The Object Model 1

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

Contents

This chapter contains the following sections.

1.1 Overview

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

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

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

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

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

There are some other characteristics of object systems that are outside the scope of the object model. Some of these concepts are aspects of application architecture, some are associated with specific domains to which object technology is applied. Such concepts are more properly dealt with in an architectural reference model. Examples of excluded concepts are compound objects, links, copying of objects, change management, and transactions. Also outside the scope of the object model are the details of control structure: the object model does not say whether clients and/or servers are single-threaded or multi-threaded, and does not specify how event loops are programmed nor how threads are created, destroyed, or synchronized.

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

1.2 Object Semantics

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

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

1.2.1 Objects

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

1.2.2 Requests

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

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

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

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

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

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

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

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

The following semantics hold for all requests:

- Any aliasing of parameter values is neither guaranteed removed nor guaranteed to be preserved.
- The order in which aliased output parameters are written is not guaranteed.

- The return result and the values stored into the output and input-output parameters are undefined if an exception is returned.

For descriptions of the values and exceptions that are permitted, see “Types” on page 1-4 and “Exceptions” on page 1-7.

1.2.3 Object Creation and Destruction

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

1.2.4 Types

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

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

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

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

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

Basic types:

- 16-bit, 32-bit, and 64-bit signed and unsigned 2’s complement integers.
- Single-precision (32-bit), double-precision (64-bit), and double-extended (a mantissa of at least 64 bits, a sign bit and an exponent of at least 15 bits) IEEE floating point numbers.
- Fixed-point decimal numbers of up to 31 significant digits.
- Characters, as defined in ISO Latin-1 (8859.1) and other single- or multi-byte character sets.
- A boolean type taking the values TRUE and FALSE.
- An 8-bit opaque detectable, guaranteed to *not* undergo any conversion during transfer between systems.
- Enumerated types consisting of ordered sequences of identifiers.
- A string type, which consists of a variable-length array of characters (a null character is one whose character code is 0); the length of the string is a positive integer, and is available at run-time.
- A container type “any,” which can represent any possible basic or constructed type.
- Wide characters that may represent characters from any wide character set.

- Wide character strings, which consist of a length, available at runtime, and a variable-length array of (fixed width) wide characters.

Constructed types:

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

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

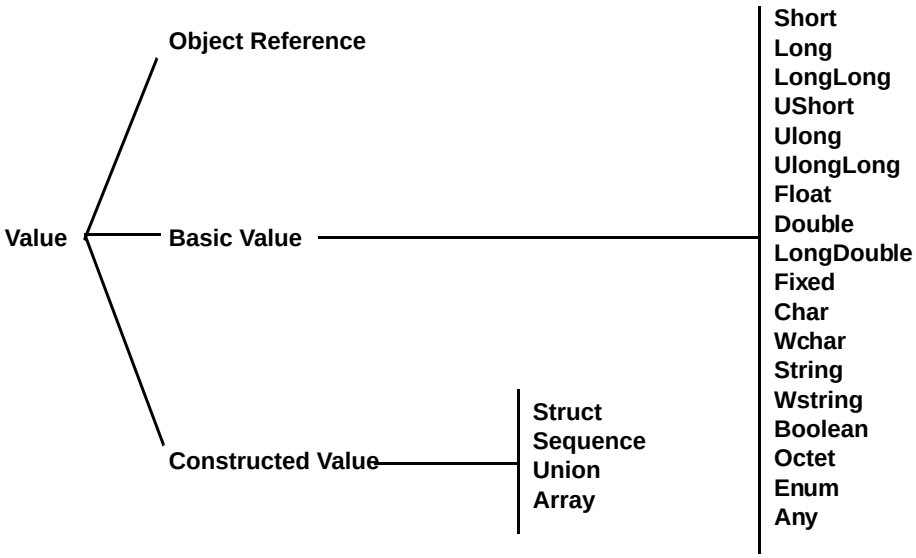

Figure 1-1 Legal Values

1.2.5 *Interfaces*

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

An *object type* is a type that is satisfied by any object reference whose referent satisfies an interface that describes the object type.

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

1.2.6 Operations

An *operation* is an identifiable entity that denotes a service that can be requested and is identified by an *operation identifier*. An operation is not a value.

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

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

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

The general form for an operation signature is:

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

where:

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

Parameters

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

Return Result

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

Exceptions

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

The additional, exception-specific information is a specialized form of record. As a record, it may consist of any of the types described in "Types" on page 1-4.

All signatures implicitly include the system exceptions; the standard system exceptions are described in "Standard Exceptions" on page 3-37.

Contexts

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

Execution Semantics

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

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

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

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

1.2.7 Attributes

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

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

1.3 Object Implementation

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

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

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

1.3.1 The Execution Model: Performing Services

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

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

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

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

1.3.2 The Construction Model

A computational object system must provide mechanisms for realizing behavior of requests. These mechanisms include definitions of object state, definitions of methods, and definitions of how the object infrastructure is to select the methods to execute and

to select the relevant portions of object state to be made accessible to the methods. Mechanisms must also be provided to describe the concrete actions associated with object creation, such as association of the new object with appropriate methods.

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