A Discussion of the Object Management Architecture

January 1997

NOTICE

The information contained in this document is subject to change without notice.

The material in this document details an Object Management Group specification in accordance with the license and notices set forth on this page. This document does not represent a commitment to implement any portion of this specification in any company's products.

WHILE THE INFORMATION IN THIS PUBLICATION IS BELIEVED TO BE ACCURATE, THE OBJECT MANAGEMENT GROUP AND THE COMPANIES LISTED ABOVE MAKE NO WARRANTY OF ANY KIND WITH REGARD TO THIS MATERIAL INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. The Object Management Group and the companies listed above shall not be liable for errors contained herein or for incidental or consequential damages in connection with the furnishing, performance or use of this material.

The copyright holders listed above acknowledge that the Object Management Group (acting itself or through its designees) is and shall at all times be the sole entity that may authorize developers, suppliers and sellers of computer software to use certification marks, trademarks or other special designations to indicate compliance with these materials.

RESTRICTED RIGHTS LEGEND. Use, duplication, or disclosure by government is subject to restrictions as set forth in subdivision (c) (1) (ii) of the Right in Technical Data and Computer Software Clause at DFARS 252.227.7013

Table of Contents

Introduction 1

1.1 The OMG Vision

The CPU as an island, contained and valuable in itself, is dying in the nineties. The next paradigm of computing is distributed or cooperative computing. This is driven by the very real demands of corporations recognizing information as an asset, perhaps their most important asset.

To make use of information effectively, it must be accurate and accessible across the department, even across the world. This means that CPUs must be intimately linked to the networks of the world and be capable of freely passing and receiving information, not hidden behind glass and cooling ducts or the complexities of the software that drives them.

1.1.1 Current Problems

The major hurdles in entering this new world are provided by software: the time to develop it, the ability to maintain and enhance it, the limits on how complex a given program can be in order to be profitably produced and sold, and the time it takes to learn to use it. This leads to the major issue facing corporate information systems today: the quality, cost, and lack of interoperability of software. While hardware costs are plummeting, software expenses are rising.

As information systems attain strategic importance and represent the key competitive edge to the industry leaders, the cost of inaccuracies or delayed implementations is attenuating entire MIS departments. As systems departments require information among a diversity of inhouse, brought-in, supplier, customer, and commercial applications, those applications become increasingly difficult and complex.

1.1.2 The OMG Solution

The Object Management Group (OMG) was formed to help reduce complexity, lower costs, and hasten the introduction of new software applications. The OMG plans to accomplish this through the introduction of an architectural framework with supporting detailed interface specifications. These specifications will drive the industry towards interoperable, reusable, portable software components based on standard object-oriented interfaces.

The OMG is an international trade association incorporated as a nonprofit in the United States. The OMG receives funding on a yearly dues basis from its diverse membership of more than 700 corporations. The OMG is headquartered in Framingham, Massachusetts, and has marketing offices in Frankfurt, Germany; Tokyo, Japan; and Hounslow, England. The OMG also sponsors the world's largest exhibition and conference on object technology, Object World. The mission of the OMG is as follows:

- The Object Management Group is dedicated to maximizing the portability, reusability, and interoperability of software. The OMG is the leading worldwide organization dedicated to producing a framework and specifications for commercially available object-oriented environments.

- The Object Management Group provides a Reference Architecture with terms and definitions upon which all specifications are based. Implementations of these specifications will be made available under fair and equitable terms and conditions. The OMG will create industry standards for commercially available object-oriented systems by focusing on Distributed Applications, Distributed Services, and Common Facilities.

- The OMG provides an open forum for industry discussion, education, and promotion of OMG-endorsed object technology. The OMG coordinates its activities with related organizations and acts as a technology/marketing center for object-oriented software.

The OMG defines the object management paradigm as the ability to encapsulate data and methods for software development. This models the "real world" through representation of program components called "objects." This representation results in faster application development, easier maintenance, reduced program complexity, and reusable components. A central benefit of an object-oriented system is its ability to grow in functionality through the extension of existing components and the addition of new objects to the system.

The software concept of "objects," as incorporated into the technology of the Object Management Group, will provide solutions to the software complexities of the 1990s. Object-oriented architectures will allow applications acquired from different sources and installed on different systems to freely exchange information. Software "objects" will mirror the real world business objects they support, in the sense that the architect's blueprint mirror a building. The OMG envisions a day where users of software start

up applications as they start up their cars, with no more concern about the underlying structure of the objects they manipulate than the driver has about the molecular construction of gasoline.

1.1.3 Goals of the OMG

The members of the Object Management Group have a shared goal of developing and using integrated software systems. These systems should be built using a methodology that supports modular production of software; encourages reuse of code; allows useful integration across lines of developers, operating systems and hardware; and enhances long-range maintenance of that code. Members of the OMG believe that the object-oriented approach to software construction best supports their goals.

Object orientation, at both the programming language and applications environment levels, provides a terrific boost in programmer productivity, and greatly lends itself to the production of integrated software systems. While not necessarily promoting faster programming, object technology allows you to construct more with less code. This is partly due to the naturalness of the approach, and also to its rigorous requirement for interface specification. The only thing missing is a set of standard interfaces for interoperable software components. This is the mission of the OMG.

Member companies join the OMG because they see themselves in a position to capitalize on a decade's work in object-oriented development by constructing a real system based on a vision of a distributed object-oriented architecture for application development. A major goal is to define a living, evolving standard with realized parts, so that applications developers can deliver their applications with off-the-shelf components for common facilities like object storage, class structure, peripheral interface, user interface, etc. The function of the OMG is then to promulgate the standard specifications throughout the international industry, and to foster the development of tools and software components compliant with the standard.

1.1.4 The OMG Process

The OMG Board of Directors approves the standard by explicit vote on a technology-by-technology basis. The OMG Board of Directors bases its decisions on both business and technical merit. As portions of the reference model are proposed to be filled by various vendors' software specifications, the standard grows. The purpose of the OMG Technical Committee (TC) is to provide technical guidance and recommendations to the Board in making these technology decisions. An end-user special interest group likewise guides the Board toward decisions in the best interests of technology users.

The TC is composed of representatives of all OMG member companies (Corporate, Associate, and End User), with similar voting provisions to the Board's voting structure. It is operated by a Vice President of Technology, working full-time for the OMG (as opposed to being employed by a member company). The TC operates in a Request for Proposal (RFP) mode, requesting technology to fill open portions of the reference model from the international industry. (This document lays the groundwork for technology response to our Requests for Proposals and subsequent adoption of

specifications.) The responses to an RFP, taken within a specific response period, are evaluated by a Task Force of the Technical Committee. Then, the full TC votes on a recommendation to the Board for approval of the proposed addition to the standard. Once a technology specification (not source code or product) has been adopted, it is promulgated by the OMG to the industry through a variety of distribution channels. There also exists a somewhat faster model for adopting standards, one that is based on Requests for Public Comment (RFC).

1.2 Benefits of Object Management

As previously mentioned, the technological approach of object technology (or object orientation) was chosen by the OMG founders not for its own sake, but in order to attain a set of end user goals. End users benefit in a number of ways from the object-oriented approach to application construction:

- An object-oriented user interface has many advantages over more traditional user interfaces. In an object-oriented user interface, Application Objects (computer simulated representations of real world objects) are presented to end users as objects that can be manipulated in a way that is similar to the manipulation of the real world objects. Examples of such object-oriented user interfaces are realized in systems such as Xerox Star, Apple Macintosh, NeXTStep from NeXT Computer, OSF Motif and HP NewWave, and to a limited degree, Microsoft Windows. CAD systems are also a good example in which components of a design can be manipulated in a way similar to the manipulation of real components. This results in a reduced learning curve and common "look and feel" to multiple applications. It is easier to see and point than to remember and type.

- A more indirect end-user benefit of object-oriented applications, provided that they cooperate according to some standard, is that independently developed general purpose applications can be combined in a user-specific way. It is the OMG's central purpose to create a standard that realizes interoperability between independently developed applications across heterogeneous networks of computers. This means that multiple software programs appear as "one" to the user of information no matter where they reside.

- Common functionality in different applications (such as storage and retrieval of objects, mailing of objects, printing of objects, creation and deletion of objects, or help and computer-based training) is realized by common shared objects leading to a uniform and consistent user interface.

- Sharing of information drastically reduces documentation redundancy. Consistent access across multiple applications allows for increased focus on application creation rather than application education.

- Transition to object-oriented application technology does not make existing applications obsolete. Existing applications can be embedded (with different levels of integration) in an object-oriented environment.

- Pragmatic migration of existing applications gives users control over their computing resources, and how quickly these resources change.

- Likewise, application developers benefit from object technology and object-oriented standards. These benefits fall into two categories:

  - Through encapsulation of object data (making data accessible only in a way controlled by the software that implements the object) applications are built in a truly modular fashion, preventing unintended interference. In addition, it is possible to build applications in an incremental way, preserving correctness during the development process.

  - Reuse of existing components. Specifically, when the OMG standard is in effect, thereby standardizing interaction between independently developed applications (and application components), cost and lead time can be saved by making use of existing implementations of object classes.

- In developing standards, the OMG keeps these benefits of object orientation in mind, together with a set of overall goals:

  - Heterogeneity. Integration of applications and facilities must be available across heterogeneous networks of systems independent of networking transports and operating systems.

  - Customization options. Common Facilities must be customizable in order to meet specific end-user or organizational requirements and preferences.

  - Scope. The scope of OMG adopted technology is characterized by both work group support and mission critical applications.

  - Management and control. Issues such as security, recovery, interruptibility, auditing, and performance are examined.

  - Internationalization. As the OMG is itself an international group, the standard reflects built-in support for internationalization of software.

  - Technical standards. Standards to meet these user goals are the central goal of the OMG, as well as the content of this manual.

Overview 2

2.1 Introduction

This chapter uses typical problems and their OMG solutions to describe what the OMG's Object Management Architecture tries to achieve. The examples in this chapter should put the rest of the guide in perspective and help the user to appreciate the proposed technology, as outlined in the following chapters.

This chapter makes several assumptions: first, that the OMG's Object Management Architecture is mature; second, that conforming platforms and applications are abundant; and third, that the examples, taken from the mechanical CAD world, apply to other application areas as well.

2.2 The Advantages of OMA for System Vendors

Situation

A supplier of CAD workstations has, on the basis of hardware and a CAD-M application, a good position in the CAD-M market.

Problem 1: A large customer requires that the manuals related to his design be created on another supplier's publishing workstations, directly accessing the design information.

Solution: With the OMG's Object Management Architecture in place, the CAD-M design objects are stored on a database server. These objects are accessed by CAD application functions (methods) for drawing, parts explosion, and others. The new documentation requirements can be implemented by connecting the publishing equipment to the CAD network. A multimedia editor runs on the publishing workstations and addresses the CAD-M design objects to generate the drawings to be included in the documentation. Some editing is of course required: an illustration in a

manual usually takes a different format than a design drawing. The design objects may require a method to generate a representation in a format compatible with the publishing software, since there are many standards in the CAD world.

Equipment as well as software from different sources can interoperate. Substantial functional extensions can be made by applying a minimum of glue between nonrelated but existing applications. Software is reusable and extensible.

Problem 2: A major competitor expands its product with support for circuit board design. To be competitive, the system vendor must port software supplied by a leading OEM.

Solution: The solution of the second problem is even simpler: if the system vendor's hardware and systems software is indeed a platform supported by the Object Management Architecture and the circuit board design application conforms to the Object Management Architecture, the porting of the circuit board design application is straightforward and the system vendor as well as the OEM software supplier can enjoy a productive OEM contract. Applications written in conformance with the Object Management Architecture are portable. Translating these technical opportunities in business terms, it means that the system vendor can profit from the OMG's Object Management Architecture due to:

- Reduced initial cost to make new functionality available (alternatively, more functionality offered to the customers at the same cost) since existing software (from the System Vendor or from OEM sources) can easily be integrated.
- Early availability of a larger number of functions, leading to an extension of the market size.

2.3 The Advantages of OMA for Independent Software Vendors

Situation

An independent software vendor (ISV) has a position in stress analysis of mechanical parts.

Problem: The ISV wants to extend the functionality with software for the simulation of dynamic behavior of mechanical designs. The target is a set of existing CAD-M platforms.

Solution: Object Management Architecture is not magic: complex applications (like simulation of dynamic behavior of mechanical designs) require hard work. However, OMG's technology allows the ISV in this example to concentrate on the essentials: the complexity of simulation.

The Object Management Architecture comes with standardized Object Services (for application controlled management of objects) and a set of Common Facilities, in this case, helping to visualize the results of the simulation.

When used in a CAD-M environment that provides methods for parts explosion and retrieval of design data, OMG technology results in extended end-user functionality while conserving the existing design environment.

Since the simulation of dynamic behavior may involve heavy number crunching, calculations can be delegated to a dedicated server, providing the necessary megaflops, and leaving the CAD workstation resources free for interactive design work.

The Object Management Architecture improves productivity by reusing existing parts (and thus focusing on essentials). Object-oriented software design using the standard components of the Object Management Architecture imposes a design philosophy that leads to less iteration during the design phase. Object Management Architecture provides transparency over heterogeneous networks, allowing specific tasks to be delegated to specific machines.

2.4 The Advantages of OMA for End Users

Situation

A central CAD department of a pump factory supplying a large variety of pumping equipment.

Problem: The management of the central CAD department of the pump factory gets the following tasks:

- Fine-tune the design of a series of high-precision pumps. A CAD-M system is available but fine-tuning requires stress analysis and simulation of dynamic behavior of the moving parts in the pump's design.

- Refine reporting of design costs to reflect the cost per product line. The accounting department is equipped with PC workstations connected to the factory's LAN. Reporting from accounting is based on regular word processing and spreadsheet applications.

- Propose, in cooperation with the documentation department, an improved procedure for the production of manufacturing and service documentation. The documentation department uses desktop publishing equipment, connected to the factory's LAN. The supplier of the CAD equipment has a stress analysis package in its catalogue and is able to install that package on the current equipment. (Refer to 1.) The same supplier has recently announced the availability of an extension allowing simulation of dynamic behavior of mechanical designs. A contact with the ISV supplying that software indicates that the necessary vibration analysis can indeed be done but will require more number-crunching capacity than is currently available.

Solution: The pump factory's computer consultant indicates how the classes of the CAD-M package can be extended with an automatic time-stamp facility. Each week, the collection of time stamps is put into a spreadsheet format and is sent to Accounting. The implementation work needed is assigned to a software house specializing in Object Management Architecture–conforming software.

Another extension to the CAD-M classes is an option to extract drawings from the CAD-M object database in a format required by the desktop publishing software. The documentation department gets (read-only) access to released design objects in the database of the CAD department.

For the end user, standardized interfaces for object-oriented application software provide the option of extensibility of existing software. In addition, through encapsulation of non-conforming applications (like the spreadsheet package of Accounting), the transition to the new technology can be a gradual one. The productivity of an organization can be improved through exploitation of the interoperability of conforming software over heterogeneous networks. In addition, the object-oriented approach to application software guarantees an intuitive user interface: computer-simulated objects correspond with real world objects, whether they are the blades of a pump, a pump's service manuals, or weekly reports on the time spent on a specific design. Standardization of object-oriented technology creates uniformity and consistency over different and independently developed applications.

2.5 Object Management Architecture

The Object Request Broker component of the Object Management Architecture is the communications heart of the standard. This is referred to commercially as CORBA (Common Object Request Broker Architecture). It provides an infrastructure allowing objects to communicate, independent of the specific platforms and techniques used to implement the addressed objects. The Object Request Broker component will guarantee portability and interoperability of objects over a network of heterogeneous system. Specifications for the Common Object Request Broker are contained in *CORBA: Common Object Request Broker Architecture and Specification.*

- The Object Services component standardizes the life cycle management of objects. Functions are provided to create objects (the Object Factory), to control access to objects, to keep track of relocated objects and to consistently maintain the relationship between groups of objects. The Object Service components provide the generic environment in which single objects can perform their tasks. Standardization of Object Services leads to consistency over different applications and improved productivity for the developer. Specifications for the Object Services that have been adopted as standards by the OMG are contained in *CORBAservices: Common Object Services Specifications.*

- The Common Facilities component provides a set of generic application functions that can be configured to the requirements of a specific configuration. Examples are printing facilities, database facilities, and electronic mail facilities.

Standardization leads to uniformity in generic operations and to options for end users to configure their configurations (as opposed to configuring individual applications).

- The Application Objects part of the architecture represents those application objects performing specific tasks for users. One application is typically built from a large number of basic object classes, partly specific for the application, partly from the set of Common Facilities. New classes of application objects can be built by modification of existing classes through generalization or specialization of existing classes (inheritance) as provided by Object Services. The multi-object class approach to application development leads to improved productivity for the developer and to options for end users to combine and configure their applications.

- Domain Interfaces are domain-specific interfaces for application domains such as Finance, Healthcare, Manufacturing, Telecom, Electronic Commerce, and Transportation.

The Object Model 3

3.1 Overview

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

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

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

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

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

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

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

3.2 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.

3.2.1 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.

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

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

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

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

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

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

The following semantics hold for all requests:

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

Descriptions of the values and exceptions that are permitted, see 3 and 7.

3.2.3 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.

3.2.4 Types

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

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

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

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

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

Basic types:

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

Constructed types:

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

Values in a request are restricted to values that satisfy these type constraints. The legal values are shown in on page 3-5. No particular representation for values is defined.

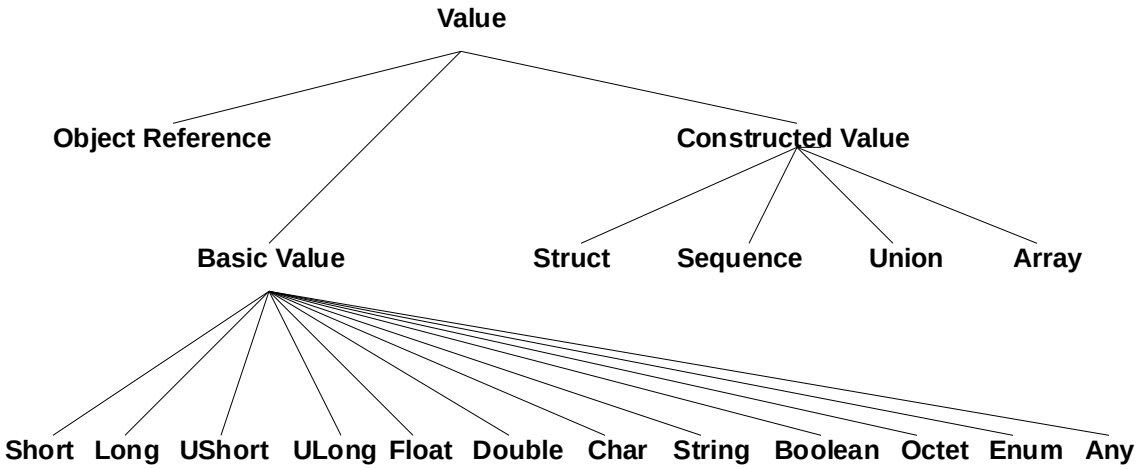

Figure 3-1 Legal Values

3.2.5 Interfaces

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

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

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

3.2.6 Operations

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

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

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

- A specification of the parameters required in requests for that operation
- A specification of the result of the operation
- A specification of the exceptions that may be raised by a request for the operation and the types of the parameters accompanying them

- A specification of additional contextual information that may affect the request
- An indication of the execution semantics the client should expect from a request for the operation

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

The general form for an operation signature is:

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

where:

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

Parameters

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

Return Result

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

Exceptions

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

The additional, exception-specific information is a specialized form of record. As a record, it may consist of any of the types described in Section 3.2.4.

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.

3.2.7 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.

3.3 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.

3.3.1 The Execution Model: Performing Services

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

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

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

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

3.3.2 The Construction Model

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

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

Reference Model 4

4.1 Object Management Architecture

4.1.1 Introduction

The *Object Management Architecture Guide* (OMAG) describes OMG's technical objectives and terminology and provides the conceptual infrastructure upon which supporting specifications are based. The guide includes the *OMG Object Model,* which defines common semantics for specifying the externally visible characteristics of objects in a standard implementation-independent way, and the *OMA Reference Model.*

Through a series of RFPs, OMG is populating the OMA with detailed specifications for each component and interface category in the Reference Model. Adopted specifications include the Common Object Request Broker Architecture (CORBA), CORBAservices, and CORBAfacilities.

The wide-scale industry adoption of OMG's OMA provides application developers and users with the means to build interoperable software systems distributed across all major hardware, operating system, and programming language environments.

4.1.2 Reference Model Overview

The Reference Model identifies and characterizes the components, interfaces, and protocols that compose the OMA. This includes the *Object Request Broker* (ORB) component that enables clients and objects to communicate in a distributed environment, and four categories of object interfaces:

- *Object Services* are interfaces for general services that are likely to be used in any program based on distributed objects
- *Common Facilities* are interfaces for horizontal end-user-oriented facilities applicable to most application domains

- *Domain Interfaces* are application domain-specific interfaces
- *Application Interfaces* are non-standardized application-specific interfaces

These interface categories are shown in Figure 4-1.

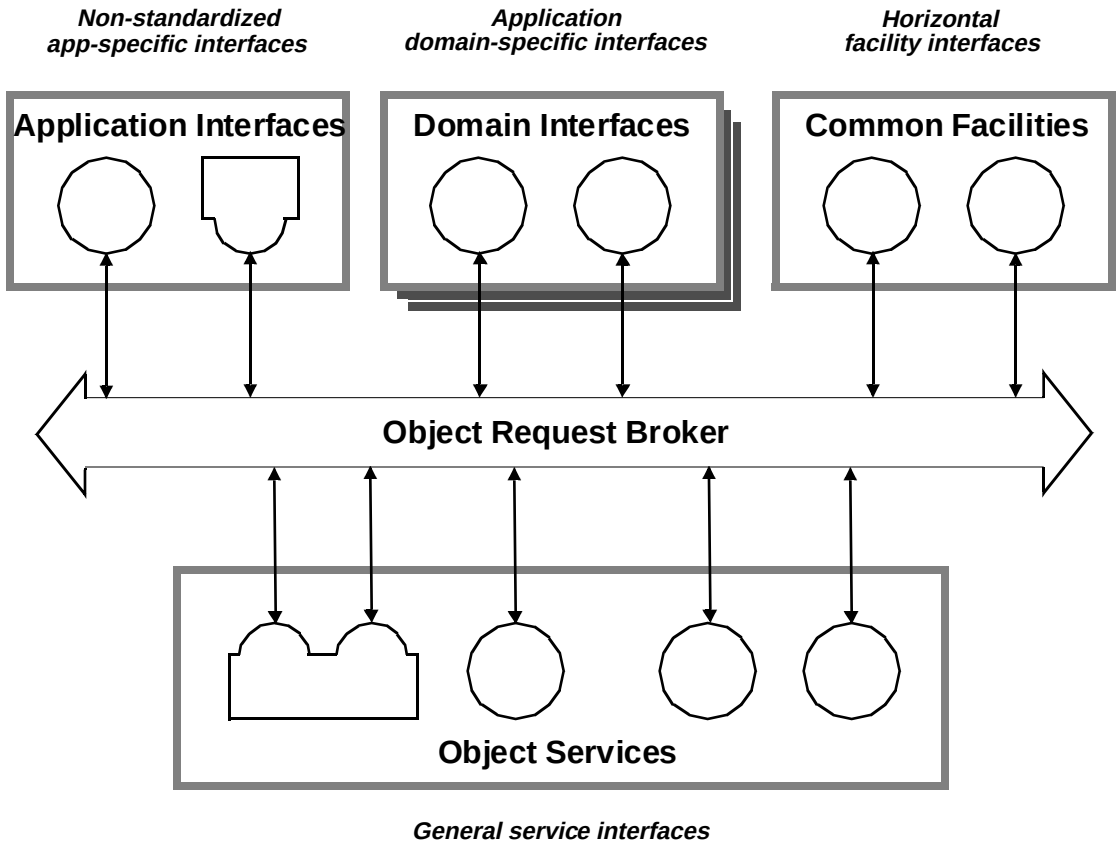

Figure 4-1 OMA Reference Model: Interface Categories

A second part of the Reference Model, shown in Figure 4-2, focuses on interface *usage* and introduces the notion of domain-specific *Object Frameworks*. An Object Framework component is a collection of cooperating objects that provide an integrated solution within an application or technology domain and which is intended for customization by the developer or user. Object Frameworks are explained in more detail below.

4.1.3 Interface versus Implementation

It is important to note that applications need only support or use OMG-compliant interfaces to participate in the OMA. They need not themselves be constructed using the object-oriented paradigm. Figure 4-1 shows, in the case of Object Services, how existing non-object-oriented software can be embedded in objects (sometimes called object wrappers) that participate in the OMA.

4.1.4 Object Request Broker

The *Common Object Request Broker Architecture* defines the programming interfaces to the OMA ORB component. An ORB is the basic mechanism by which objects transparently make requests to - and receive responses from - each other on the same machine or across a network. A client need not be aware of the mechanisms used to communicate with or activate an object, how the object is implemented, nor where the object is located. The ORB thus forms the foundation for building applications constructed from distributed objects and for interoperability between applications in both homogeneous and heterogeneous environments.

The *OMG Interface Definition Language* (IDL) provides a standardized way to define the interfaces to CORBA objects. The IDL definition is the contract between the implementor of an object and the client. IDL is a strongly typed declarative language that is programming language-independent. Language mappings enable objects to be implemented and sent requests in the developer's programming language of choice in a style that is natural to that language.

4.1.5 Object Services

Object Services are general purpose services that are either fundamental for developing useful CORBA-based applications composed of distributed objects, or that provide a universal - application domain-independent - basis for application interoperability.

Object Services are the basic building blocks for distributed object applications. Compliant objects can be combined in many different ways and put to many different uses in applications. They can be used to construct higher level facilities and object frameworks that can interoperate across multiple platform environments.

Adopted OMG Object Services are collectively called CORBAservices and include Naming, Events, LifeCycle, Persistent Object, Transactions, Concurrency Control, Relationships, Externalization, Licensing, Query, Properties, Security, Time, Collections, and Trader. See “4.2 Summary of Object Services” for additional information.

4.1.6 Common Facilities

Common Facilities are interfaces for horizontal end-user-oriented facilities applicable to most application domains. Adopted OMG Common Facilities are collectively called CORBAfacilities and include an OpenDoc-based Distributed Document Facility.

A specification of a Common Facility or Object Service typically includes the set of interface definitions - expressed in OMG IDL - that objects in various roles must support in order to *provide*, *use* or *participate in* the facility or service. As with all specifications adopted by OMG, facilities and services are defined in terms of interfaces and their semantics, and not a particular implementation.

4.1.7 Domain Interfaces

Domain Interfaces are domain-specific interfaces for application domains such as Finance, Healthcare, Manufacturing, Telecom, Electronic Commerce, and Transportation. Figure 4-1, highlights the fact that Domain Interfaces will be grouped by application domain by showing a possible set of collections of Domain Interfaces.

4.1.8 Object Frameworks

Unlike the interfaces to individual parts of the OMA “plumbing” infrastructure, Object Frameworks are complete higher level components that provide functionality of direct interest to end-users in particular application or technology domains.

Object Frameworks are collections of cooperating objects categorized into *Application, Domain, Facility,* and *Service Objects*. Each object in a framework supports (for example, by virtue of interface inheritance) or makes use of (via client requests) some combination of Application, Domain, Common Facility, and Object Services *interfaces*.

A particular Object Framework may contain zero or more Application Objects, zero or more Domain Objects, zero or more Facility Objects, and zero or more Service Objects. Service Objects support Object Services (OS) interfaces; Facility Objects support interfaces that are some combination of Common Facilities (CF) interfaces and potentially inherited OS interfaces; Domain Objects support interfaces that are some combination of Domain Interfaces (DI) and, potentially, inherited CF and OS interfaces; and so on for Application Objects. Thus, higher level components and interfaces build on and reuse lower level components and interfaces.

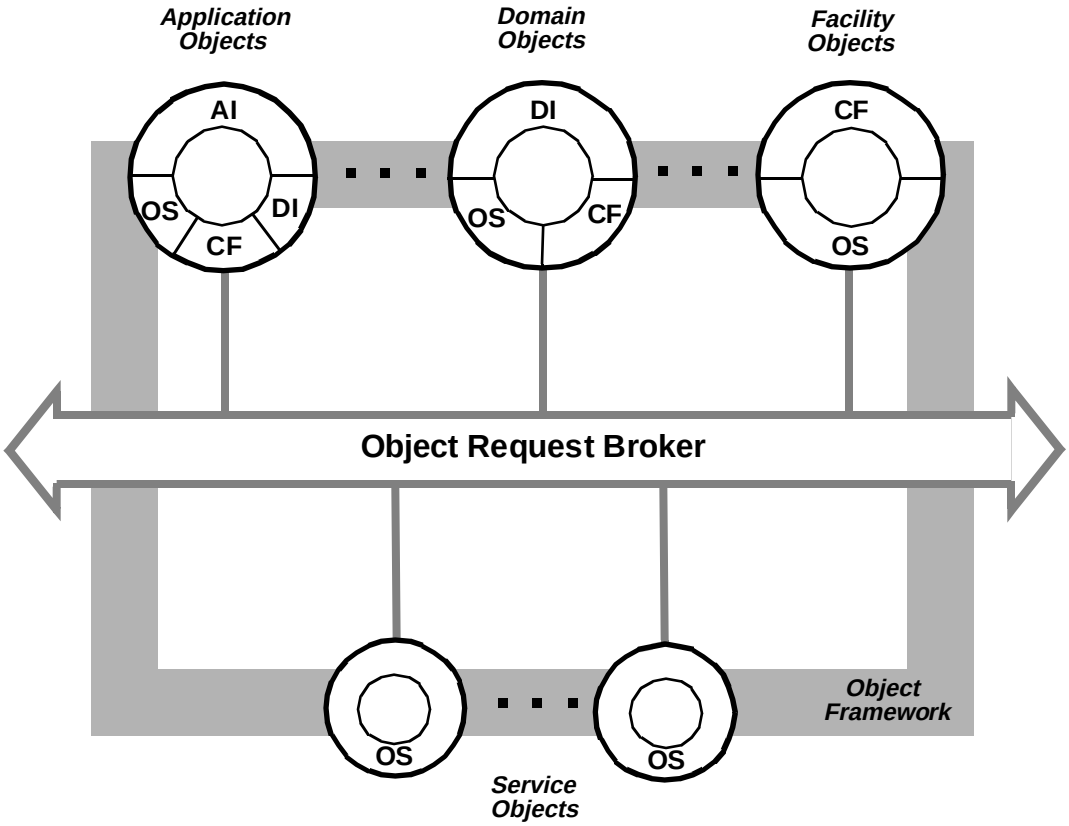

Figure 4-2 OMA Reference Model: Interface Usage

The concept of an Object Framework is illustrated in Figure 4-2. Objects are shown as an implementation “core” surrounded by a partitioned concentric shell (or “donut”) representing the interfaces that the object supports.

The picture shows the most general case where objects support all the possible interfaces for their category. In any given specific situation, degenerative cases may exist, such as Domain Objects that support only inherited Object Services interfaces (e.g. the event channel pull consumer interface) and no Common Facility interfaces, or Domain Objects that support neither Object Services or Common Facility interfaces in order to provide their functionality.

Figure 4-3 shows how objects in an Object Framework can make requests to other objects in the framework in order to provide the overall functionality of the framework. The picture shows three requests: one from an Application Object to a Service Object;

one from a Facility Object to a Service Object; and one from a Domain Object to an Application Object which could, for example, be a “call back” to a Domain Interface supported by the Application Object.

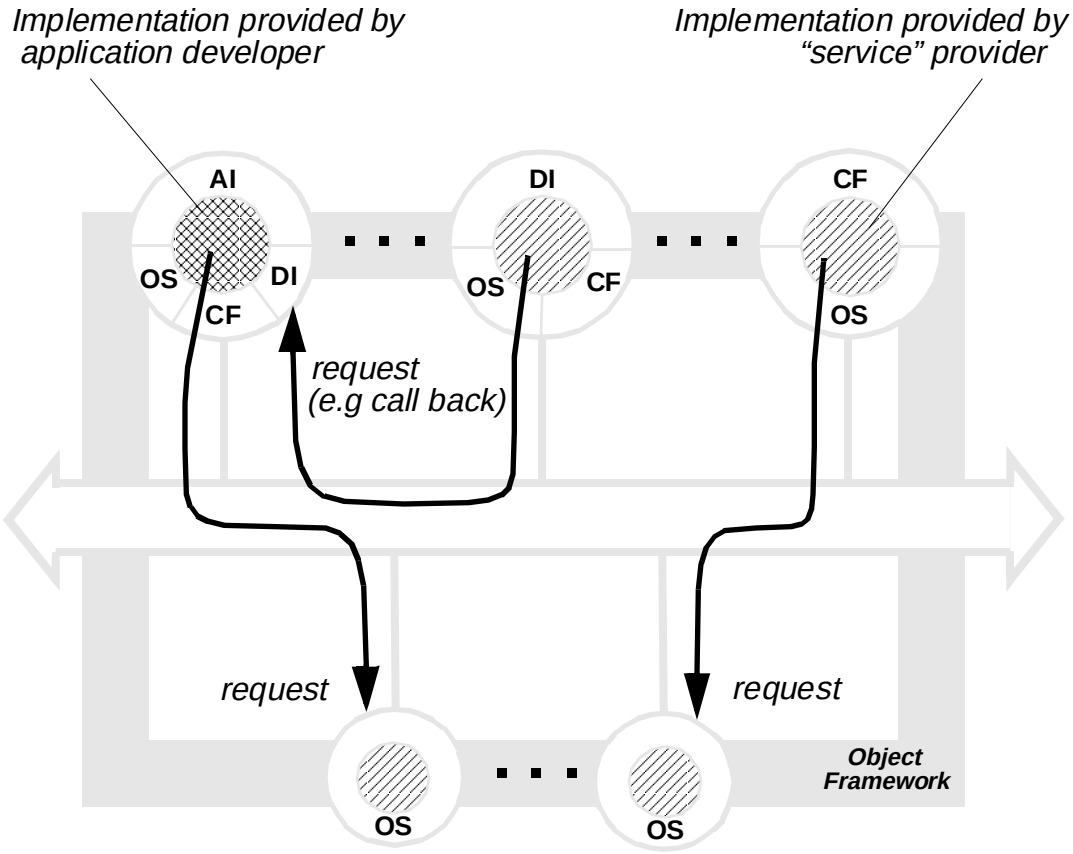

Figure 4-3 Example request flow (runtime reuse)

4.1.9 Object Framework Specifications

A specification of an Object Framework defines such things as the structure, interfaces, types, operation sequencing, and qualities of service of the objects that make up the framework. This includes requirements on implementations in order to guarantee application portability and interoperability across different platforms. Object Framework specifications may include new Domain Interfaces for particular application domains.

The application-specific part of an Application Object's interface is, by definition, not included in the specification of an Object Framework. This part is totally defined by the application developer. On the other hand, standardized interfaces that must be inherited and supported in order that the Application Object can function in the framework may form part of the framework specification.

4.2 Summary of Object Services

This section provides a brief description of each Object Service.

- The Naming Service provides the ability to bind a name to an object relative to a naming context. A naming context is an object that contains a set of name bindings in which each name is unique. To resolve a name is to determine the object associated with the name in a given context. Through the use of a very general model and in dealing with names in their structural form, Naming Service implementations can be application specific or be based on a variety of naming systems currently available on system platforms.

  Graphs of naming contexts can be supported in a distributed, federated fashion. The scalable design allows the distributed, heterogeneous implementation and administration of names and name contexts.

  Because name component attribute values are not assigned or interpreted by the Naming Service, higher levels of software are not constrained in terms of policies about the use and management of attribute values.

- The Event Service provides basic capabilities that can be configured together flexibly and powerfully. The service supports asynchronous events (decoupled event suppliers and consumers), event "fan-in," notification "fan-out,"—and through appropriate event channel implementations—reliable event delivery.

  The Event Service design is scalable and is suitable for distributed environments. There is no requirement for a centralized server or dependency on any global service. Both push and pull event delivery models are supported; that is, consumers can either request events or be notified of events.

  Suppliers can generate events without knowing the identities of the consumers. Conversely, consumers can receive events without knowing the identities of the suppliers. There can be multiple consumers and multiple suppliers of events. Because event suppliers, consumers, and channels are objects, advantage can be taken of performance optimizations provided by ORB implementations for local and remote objects. No extension is required to CORBA.

- The Life Cycle Service defines operations to copy, move, and remove graphs of related objects, while the Relationship Service allows graphs of related objects to be traversed without activating the related objects. Distributed implementations of the Relationship Service can have navigation performance and availability similar

to CORBA object references: role objects can be located with their objects and need not depend on a centralized repository of relationship information. As such, navigating a relationship can be a local operation.

- The Persistent Object Service (POS) provides a set of common interfaces to the mechanisms used for retaining and managing the persistent state of objects. The object ultimately has the responsibility of managing its state, but can use or delegate to the Persistent Object Service for the actual work. A major feature of the Persistent Object Service (and the OMG architecture) is its openness. In this case, that means that there can be a variety of different clients and implementations of the Persistent Object Service, and they can work together. This is particularly important for storage, where mechanisms useful for documents may not be appropriate for employee databases, or the mechanisms appropriate for mobile computers do not apply to mainframes.

- The Transaction Service supports multiple transaction models, including the flat (mandatory in the specification) and nested (optional) models. The Transaction Service supports interoperability between different programming models. For instance, some users want to add object implementations to existing procedural applications and to augment object implementations with code that uses the procedural paradigm. To do so in a transaction environment requires the object and procedural code to share a single transaction. Network interoperability is also supported, since users need communication between different systems, including the ability to have one transaction service interoperate with a cooperating transaction service using different ORBs.

  The Transaction Service supports both implicit (system-managed transaction) propagation and explicit (application-managed) propagation. With implicit propagation, transactional behavior is not specified in the operation's signature. With explicit propagation, applications define their own mechanisms for sharing a common transaction.

  The Transaction Service can be implemented in a TP monitor environment, so it supports the ability to execute multiple transactions concurrently, and to execute clients, servers, and transaction services in separate processes.

- The Concurrency Control Service enables multiple clients to coordinate their access to shared resources. Coordinating access to a resource means that when multiple, concurrent clients access a single resource, any conflicting actions by the clients are reconciled so that the resource remains in a consistent state.

  Concurrent use of a resource is regulated with locks. Each lock is associated with a single resource and a single client. Coordination is achieved by preventing multiple clients from simultaneously possessing locks for the same resource if the client's activities might conflict. Hence, a client must obtain an appropriate lock before accessing a shared resource. The Concurrency Control Service defines several lock modes, which correspond to different categories of access. This variety of lock modes provides flexible conflict resolution. For example,

providing different modes for reading and writing lets a resource support multiple concurrent clients on a read-only transaction. The Concurrency Control service also defines Intention Locks that support locking at multiple levels of granularity.

- The Relationship Service allows entities and relationships to be explicitly represented. Entities are represented as CORBA objects. The service defines two new kinds of objects: *relationships* and *roles*. A role represents a CORBA object in a relationship. The Relationship interface can be extended to add relationship-specific attributes and operations. In addition, relationships of arbitrary degree can be defined. Similarly, the *Role* interface can be extended to add role-specific attributes and operations. Type and cardinality constraints can be expressed and checked: exceptions are raised when the constraints are violated.

- The Externalization Service defines protocols and conventions for externalizing and internalizing objects. Externalizing an object is to record the object state in a stream of data (in memory, on a disk file, across the network, and so forth) and then be internalized into a new object in the same or a different process. The externalized object can exist for arbitrary amounts of time, be transported by means outside of the ORB, and be internalized in a different, disconnected ORB. For portability, clients can request that externalized data be stored in a file whose format is defined with the Externalization Service Specification.

  The Externalization Service is related to the Relationship Service and parallels the Life Cycle Service in defining externalization protocols for simple objects, for arbitrarily related objects, and for facilities, directory services, and file services.

- The Licensing Service provides a mechanism for producers to control the use of their intellectual property. Producers can implement the Licensing Service according to their own needs, and the needs of their customers, because the Licensing Service does not impose its own business policies or practices.

  A license in the Licensing Service has three types of attributes that allow producers to apply controls flexibly: time, value mapping, and consumer. Time allows licenses to have start/duration and expiration dates. Value mapping allows producers to implement a licensing scheme according to units, allocation (through concurrent use licensing), or consumption (for example, metering or allowance of grace periods through “overflow” licenses). Consumer attributes allow a license to be reserved or assigned for specific entities; for example, a license could be assigned to a particular machine. The Licensing Service allows producers to combine and derive from license attributes.

  The Licensing Service consists of a *LicenseServiceManager* interface and a *ProducerSpecificLicenseService* interface: these interfaces do not impose business policies upon implementors.

- The Query Service allows users and objects to invoke queries on collections of other objects. The queries are declarative statements with predicates and include the ability to specify values of attributes; to invoke arbitrary operations; and to invoke other Object Services.

  The Query Service allows indexing; maps well to the query mechanisms used in database systems and other systems that store and access large collections of objects; and is based on existing standards for query. The Query Service provides an architecture for a nested and federated service that can coordinate multiple, nested query evaluators.

- The Property Service provides the ability to dynamically associate named values with objects outside the static IDL-type system. It defines operations to create and manipulate sets of name-value or name-value-mode tuples. The names are simple OMG IDL strings. The values are OMG IDL *anys*. The use of type *any* is significant in that it allows a property service implementation to deal with any value that can be represented in the OMG-IDL-type system.

  The Property Service was designed to be a basic building block, yet robust enough to be applicable for a broad set of applications. It provides “batch” operations to deal with sets of properties as a whole. The use of “batch” operations is significant in that the systems and network management (SNMP, CMIP,...) communities have proven such a need when dealing with “attribute” manipulation in a distributed environment.

- The Security Service comprises:

  - **Identification** and **authentication** of principals (human users and objects which need to operate under their own rights) to verify they are who they claim to be.

  - **Authorization** and **access control** - deciding whether a principal can access an object, normally using the identity and/or other privilege attributes of the principal (such as role, groups, security clearance) and the control attributes of the target object (stating which principals, or principals with which attributes) can access it.

  - **Security auditing** to make users accountable for their security related actions. It is normally the human user who should be accountable. Auditing mechanisms should be able to identify the user correctly, even after a chain of calls through many objects.

  - **Security of communication** between objects, which is often over insecure lower layer communications. This requires trust to be established between the client and target, which may require **authentication of clients to targets** and **authentication of targets to clients**. It also requires **integrity protection** and (optionally) **confidentiality protection** of messages in transit between objects.

- **Non-repudiation** provides irrefutable evidence of actions such as proof of origin of data to the recipient, or proof of receipt of data to the sender to protect against subsequent attempts to falsely deny the receiving or sending of the data.

- **Administration** of security information (for example, security policy) is also needed.

- The Time Service enables the user to obtain current time together with an error estimate associated with it. It ascertains the order in which "events" occurred and computes the interval between two events.

  Time Service consists of two services, hence defines two service interfaces:

  - Time Service manages Universal Time Objects (UTOs) and Time Interval Objects (TIOs), and is represented by the *TimeService* interface.

  - Timer Event Service manages Timer Event Handler objects, and is represented by the *TimerEventService* interface.

- The Collections Service provides a uniform way to create and manipulate the most common collections generically. Collections are groups of objects which, as a group, support some operations and exhibit specific behaviors that are related to the nature of the collection rather than to the type of object they contain. Examples of collections are sets, queues, stacks, lists, and binary trees.

  For example, sets might support the following operations: insert new element, membership test, union, intersection, cardinality, equality test, emptiness test, etc. One of the defining semantics of a set is that, if an object O is a member of a set S, then inserting O into S results in the set being unchanged. This property would not hold for another collection type called a bag.

- The Trader Service provides a matchmaking service for objects. The service provider registers the availability of the service by invoking an export operation on the trader, passing as parameters information about the offered service. The export operation carries an object reference that can be used by a client to invoke operations on the advertised services, a description of the type of the offered service (i.e., the names of the operations to which it will respond, along with their parameter and result types), information on the distinguishing attributes of the offered service.

  The offer space managed by traders may be partitioned to ease administration and navigation. This information is stored persistently by the Trader. Whenever a potential client wishes to obtain a reference to a service that does a particular job, it invokes an import operation, passing as parameters a description of the service required. Given this import request, the Trader checks appropriate offers for acceptability. To be acceptable, an offer must have a type that conforms to that requested and have properties consistent with the constraints specified by an importer.

Trading service in a single trading domain may be distributed over a number of trader objects. Traders in different domains may be federated. Federation enables systems in different domains to negotiate the sharing of services without losing control of their own policies and services. A domain can thus share information with other domains with which it has been federated, and it cannot be searched for appropriate service offers.

Glossary *A*

This glossary contains terms used throughout this guide. Text in *italics* is for clarification only.

For more information about object-oriented terminology, refer to the Semaphore *Glossary of Object-Oriented Terminology*. (Semaphore email: 74743.16@compuserve.com).

activation Copying the persistent form of methods and stored data into an executable address space to allow execution of the methods on the stored data.

application A dynamic object-based application is the (end-user) functionality provided by one or more programs consisting of a collection of interoperating objects. *In common terminology this is usually referred to as a running application or process.* A static object-based application is a set of related types and classes specific to a particular (end user) objective. *In common terminology this is usually referred to as a program*.

application facilities Comprise facilities that are useful within a specific application domain. See *common facilities*.

application objects Applications and their components that are managed within an object-oriented system. Example operations on such objects are open, install, move and remove.

asynchronous request A request where the client object does not pause to wait for results.

atomicity The property that ensures that an operation either changes the state associated with all participating objects consistent with the request, or changes none at all. If a set of operations is atomic, then multiple requests for those operations are serializable.

attribute A conceptual notion. An attribute of an object is an identifiable association between the object and some other entity or entities. Typically, the association is revealed by an operation with a single parameter identifying the object. See related definition for *property*.

behavior The behavior of a request is the observable effects of performing the requested service (including its results).

behavior consistency Ensures that the behavior of an object maintains its state consistency.

binding (or, more specifically, **method binding**) The selection of the method to perform a requested service and of the data to be accessed by that method. See also *dynamic binding* and *static binding*.

class An implementation that can be instantiated to create multiple objects with the same behavior. An object is an instance of a class. Types classify objects according to a common interface; classes classify objects according to a common implementation.

class inheritance The construction of a class by incremental modification of other classes.

class object An object that serves as a class. A class object serves as a **factory**. See *factory*.

client object An object issuing a request for a service. See also *server object. A given object may be a client for some requests and a server for other requests.*

common facilities Provides facilities useful in many application domains and which are made available through OMA-compliant class interfaces. See also *application facilities*.

component A conceptual notion. A component is an object that is considered to be part of some containing object.

compound object A conceptual notion. A compound object is an object that is viewed as standing for a set of related objects.

conformance A relation defined over types such that type x conforms to type *y* if any value that satisfies type *x* also satisfies type *y*.

context-independent operation An operation where all requests that identify the operation have the same behavior. (In contrast, the effect of a context-dependent operation might depend upon the identity or location of the client object issuing the request.)

core object model Basic object model that forms the basis for the OMG's Object Management Architecture; formally defined in Chapter 4 of this guide. The Core Object Model defines concepts such as object and non-object types, operations, signatures, parameters, return values, interfaces, substitutability, inheritance, and subtyping.

data model A collection of entities, operators, and consistency rules.

delegation The ability for a method to issue a request in such a way that self-reference in the method performing the request returns the same object(s) as self-reference in the method issuing the request. See *self-reference*.

dynamic binding Binding that is performed after the request is issued (see *binding*).

embedding Creating an object out of a non-object entity by wrapping it in an appropriate shell.

exchange format The form of a description used to import and export objects.

export To transmit a description of an object to an external entity.

extension of a type The set of values that satisfy the type.

factory A conceptual notion. A factory provides a service for creating new objects.

generalization The inverse of the specialization relation.

generic operation A conceptual notion. An operation is generic if it can be bound to more than one method.

handle A value that unambiguously identifies an object. See also *object name*.

implementation A definition that provides the information needed to create an object, allowing 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 also typically includes information about the intended type of the object.

implementation inheritance The construction of an implementation by incremental modification of other implementations.

import Creating an object based on a description of an object transmitted from an external entity.

inheritance The construction of a definition by incremental modification of other definitions. See also *implementation inheritance*.

instance An object created by instantiating a class. An object is an instance of a class.

instantiation Object creation.

interface A description of a set of possible uses of an object. Specifically, an interface describes a set of potential requests in which an object can participate meaningfully. See also *object interface*, *principal interface*, and *type interface*.

interface inheritance The construction of a new interface using one or more existing interfaces as its basis. The new interface is called a subtype and the existing interfaces are its supertypes.

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

interoperability The ability to exchange requests using the ORB in conformance with the OMG Architecture Guide. *Objects interoperate if the methods of one object request services of another.*

link A conceptual notion. A relation between two objects.

literal A value that identifies an entity that is not an object. See also *object name*.

meaningful request A request where the actual parameters satisfy the signature of the named operation.

metaobject An object that represents a type, operation, class, method, or other object model entity that describes objects.

method Code that may be executed to perform a requested service. *Methods associated with an object may be structured into one or more programs.*

method binding See *binding*.

multiple inheritance The construction of a definition by incremental modification of more than one other definition.

non-object A member of the set of denotable values. Non-objects are not labeled by an object reference.

object A combination of a state and a set of methods that explicitly embodies an abstraction characterized by the behavior of relevant requests. An object is an instance of a class. *An object models a real world entity and is implemented as a computational entity that encapsulates state and operations (internally implemented as data and methods) and responds to requests for services.*

A basic characteristic of an object is its distinct object identity, which is immutable, persists for as long as the object exists, and is independent of the object's properties or behavior.

Methods can be owned by one or more objects.

Requests can be sent to zero, one, or more objects.

State data can be owned by one or more objects.

State data and methods can be located at one or more locations.

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 and its associated resources to become available for reuse.

object interface A description of a set of possible uses of an object. Specifically, an interface describes a set of potential requests in which an object can meaningfully participate as a parameter. It is the union of the object's type interfaces.

object name A value that identifies an object. See also *handle*.

object services A collection of interfaces and objects that support basic functions for using and implementing objects. Object Services are necessary to construct any distributed application and are independent of application domains. Interfaces for object services are specified by OMG in *CORBAservices* and currently include Life Cycle, Events, Naming, Persistent Object, Transaction, Concurrency Control, Relationships, and Externalization.

object type A type the extension of which is a set of objects (literally, a set of values that identify objects). In other words, an object type is satisfied only by (values that identify) objects. See also *interface type*.

OMA-compliant application An application consisting of a set of interworking classes and instances that interact via the ORB. Compliance therefore means conformance to the OMA protocol definitions and interface specifications outlined in this document.

OMG IDL Object Management Group Interface Definition Language. A programming language–independent way to specify object interfaces. OMG IDL must be used to specify all object interfaces in a CORBA-compliant system; it is used only for specifications, not for programming. The specification for OMG IDL is contained in *CORBA*.

operation A service that can be requested. An operation has an associated signature, which may restrict which actual parameters are possible in a meaningful request.

operation name A name used in a request to identify an operation.

ORB (Object Request Broker) provides the means by which objects make and receive requests and responses.

parameter Part of an operation's signature. It gives the type and name of an argument to the operation.

participate An object participates in a request if one or more of the actual parameters of the request identifies the object.

passivation The reverse of activation.

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

principal interface The interface that describes all requests in which an object is meaningful.

property A conceptual notion. An attribute the value of which can be changed.

protection The ability to restrict the client objects for which a requested service will be performed.

query An activity that involves selecting objects from implicitly or explicitly identified collections based on a specified predicate.

referential integrity The property that ensures that a handle which exists in the state associated with another object reliably identifies a single object.

request An event consisting of an operation and zero or more actual parameters. A client issues a request to cause a service to be performed. Also associated with a request are the results that may be returned to the client. *A message can be used to implement (carry) a request and/or a result.*

result 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.

security domain An identifiable subset of computational resources used to define security policy.

self-reference The ability of a method to determine the object(s) identified in the request for the service being performed by the method. (Self-reference in Smalltalk is indicated by the keyword *self*. See also *delegation*.)

server object An object providing response to a request for a service. See also *client object*. *A given object may be a client for some requests and a server for other requests.*

service A computation that may be performed in response to a request.

signature Defines the types of the parameters for a given operation.

single inheritance The construction of a definition by incremental modification of one definition. See also *multiple inheritance*.

specialization A class *x* is a specialization of a class *y* if *x* is defined to directly or indirectly inherit from *y*.

state The information about the history of previous requests needed to determine the behavior of future requests.

state consistency Ensures that the state associated with an object conforms to the data model.

state integrity Requires that the state associated with an object is not corrupted by external events.

state-modifying request A request that by performing the service alters the results of future requests.

state variable Part of the state of an object.

static binding Binding that is performed prior to the actual issuing of the request. See also *binding*.

supertype When an object of type A is substitutable for an object of type B, A is a subtype of B, and B is a supertype of A. Although the Core Object Model effectively defines its inheritance to be the same as subtyping, the two concepts are separate. Subtyping determines substitutability, whereas inheritance is a mechanism for specifying one entity in terms of another.

synchronous request A request where the client object pauses to wait for completion of the request.

transient object An object the existence of which is limited by the lifetime of the process or thread that created it.

type A predicate (Boolean function) defined over values that can be used in a signature to restrict a possible parameter or characterize a possible result. Types classify objects according to a common interface; classes classify objects according to a common implementation.

type interface Defines the requests in which instances of this type can meaningfully participate as a parameter. *Example: given that document type and product type the interface to document type comprises edit and print, and the interface to product type comprises set price and check inventory, then the object interface of a particular document which is also a product comprises all four requests.*

type object An object that serves as a type.

value Any entity that can be a possible actual parameter in a request. Values that serve to identify objects are called "object names." Values that identify other entities are called "literals."

value-dependent operation An operation where the behavior of the corresponding requests depends upon which names are used to identify object parameters (if an object can have multiple names).