6.2.1 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.

4.6.2.2 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>
```

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 TBL. 3 on page 62.

TBL. 3 Case Label Matching

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

Name scoping rules require that the element declarators in a particular union be unique. The value of a union is the value of its discriminator and one of its members.

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

*** It is unresolved as to how to map IDL union data structures to languages other than C.

4.6.2.3 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.

4.6.3 Template Types

The template types are:

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

4.6.3.1 Sequences

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> "," <template_bound> ">"
<template_bound>    ::= <positive_int_const>
                    |   "UNBOUNDED"
```

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.

A maximum size of UNBOUNDED indicates that the size of the sequence is unspecified. 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.

4.6.3.2 Strings

IDL defines the string type **string**. 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" "<" <template_bound> ">"
```

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.

A maximum size of UNBOUNDED indicates that the size of the string is unspecified.

Strings are singled out as a separate, constructed type since many language 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.

4.6.4 Complex Declarator

4.6.4.1 Arrays

IDL defines fixed 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.

4.7 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_list> "}"**

A set of standard exceptions are defined to correspond to standard run-time errors which may occur during the execution of a request. These standard exceptions are documented in §4.12 on page 70.

*** It is unresolved whether it is possible to return arbitrary values along with an exception using existing programming language exception mechanisms.

4.8 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" <simple_type_spec> <declarators>**

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 specification fragment:

```
   . . .
   float       get_radius();
   boolean     set_radius(in float r);
   material_t  get_material();
   boolean     set_material(in material_t m);
   position_t  get_position();
   . . .
```

Attributes are inherited. An attribute name *cannot* be redefined to be a different type. See §4.10 on page 68 for more information on redefinition constraints and the handling of ambiguity.

4.9 Operation Declaration

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

<op_dcl> ::= [**<op_attribute>**] **<op_type_spec>** **<identifier>** **<parameter_dcls>**
[**<throw_expr>**] [**<context_expr>**]

<op_type_spec> ::= **<simple_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 §4.9.1 on page 66.
- The type of the operation's return result; the type may be any type which can be defined in IDL. Operations that do not return a result must specify the **void** type.
- An identifier that names the operation.
- A parameter list that specifies zero or more parameter declarations for the operation. Parameter declaration is described in §4.9.2 on page 66.

- An optional throw expression which indicates which exceptions may be returned as a result of an invocation of this operation. Throw expressions are described in Section §4.9.3 on page 67.
- 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 §4.9.4 on page 67.

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

4.9.1 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"
```

The **oneway** keyword in an operation declaration indicates that the operation's caller does not expect (and may not be able to handle) a response. 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.

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

4.9.2 Parameter Declarations

Parameter declarations in IDL operation declarations have the following syntax:

```
<parameter_dcls>        ::= "(" <param_dcl> { "," <param_dcl> }* ")"
                        |   "(" ")"
<param_dcl>             ::= <param_attribute> <simple_type_spec> <declarator>
<param_attribute>       ::= "in"
                        |   "out"
                        |   "inout"
```

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 a server will *not* attempt to modify an **in** parameter. The ability to even attempt to do so is language-mapping specific; the affect of such an action is undefined.

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

4.9.3 Throw Expressions

A throw expression specifies which exceptions may be returned (*thrown*) as a result of an invocation of the operation. The syntax for its specification is as follows:

<throw_expr> ::= **"throw" "(" <scoped_name> { "," <scoped_name> }* ")"**

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

In addition to any operation-specific exceptions specified in the throw expression, there are a standard set of exceptions that may be signalled by the ORB. These standard exceptions are described in §4.12 on page 70.

The absence of a throw 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.

4.9.4 Context Expressions

A context expression specifies which elements of the client's context may affect method binding in 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 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 an empty request context is to be associated with requests for this operation.

The <string_literal>'s must satisfy the following constraints:

- The first character must be one in the range a-z or A-Z.
- All subsequent letters must be from the set a-z A-Z 0-9.
- The last letter may be an asterisk, '*'.

The mechanism by which a client associates values with the context identifiers is described in Chapter 6.

4.10 Names and Scoping

When an <identifier> is encountered in an IDL specification, it must be interpreted with respect to the notion of a current naming scope. The IDL global name of an identifier (<scoped-name>) consists of the concatenation of the current scope, the string "::", and the identifier. The following identifiers are scoped:

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

Prior to starting to scan a file containing a IDL specification, the name of the current scope is initially empty, "". Whenever an **interface**, **struct**, or **union** 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**, or **union**, the trailing "::" and identifier are deleted from the name of the current scope. As such, the scopes nest. 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. For example, an IDL global name is of the form ::A::B::C.

*** Since IDL interface names have global scope, they must all be unique in any particular context. This may limit the use of interfaces by a client because of name clashes. It is unresolved how to handle this name scoping problem.

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

NOTE *It is currently illegal to redefine an operation name or attribute name. This restriction may be removed in a future version of the language.*

Due to possible restrictions imposed by future language bindings, IDL identifiers are case insensitive—i.e. two identifiers that differ only in the case of their characters are considered redefinitions of one another.

Type names defined in a scope are available for immediate use within that scope.

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

4.10.1 Handling Ambiguity in Types, Constants, Enumeration Values, and Exceptions

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

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

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

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

The attribute declaration in C is ambiguous, since the compiler does not know which string_t is desired. The programmer must disambiguate the specification by providing a scoped name for the type of the Title attribute, either A::string_t or B::string_t.

4.11 Differences from C++

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

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

4.12 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.

*** The list of IDL standard exceptions is provisional. In addition, a listing of "well-known types" (such as objref_t) needs to be added to the end of this chapter.

```
const unsigned long except_ok = 0;
const unsigned long except_user_supplied = except_ok + 1;

const unsigned long except_ORB_failure = except_user_supplied + 1;
exception ORB_failure {// unspecified failure
   string<256> reason;
};

const unsigned long except_ORB_einval = except_ORB_failure + 1;
exception ORB_einval {// bad argument presented
   string<256> reason;
};

const unsigned long except_ORB_enomem = except_ORB_einval + 1;
exception ORB_enomem {// ran out of memory
   string<256> reason;
};

const unsigned long except_ORB_etoobig = except_ORB_enomem + 1;
exception ORB_etoobig {// surpassed implementation limit
   string<256> reason;
};

const unsigned long except_ORB_messaging = except_ORB_etoobig + 1;
exception ORB_messaging {// messaging not possible
   string<256> reason;
};

const unsigned long except_ORB_objref = except_ORB_messaging + 1;
exception ORB_objref {// invalid object reference
   string<256> reason;
};

const unsigned long except_ORB_eperm = except_ORB_objref + 1;
exception ORB_eperm {// no permission for operation
   string<256> reason;
};

const unsigned long except_ORB_internal = except_ORB_eperm + 1;
exception ORB_internal {// ORB internal error
   string<256> reason;
};

const unsigned long except_ORB_marshal = except_ORB_internal + 1;
exception ORB_marshall {// error marshalling argument/result
   string<256> reason;
};

const unsigned long except_ORB_initialize = except_ORB_marshal + 1;
exception ORB_initialize {// ORB initialization failure
```

```
   string<256> reason;
};

const unsigned long except_ORB_class_epv = except_ORB_initialize + 1;
exception ORB_class_epv {// epv not found
   string<256> reason;
};

const unsigned long except_ORB_obj_act = except_ORB_class_epv + 1;
exception ORB_obj_act {// object activation function not found
   string<256> reason;
};
```

5 C Language Stub Mapping

This chapter describes how concepts used in IDL are mapped into The C Programming Language—that is, how a C programmer uses an interface defined in IDL.

5.1 Scoped Names

The C programmer must always use the global name for a type, constant, exception, or operation. The C global name corresponding to an IDL global name is derived by converting occurrences of "::" to "_" (an underscore) and eliminating the leading underscore.

Consider the following example:

```
typedef string<256> filename_t;
interface example0 {
   enum color {red, green, blue};
   union bar switch (enum foo {room, bell}) { ... };
   • • •
};
```

Code to use this interface would look as follows:

```
#include "example0.h"

filename_t FN;
example0_color C = example0_red;
example0_bar myUnion;

switch (myUnion._d) {
case example0_bar_room: • • •
case example0_bar_bell: • • •
};
```

Note that the use of underscores to replace the "::" separators can lead to ambiguity if the IDL specification contains identifiers with underscores in them. Consider the following example:

```
typedef long foo_bar;
interface foo {
   typedef short bar;/* A legal IDL statement, but ambiguous in C */
   • • •
};
```

Due to such ambiguities, it is advisable to avoid the indiscriminate use of underscores in identifiers.

5.2 Mapping for Interfaces

All interfaces must be defined at global scope (*no* nested interfaces). The mapping for an interface declaration is as follows:

```
interface example1 {
   long op1(in long arg1);
};
```

The preceding example generates the following C declarations[1]:

```
typedef objref_t example1;
extern long example1_op1(example1 object,
   long arg1, exception_t *except);
```

All object references (actually typed interface references to an object) are of the well-known, opaque type **objref_t**. To permit the programmer to decorate a program with typed references, a type with the name of the interface is defined to be an objref_t. The literal

1. §5.12 on page 81 describes the additional arguments added to an operation in the C mapping.

OBJREF_NIL is legal wherever an objref_t may be used; it is guaranteed to fail the equality test with any real objref_t.

IDL permits specifications in which arguments, return results, or components of constructed types may be interface references.Consider the following example:

```
#include "example1.idl"

interface example2 {
   example1 op2();
};
```

This is equivalent to the following C declaration:

```
#include "example1.h"

typedef objref_t example2;
extern example1 example2_op2(example2 object, exception_t *except);
```

A C fragment for invoking such an operation is as follows:

```
#include "example2.h"

example1 ex1;
example2 ex2;
exception_t except;

/* code for binding ex2 */

ex1 = example2_op2(ex2, &except);
```

5.3 Inheritance and Operation Names

IDL permits the specification of interfaces that inherit operations from other interfaces. Consider the following example:

```
interface example3 : example1 {
   void op3(in long arg3, out long arg4);
};
```

This is equivalent to the following C declarations:

```
typedef objref_t example3;
extern long example3_op1(example3 object,
   long arg1, exception_t *except);
extern void example3_op3(example3 object,
   long arg3, long *arg4, exception_t *except);
```

As a result, an object written in C can access **op1** as if it was directly declared in **example3**. Of course, the programmer could also invoke **example1_op1** on an **objref_t** of type **example3**; the virtual nature of operations in interface definitions will cause invocations of either function to cause the same method to be invoked.

5.4 Mapping for Constants

Constant identifiers can be referenced at any point in the user's code where a literal of that type is legal. In C, these constants are **#defined**.

The fact that constants are **#defined** may lead to ambiguities in code. All names which are mandated by the mappings for any of the structured types below start with an underscore. As a result, these possible ambiguities may be avoided if the programmer avoids using any identifiers in IDL which start with an underscore.

5.5 Mapping for Basic Data Types

The basic data types have the mappings shown in TBL. 4 on page 76.

TBL. 4 Data Type Mappings

IDL	C
short	short
long	long
unsigned short	unsigned short
unsigned long	unsigned long
float	float
double	double
char	char
boolean	long
octet	unsigned char
enum	unsigned long

5.6 Mapping for Structure Types

IDL structures map directly onto C **structs**.

5.7 Mapping for Union Types

IDL discriminated unions are mapped onto C **structs**. Consider the following IDL declaration:

```
union Foo switch (long) {
   case 1: long x;
   case 2: float y;
   default: char z;
};
```

This is equivalent to the following **struct** in C:

```
typedef struct {
   long _d;
   union {
      long x;
      float y;
      char z;
   } _u;
} Foo;
```

The discriminator in the struct is always referred to as **_d**; the union in the struct is always referred to as **_u**.

Reference to union elements is as in normal C:

```
Foo *v;

/* make a call that returns a pointer to a Foo in v */

switch(v->_d) {
   case 1:  printf("x = %ld\n", v->_u.x); break;
   case 2:  printf("y = %f\n", v->_u.y); break;
   default: printf("z = %c\n", v->_u.z); break;
}
```

5.8 Mapping for Sequence Types

The sequence IDL data type permits passing of open arrays between objects. Consider the following IDL declaration:

```
typedef sequence<long,10> vec10;
```

In C, this is converted to:

```
typedef struct {
   unsigned long _maximum;
   unsigned long _length;
   long *_buffer;
} vec10;
```

An instance of this type is declared as follows:

```
vec10 x = {10L, 0L, (long *)NULL);
```

Prior to passing **x** as an **in** parameter, the programmer must set the *_buffer* member to point to a **long** array of 10 elements, and must set the *_length* member to the actual number of elements to transmit.

Prior to passing **&x** as an **out** parameter (or receiving a **vec10** as the function return), the programmer does nothing. The client stub will allocate storage for the returned buffer using malloc(); for bounded sequences, it allocates a buffer of the specified size, while for unbounded sequences, it allocates a buffer big enough to hold what was returned by the object. Upon successful return from the invocation, the *_maximum* member will contain the size of the allocated array, the *_buffer* member will point at the allocated storage, and the *_length* member will contain the number of values that were returned in the *_buffer* member. The client is responsible for freeing the allocated storage using free(). The current contents of the *_buffer* member is overwritten with a pointer to the storage allocated by the stub.

Prior to passing **&x** as an **inout** parameter, the programmer must set the *_buffer* member to point to a **long** array of 10 elements. For an unbounded sequence, the programmer **must** set the *_maximum* member to the actual size of the array. The *_length* member must be set to the actual number of elements to transmit. Upon successful return from the invocation, the *_length* member will contain the number of values that were copied into the buffer pointed to by the *_buffer* member.

For bounded sequences, it is an error to set the *_length* or *_maximum* member to a value larger than the specified bound.

Two sequence types are the same type if their sequence element type and size arguments are identical. For example,

```
const long SIZE = 25;
typedef int seqtype;

typedef sequence<int, SIZE> s1;
typedef sequence<int, 25> s2;
typedef sequence<seqtype, SIZE> s3;
typedef sequence<seqtype, 25> s4;
```

declares s1, s2, s3, and s4 to be of the same type.

The IDL type

```
sequence<type,size>
```

maps to

```
#ifndef _IDL_SEQUENCE_type_size_defined
#define _IDL_SEQUENCE_type_size_defined
typedef struct {
   long maximum;
   long length;
   type *buffer;
} _IDL_SEQUENCE_type_size
#endif/* _IDL_SEQUENCE_type_size_defined */
```

The ifdef's are needed to prevent duplicate definition where the same type is used more than once. The type name used in the C mapping is the type name of the effective type, e.g. in

```
typedef long FRED;
typedef sequence<FRED,10> FredSeq;
```

the sequence is mapped onto struct { ... } _IDL_SEQUENCE_long_10;

These generated type names may be used to declare instances of a sequence type.

5.9 Mapping for Strings

IDL strings are mapped to 0-byte terminated character arrays; i.e. the length of the string is encoded in the character array itself through the placement of the 0-byte. Consider the following IDL declarations:

```
typedef string<10> sten;
typedef string<UNBOUNDED> sinf;
```

In C, this is converted to:

```
typedef char *sten;
typedef char *sinf;
```

Instances of these types are declared as follows:

```
sten s1 = NULL;
sinf s2 = NULL;
```

Two string types are the same type if their size arguments are identical. For example,

```
const long SIZE = 25;

typedef string<SIZE> s1;
typedef string<25> s2;
```

declares s1 and s2 to be of the same type.

Prior to passing **s1** or **s2** as an **in** parameter, the programmer must assign the address of a character buffer containing a 0-byte terminated string to the variable.

Prior to passing **&s1** or **&s2** as an **out** parameter (or receiving an **sten** or **sinf** as the return result), the programmer does nothing. The client stub will allocate storage for the returned buffer using malloc(); for bounded strings, it allocates a buffer of the specified size, while for unbounded strings, it allocates a buffer big enough to hold the returned string. Upon successful return from the invocation, the character pointer will contain the address of the allocated buffer. The client is responsible for freeing the allocated storage using free(). If the pointer is non-NULL when a call is made, it is overwritten with a pointer to the storage allocated by the stub.

Prior to passing **&s1** or **&s2** as an **inout** parameter, the programmer must assign the address of a character buffer containing a 0-byte terminated array to the variable. Upon successful return from the invocation, the returned 0-byte terminated array is copied into the same buffer. If it was a bounded string, then the size of the returned string is limited by the declared size of the string type; if it was an unbounded string, then the size of the returned string is limited by the size of the string passed as input. Due to this restriction, use of **inout** string parameters is deprecated.

5.10 Mapping for Arrays

IDL arrays map directly to C arrays. All array indices run from 0 to <**size** - 1>.

If a return result to an operation is an array, the array storage is dynamically allocated by the stub and the following structure is returned to the caller:

```
struct array_desc {
   void *_buffer;
};
```

The *_buffer* member of the structure contains the address of the first element of the dynamically allocated array, cast to a (`void *`). The caller must cast this to a pointer of the correct type to use the array. When the data is no longer needed, the programmer should return the dynamically allocated storage by calling free().

5.11 Mapping for Exception Types

Each defined exception type is defined as a struct tag and a typedef with the C global name for the exception. An identifier for the exception, in string literal form, is also **#defined**. For example:

```
exception foo {
   long dummy;
};
```

yields the following C declarations:

```
typedef struct foo {
   long dummy;
} foo;
#define ex_foo "unique identifier for ex_foo"
```

The above definition of the exception identifier is simply an example. The exact format of this identifier will be described in later versions of this document.

5.12 Implicit Arguments to Operations

From the point of view of the C programmer, all operations declared in an interface have an implicit leading **objref_t** input parameter (the target object) and a trailing (**exception_t** *) output parameter. The leading parameter permits the programmer to designate which object is to process the request; the trailing parameter permits the return of exception information. See examples in previous sections.

If an operation in an IDL specification has a **context** specification, then an argument of type **context_t** follows the leading **objref_t** and precedes any operation-specific arguments.

As described above, the **objref_t** type is an opaque type. The **exception_t** type is partially opaque; a subsequent section provides a description of the non-opaque portion of the exception structure and an example of how to handle exceptions in client code. The **context_t** type is opaque; see Chapter 6 for more information on how to create and manipulate context objects.

5.13 Interpretation of Functions with Empty Argument Lists

As in C++, a function declared with an empty argument list is defined to take *no* operation-specific arguments.

5.14 Argument Passing Considerations

Regardless of the IDL type being passed, if the IDL signature specifies that an argument is an **out** or **inout** parameter, then the caller must always pass the address of a variable of that type (or the value of a pointer to that type); the callee must always dereference to get to the type. For **in** parameters, the value of the parameter is passed.

Consider the following IDL specification:

```
interface foo {
   typedef long Vector[25];

   void bar(out Vector x);
};
```

Client code for invoking the **bar** operation would look like:

```
foo object;
Vector x;
exception_t except;

/* code to bind objref_t to instance of foo */

foo_bar(object, &x, &except);
```

Classic C compilers, when asked to take the address of an array, will invariably return the address of the first element. ANSI C, on the other hand, defines the address of the array to be different from the address of the first element. As a result, it is imperative that one compile both the client code and stub code with the same compiler so as not to generate a disconnect across the stub interface.

5.15 Return Result Passing Considerations

When an operation is defined to return a non-void return result, the following rules hold:

1. If the return result is one of the types float, double, long, short, unsigned long, unsigned short, char, boolean, octet, objref_t, or an enumeration, then the value is returned as the operation result.
2. If the return is any other type, then a pointer to the return value is returned as the operation result.

Consider the following interface:

```
interface X {
   struct y {
      long a;
```

```
        float b;
    };

    long op1();
    y op2();
}
```

The following C declarations ensue from processing the specification:

```
typedef objref_t X;
typedef struct X_y {
    long a;
    float b;
} X_y;

extern long X_op1(X object, exception_t *except);
extern X_y *X_op2(X object, exception_t *except);
```

Summary of Argument/Result Passing

Data Type	Pass In	Pass Out/inout	Return Result
short	value	addr of variable to hold value	receive value
long	value	addr of variable to hold value	receive value
unsigned short	value	addr of variable to hold value	receive value
unsigned long	value	addr of variable to hold value	receive value
float	value	addr of variable to hold value	receive value
double	value	addr of variable to hold value	receive value
boolean	value	addr of variable to hold value	receive value
char	value	addr of variable to hold value	receive value
octet	value	addr of variable to hold value	receive value
enumeration	value	addr of variable to hold value	receive value
object reference	objref_t value	addr of variable to hold objref_t	receive value of objref_t
struct	addr of struct	addr of variable to hold struct	receive value of struct
union	addr of struct	addr of variable to hold struct	receive value of struct
string	addr of 1st char	addr of (char *) variable	receive char *
sequence	addr of seq. struct	addr of variable holding seq struct	receive value of sequence struct
array	addr of 1st elem	addr of 1st element	receive value of array descriptor

5.16 Dynamic Storage Management

Client stubs must allocate storage for unbounded sequences and unbounded strings passed as `out` parameters; they must also allocate storage for unbounded sequences, unbounded strings, and arrays which are return results. It is the responsibility of the caller to free this storage using free().

5.17 Handling Exceptions in Client CDL

The **exception_t** type is partially opaque; the C declaration contains at least the following:

```
typedef struct exception_t {
   unsigned long _id;
   char *_ex_id;
   void *_value;
} exception_t;
```

Upon return from an invocation, the trailing exception parameter indicates whether the invocation terminated successfully; if not, it gives an indication of which exception occurred and provides access to any exception parameters signalled by the object.

Consider the following example:

```
interface exampleX {
   exception BadCall {
      string<80> reason;
   };

   void op() throw(BadCall);
};
```

This interface defines a single operation which returns no results and can signal a BadCall exception. The following user code shows how to invoke the operation and recover from an exception:

```
#include "exampleX.h"

exception_t ex;
exampleX obj;
exampleX_BadCall *bc

/*
 * some code to initialize obj to a reference
 * to an object supporting the exampleX interface
 */

exampleX_op(obj, &ex);
if (ex._id == except_ok) {
   /* process out and inout arguments */
} else {
   if (ex._id != except_user_defined) {
      /* process standard exceptions */
   } else {
      if (strcmp(ex._ex_id, ex_exampleX_BadCall) == 0) {
         bc = (exampleX_BadCall *)(ex._value);
         fprintf(stderr, "exampleX_op() failed - reason: %s\n",
            bc->reason);
      } else {              /* should never get here ... */
         fprintf(stderr, "unknown exception - %s\n",
            ex.user_identifier);
      }
   exception_free(&ex); /* free any storage associated with exception */
   }
}
```

Three cases can occur:

1. the exception identifier is except_ok, in which case the request was performed successfully
2. the exception identifier is except_user_defined, in which case it is a user-defined exception
3. the exception identifier identifies a standard exception

In the last two cases, the _value member of the exception_t points to the structure corresponding to the exception with that exception identifier. After processing an exception, the program code should invoke the following function:

```
extern void exception_free(exception_t *except);
```

This function will return any storage which was allocated in the construction of the exception_t.

5.18 Method Routine Signatures

The signatures of the methods used to implement the object depend not only on the language binding, but also on the choice of object adapter. Different object adapters may provide additional parameters to access object adapter-specific features.

Most object adapters are likely to provide method signatures that are similar in most respects to those of the client stubs. In particular, the mapping for the operation parameters expressed in IDL should be the same as for the client side.

See §9.3 on page 132 for the description of method signatures for implementations using the Basic Object Adapter.

5.19 Include Files

Multiple interfaces may be defined in a single source file. By convention, each interface is stored in a separate source file. All IDL compilers will, by default, generate a header file named **Foo.h** from **Foo.idl**. This file should be **#included** by clients and implementations of the interfaces defined in **Foo.idl**.

Inclusion of **Foo.h** is sufficient to define all global names associated with the interfaces in **Foo.idl** and any interfaces from which they are derived.

A source file containing interface specifications written in IDL must have a "**.idl**" extension.

6 Dynamic Invocation Interface

6.1 Overview

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

A request is comprised of an object handle, 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 the a NamedValue (see below).

*** Datatypes are normally provided through the interface in native (eg. compiler generated) format. It is an issue, in addition to passing structures in native form as to whether it will be allowed for constructed datatypes to be provided as an exploded structure defined as a list of lists for the dynamic invocation interface. Note that however provided on the client side, the datatypes always appear in native format to the object implementation.

*** Individual elements can themselves be lists, so that arbitrary constructed types can be represented as a nested list.

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 IDL datatype set does not currently support the concept of "void *" (eg. *any*) which is necessary for certain servers. The dynamic invocation interface currently does define this concept. The use of this construct is not resolved (resolution depends in part on the issue of datatype representation).

The following type should be a well-known datatype in IDL. It can be used either as a parameter type directly or as the building block for parameter lists. NamedValue is defined in the 'C' language by:

```
typedef struct
   {
   ArgName      Name;          /* argument name */
   typecode_t   Datatype;      /* argument datatype */
   (void *)     Value;         /* argument value */
   long         Len;           /* length/count of argument value */
   NVFlags      Flags;         /* argument flags */
   } Name

dValue, * NVList;
```

typecode_t a well-known opaque type that provides a mechanism for identifying all of the simple types(float, double, ...), constructed types (union, sequence,..) and well-known types (objref_t, NamedValue, NVList, typecode_t, ...), and can be extended to include arbitrary types defined in an IDL type specification. The list of valid typecode_t mnemonics for types and their type codes that must be supplied in an implementation is described in section 7.6 on page 115.

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

IN_ARG	the associated value is an input only argument
OUT_ARG	the associated value is an output only argument
INOUT_ARG	the associated value is an in/out argument
IN_COPY_VALUE	copies the value (see below)
OUT_LIST_MEMORY	use list memory for output values (see below)

DEPENDENT_LIST see below

The following routines return a status typedef'ed as **ORBStatus**. This status code is the same as the "well-known" status code returned in the exception structure.

6.2 Request Routines

6.2.1 ORB_CreateRequest

```
ORBStatus ORB_CreateRequest (
   objref_t     Objref,      /* object on which to invoke req */
   OpnName      Operation,   /* intended operation on object */
   NVList       ArgList,     /* (in,out) arguments to operation */
   CtxObj       Context,     /* context object to be used during opn */
   NVList       CtxOverrides, /* override context */
   NamedValue   * Result,    /* (in,out) operation result */
   ReqHnd       * Request,   /* (out) newly created request */
   ORBExc       * Exc);      /* (in,out) exception */
```

Creates a request "object". The 'ArgList', if specified, contains a list of arguments (input and output) which become associated with the request. If 'ArgList' is omitted, the arguments (if any) must be specified using the AddArgToRequest call below.

If specified, the 'ArgList' becomes associated with the request; and until the InvokeRequest call has completed (or the request has been deleted), the 'ArgList' (and any data it points to) is assumed to be unchanged.

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

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

*** The memory management definition used in the dynamic invocation mechanism is not agreed to.

Setting the OUT_LIST_MEMORY flag for an argument controls the memory allocation mechanism for output arguments that are dynamically allocated. Output arguments of this type are dynamically allocated off of, and associated with, the list structure (if passed in). When the list structure has been freed, the associated dynamically allocated memory is also freed.

*** It is unresolved as to whether the context override list will be allowed in a dynamic invocation.

Context values from the specified context object are effectively passed to the intended method. If 'CtxOverrides' are specified, these values logically override those, and are merged with, values in the context object. If no context object is specified, the default context object is used. See section §6.5 on page 94 for a further description of context objects.

The operation name is a string that conforms to the IDL rules for naming identifiers.

6.2.2 ORB_AddArgToRequest

```
ORBStatus ORB_AddArgToRequest (
   ReqHnd    Request,     /* request to be modified */
   ArgName   Name,        /* argument name */
   ArgType   Datatype,    /* argument datatype */
   (void *)  Value,       /* argument value to be added */
   long      Len,         /* length/count of argument value */
   NVFlags   Flags,       /* flags */
   ORBExc    * Exc);      /* (in,out) exception */
```

AddArgToRequest incrementally adds arguments to the request.

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

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

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

Arguments may be associated with a request by specifying them on the CreateRequest call or by adding them via AddArgToRequest. Using both methods for specifying arguments on the same request is not currently supported.

If the IN_COPY_VALUE flag is set, the argument value is first copied (presumably so that memory associated with the value can be used for some other purpose). This flag is ignored for in/out and out arguments.

6.2.3 ORB_InvokeRequest

```
ORBStatus ORB_InvokeRequest (
  ReqHnd   Request      /* request to be invoked */
  InvFlags Flags,       /* invocation flags */
  ORBExc   * Exc);      /* (in,out) exception */
```

Calls the ORB, which performs method resolution and invokes the associated method.

6.2.4 ORB_DeleteRequest

```
ORBStatus ORB_DeleteRequest (
   ReqHnd   Request);       /* the request to be deleted */
   ORBExc   * Exc);         /* (in,out) exception */
```

Deletes the request and all its associated memory.

6.3 Deferred Synchronous Routines

6.3.1 ORB_SendRequest

```
ORBStatus ORB_SendRequests (
   ReqHnd   * Requests,     /* request(s) to be sent */
   long     Count,          /* number of requests to be sent */
   InvFlags Flags,          /* invocation flags */
   ORBExc   * Exc);         /* (in,out) exception */
```

Dispatches each request in the list. One or more requests can be specified using this routine. In addition to supporting the aforementioned invocation flags.

The following invocation flags are currently defined for the SendRequest:

INV_TERM_ON_ERR — If one of the requests causes an error, the remaining requests are not sent.

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 one-way.

6.3.2 ORB_GetResponse

```
ORBStatus ORB_GetResponse (
   ReqHnd     * Request,     /* (in,out) the request */
   RespFlags Flags,          /* response flags */
   ORBExc    * Exc);         /* (in,out) exception */
```

Receives the response of the next request that has completed, or the response of a specified request. If the 'Request' is NULL, the caller receives the next response; otherwise the response for the indicated request is returned.

The following response flags are defined for GetResponse:

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

6.4 List Routines

*** Memory management support in the List routines of the dynamic request invocation is not resolved. The particular issue is whether the ORB will allocate storage for return values and out parameters according to the list allocation mechanism described here, or using the same policy as for stubs. Alternatively stated, It is still unresolved whether an ORB mechanism, such as lists, is used to manage memory in both the dynamic and static interfaces; or memory managment is always handled in a language specific manner.

6.4.1 ORB_CreateItemList

```
ORBStatus ORB_CreateItemList (
  long Count,             /* number of items to allocate for list */
  NVList * List);         /* (out) newly created list */
  ORBExc    * Exc);       /* (in,out) exception */
```

Allocates a list of the specified size, and clears it for initial use. List items may be added to the list using the AddItemToList routine. Alternatively, they may be added by indexing directly into the list structure.

6.4.2 ORB_AddItemToList

```
ORBStatus ORB_AddItemToList (
  NVList List,          /* (in,out) list to be modified */
  ArgName Name,         /* name of item */
  ArgType Datatype,     /* item datatype */
  (void *) Value,       /* item value */
  long Len,             /* length of item value */
  NVFlags Flags);       /* item flags */
  ORBExc   * Exc);      /* (in,out) exception */
```

Adds a new item to the indicated list. The item is added after the previously added item. If the IN_COPY_VALUE flag has been set, a copy of the value is made and associated with the list (i.e. the program variable passed in may be used for some other purpose).

If a list structure is added as an item (e.g. a "sublist") the DEPENDENT_LIST flag may be specified which indicates that the sublist should be freed when the parent list is freed.

6.4.3 ORB_FreeList

```
ORBStatus ORB_FreeList (
  NVList List );        /* list to be freed */
  ORBExc   * Exc);      /* (in,out) exception */
```

Frees the list structure and any associated memory.

6.4.4 ORB_AllocateListMemory

```
ORBStatus ORB_AllocateListMemory (
   NVList   List,     /* list */
   long     Size,     /* amount of memory to allocate */
   (void *) Buf,      /* (out) allocated memory */
   ORBExc   * Exc);   /* (in,out) exception */
```

The requested amount of memory is allocated and is associated with the indicated list. When the list is freed all associated memory is freed along with it.

6.4.5 ORB_FreeListMemory

```
ORBStatus ORB_FreeListMemory (
  NVList   List );      /* list from whose memory is to be freed */
  ORBExc   * Exc);      /* (in,out) exception */
```

Frees any dynamically allocated memory associated with the list (e.g. those items that were copied on a list operation, or output arguments whose memory was dynamically allocated and associated with the list). The list structure itself is left unchanged.

6.4.6 ORB_GetListCount

```
ORBStatus ORB_GetListCount (
  NVList List,          /* list from which to retrieve the count */
  long * Count);        /* (out) size of list */
  ORBExc   * Exc);      /* (in,out) exception */
```

Returns the total number of items allocated for this list.

6.5 Context Objects

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

Context properties can represent a portion of a client's or application's environment that is meant be propagated to (and made implicitly part of) a server's environment (for example, the XWindows window identifier, or user preference information). Once a server has been invoked (i.e., subsequent to the properties being 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.

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. Persistent context objects have a name associated with them and are called "named context objects".

*** It is unresolved whether the ORB directly supports persistent context objects, or if the ORB is simply a client of context objects, which could have a variety of persistent or non-persistent implementations.

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).

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

Through the dynamic invocation interface you can provide a list of name-value pairs that override the corresponding properties values in the context argument; and can serve to temporarily change environment information or method resolution behavior.

The ORB may choose to pass more properties than those specified in the operation declaration. These properties may be used to supplement those properties specified on the operation definition and may also be propagated to the server's environment.

6.5.1 Logical structure of the Context Object

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 "failover" options on the GetCtxValues call.

Context objects may be named.

6.6 Context Object Routines

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 4.1.3 on page 47 for the valid set of characters that are allowed). Property names are stored preserving their case, however names cannot differ simply by their case.

*** The Context Object routines have not yet been cast as IDL-compliant interfaces.

6.6.1 ORB_GetDefaultCtx

```
ORBStatus ORB_GetDefaultCtx (
  CtxObj * CtxObject,      /* (out) context object */
  ORBExc * Exception);     /* (in,out) exception */
```

Returns a reference to the default process context object.

6.6.2 ORB_SetCtxValues

```
ORBStatus ORB_SetCtxValues (
  CtxObj CtxObject,        /* context object to modify */
  NVList AttValues,        /* property values to be changed */
  ORBExc * Exception);     /* (in,out) exception */
```

Sets one or more property values in the context object.

6.6.3 ORB_GetCtxValues

```
ORBStatus ORB_GetCtxValues (
  CtxObj CtxObject,        /* context object */
  CtxName StartScope,      /* search scope */
  CtxFlags OpFlags,        /* operation flags */
  string AttName,          /* name of property to retrieve */
  NVList * AttValues,      /* (out) requested property(s) */
  ORBExc * Exception);     /* (in,out) exception */
```

Retrieves the indicated context property value(s). If 'AttName' has a trailing wildcard character ("*"), then 'AttValues' may contain more than one property value.

Scope indicates the context object level at which to initiate the search for the specified properties (e.g. User, Group, System). If the property is not found at the indicated level, the search continues ("fails over") up the context object tree until a match is found or all context objects in the chain have been exhausted. Valid scope names are implementation specific.

If scope name is omitted, the search begins with the specified context object. It is invalid to specify a scope name that is at a "lower" level than that of 'CtxObject'.

The following operation flags may be specified:

CTX_NO_FAILOVER	Specifies that search failover is not to be performed; i.e. the search is limited to the specified (or defaulted) scope.

6.6.4 ORB_DeleteCtxValues

```
ORBStatus ORB_DeleteCtxValues (
  CtxObj CtxObject,        /* context object to modify */
  NVList AttValues,        /* property values to be deleted */
  ORBExc * Exception);     /* (in,out) exception */
```

Deletes the specified property and its value from the context object.

6.6.5 ORB_CreateCtx

```
ORBStatus ORB_CreateCtx (
  CtxObj ParentCtx,        /* context object to act as "parent" */
  CtxObj * CtxObject,      /* (out) context object */
  ORBExc * Exception);     /* (in,out) exception */
```

Creates a transient context object. If 'ParentCtx' is specified, the returned context object is chained into the parent context. That is, searches on the created context object will "failover", if necessary, to its parent. If no parent context is specified, an orphan context object is returned and no failover will be done.

6.6.6 ORB_DeleteCtx

```
ORBStats ORB_DeleteCtx (
  CtxObj CtxObject,        /* context object to be deleted */
  CtxFlags Flags,          /* operation flags */
  ORBExc * Exception);     /* (in,out) exception */
```

Deletes the indicated context object.

The following option flags may be specified:

CTX_DELETE_CHILDREN

Deletes the indicated context object and all of its children context objects, as well.

CTX_DISCONNECT_CHILDREN

Disconnects parent context object from its children, prior to being deleted. If neither option is specified, and there are one or more children context objects, an error will be returned.

6.6.7 ORB_OpenCtx

```
ORBStatus ORB_OpenNamedCtx (
  CtxName Name,            /* name of context object */
  CtxModes Mode,           /* read,write access */
  CtxObj * CtxObject,      /* (out) context object */
  ORBExc * Exception);     /* (in,out) exception */
```

Opens a named context object for reading or writing.

This routine allows access to named (persistent) context objects. Changes to a context object opened in this fashion are made permanent.

7 The Interface Repository

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

7.1 Philosophy

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

In order for an ORB to function correctly, it may require access to the definitions of the objects it is handling. Object definitions can be made available to an ORB in one of two forms:

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

Note that the IDL compiler and the Interface Repository provide similar services within the OMA. IDL specifications are input into both and both output interface information (either in a canonical or procedural form) which is used by clients, language bindings, and ORBs. It may be best to consider the Interface Repository as "directory software" that manages the storage, distribution, and management of interface specifications and the IDL compiler as the implementation of the primary operation provided on the "interface specification objects" maintained in the Interface Repository.

In particular, the ORB may use object definitions maintained in the Interface Repository to interpret/handle the values provided in a request

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

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

- To manage the installation and distribution of interface definitions
- To provide components of a CASE environment (e.g., an interface browser)
- To provide interface information to language-bindings (e.g., a compiler)
- To provide components of end-user environments (e.g., a menu bar constructor).

The interfaces defined here are the read-only interfaces required by clients of an ORB to enable their use of the dynamic invocation interface. In particular, the interfaces specified here are designed to allow clients to dynamically collect the information needed to construct requests. Note that the ORB provides an operation that allows a client to map any object reference to the object reference of the interface to which the object belongs; i.e., to an entry in the Interface Repository. This allows clients to discover the operations (and their signature) supported by an object.

7.2 Scope of an Interface Repository

Interface definitions are maintained in the Interface Repository as a set of objects which are accessed through a set of IDL-specified interface definitions.

Each Interface Repository maintains a collection of related interface definitions called an interface set.[1] An interface set may be local to a user (one set per user), may be local to a group of users on a single system, or may be globally available to all users within a network domain. Similarly, an interface set may describe all objects a client may use or it

may define only a subset of the objects a client may use (i.e., a client may need to use more than one interface set).

Each implementation of the Interface Repository determines the scope of an interface set. The scope of an interface set determines:

- The breadth of availability: the number of clients and services (ORBs) who use an instance of an Interface Repository.
- The width of availability: the number of Interface Repositories a client or service (ORB) may/must use.

The standard does not specify how interface sets relate to each other—how they are combined. For example, one implementation may require that all interface definitions an application uses (from the "root interfaces" down) must be in a single interface set (of which there is only one logical copy shared by many users) while another implementation may provide a scheme for combining interface sets (e.g., it might provide for federated hierarchies of named interface sets).

Regardless of how an implementation relates interface sets together:

- All interface names within an interface set must be unique
- Any single client's view of the interfaces within an interface set must be consistent.

7.3 Implementation Dependencies

An implementation of an Interface Repository requires some form of persistent object store. Normally the type of persistent object store used to implement an Interface Repository determines how an interface set is 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 an interface set maintained on a single machine. Alternatively, if an OODB is used to provide object storage, multiple copies of an interface set may be maintained each of which is distributed across several machines to provide both high- availability and load-balancing. The type of object store used may determine the scope of interface sets provided by an implementation of the Interface Repository. For example, it may determine whether each user has a local copy of an interface set or if there is one copy per community of users. The object store may also determine how consistent are views of an interface set within a community of users -- 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.

1. An instance of an Interface Repository may contain more than one interface set.

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 type and granularity of access controls available to constrain which users may use which interfaces and which parts of which interfaces. For example, the security mechanism may support only the provision of ACL-based controls which apply to interfaces in toto (i.e., whether a user may or may not use an interface). Alternatively, a more sophisticated security mechanism may provide access controls which permit selective protection of an interface's individual operations.

7.4 Basics of the Interface Repository Interface

7.4.1 Types of Interface Objects

Each interface managed in an Interface Repository is maintained as a collection of five types of objects:[2]

1. Interface_Def represents an interface type definition; contains lists of operations and attributes
2. Operation the definition of an operation on the interface; contains a list of parameters
3. Parameter the definition of an argument to an operation
4. Attribute the definition of an attribute of the interface.
5. Exception the definition of an exception for an operation.

7.4.2 Instances of Interface Objects

Each interface in the Repository is uniquely identified by the object reference assigned to its Interface_Def object. (This reference is the "interface reference"). In addition, each interface has a unique, implementation dependent identifier assigned that allows the interface definition object to be duplicated in other Interface Repositories without confusion.

The set of interfaces maintained by an Interface Repository is maintained in a (flat) container called Interface_Bin. [3]

2. The interface specifications listed in §7.5.1 on page 103 also define a set of "abstract" interfaces that are used to define generic attributes and operations. An implementation of the Interface Repository need not use these specific abstract interfaces to provide these same generic attributes and operations. In particular, implementations of ORB's that provide interfaces derived from the interfaces defined here are OMG compliant.
3. Note that each interface set is maintained in a flat container (i.e., as a list of interface objects) as opposed to being structured according to the inheritance graph. An implementation of the Interface Repository may choose to also maintain an interface set as a graph, but this is not required.

An instance of an interface set has the following containment structure:

Interface_Bin	one per interface set
Interface_Def(n)	one per interface in the set
Attribute(n)	one per attribute per interface
Operation(n)	one per operation per interface
Parameter(n)	one per parameter per operation
Exception(n)	one per exception per operation

7.4.3 Attributes of Interface Objects

The interface specifications for each type of interface object specifies the attributes maintained by that object (see below, section 7.5 on page 103). These attributes correspond directly to IDL statements. An implementation may choose to maintain additional attributes to facilitate managing the Repository or to record additional (proprietary) information about an interface.

7.4.4 Operations on Interface Objects

The interface specifications for the interface objects given here define a set of basic operations for clients wishing to access the interface objects. They are not intended to provide sufficient semantics for the construction of basic interface browsers or command-line interfaces to the Interface Repository, nor to provide an administrative interface. Implementations of the Interface Repository must have additional operations for creation of the component objects within an Interface object, but these need not be part of the public interface.

Only a minimal set of operations have been specified for interface objects. Additional operations and attributes that an implementation of the Interface Repository may provide could support the versioning of interfaces, deletion of interfaces and reverse compilation of specifications (i.e., the generation of a files containing an object's IDL specification).

7.5 Interface Repository Interface

7.5.1 Definitions

The inheritance relationships for interfaces comprising the Interface Repository are shown in the following nested list:

Intf_Root(*)
    Attribute
    Exception
    Parameter
    Container(*)
        Interface_Bin
        Interface_Def
        Operation

* This interface is abstract (i.e., not instantiable)

The Containment relationships for the interface objects in the Interface Repository are shown in the following nested list:

```
Interface_Bin(1)          /* 1 per 'interface set' */
   Interface_Def(n)       /* independent child objects */
      Attribute(n)        /* dependent child objects */
      Operation(n)        /* independent child objects */
         Parameter(n)     /* dependent child objects */
         Exception(n)     /* dependent child objects */
```

Orb_idl.h is a standard include file which contains at least the following declarations for the IDL transformation:

```
typedef long dtime_t;
typedef string <UNBOUNDED> interface_id_t;
typedef string <UNBOUNDED> typecode_t;
typedef ... Name_Value; /* the "well-known any type" */
typedef sequence <Name_Value,UNBOUNDED> N_V_List;
typedef sequence <objref_t,UNBOUNDED> objref_list_t;
typedef struct nm_tag {
     string<ORBNAMELEN> name;
     string opaque; } name_t;
enum io_mode {IN, OUT, INOUT};
```

*** This definition of the interface repository depends on the "any" type (see the issue about datatypes in the dynamic invocation interface).

*** It is unresolved if all of the attributes (eg. Time_Created) specified for the three types of "Container" objects are to be included in the interface.

*** It is unresolved as to whether the "survey" interfaces should be included.

7.5.2 Interface Definition for Type Intf_Root

```
#include "orb_idl.h"

/*
 * The Intf_Root interface represents the most generic form of
 * interface from which all other type repository interfaces are derived.
 */
interface Intf_Root {

/* attributes */

 attribute string type_name<ORB_NAME_LEN>; /* the name of the */
 /* interface to which */
 /* the object belongs */

 readonly attribute objref_t object_id; /* the object reference */
 /* for the object */

 attribute string Object_Name<ORB_NAME_LEN>;/* Name of this object */

/* operations */

 ORBSTATUS Get_Attribute(in N_V_List names,
 out N_V_List values);

 /* Description:
 *
 * Attribute values are returned in the same order as
 * the attribute names passed in.
 *
 * Errors:
 *
 * ORB_ATTR_NAME_INVALID The attribute name is not valid
 */

 ORBSTATUS Contained_By(out objref_list_t Container_List);

 /* Description:
 *
 * Retrieves a list of container objects that this
 * object is inside, and returns them in Container_List
 *
 * Errors:
 *
 * No additional errors.
 */

} /* end interface Intf_Root */
```

7.5.3 Interface Definition for Type Container

```
typedef struct obj_record {
 objref_t object_id;
 N_V_List *column_values;
} survey_record_t;

typedef sequence <survey_record_t,UNBOUNDED> survey_list_t;

 /*
 * This type defines the behavior and attributes of
 * generic container objects. This interface supports
 * queries that return a list of contained objects and
 * attribute values for each of the contained objects.
 * The caller controls the selection, ordering, and projection
 * criteria.
 */

interface Container : Intf_Root {


 /* attributes */

 attribute string Comments<128>; /* A description field */

 readonly attribute dtime_t Time_Created; /* Time object was created */

 readonly attribute dtime_t Time_Modified; /* Time object was modi-
fied.*/
 /* It should be noted */
 /* that the object is con- */
 /* sidered to be modified if*/
 /* any of its attributes have*/
 /* changed, or its array */
 /* members or child objects */
 /* have been added or deleted.*/

 readonly attribute string Created_By<ORB_USER_NAME_LEN>;
 /* User who created the object */

 readonly attribute string Modified_By<ORB_USER_NAME_LEN>;
 /* User who last updated the */
 /* object. The object is con- */
 /* sidered to be modified if its */
 /* attributes have been modified */
 /* or if its array members or */
 /* child objects have been added */
 /* or deleted. */
 attribute string Select_Format<ORB_SELECT_FORMAT_LEN>;
```

```
/* This is the default rule for selecting which objects are returned.
* A rule consists of attribute names, relational operators, values,
* ANDs * and ORs.
*
* The syntax for a selection rule is:
*
* rule -> item ({AND|OR} item)*
*
* item -> Attr Relop Value
*
* where Attr is an attribute name; Relop is one of =, <>,
* >, <, >=, <=; The case of AND and OR is not important.
   * Parentheses are not supported at this
* time. Character strings must be enclosed in single quota-
* tion marks, numeric values need not be. Attribute names
* are not enclosed in single quotation marks. The maximum
* number of select criteria allowed is
* ORB_MAX_SELECT_CRITERIA.
*
* Example:
*
* Created_By = 'George' AND Time_Modified > 615215627 AND
* Time_Modified = Time_Created (June 30, 1989 at about 9:13 am)
*/

attribute string Sort_Format<ORB_SORT_FORMAT_LEN>;

/* Default list of attributes to sort by, along with a single
* character for the sort order; i.e., either A(scend)
* or D(escend).
*
* Example:
*
* Created_By A Time_Created D
*
* The maximum number of sort orders allowed is
* ORB_MAX_SORT_ORDERS.
*/

attribute string Attribute_Format<ORB_ATTR_FORMAT_LEN>;

/* names of attribute values to return
*
* Example:
*
* Created_By Object_Name
*
* Header strings must be enclosed in single quotation
* marks if there is embedded white space (space, tab,
```

```
* nylon). Note that the object reference for each child object
* is always returned, and always in the first position.
* If that is all the information that is desired,
* Attribute_Format can be empty. The maximum number of columns
* allowed is ORB_MAX_COLUMNS, not counting the object reference,
* which is always returned.
*
* Select_Format, Sort_Format and Attribute_Format can be
* overridden on an Get_Survey call. See Get_Survey below.
*/


/* operations */

ORBSTATUS Get_Children (
   in name_t Child_Interface,
   out objref_list_t Children
));

/* Description:
*
* Get_Children will return object references of a particular
  * interface type that are contained within the container.
*/

ORBSTATUS Get_Survey(
in long Return_Data;
in name_t Child_Interface_Name;
in string Select_Format<UNBOUNDED>;
in string Sort_Format<UNBOUNDED>;
in string Attribute_Format<UNBOUNDED>;
out long Entries_Returned;
out long Total_Entries;
out long Num_Attributes;
out sequence<survey_record_t,UNBOUNDED> Survey_Buffer;

);

/* Description:
*
* Get_Survey freezes a view of a container/contained
* relationship in this object. The container is not
* locked while the frozen view exists. Additional
* objects may be added to the container or objects can
* be deleted in the background while the view is in
* force. For that reason, some of the object handles
* in the survey may not be valid. The caller should be
* prepared to handle this gracefully. The frozen view
* allows the caller to create a scrolling display
* without worrying about the previously retrieved parts
```

```
* of the survey changing.
*
* Return_Data is used to control how much survey data
* is to be returned by Get_Survey. If set to 0, no
* data is returned; in this case, Entries_Returned will
* be 0 and Total_Entries will contain the number of
* items in the survey. If Return_Data is set to a
* positive integer value, that many items will be
* returned.
*
* Normally Get_Survey uses attributes of the container
* object to determine what objects to retrieve,
* how to sort them, and what attributes are
* desired. The attributes Select_Format, Sort_Format,
* and Attribute_Format are used for this purpose (see the
* attributes definitions). These values can temporarily be
* over-ridden by supplying them as input to the call.
* If the attribute is to be over-ridden, and the
* desired format is for it to be empty, then the string
* ORB_NULL_FORMAT should be placed in the appropriate
* location. It is possible to override any combination
* of Select_Format, Sort_Format, and Attribute_Format.
*
* The output parameters Entries_Returned and Total_Entries
* contain the number of entries returned in
* this Get_Survey call, and the total number of
* entries in the survey, respectively.
*
* The Survey Handle must be used to read additional data from
* the survey. It is used in other survey operations. It
* is not an object handle and should not be treated as
* such (there is no need to perform a Use operation on
* it, for example).
*
* Num_Attributes is the number of columns in the output as
* determined by Attribute_Format. The value of Num_Attributes
* includes the object reference of the objects, which is re-
* turned regardless of the contents of the Attribute_Format.
*
* The Survey_Buffer is a sequence of survey records, each of
* which contains an object reference and a list of Name_Value
* items corresponding to the attributes to be returned.
*
* Errors:
*
* Major error codes:
*
* ORB_RETURN_DATA_INVALID A negative value was speci-
* fied for Return_Data.
*
```

```
* ORB_SELECT_FORMAT_INVALID The select_format used con-
* tains a syntax error or
* specifies a nonexistent
* attribute.
*
* ORB_SORT_FORMAT_INVALID The sort_format used contains
* a syntax error or specifies a
* nonexistent attribute.
*
* ORB_ATTR_FORMAT_INVALID The cols_format used contains
* a syntax error or specifies a
* nonexistent attribute.
*
* ORB_ATTR_NAME_INVALID Invalid attribute name in
* format.
*
* ORB_ATTR_NOT_SURVEYABLE An attribute specified in the
* Attribute_Format is not survey-
* able.
*
* ORB_TOO_MANY_ATTRS The number of attributes
* named is greater than
* ORB_MAX_SELECT_CRITERIA,
* ORB_MAX_SORT_ORDERS, and
* ORB_MAX_COLUMNS for
* Select_Format, Sort_Format,
* and Attribute_Format,
* respectively.
*
* ORB_RELOP_INVALID The relational operator is
* invalid in Select_Format.
*
* ORB_ATTR_VALUE_INVALID The value provided for the
* attribute in the select cri-
* teria is invalid for that
* type of attribute.
*
* ORB_REL_CON_INVALID The relational connector in
* the Select_Format is invalid.
*
* ORB_SORT_FLAG_INVALID The flag in the Sort_Format
* was not of a valid value.
* See the description of
* Sort_Format for the valid
* Sort_Flag values.
*
* ORB_DUPLICATE_ATTR The same attribute is
* specified more than once in
* Attribute_Format.
*
```

```
 * Additional data may be returned in the attribute name
 * field of the error return buffer, for example, the
 * name of an invalid attribute.
 */


} /* end interface Container */
```

7.5.4 Interface Definition for Type Exception

```
#include "orb_idl.h"

interface Exception: Intf_Root {

 /* attributes */

 attribute typecode_t exception_type;


} /* end interface definition for type Exception */
```

7.5.5 Interface Definition for Type Attribute

```
#include <orb_idl.h>

interface Attribute : Intf_Root {

 /* attributes */

 attribute typecode_t attribute_type;


 attribute boolean readonly;

 /*
 * READONLY or READ/WRITE access for this attribute.
 */

 attribute Interface_Def inherited_from;

 /*
 * The Interface from which the attribute is inherited.
 * If the attribute is new at a given Interface level, the value
 * is the current Interface.
 */


} /* end interface Attribute */
```

7.5.6 Interface Definition for Operation

```
#include <orb_idl.h>

interface Operation : Container {


 /*
 * Definitions of operations are stored as Operation objects.
 * Each operation has a list of Parameters and a list of
 * Exceptions as dependent member objects.
 */

 /* attributes */

 attribute Interface_Def inherited_from;

 /*
 * The Interface from which the Operation is inherited.
 * If the Operation is new at a given Interface level, the value
 * is the current Interface.
 */

 attribute typecode_t return_type;

 /*
      * The return type specified for this Operation in IDL
      */

 attribute sequence<string,UNBOUNDED> context_names;
 /*
 * list of names of the context values declared for
 * this operation
 */

 /* operations */

 ORBSTATUS describe_operation (
   in NV_List operation_attributes,
   in NV_List parameter_attributes,
   in NV_List exception_attributes,
   out NV_List attribute_info
 );

 /*
 * Description:
 *
 * describe_operation is used to gather attributes of an
 * operation, attributes of all its parameters and attributes
```

```
 * of all its exceptions. The client sets the attributes desired
 * in the input lists.
 * On output the client receives a list in the following form:
 * the list of operation_attributes
 * a list of parameter object references and for each the requested
 * attributes
 * a list of exception object references and for each the requested
 * attributes
 */

 ORBSTATUS prep_arglist (
    out ArgList arguments
 );

 /*
 * Description:
 *
 * prep_arglist scans the list of parameters and returns an arglist
 * so that a client can assign values to each entry and use the list
 * in a CreateRequest call
 */

} /* end interface operation */
```

7.5.7 Interface Definition for Parameter

```
#include <orb_idl.h>

interface Parameter : Intf_Root {

 /*
 * This type defines the definition of an argument to an operation.
 */

 /* attributes */

 attribute typecode_t type;

 attribute io_mode inout; /* IN, OUT, or INOUT */


} /* end interface Parameter */
```

7.5.8 Interface Definition for Interface_Def

```
#include <orb_idl.h>

 /*
 * An Interface_Def represents an interface type definition.
 * An Interface_Def can contain Operation objects as
 * independent child objects and Attribute objects as dependent
 * member objects
 */

interface Interface_Def : Container {

 /* attributes */

 attribute sequence<name_t,UNBOUNDED> super_types;

 attribute interface_id_t interface_id;

 attribute boolean instantiable;

 /*
 * TRUE if a user may create an object of this type.
 * FALSE if an abstract (place-holder) type.
 */

 /* operations */


 /*
 * A client can retrieve a list of Operations or Attributes comprising
 * the Interface_Def by using the Get_Children operation inherited from
 * Container.
 */

} /* end interface Interface_Def */
```

7.5.9 Interface Definition for Interface_Bin

```
#include <orb_idl.h>

 /*
 * An instance of Interface_bin is a container holding
 * instances of Interface_Def.
 *
 */


interface Interface_Bin : Container {

 /* operations */

 /*
 * the only legal value for input to the Get_Children operation
 * (inherited from Container) is 'Interface_Def'
 */

} /* end interface Interface_Bin */
```

7.6 Typecodes

Typecodes are values that represent invocation argument types and attribute types. They can be obtained from the type repository or from IDL compilers. In particular, clients who use the dynamic invocation interface may need to use the typecode as a means to decipher information about objects that were not known to the client at compile time. Complex datatypes require a typecode representation that allows construction of a representation of the datatype either as a list or as native structure in the language binding.

Typecodes are themselves values that can be passed as invocation arguments. In order to allow different ORB implementations to hide extra information in typecodes, the representation of typecodes will be opaque (like object references). However, we will assume that the representation is such that typecode "literals" can be placed in C include files.

Abstractly, typecodes consist of a "kind" field, and a "parameter list" field. Two typecodes are equal if these two fields are equal. However, because of extra information, certain interfaces may require the use of repository or the operations below.

The following is a C binding for typecodes.

*** The exact definition of typecodes has not yet been resolved. What is required is that all the types representable in IDL be handled.

NOTE *There should be an entry in TC_KIND for every base data type defined in IDL and for every well-known opaque type that is a standard part of the standard ORB.*

The basic operations on typecodes are

```
typedef ... typecode_t;
typedef union {typecode_t type; long num; string name;} TC_param;
typedef long bool;
typedef enum { tk_short, tk_long, tk_ushort, tk_ulong,
    tk_bool, tk_char, tk_float, tk_double,
    tk_octet, tk_named_value, tk_typecode,
    tk_objref, tk_struct, tk_union, tk_enum,
    tk_string, tk_sequence, tk_array } TC_kind;

bool typecode_equal (typecode_t x, typecode_t y);

TC_kind typecode_kind (typecode_t x);

int typecode_num_params (typecode_t x);

TC_param typecode_param (typecode_t x, int index);

typecode_t parse_typecode (typecode_t tc_in, typecode_t tc_tail));

typecode_t typecode_create (TC_kind kind,
   int count; TC_param* params);
```

The create operation returns NULL for an illegal combination of kind and parameters. The legal combinations are:

```
tk_short        {}
tk_long         {}
tk_ushort       {}
tk_ulong        {}
tk_bool         {}
tk_char         {}
tk_float        {}
tk_double       {}
tk_octet        {}
tk_named_value  {}
tk_typecode     {}
tk_objref       { interface-name-string }
tk_struct       { field-name, typecode, ... (repeat pairs) }
tk_union        { switch-type, field-name, typecode , ...
                    (repeat pairs) }
tk_enum         { enum-id-string, ... }
tk_string       { maxlen-integer }
tk_sequence     { typecode, maxlen-integer }
tk_array        { typecopde, length-integer }
```

Clearly we could make the programmer's life slightly easier if we provided special constructor/deconstructor operations for the individual composite types. Given a complete set of these, we could dump the general create/ num_params/param operations.

The predefined typecode literals are:

```
TC_SHORT
TC_LONG
TC_USHORT
TC_ULONG
TC_BOOL
TC_CHAR
TC_FLOAT
TC_DOUBLE
TC_OCTET
TC_NAMED_VALUE
TC_TYPECODE
TC_OBJECT= tk_objref {"root-object-type"}
```

And a number of others for well-known composite types, as seems appropriate.

If "typedef ... FOO" is a IDL type declaration, the IDL compiler could (if asked) produce a declaration of TC_FOO for use with the dynamic invocation interface.

8 ORB Interface

The ORB interface is the interface to those ORB functions that do not depend on which object adapter is used. These operations are the same for all ORBs and all object implementations, and can be performed either by clients of the objects or implementations. Some of these operations appear to be on the ORB, others appear to be on the object reference. Because the operations in this section are implemented by the ORB itself, they are not in fact operations on objects, although they may be described that way and the language binding will, for consistency, make them appear that way.

8.1 Converting Object References to Strings

Because an object reference is opaque and may differ from ORB to ORB, the object reference itself is not a convenient value for storing references to objects in persistent storage or communicating references by means other than invocation. Two problems must be solved: allowing an object reference to be turned into a value that a client can store in some other medium, and ensuring that the value can subsequently be turned into the appropriate object reference.

An object reference may be translated into a string by the operation `objref_to_string`. The value may be stored or communicated in whatever ways strings may be manipulated.

Subsequently, the `string_to_objref` operation will accept a string produced by `objref_to_string` and return the corresponding object reference.

```
interface ORB {
   string                          objref_to_string (in Objref objref);
   Objref       string_to_objref (in string str);
};
```

To guarantee that an ORB will understand the string form of an object reference, that ORB's `objref_to_string` operation should be used to produce the string. Since in general a client does not know or care which ORB is used for a particular object reference, the client can choose whatever ORB is convenient.

8.2 Object Reference Operations

There are some operations that can be done on any object. These are not operations in the normal sense, in that they are implemented directly by the ORB, not passed on to the object implementation. We will describe these as being operations on the object reference, although the interfaces actually depend on the language binding. As above, where we used interface Objref to represent the object reference, we will define an interface for Objref:

```
interface Objref {
   Implementation_Def   get_implementation ();
   Interface_Def        get_interface ();
   boolean              equal (in Objref objref);
   Objref               duplicate ();
   void                 release ();
};
```

8.2.1 Determining the Object Implementation and Interface

An operation on theobject reference, `get_interface`, returns an object in the Interface Repository, which provides type information that may be useful to a program. See Chapter 7 for a definition of operations on the Interface Repository. An operation on the Objref called `get_implementation` will return an object in an implementation repository that describes the implementation of the object. See Chapter 9 for information about the Implementation Repository.

```
   Interface_Def        get_interface ();
   Implementation_Def   get_implementation ();
```

8.2.2 Duplicating and Releasing Copies of Object References

Because the object reference is 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.

```
Objref   duplicate ();
void     release ();
```

If more than one copy of an object reference is needed, the client may create a **duplicate**. Note that the object implementation is not involved in creating the duplicate, and that the implementation cannot distinguish whether the original or a duplicate was used in a particular request. The **equal** operation will return **TRUE** when applied to an objref and a duplicate.

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.

8.2.3 Equality of Two Object References

It is often useful for the client to be able to determine if two object references are identical. The equal operation returns true if the ORB considers the two object references to be identical. Two object references that are different as far as the ORB is concerned might refer to the same object, depending on the object semantics.. The object implementation is not involved in the equality test.

```
boolean  equal (in Objref objref);
```

9 The Basic Object Adapter

An Object Adapter is the primary interface that an implementation uses to access ORB functions. The *Basic Object Adapter* (BOA) is an interface intended to be widely available and to support a wide variety of common object implementations. It includes convenient interfaces for generating object references, registering implementations that consist of one or more programs, activation of implementations, and authentication of requests. It also provides a limited amount of persistent storage for objects that can be used for connecting to a larger or more general storage facility, for storing access control information, or other purposes.

Most of the Basic Object Adapter interface can be expressed in IDL, since the interface is to the operations on the object adapter. Some of the operations to bind the implementation to the object adapter depend on the language mapping. We will note such dependencies, but still use IDL as the means to describe the interface.

9.1 Role of the Basic Object Adapter

One object adapter, called the Basic Object Adapter, should be available in every ORB implementation; although the BOA will generally have an ORB-dependent implementa-

tion, object implementations that use it should be able to run on any ORB that supports the required language mapping, assuming they have been installed appropriately.

Other Object Adapters are likely to be created. Ordinarily, it is not necessary for a client of an object to be concerned about which Object Adapter is used by the implementation.

The following functions are provided through the Basic Object Adapter:

- Generation and interpretation of object references,
- Authentication of the principal making the call,
- Activation and deactivation of the implementation,
- Activation and deactivation of individual objects, and
- Method invocation through skeletons.

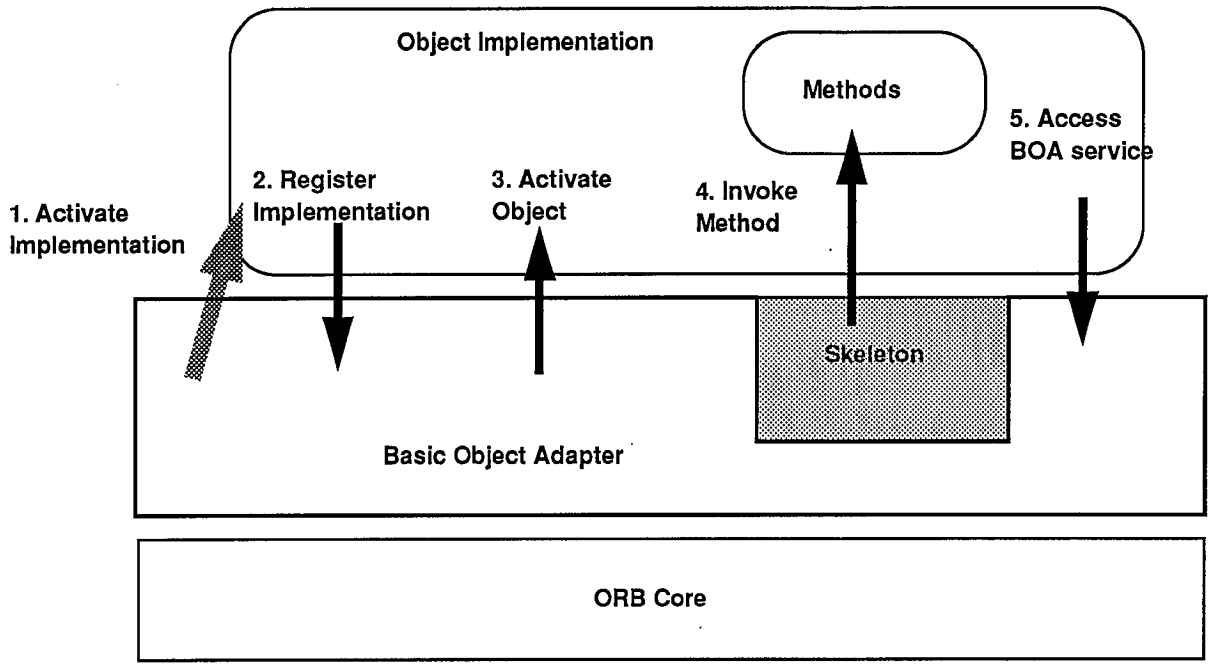
The Basic Object Adapter supports object implementations that are constructed from one or more programs[1]. The BOA activates and communicates with these programs using operating system facilities that are not part of the ORB. Therefore the BOA requires some information that is inherently non-portable. Although not defining this information, the BOA does define the concept of an Implementation Repository which can hold this information, allowing each system to install and start implementations in the way that is appropriate for that system.

The mechanism for binding the program to the BOA and ORB is also not specified because it is inherently system and language-dependent. We assume that the BOA can connect the methods to the skeleton by some means, whether at the time the implementation is compiled, installed, or activated, etc. Subsequent to activation, the BOA can make calls on routines in the implementation and the implementation can make calls on the BOA.

FIG. 11 on page 125 shows the structure of the Basic Object Adapter, and some of the interactions between the BOA and an Object Implementation. The Basic Object Adapter will start a program to provide the Object Implementation, in this example, a per-class server (1). The Object Implementation notifies the BOA that it has finished initializing and is prepared to handle requests (2). When the first request for a particular object arrives, the implementation is notified to activate the object (3). On subsequent requests, the BOA calls the appropriate method using the per-interface skeleton (4). At various times, the implementation may access BOA services such as object creation, deactivation, etc. (5).

1. The term "program" is meant to include a wide range of possible constructs, including scripts, loadable modules, etc., in addition to the traditional notions of an application or server.

FIG. 11 The Structure and Operation of the Basic Object Adapter



The BOA exports operations that are accessed by the Object Implementation. The BOA also calls the Object Implementation under certain circumstances. The interface between a particular version of the BOA and the ORB it runs on is private, as is the interface between the BOA and the skeletons. Thus, the BOA can exploit features or overcome limitations of a specific ORB, and can cooperate with the ORB and skeletons to provide a set of portable interfaces for the object implementation.

9.2 Basic Object Adapter Interface

The BOA interface is specified in IDL, so that the way it is accessed in any programming language is specified by the client side language mapping for that language. Some data structures used by the BOA are specific to a given language mapping, so most IDL compilers will not be able to accept this definition literally.

In practice, the BOA is most likely to be implemented partially as a separate component and partially as a library in the Object Implementation. The separate component is required to do activation when the implementation is not present. The library portion is needed to establish the linkage between the methods and the skeleton. The exact partitioning of functionality between these parts is implementation dependent. Generally, there will

appear to be a BOA object in the object implementation. When it is invoked, some operations are satisfied in the library, some in an external server, and some in the ORB.

The following is the approximate interface definition for the BOA object. More details will be provided as the operations are discussed.

```
interface Interface_Def;              // from Interface Repository
interface Implementation_Def;         // from Implementation Repository
interface Objref;                     // an object reference
interface Principal;                  // for the authentication service

typedef sequence <octet, 1024> id_t;

interface BOA {
   Objref    create (in id_t id, in Interface_Def intf,
                in Implementation_Def impl);
   void      dispose (in Objref obj);

   id_t      get_id (in Objref obj);
   void      change_implementation (in Objref obj,
                in Implementation_Def impl);

   Principal   get_principal (in Objref obj, in Request req);
   void      raise_exception (in long id, in string userid,
                in void *param);

   void      impl_is_ready (in Implementation impl);
   void      deactivate_impl (in Implementation impl);
   void      obj_is_ready (in Objref obj, in Implementation impl);
   void      deactivate_obj (in Objref obj);
};
```

Requests by an implementation on the BOA are of three kinds:

1. Operations to create or destroy object references, or query or update the information the BOA maintains for an object reference.
2. Operations associated with a particular request.
3. Operations to maintain a registry of active objects and implementations.

Requests by the BOA to an implementation are made with skeletons or using an implementation's runtime language mapping information, and are of three kinds:

1. Activating an implementation.
2. Activating an object.
3. Performing an operation (through a skeleton method).

Each of the BOA operations is described in detail later in this section; the requests of the BOA to an implementation are described in the language mapping section.

9.2.1 Registration of Implementations

The Basic Object Adapter expects information describing the implementations to be stored in an *Implementation Repository.* The Implementation Repository ordinarily is updated at program installation time, but may be set up incrementally or otherwise. There are objects with an IDL interface called `Implementation_Def`, which capture this information. The Implementation Repository may contain additional information for debugging, administration, etc. Note that the Implementation Repository is logically distinct from the Interface Repository, although they may in fact be implemented together.

The *Interface Repository* contains information about interfaces. There are objects with an IDL interface called `Interface_Def`, which capture this information. The Interface Repository may contain additional information for debugging, administration, browsing, etc. The ORB Core may or may not make use of the Interface Repository or the Implementation Repository, but the ORB and BOA use these objects to associate object references with their interfaces and implementations.

9.2.2 Activation and Deactivation of Implementations

There are two kinds of activation that a BOA needs to perform as part of operation invocation. The first, discussed in this section, is *implementation activation*, which occurs when no implementation for an object is currently available to handle the request. The second, discussed later, is *object activation*, which occurs when no instance of the object is available to handle the request.

Implementation activation requires coordination between the BOA and the program(s) containing the implementation. We use the term *server* as the separately executable entity that the BOA can start on a particular system. In a POSIX environment, a server would be a process. In most systems, a server corresponds to the notion of a program, but it can correspond to whatever the appropriate system facility is in a particular environment.

The BOA initiates activity by the implementation by starting the appropriate server, probably in an operating system-dependent way. The implementation initializes itself, then notifies the BOA that it is prepared to handle requests by calling `impl_is_ready` or `obj_is_ready`[2]. Between the time that the program is started and it indicates it is ready, the BOA will hold further requests for that server pending. After that point, the BOA, through the skeletons, will make calls on the methods of the implementation.

2. The latter is for per-object servers.

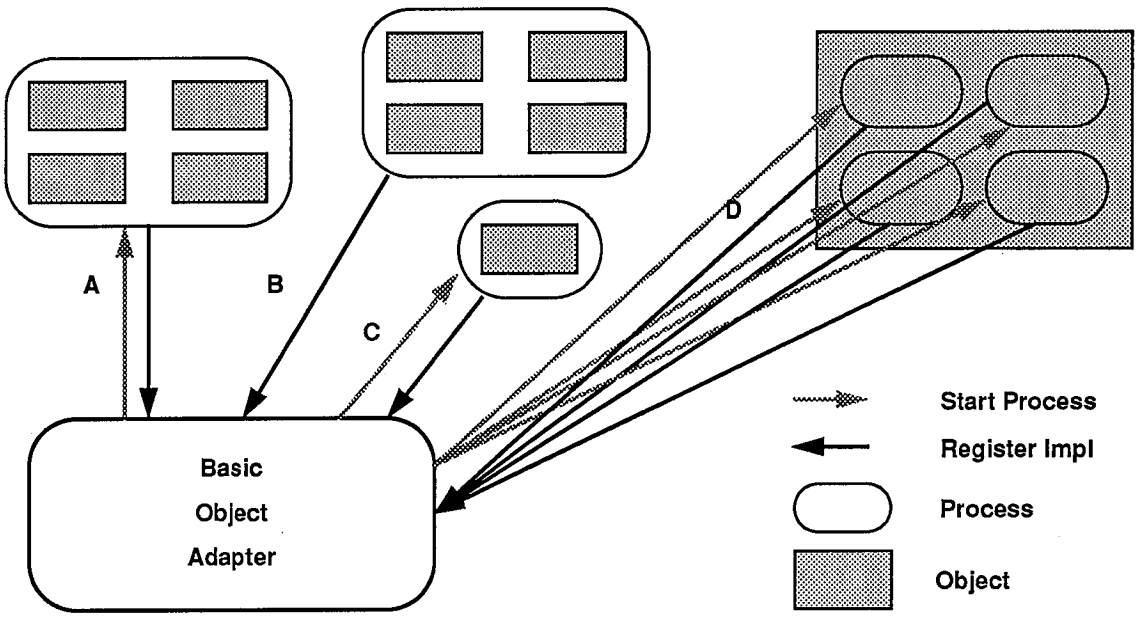
```
void  impl_is_ready (in Implementation_Def impl);
void  obj_is_ready (in Objref obj, in Implementation_Def impl)
```

An *activation policy* describes the rules that a given implementation follows when there are multiple objects or implementations active. There are four policies that all BOA implementations support for implementation activation:

- A *shared server* policy, where multiple active objects of a given implementation share the same server.
- An *unshared server* policy, where only one object of a given implementation at a time can be active in one server.
- A *server-per-method* policy, where each invocation of a method is implemented by a separate server being started, with the server terminating when the method completes.
- *Persistent server* policy, where the server is activated by something outside the BOA. The server nonetheless must register with the BOA to receive invocations. A persistent server is assumed to be shared by multiple active objects.

These kinds of implementation activation are illustrated in FIG. 12 on page 129. Case A is a shared server, where the BOA starts a process which then registers itself with the BOA. Case B is the case of a persistent server, which is very similar but just registers itself with the BOA, without the BOA having had to start a process. An unshared server is illustrated in case C, where the process started by the BOA can only hold one object; the server-per-method policy in case D causes each method invocation to be done by starting a process.

FIG. 12 Implementation Activation Policies



9.2.2.1 Shared Server Activation Policy

In a shared server, multiple objects may be implemented by the same program. This is likely to be the most common kind of server. The server is activated the first time a request is performed on any object implemented by that server. When the server has initialized itself, it notifies the BOA that it is ready by calling `impl_is_ready`. Subsequently, the BOA will deliver requests or object activations for any objects implemented by that server. The server remains active and will receive requests until it calls `deactivate_impl`. The BOA will not activate another server for that implementation if one is active.

Before the first request is delivered for a particular object, the object activate routine of the server is called. An object remains active as long as its server is active, unless the server calls `deactivate_obj` for that object.

9.2.2.2 Unshared Server Activation Policy

In an unshared server, each object is implemented in a different server. This kind of server is convenient if a object is intended to encapsulate an application or if the server requires exclusive access to a resource such as a printer. A new server is activated the first time a request is performed on the object. When the server has initialized itself, it notifies the BOA that it is ready by calling `obj_is_ready`. Subsequently, the BOA will deliver

requests for that object. The server remains active and will receive requests until it calls `deactivate_obj`.

A new server is started whenever a request is made for an object that is not yet active, even if a server for another object with the same implementation is active.

9.2.2.3 Server-per-Method Activation Policy

Under the server-per-method policy, a new server is always started each time a request is made. The server runs only for the duration of the particular method. Several servers for the same object or even the same method of the same object may be active simultaneously. Because a new server is started for each request, it is not necessary for the implementation to notify the BOA when an object is ready or deactivated.

The BOA activates an implementation for each request, whether or not another request for that operation, object, or implementation is active at the same time.

9.2.2.4 Persistent Server Activation Policy

Persistent servers are those servers which are activated by means outside the BOA. Such implementations notify the BOA that they are available using the `impl_is_ready` operation. Once the BOA knows about a persistent server, it treats the server as a shared server, sending it activations for individual objects and method calls. If no implementation is ready when a request arrives, an error is returned for that request.

9.2.3 Generation and Interpretation of Object References

Object references are generated by the BOA using the ORB when requested by an implementation. The BOA and the ORB work together to associate some information with a particular object reference. This information is later provided to the implementation upon the activation of an object. Note that this is the only information an implementation may use portably to distinguish different object references. The BOA operation used to create a new object reference is:

```
Objref    create (in id_t id, in Interface_Def intf,
            in Implementation_Def impl);
```

The `id` is immutable identification information, chosen by the implementation at object creation time, and never changed during the lifetime of the object. The `intf` is the Interface Repository object that specifies the complete set of interfaces implemented by the object. The `impl` is the Implementation Repository object that specifies the implementation to be used for the object.

A typical implementation will use the `id` value to distinguish different objects, but it is free to use it in any way it chooses or to assign the same value to different object references. Two object references created with the same parameters are *not* the same object ref-

erence (for example, they are not equal) as far as the ORB is concerned, although the implementation may or may not treat them as references to the same object. Note that the object reference itself is opaque and may be different for different ORBs, but the `id` value is available portably in all ORBs. Only the implementation can normally interpret the `id` value. The operation to get the `id` is a BOA operation:

```
id_t        get_id (in Objref obj);
```

It is possible for the implementation associated with an object reference to be changed. This will cause subsequent requests to be handled according to the information in the new implementation. The operation to set the implementation is a BOA operation:

```
void        change_implementation (in Objref obj,
              in Implementation_Def impl);
```

NOTE *Care must be taken in order to change the implementation after the object has been created. There are issues of synchronization with activation, security, and whether or not the new implementation is prepared to handle requests for that object. The* `change_implementation` *operation affects all copies of that particular object reference.*

If an object reference is copied, all copies have the same `id`, `interface`, and `implementation`.

An implementation is allowed to dispose of an object it has created by asking the BOA to invalidate the object reference. The implementation is responsible for deallocating all other information about the object. After a `dispose` is done, the ORB and BOA act as if the object had never been created, and attempts to issue requests on any existing object references for that object will fail.

```
void        dispose (in Objref obj);
```

Note that all of the operations on object references in this section may be done whether or not the object is active.

9.2.4 Authentication and Access Control

The BOA does not enforce any specific style of security management. It guarantees that for every method invocation (or object activation) it will identify the principal on whose behalf the request is performed. The object implementation can obtain this principal by the operation:

```
Principal get_principal (in Objref obj, in Request req);
```

The `obj` parameter is the object reference passed to the method. If another object is used the result is undefined. The `req` parameter is the language-mapping-specific request identifier passed to the method.

The meaning of the principal depends on the security environment that the implementation is running in. The decision of whether or not to permit a particular operation is left up to the implementation. Typically, an implementation will associate access rights with particular objects and principals, and will examine those access rights to determine if the principal making the request has the privileges required by the particular method. An implementation could store the access control information, or a reference to the access control information for this object, in the `id` for an object.

9.2.5 Persistent Storage

Objects (or, more precisely, object references) are made persistent by the BOA and the ORB, in that a client that has an object reference can use it at any time without warning, even if the implementation has been deactivated or the system has been restarted. Although the ORB and BOA maintain the persistence of object references, the implementation must participate in keeping any data outside the ORB and BOA persistent.

Toward this end, the BOA provides a small amount of storage for an object in the `id` value. In most cases, this storage is insufficient and inconvenient for the complete state of the object. Instead, the implementation provides and manages that storage, using the `id` value to locate the actual storage. For example, the `id` value might contain the name of a file, or a key for a database system that holds the persistent state.

9.3 C Language Mapping for Object Implementations

Different programming languages may provide access to the basic ORB functionality in different ways. Most of the issues of language mapping apply to all operations on all interfaces, and address such questions as the programming language view of the object reference, conventions for calling stubs and being called by skeletons, means of passing exception information, etc. There are a few details that apply specifically to the object adapter, such as how the implementation methods are connected to the skeleton.

9.3.1 Operation-specific details

Chapter 5 defines most of the details of naming of parameter types and parameter passing conventions. Generally, for those parameters that are operation-specific, the method implementing the operation appears to receive the same values that would be passed to the stubs.

9.3.2 Method signatures

With the BOA, implementation methods have signatures that are similar to the stubs. The primary difference is that there is a trailing parameter that provides request-specific information and is used to specify exception returns.

*** The syntax for method signatures in an object implementation is provisional.

If the following interface is defined in IDL:

```
interface example4 {
   long op5(in long arg6);
};
```

a method for the **op5** routine must have the following function signature:

```
long example4_op5(example4 objref, long arg6, request_t *req);
```

The `objref` parameter is the object reference that was invoked. The method can identify which object was intended by using the `get_id` BOA operation. The `req` parameter is used for authentication on the `get_principal` BOA operation, and is used for indicating exceptions.

The method terminates successfully by executing a `return` statement returning the declared operation value. Prior to returning the result of an invocation, the method code must assign legal values to all **out** and **inout** parameters.

The method terminates with an error by executing the `raise_exception` BOA operation, which has the following C language definition:

```
void     raise_exception (request_t req, long exceptid,
                          char * exceptname, void *param);
```

The `req` parameter is the last parameter passed into the method. The `exceptid` parameter is either a standard exception number, in which case the `exceptname` parameter must be NULL, or `except_user_defined`, in which case the `exceptname` parameter is a string naming the exception. In either case, if the exception is declared to take parameters, the `param` parameter must be the address of a struct containing the parameters according to the C language mapping. If the exception takes no parameters, `param` must be NULL.

9.3.3 Binding methods to skeletons

It is not specified as part of the language mapping how the skeletons are connected to the methods. Different means will be used in different environments, for example, the skeletons may make references to the methods that are resolved by the linker or there may be a system-dependent call done at program startup to specify the location of the methods.

9.3.4 BOA and ORB routines

The operations on the BOA defined earlier in this chapter and the operations on the ORB defined in Chapter 8 are used as if they had the IDL definitions described in the document, and then mapped in the usual way with the C language mapping.

For example, the string_to_objref ORB operation has the following signature:

```
objref_t ORB_string_to_objref (objref_t orb, char *objrefstring,
                               exception_t *ex);
```

The `get_id` BOA operation has the following signature:

```
objref_t BOA_create (objref_t boa, _IDL_SEQUENCE_octet_1024 *id,
                     objref_t intf, objref_t impl, exception_t *ex);
```

The equal object reference operation has the following signature:

```
long Objref_equal (objref_t objref1, objref_t objref,
                               exception_t *ex);
```

Although in each example, we are using an "object" that is special (an ORB, an object adapter, or an object reference), the method name is generated as `interface_operation` in the same way as ordinary objects. Also, the signature containes an `exception_t` parameter at the end for error indications.

In the first two cases, the signature calls for an object reference to represent the particular ORB or object adapter being manipulated. Programs may obtain these objects in a variety of ways, for example, in a global variable before program startup if there is only one ORB or BOA that makes sense, or by obtaining them from a name service if more than one is available. In the third case, the object reference being operated on is specified as the first parameter.

Following the same procedure, the C language binding for the remainder of the ORB, BOA, and object reference operations may be determined.

10 Interoperability

It is an explicit goal of the Common ORB Architecture to allow interoperation between different object systems and ORBs. The large diversity of ORB implementation techniques means that a single strategy or technology for interoperation would be infeasible. However, there is substantial experience in the industry in connecting networks with different protocols, and we look to those working examples that are in everyday use for the model of how to connect ORBS.

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 coexist, and there are ways to bridge between environments that share no protocols. These same truths will hold for ORBs as well.

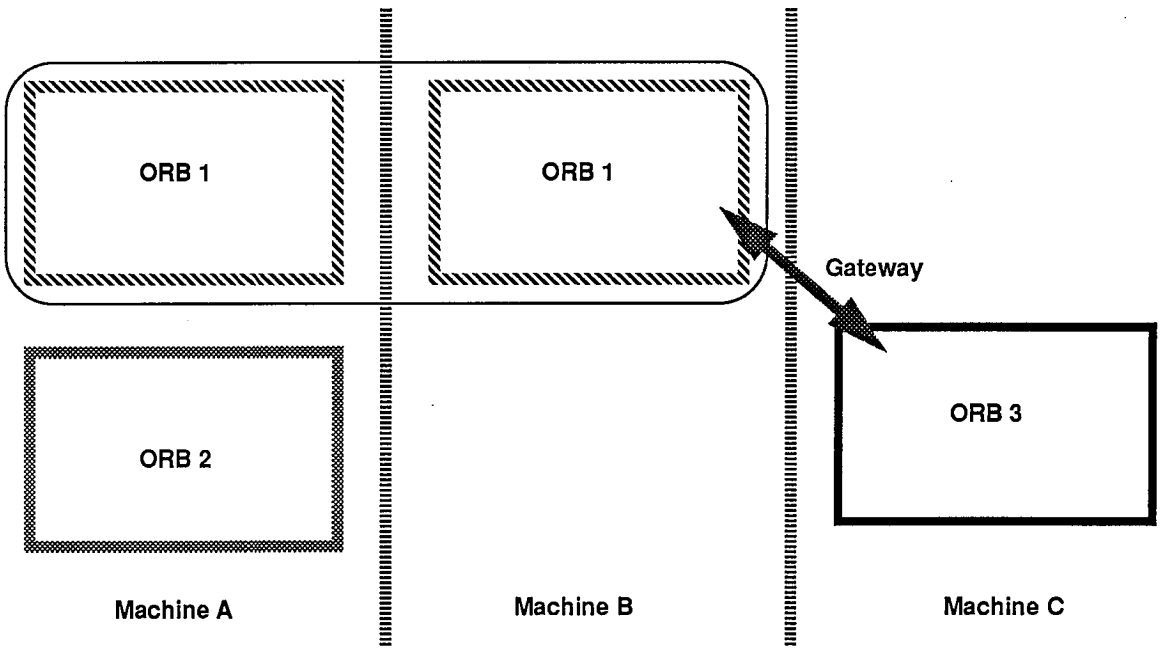
The primary requirement to allow convenient interoperation is to have a higher-level model that spans the differences. In the case of the ORB, there is an obvious higher-level model—IDL-defined object-oriented invocation. Because IDL is defined in an ORB-independent way, and because clients and object implementations can be built in an ORB-independent way, it is possible for a particular request to pass through multiple ORBs, preserving the invocation semantics transparent to clients and implementations.

10.1 The Organization of Multiple ORBs

FIG. 13 on page 136 shows three possible scenarios in which multiple ORBs coexist (although we will describe the first scenario as a single ORB).

1. ORB 1 is implemented on both Machine A and Machine B. Both implementations use the same object references and communication mechanism, and an object reference can be freely passed from Machine A to Machine B. We actually consider this case to be a single ORB implemented on two machines since no transformation is needed to move object references from one machine to the other.
2. On Machine A, the same client may have some objects implemented by ORB 1 and some by ORB 2. It is thus possible to an invoke an object reference in one ORB and pass as a parameter an object reference from another ORB. In any particular computing environment, an ORB must be able to distinguish its own object references from others', and must be able to pass other ORB's object references as parameters.
3. Between Machine B and Machine C, there are no common ORBs. In order to pass (and subsequently invoke) objects between Machine B and Machine C, it is necessary to construct a gateway to translate object references and requests in one ORB to object references and requests in the other.

FIG. 13 Multiple ORBs



There are many possible ways to connect two ORBS together, but they tend to fall into two categories: embedding of object references and protocol translation. Another technique is to allow object implementations to move objects between ORBs.

10.1.1 Reference Embedding

With reference embedding, an object in one ORB appears to be an object in a second ORB. An invocation on the object in the second ORB arrives at an implementation whose job it is to perform an invocation in the first ORB. On Machine A in FIG. 13 on page 136 an object implemented using ORB 1 might be made available in ORB 2 make creating an implementation in ORB 2 that, when invoked, simply invokes the corresponding object in ORB 1. A common use for reference embedding is when one ORB is a library ORB or other optimized implementation that cannot be accessed remotely. By embedding those objects that must be accessed remotely, most of the benefits of the optimized ORB can be had without sacrificing generality.

10.1.2 Protocol Translation

When two ORBs differ in their implementation details but have similar functionality, it will often be possible to translate requests in one ORB to be requests in the other ORB. For example, two RPC-based ORBs might differ in their object reference representation and packet formats, but otherwise present the same semantics. If it is possible to map object references in one ORB into object references in the other domain, and translate packets from one format to the other, a gateway could be constructed to pass requests back and forth.

10.1.3 Alternate ORBs

An object implementation implicitly chooses an ORB when it binds to a particular object adapter. If a machine supports multiple ORBs and the same object adapter interface is available on more than one ORB, an object implementation may choose to make the same object available through multiple ORBs. Because the object adapter interface is the same in each case, few changes would be necessary to the object implementation code to support multiple ORBs. However, the scope and performance of the different ORBs might be quite different.

In this case, an object implementation might generate object references in different ORBs. The interface to the object could define operations that would allow a client to obtain an equivalent object reference in a different ORB.

11 Glossary

activation — Preparing an object to execute an operation. For example, copying the persistent form of methods and stored data into an executable address space to allow execution of the methods on the stored data.

adapter — Same as object adapter.

attribute — An identifiable association between an object and a value. An attribute A is made visible to clients as a pair of operations: get_A and set_A. Readonly attributes only generate a get operation.

basic object adapter — The object adapter described in Chapter 9.

behavior — The observable effects of an object performing the requested operation including its results)binding. Not used. See language binding, dynamic invocation, static invocation, or method resolution for alternatives.

class — Not used. See interface and implementation for alternatives.

client The code or process that invokes an operation on an object.

context object A collection of name-value pairs that provides environmental or user-preference information. See Chapter 6.

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.

dynamic invocation Constructing and issuing a request whose signature is possibly not known until runtime.

externalized object reference An object reference expressed as an ORB-specific string. Suitable for storage in files or other external media.

handle Same as object reference.

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, 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. Also called an object interface or an object's 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 object interface information.

interface type — Same as interface or object 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 requestor 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 an object to exist 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 interface — Same as interface.

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 a operation parameter. The parameter passing modes are IN, OUT, and INOUT.

persistent object — An object that can survive the process or thread that created it. A persistent object exists until it is explicitly deleted.

referential integrity — The property ensuring that an object reference that exists in the state associated with an object reliably identifies a single object.

repository — See interface repository and implementation repository.

request — A client issues a request to cause a service to be performed. A request consists of an operation and zero or more actual parameters. Also associated with a request are the results that may be returned to the client.

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 object interface.

value — Any entity that may be a possible actual parameter in a request. Values that serve to identify objects are called object references.