This report defines an abstract object model for object-oriented systems. The abstract object model provides an organized presentation of a core set of concepts and terminology. It defines a partial model of computation as seen by applications built upon a distributed object management infrastructure. Its purpose is to provide a general framework that embodies the essential characteristics of object systems. Within this framework, it should be possible to define many specific object models by elaborating (refining) and populating (defining specific entities of) the general model. The abstract model was developed for use by the Object Management Group as a common conceptual framework for requesting, describing, and evaluating proposed object technologies.
Preface

This report was created for use by the technical committee of the Object Management Group. It has been proposed to the OMG technical committee to be included as part of a larger document that describes the status of the developing OMG standard. This report is not an official document of the OMG. For further information on the status of the OMG standard, please contact the Object Management Group, Framingham Corporate Center, 492 Old Connecticut Path, Framingham, Massachusetts 01701.

Acknowledgements

This report documents work performed as an activity of the technical committee of the Object Management Group. It has benefited from discussions with and feedback from many people, including Jeff Eastman, Jon Gustafson, Brian McBride, Ian Thomas, Jan te Kiefte, Lee Scheffler, Rod Bark, Pete Showman, Dennis Tenney, and Tom Allen. Most notably, it has benefited from extensive discussions with Bill Kent, particularly on the topics of object interfaces, object identity, and object state. Bill provided the initial challenge to incorporate the generalized object model, and forced me on many occasions to explain things more carefully and clearly. Bill's input to this document should not be taken to imply consensus on all issues. Finally, this work is an outgrowth of the previous work of the HP "object task force", which was initiated by Alain Couder.
1 Introduction

The OMG object model provides an organized presentation of the OMG object concepts and terminology. The object model partially defines the model of computation seen by OMG compliant applications, and through applications by end users. Its purpose is to define a conceptual framework for OMG proposed technologies, and in particular, to motivate the basic design choices to be made in proposing and adopting specific technologies.

The OMG object model does not define the components of an OMG system or their interfaces. The object model does not define the structure of OMG applications. The object model does not define specific objects, specific kinds of objects, or specific interfaces to objects. These issues are addressed by the OMG reference model.

It is expected that any proposed OMG technology will both elaborate and populate the model described here. Elaborating the model means making it more specific, for example, by defining the form of request parameters or the language used to specify types. Populating the model means introducing specific instances of entities defined by the model, for example, specific objects, specific operations, or specific types.

One goal of this object model is to avoid placing unnecessary restrictions on the possible models of proposed technologies. The requirements embodied in the OMG object model are those considered “essential” to the concept of object technology [3].

The fundamental characteristic of object systems is the distinction between the semantics of objects as perceived by object clients and the implementation of those semantics in terms of actual data representations and executable code. Object semantics includes such concepts as object creation and identity, requests and operations, types and signatures. Object implementation includes such concepts as methods, data structures, classes (as implementation definitions), and implementation inheritance.

The object model first describes concepts related to object semantics and then describes concepts related to object implementation. Because object semantics are visible to clients, the object model is most specific and prescriptive in defining object semantics concepts. 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 other characteristics of object systems that are outside the scope of the OMG object model. Some of these concepts are aspects of application architectures, 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, attributes and links, copying of objects, change management, and transactions. Also outside the scope of the OMG object model is the model of control and execution.

In the object model definition, *italics* are used to introduce or define new terms. Parenthesized sentences and indented paragraphs are commentary and are not part of the object model.
1.1 Classical vs. Generalized Object Models

As defined below, the OMG object model is a generalized object model. As such, it differs profoundly from the object models most familiar in current systems. It is intentionally a significant generalization of the more familiar object models. Before describing the OMG object model, it is helpful to compare and contrast the two different kinds of object model. The terms used in this description are defined later, but the intuition should be clear.

A classical object model, used in most existing object technologies, is one 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 request identifies an object and zero or more actual parameters. 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 by the system.

A generalized object model, used for example in the Common Lisp Object System [2] and the Iris database [1], is one where a client issues a request that identifies an operation and zero or more parameters, any of which may identify an object. In the generalized object model, method selection may be based on any of the objects identified in the request, as well as the operation. Because method selection may be based on multiple objects, it is best viewed as being done by the system, rather than by an object. Operationally, of course, method selection could be performed in multiple stages, with the final stage being carried out by an object.

The classical object model is a special case of the generalized object model. The classical sending of a message to an object is equivalent to a generalized request where method selection is based strictly on the operation and the object identified by the first parameter. For example, sending the message print a-printer to the object a-spreadsheet in the classical model is equivalent to issuing the request print a-spreadsheet a-printer in the generalized model, under the assumption that method selection is based on a-spreadsheet and not a-printer.

The generalized object model allows an object technology to provide additional functionality. For example, on object technology could support defining a method that is invoked only when the print operation is requested on a spreadsheet and a particular kind of printer. This specialized method could take advantage of the unique capabilities of the particular kind of printer.

Distributed implementations of classical object models generally assign locations to objects. With generalized models, it is also possible to assign locations to operations. Thus, remote procedure calls are also a special case of the generalized object model.

2 Overview

The OMG system is a computational system that includes code and data.
The viewpoint taken in this document is that there is exactly one OMG computational sys-
tem, although it may consist of many isolated (disconnected) islands of computation. The
motivation for this viewpoint is that the object model should gracefully handle the situation
where two previously disconnected “islands” are later interconnected. This statement is not
a requirement that a specific object technology support the arbitrary interconnection of in-
dependent object systems (networks). In other words, the object model does not require a
universal naming architecture.

The entities of the OMG system include OMG objects, values (including object names and han-
dles), operations, signatures, and types (including interface types).

3 Object Semantics

This section defines the concepts associated with object semantics, i.e., the concepts relevant to
using objects. (See Figure 1.)

3.1 Objects

The OMG computation model includes entities known as OMG objects. (OMG objects are not
to be confused with other possible entities also called objects, such as C++ objects.) For brevity,
OMG objects will be called objects henceforth. Objects have certain characteristics, which are
explained below.

An object participates in providing services to clients. The clients of a service are any entity
capable of requesting the service. (Clients may include application programs, objects, agents,
and end users acting through a user interface.)

3.2 Requests

Clients request services by issuing requests. A request is an event, i.e., something that occurs at
a particular time during the execution of the computational system. The information associated
with a request consists of an operation and zero or more (actual) parameters.

A request is to be distinguished from the static program text whose evaluation causes a request
to be issued. The object model does not define the syntax by which programs cause requests
to be issued. For example, it does not require that the operation appear first in a request-
issuing construct. The object model does not define the evaluation process by which forms in
a program are resolved to particular operations or particular parameter values.

An operation is identified in a request in some form, which is called an operation name. The
object model does not describe or define how clients name operations. It is possible that
different clients could use different names to refer to the same operation, or use the same
name to refer to different operations. The notion of operation identity is defined below.
The intended meaning of a request is that the operation identifies the service to be performed
and the parameters specify both the objects that are to participate in providing the service
and any other information needed to specify the result desired by the client. This object
model allows a request to name zero objects or multiple objects, in addition to the usual case
of a single object.

Operations are (potentially) generic, meaning that a single operation can be requested of objects
with different implementations, resulting in observably different behavior.

This is a philosophical principle, intended to characterize the power of the computational
model. The intuition is that when a given operation is requested of different objects, the code
that is executed to perform the requested service may be different, and may have observably
different behavior. In particular, new kinds of objects may be introduced into the system that
provide specialized behavior for existing operations. One would expect the different behaviors
for a given operation to share some basic intent, such as the intent to cause something to be
printed; however, this expectation is not enforceable by the system.

Operations are created by explicit action; each such action creates an operation that is distinct
from operations created previously or in the future.

This statement defines the notion of operation identity, which is equivalent to the notion
of object identity described below. It implies that at a fundamental level, there can be no
"accidental" overloading of operations: for two clients to refer to the same operation, the
identity of the operation must have been communicated from a common origin to each client.
A developer can therefore define a new service (by creating a new operation), with the certainty
that the service will be distinguishable from all other services. A system that predefines certain operations could be viewed as creating those operations automatically during system initialization.

The object model does not specify whether an operation is an object or whether an operation name is a value (i.e., can be used as a parameter).

A value is anything that is a possible (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. (See Figure 2.)

There may be values that identify abstract entities other than objects. Such values are called literals. (The term literal is not intended to suggest that these values are necessarily compile-time constants.) For example, in a particular system, integers might be identifiable in requests, but they might not have all of the characteristics of objects, as defined by that system. In other words, the object model permits but does not require that all referenceable entities be objects. There also may be values that are meaningless, i.e., do not reference any existing entity. For example, some values may be reserved for identifying objects that have not yet been created. This case is ignored in the subsequent discussion.

A handle is an object name that unambiguously identifies a particular object. Within certain pragmatic limits of space and time, a handle will reliably identify the same object each time the handle is used in a request.
An object technology might provide object names that identify different objects at different times or in different "locations", for example, a value that denotes the nearest available printer of a particular kind. These values are object names, but not handles. While both are useful, the existence of reliable, unambiguous object names (handles) is an essential characteristic of an object system.

The distinction between object names and handles can be illustrated by analogy with the Unix file system. Assume Unix files are objects, and consider three possible ways of identifying a Unix file: pathnames, file descriptors, and inodes. A pathname is an object name, but not a handle, since the file named by a pathname can change at any instant as the result of renaming files or directories. (A relative pathname has the additional ambiguity of being dependent for its interpretation on the file directory identified as "current" by a given process at a given time.) A file descriptor is a handle, because within a single process (or process tree, if file descriptors are shared), it will always identify the same file object. Although an inode unambiguously identifies a particular file in the context of a single file system, an inode is not an object name, because an inode is not a value (it cannot be used in normal Unix system calls). Having both pathnames and file descriptors is useful. A major limitation of Unix is the lack of a persistent form of a file descriptor.

The object model allows an object to have multiple handles, in a single context or in different contexts. For example, a handle in the context of a Unix process might be a pointer into the address space of that process. In different processes, the same object would be identified by different pointers. Such handles are valid only within the address space of the process and only during the lifetime of the process. (The Unix file system example above illustrates multiple handles for an object in a single context: A Unix process can have multiple file descriptors that identify the same file. The file descriptors differ in having distinct associated positions into the file.)

An object is defined to participate in a request if one or more of the actual parameters of the request identifies the object. (The word "participate" is not intended to imply any particular degree of involvement by the object in actually providing the requested service.)

A request causes a service to be performed on behalf of the client. One outcome of performing a service may be that some results are returned to the client. The results associated with a request may include values as well as status information indicating that exceptional conditions were raised in attempting to perform the requested service.

The object model does not specify how parameters, results, and exceptional conditions are identified by a client. Possibilities for identifying parameters and results include sequential/positional or by name. These details would be specified by a particular object technology that elaborates this model.

Note that the object model does not specify that the operation must be statically identified in client programs. For example, a programming language interface might provide a construct for issuing a request in which the operation name is a variable. Such a construct would be similar to the `funcall` construct in Lisp.
3.3 Behavior and Abstraction

The behavior of a request is the observable effects resulting from performing the requested service. (The effects may be visible to parties other than the requesting client.) The behavior of a request includes the results returned to the client (including both the values returned and the exceptional conditions reported), as well as indirect effects on the results of future requests (by the same or different client).

In general, the possible behaviors of a request are any arbitrary computation, including computations that issue other requests. (This statement characterizes the intended power of the computational model.)

The behavior of a request generally depends both on the actual parameters in the request and the state of the computational system. The state of the computational system is a representation of the history of past requests; specifically, it represents the effect of past requests on future behavior. A behavior can thus be modeled as a function whose parameters include the actual arguments of the request and the state of the system. (Technically, this technique is called denotational semantics.)

The state of the system is physically represented by data that persists between the execution of requests. This “persistent data” may include what are effectively references to objects. The object model does not say anything about the stored form of such references. Persistent object references are visible to clients only via the values (object names) that are passed as request arguments and results. For example, a persistent object reference might be revealed to a client as the value that is the result of a service that returns a particular attribute of an object.

The behavior of a request may depend on contextual information (such as the identity of the client), as well as the state of other entities interfaced to the system (such as the state of an attached hardware device) or the effects of direct interactions with users. These additional influences would be modeled as additional arguments to the behavior function.

An object explicitly embodies an abstraction, which is characterized by the behavior of relevant requests. The abstraction embodied by an object is meaningful to its clients. An object that is visible to an end user typically models some real-world object that is familiar to the end user.

These philosophical principles describe the intent to use objects to embody meaningful abstractions. A client that expects to manipulate spreadsheets should be provided with objects that perform the services expected of spreadsheets. The fact that a spreadsheet object behaves as a spreadsheet is inherent in the object and not dependent upon the client’s knowledge. A contrasting example from Unix may help to clarify. A Unix file is a low level abstraction of a mutable byte sequence. No additional semantics are attached to a Unix file; instead, the interpretation of a Unix file is left to applications that manipulate the file. A Unix file might represent a spreadsheet, but the Unix file system does not record this intent. Instead, it is up to the client to know to invoke the spreadsheet application when operating upon this particular Unix file. These statements are not intended to rule out systems that support multiple levels of abstraction.
3.4 Object Creation and Identity

An object is created by an explicit action. The object has identity: it is distinct from every object that has previously been created or will be created in the computational system.

This notion of object identity can be formally modeled by assuming an infinite set of object identifiers, from which is allocated at each object creation event an object identifier never before allocated. Object identity is then modeled as equivalence of object identifiers. (See below for a discussion of abstract object identity.) Note that the object model does not require the explicit use of unique object identifiers in the computational system. Unique identifiers are an implementation technique. Even if unique identifiers are used, they need not be revealed to clients.

Object creation is made available to clients through operations. The result of object creation is revealed to the client in the form of a handle that identifies the newly-created object.

The behavior of a request may depend upon the identity of the participating objects. In particular, a component of the state of the computational system may be uniquely associated with a particular object. In general, such state cannot be modified without issuing specific requests in which the object participates. In other words, distinct objects may have distinct state that is encapsulated by operations.

These statements describe the intended power of the computational system. It must be possible to define objects with the stated properties. The classic example of encapsulated state is a stack object that provides the services push and pop: the state implicitly associated with a stack object is the sequence of elements that have been pushed but not yet popped. This state is altered only by push and pop. It is revealed to clients only by the time-varying behavior of pop.

There may be additional components of the state of the computational system that are not uniquely associated with individual objects. For example, consider the operations that define the created and created-by relations over two sets of objects, program source objects and software developer objects. One could think of the state introduced by these operations as being associated with pairs of objects, or alternatively with the operations themselves. The notion of associated state is not intended to restrict the form of the persistent representation of the state.

The object model does not define any further details of object creation. The object model does not define a notion of object destruction.

3.5 Abstract Object Identity

Objects are distinguishable by clients to the extent that they affect the behavior of requests.

In other words, the object model allows but does not require that object identity, as defined above, be directly revealed to clients. There are other possible notions of object identity that
may be more appropriate for clients, as illustrated below. While not supported directly by the object model, it should be possible to use the abstraction capabilities of the system to provide these more abstract notions of object identity. Such notions would be realized in the form of specific services that can be requested by clients. For example, a service could be provided that compares two values to see if they identify the "same" object according to some notion of abstract object identity. Based on the intended notion of abstract object identity, this service may report that values identifying two distinctly-created objects are the "same". The object model does not specify any particular such service. Implementing such a service would require the ability to test objects for distinct creation.

One example of a more abstract notion of object identity is the notion of "merging identity". This notion would allow two originally distinct objects to become a single, indistinguishable entity, from the point of view of clients. It could be useful when objects are used to model an evolving understanding of the real world, for example, to model the realization that the entities "Joe's father" and "Bob's brother" are actually the same person.

Another example of a more abstract notion of object identity is the notion of (immutable) mathematical abstractions. An example of a mathematical abstraction is the set whose elements are the integers 1 and 2. There is only one such set. In a system that uses objects to represent mathematical abstractions, there could be multiple objects representing the same abstraction. Clients, however, should see these objects as "identical". A service could be defined that would provide that illusion to clients.

The motivation for supporting abstract object identity, rather than providing more complex forms of identity directly in the object model, is to allow flexibility in defining new concepts of identity and to avoid requiring an object technology to support concepts of object identity that may be very difficult to implement. More complex forms of object identity are not prohibited in a particular object model.

### 3.6 Meaningful Requests

Not all possible requests are meaningful: an object cannot participate in providing all possible services, and there may be other constraints on the actual parameters in a request.

The notion of meaningful request corresponds to the notion of type correctness in typed systems. It corresponds to the absence of message not understood errors in systems like Smalltalk.

Each operation has an associated signature that may restrict the possible (actual) parameter values that are meaningful in requests that name that operation. A request whose actual parameters do not satisfy the signature associated with the operation named in the request is meaningless. Issuing a meaningless request may result in an exception condition being reported to the client.

For those readers familiar with typed programming languages, a signature is like a procedure type or a function type. In this particular case, a signature is like the type of a generic function (one that accepts arguments of multiple types). For example, the signature of the plus operation might specify that there are two parameters, the actual arguments can be any
combination of integers or reals, and the result is an integer if the two arguments are both integers, or a real otherwise.

The object model does not specify the language used to describe signatures. An elaboration of this object model would define a specific formalism for signatures.

Note that if operation names are values (meaning that operation names can be parameters in requests), then signatures can be considered to be types, as defined below. It is in this sense that Figure 1 shows operations to have types.

When an operation is created, its signature is specified.

The object model does not specify whether the signature of an operation can change after it is created.

A signature may also characterize the possible result values and exceptional conditions associated with the corresponding requests. This additional information does not affect the notion of meaningful request, but may be used by tools that perform static analysis of programs.

The object model does not specify the possible characterizations of results. Typically, signatures will characterize results in terms of their types. Signatures may include additional information such as formal or informal descriptions of the behavior of the operation, or information about whether requests are synchronous or asynchronous.

A type is a predicate (boolean function) defined over values that can be used in a signature to restrict a possible parameter or to characterize a possible result.

This statement is a very general definition of the notion of type. According to this definition, a type such as integer would be thought of as a predicate that when applied to an integer value returns true and when applied to any other value returns false. This definition allows more unusual types, such as the type of odd integers. Note that this definition allows a single value to have multiple types (satisfy multiple type predicates). For example, the value denoting the integer 1 is both an integer and an odd integer.

The object model does not specify what kinds of types may exist in a system. An elaboration of this model would define a specific formalism for types. The choice of formalism may restrict the set of predicates that can be used as types.

This definition of type allows (but does not require) types that distinguish between different values that identify the same object (or other abstract entity). If a particular object model does not allow types to distinguish different values that identify the same object (or other abstract entity), then one can view types as predicates over objects (and other abstract entities), rather than over values.

Existing object-oriented programming languages provide types that distinguish between different implementations of objects — called classes. A more general notion of interface type is defined below that captures the intuition of generic operations.

The object model does not specify whether types are objects, or whether types are invocable as operations. In some systems, type checking is performed at compile time and types are not represented during execution.
The extension of a type is the set of values that satisfy the type. The extension of a type may change over time as new objects are created, and may change as the result of side-effects on existing objects.

If types do not distinguish multiple values that identify the same object, then one can think of the extension of the type as including the objects identified by those values, rather than the values themselves.

A relation called conformance is defined over types. A type $a$ conforms to a type $b$ if any value that satisfies type $a$ also satisfies type $b$.

A particular object model will define the conformance relation, which must be consistent with logical implication. Conformance permits a hierarchical classification of objects, which is one of the traditional uses of inheritance. An example of conforming types is presented below.

An object type is a type whose extension 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.

An object may satisfy many types. For example, a Lotus 2.2 spreadsheet object might satisfy the following types (of decreasing specificity): Lotus 2.2 spreadsheet, Lotus spreadsheet, spreadsheet, printable object. In this example, the types are all related by conformance. Some systems might allow an object to satisfy types unrelated by conformance. For example, an object representing a particular individual might satisfy both the type person and the type shareholder (there can be people who are not shareholders and shareholders that are not people, e.g., corporations).

An interface is 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. An object satisfies an interface if it is meaningful in each potential request described by the interface.

In a classical object system, an interface consists of a set of operations. For example, the interface of "printable objects" would consist of the single operation print. An object satisfies a classical interface if it is meaningful as the first parameter in any request that identifies one of those operations. In other words, an object satisfies such an interface if it can provide all of the services identified in the interface. A system supporting the generalized object model might support more complex interfaces that identify specific formal parameters of operations where an object can meaningfully appear. For example, the "printable object" interface would be satisfied by those objects that can appear as the first argument to a print request, and the "printer object" interface would be satisfied by those objects that can appear as the second argument to a print request.

Interfaces are in some sense duals of signatures. One can imagine the system containing a set of "constraints" that determine the meaningfulness of requests. Signatures are an operation-centered view of this information, and interfaces are an object-centered view of this information. The object model does not specify how this information originates. An object might be
printable because it defines behavior for the *print* operation. Alternatively, an object might be printable because it has been asserted to satisfy the type *printable object*, which is the declared type of the first argument of the *print* operation.

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

An interface can both describe what a particular object can do, as well as describe how a client intends to use an object. For example, an interface type might be used to declare a formal parameter of a procedure, to indicate how the actual parameter object will be used in the procedure.

Conformance over interface types means that any object that can be used as described by the smaller interface can also be used as described by the larger interface. Conformance over interface types is related to set inclusion, at least for simple type systems. The interface consisting of the *print* and *copy* operations conforms to the interface consisting of just the *print* operation, since any object that can be printed and copied can (by implication) simply be printed.

Each object has a *principal* (maximal, most general) *interface* that describes all requests in which the object is meaningful.

The principal interface of an object may enlarge or shrink for various reasons, such as the definition of new operations. The object model permits but does not require the existence of a service by which a client can determine the principal interface of an object.

4 Object Implementation

This section defines the concepts associated with object implementation, i.e., the concepts relevant to implementing objects. (See Figure 3.)

4.1 Performing Requests

The practical effect of performing a requested service is to cause some code to execute that accesses some stored data. The stored 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. The code that is executed to perform a service is called a *method*.

An OMG system includes an infrastructure that serves as the mediator between clients and services. (Of course, a service can also be a client, by issuing requests of its own.) A primary function of the infrastructure is to select the appropriate code to perform a requested service, and to execute that code giving it access to the appropriate data.
The selection of a method to perform a requested service and the selection of the data to be accessed by the method is called *binding*. Binding can be *static*, meaning that the selection (typically, of the method) can be performed prior to the actual issuing of the request, or *dynamic*, meaning that the selection is performed after the request is issued.

Static binding is performed by compilers based on declarations. A program construct that issues a request often includes variables. The declarations in the program may restrict the possible values denoted by those variables.

The code selected to perform a requested service can depend upon several factors, in particular, the identity of the objects participating in the request.

The selection of code based on the participating objects is the basis for the description of operations as generic. In a classical object system, the selected code depends only upon the operation and the class of the object identified by the first parameter in the request.

The selected code may also depend on other factors, such as the "location" of the object or the "location" of the client. (The object model does not require a concept of location.) A specific object technology might impose constraints on the process of binding that provide increased predictability for clients. The OMG object model intentionally does not impose such constraints.

Performing a requested service causes a method to execute that may access stored data. If the persistent form of the method and the data is not executable, then it may be necessary to first copy the method and the data into an executable address space. This process is called *activation*. The reverse process is called *passivation*. 

Figure 3: Primary object implementation concepts
4.2 Realizing Behavior

An OMG system must provide mechanisms for realizing the behavior of requests. These mechanisms would include definitions of data structures, definitions of methods, and definitions of how the OMG infrastructure is to select methods to execute and to select the data to be made accessible to the methods. Mechanisms must also be provided to describe the concrete actions associated with object creation, such as the allocation of new data, and the 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 a description of the data structure used to represent the core state associated with an object, as well as definitions of the methods that access that data structure. It will also typically include information about the intended type of the object. (An implementation might also be used to add new behavior to an existing object.)

An OMG system may allow a method definition to reference the object(s) identified in the request for the service being performed by the method. This ability, called self-reference, allows a method to issue additional requests involving the same object(s). Self-reference is useful when a single method may be executed for different objects. (Self-reference in Smalltalk is indicated by the keyword self.)

4.3 Sharing Behavior

An OMG system will typically provide mechanisms that allow objects with the same behavior to share implementations. A class is an implementation that can be instantiated to create multiple objects. The resulting objects are called instances of the class. Self-reference is useful when methods are shared by multiple instances of a class.

However, there may also be several different implementations that support the same behavior. For example, there may be specific implementations for specific hardware environments. Also, there may be a series of implementations introduced over time with performance enhancements.

In most existing object systems, classes define both an implementation and an interface type. Merging these roles in a single entity makes it more difficult for a system to support different implementations of the same behavior. A class may or may not be an object.

An OMG system will typically provide mechanisms that allow objects with similar behavior to share parts of their implementations. For example, implementation inheritance may be provided to allow an implementation to be defined as an incremental refinement of other implementations.

An alternative technique for providing sharing of implementations is delegation. Delegation is 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.
5 Glossary

**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.

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

**binding.** The selection of the code to perform a requested service and of the data to be accessed by the code. See method.

**class.** A class is an implementation that can be instantiated to create multiple objects with the same (initial) behavior.

**client.** A client is any computational entity capable of requesting services.

**conformance.** Conformance is a relation defined over types that is consistent with logical implication. A type $a$ conforms to a type $b$ if any value that satisfies type $a$ also satisfies type $b$.

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

**delegation.** Delegation is 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.

**encapsulated.** An object is encapsulated if it has associated state that can be directly accessed or modified only by particular methods.

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

**generic operation.** A conceptual notion. An operation is generic if it identifies a service that can be requested of objects with different implementations, resulting in observably different behavior.

**handle.** A handle is a value that reliably identifies an object. See object name.

**identity.** A conceptual notion. An entity has identity if it is inherently distinguishable from other entities, e.g. on the basis of distinct creation events.

**implementation.** An implementation 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 a description of the data structure used to represent the core state associated with an object, as well as definitions of the methods that access that data structure. It will also typically include information about the intended type of the object.

**implementation inheritance.** The construction of an implementation by incremental refinement of other implementations.
inheritance. The construction of a definition by incremental refinement of other definitions. See implementation inheritance.

instance. An object created by instantiating a class.

instantiation. Creating an object using a class.

interface. An interface is 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.

interface satisfaction. An object satisfies an interface if it is meaningful in each potential request described by the interface.

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

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

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

method. Code that may be executed to perform a requested service.

object. An OMG object is a distinguished kind of entity in the OMG computational system.

object creation. An event that causes an object to exist that is different than any previously existing object.

object implementation. See implementation.

object name. An object name is a value that identifies an object. See handle.

object type. An object type is a type whose extension 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 interface type.

operation. An operation is an identified 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.

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.

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

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

results. The results of a request are 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.

self-reference. Self-reference is 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 delegation.

service. A service is a computation that may be performed upon request.

signature. A signature is a description of an operation that may restrict the possible (actual) parameter values that are meaningful in requests that name that operation. A request whose actual parameters do not satisfy the signature associated with the operation named in the request is meaningless.

state. The state of the computational system is the information about the history of previous requests needed to explain the behavior of future requests.

static binding. Binding that is performed prior to the actual issuing of the request, based on static properties of the client.

type. A type is a predicate (boolean function) defined over values that can be used in a signature to restrict a possible parameter or characterize a possible result.

value. A value is any entity that may 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.
References

