The Essence of Objects: Common Concepts and Terminology

Alan Snyder
Software and Systems Laboratory
HPL-91-50
June, 1991

objects, object-oriented programming, object-oriented databases, object-oriented user interfaces, distributed systems, terminology

Concepts originally developed in object-oriented programming languages are appearing in many other domains. There are object-oriented databases, object-oriented application frameworks and integration platforms, even object-oriented user interfaces. Object concepts are widely used in distributed systems and are prominent in the draft ISO standards for the management of open systems. In reviewing the "object" concepts in several systems, we believe we have identified the essential concepts that appear in most or all of these systems.

Identifying the common concepts is made more difficult by a lack of common terminology. The lack of common terminology hinders communication among researchers, developers, and users. In this report, we propose a common terminology for object concepts, chosen to be broadly applicable to many domains, not just programming languages. We describe the essential concepts, give examples, and provide a glossary defining the key terms.

Published in *IEEE Software*, January, 1993

1 Introduction

Concepts originally developed in object-oriented programming languages are appearing in many other domains. There are object-oriented databases, object-oriented application frameworks and integration platforms, even object-oriented user interfaces. Object concepts are widely used in distributed systems and are prominent in the draft ISO standards for the management of open systems.

Do these domains share a common concept of what it means to be object-oriented? In reviewing the "object" concepts in several systems, we believe we have identified the essential concepts that appear in most or all of these systems.

We present the concepts in a series of descriptive statements, summarized in the table on page 2. Each statement is illustrated by examples from various domains. Sources for examples include: Smalltalk, C++, and the Common Lisp Object System[1] (programming languages), the HP NewWave environment[2] (an application integration environment for personal computing), Xerox ViewPoint[3] (an office workstation), the HP SoftBench software development environment[4] (an integration framework for software development tools), Iris[5] (a database system), Symbolics Dynamic Windows[6] (a user interface), and the ISO draft international standard for the management of open systems.[7] These systems have all been described as object-oriented, to a greater or lesser extent.

To help explain the concepts, we describe their possible benefits and give contrasting examples of solutions that are not object-oriented. Our intent is to *describe*, not to *evaluate* the object-oriented approach. We omit the counterarguments that would be needed in a balanced evaluation. The reader should not infer that alternative approaches are without advantage.

Identifying the common concepts is made more difficult by a lack of common terminology. The problem is exacerbated by the use of object concepts in so many disparate disciplines. However, even within the programming language community alone, multiple terms are often used for the same concept, and in several cases, the same term is used with different meanings. The lack of common terminology hinders communication among researchers, developers, and users.

In this report, we propose a common terminology for object concepts. The terminology was chosen to be broadly applicable to many domains, not just programming languages. For this reason, several established terms from object-oriented programming were considered inappropriate. The descriptions of the essential concepts are followed by a glossary defining the key terms.

This work originated as an internal effort to create a terminology for object concepts to improve communication within Hewlett-Packard. We first identified and defined a set of core concepts as a basis for characterizing and contrasting the relevant technologies. Once the concepts were defined, we selected terms for those concepts. The work was refined in the development of an abstract object model for the Object Management Group[8] and is currently being used in an ANSI-sponsored effort to converge object models used in several standards efforts.

The essential concepts

- An object embodies an **abstraction** characterized by **services.**
- Clients request services from objects.
  - Clients issue **requests.**
  - Objects are **encapsulated.**
  - Requests identify **operations.**
  - Requests can identify objects.
- New objects can be **created.**
- Operations can be **generic.**
- Objects can be **classified** in terms of their services (**interface hierarchy**).
- Objects can share implementations.
  - Objects can share a common implementation (multiple **instances**).
  - Objects can share partial implementations (**implementation inheritance** or **delegation**).

2 The Essential Concepts

This section defines the essential concepts of an object system. An object system contains entities called objects that play a visible role in providing services to clients. The exact nature of clients and services depends upon the particular object system. In general, a client can be a person or a program, and a service can be any activity that can be performed at the request of a client. The essential concepts define the common characteristics of object systems. (The papers by Wegner[9] and Thomas[10] present similar material in a more tutorial form.)

An object embodies an abstraction characterized by services.

An object embodies an abstraction.

An object *explicitly* embodies an abstraction that is meaningful to its clients. Although an object may involve data, an object is not just a data structure or a collection of bits; the purpose of the data is to represent information.

Benefit: Clients manipulate meaningful entities. Clients are not responsible for providing an interpretation for data.

Contrast: A non-object model is that data is simply a string of bits that is given an interpretation only implicitly by programs that read or write the bits, possibly different interpretations by different programs. With this model, clients must know how the data is intended to be interpreted. Furthermore, the data may be misinterpreted if it is given to the wrong program.

Example: Users of ViewPoint and the HP NewWave environment are presented with objects that correspond to familiar abstractions, such as documents, folders, and charts.

Example: Smalltalk programmers are provided with objects that support familiar programming abstractions such as sequences and dictionaries.

Contrasting example: In Unix*, an ordinary file is an uninterpreted sequence of bytes. The interpretation of a file is left up to application programs. The same file may be viewed as a C program source by the C compiler and a long text string by `grep` (a string search utility). Various techniques are used to associate files with meaningful abstractions, such as conventions based on the form of file names (e.g., a file named `foo.c` names a C program source file).

Elaboration: Abstractions appear at every level of a system, forming a hierarchy: higher level abstractions are implemented in terms of lower level abstractions. A file is an abstraction of persistent storage, higher level than the actual storage devices. However, most files implicitly implement an even higher level abstraction, such as a C program or a document. It is this higher-level abstraction that is meaningful to most Unix users.

An object provides services.

The abstraction embodied by an object is characterized by a set of services that can be requested by clients. A service may access or modify data and may produce observable effects on other objects. The data that can be modified by a service is called *state.*

Elaboration: The services are appropriate for the embodied abstraction. Services are often complex; they may involve arbitrary transformations on data and the requesting of additional services that affect other objects. A record consisting only of named attributes that may be read and written is a degenerate form of an object whose services are simple read and write operations and whose associated state is its attributes. State may or may not be associated with specific objects, or may be shared by multiple objects.

Benefit: Complex services can be carefully designed to preserve critical integrity constraints. If clients are provided only primitive services, they would be forced to compose them into more complex services themselves, introducing possibilities for error.

Example: A dictionary object provides a service to client programs by which a name is presented to the dictionary and the dictionary returns the corresponding value. The implementation of this service may involve the construction and manipulation of a complex data structure, such as an ordered binary tree. Incorrect manipulation of the underlying data structure could cause the binary tree to become misordered, which could cause the dictionary services to behave incorrectly.

Example: A print spooler object in a distributed environment provides a service by which multiple users can present text files to the spooler for printing. This service is implemented using a transaction mechanism so that requests can be presented concurrently without interference. If applications were provided only low level operations to read and write data, they would have to compose these low level operations in the proper order to guarantee noninterference. In this case, the printer spooler cannot guaran-

*Unix is a registered trademark of AT&T in the U.S.A. and other countries.

tee reliability, because correct processing of concurrent requests depends on the correctness of each application.

Contrasting example: The standard Unix mail delivery service relies on applications following a specific protocol for accessing a user's mailbox file. This protocol involves a specific sequence of steps including testing for and creating a lock file. If an application fails to follow this protocol when removing new mail from the mailbox file, messages delivered concurrently may be lost or garbled.

Clients request services from objects.

Clients issue requests.

Clients respect the abstraction embodied in an object. Instead of directly accessing data, clients issue requests for the services associated with objects. A request causes code to be executed to perform the requested service. The client is concerned only that the requested service be performed, not in the details of where and how the code runs. The *behavior* of a request is the observable effect of performing the requested service (including its results).

Benefit: Clients do not have to remember what code to execute to perform a service. Clients cannot mistakenly execute the wrong code.

Example: Users of ViewPoint and the HP NewWave environment do not have to remember which application to run to manipulate a given object; the system will invoke the proper application in response to user requests (made via mouse gestures directed at an icon, for example).

Example: A Smalltalk program requests a service by sending a message to an object. The object is responsible for determining how to perform the requested service. The only type error in Smalltalk is requesting a service that is unavailable. One cannot apply code to the wrong data.

Benefit: By isolating clients from the details of the implementation of an object, it becomes possible to change how the object is implemented without having to modify the clients (assuming the services appear unchanged to the clients).

Example: A programmer using an object-oriented programming language discovers a new technique for implementing dictionary objects that dramatically improves performance in many cases. This technique uses a hash table instead of an ordered binary tree. The new implementation of dictionary objects provides the same services as the old one, although the data format is different and the code that implements the services is different. The new dictionary implementation can replace the old one in a new release of the object class library. Client programs can be relinked using the new release of the class library to take advantage of the improved implementation of dictionaries. No client source changes or recompilations are required.

Example: A user of the HP SoftBench software development environment decides to use the GNU Emacs editor instead of the standard SoftBench editor. Because the GNU Emacs editor supports the same services (to other applications) as the SoftBench editor, applications that request editor services will continue to function using the GNU Emacs editor.

Example: A foreign tool can be embedded in the HP SoftBench environment by using the HP Encapsulator program to create a wrapper for the tool that supports the services that clients request.

Example: A foreign database can be accessed via the Iris database by providing foreign functions that transform requests by clients into the appropriate calls on the foreign database. (A foreign function in Iris is a service whose implementation is a program written in a conventional programming language.)

Objects are encapsulated.

A client can access objects *only* by issuing requests for service. Clients are prevented from directly accessing or manipulating data associated with objects.

Benefit: By preventing direct client contact with data, one can *guarantee* that objects satisfy certain integrity constraints. One can also *guarantee* that clients will be unaffected by certain changes to the implementations of objects.

Example: In object-oriented programming languages like Smalltalk and C++, a client is prevented in normal usage from directly accessing the private data (instance variables or data members) of an object. (However, both languages provide explicit constructs for circumventing encapsulation in unusual circumstances.)

Requests identify operations.

A request indicates the service to be performed by identifying an *operation*. An operation is an identifiable entity that denotes a service.

Example: The HP NewWave desktop allows the user to identify operations in several ways, such as by selecting menu items or by dropping icons onto the trash can icon or the printer icon.

Example: In C++, an operation is identified by naming a particular member function in a particular class, and by identifying the types of the actual parameters. (The parameter types are used to resolve overloaded functions.) If two classes define the same function name, but the function is not a virtual member function inherited from a common ancestor class, then the functions are considered two different operations.

Example: In the Common Lisp Object System, an operation is identified by a particular generic function object. The same textual name in different contexts can refer to different generic function objects, and in that case would be considered to be different operations.

Example: In the draft ISO standard for the management of open systems, an operation is identified by a globally unique identifier assigned by a registration authority.

Requests can identify objects.

A request can have associated parameters, and the service can return one or more results. A parameter or a result may identify an object. (Although in most systems a distinguished parameter of each request identifies the object to perform the request, other possibilities exist – see the sidebar on classical and generalized object models.)

An object can be identified directly and reliably. Object identification is direct in the sense that one is naming the object, not describing it. Object identification is reliable in the sense that repeating an identification will refer to the same object (subject to certain pragmatic limits of time and space). A value that identifies an object is called an *object reference*.

Elaboration: An object with associated state has an identity that transcends the current values of its state variables; two objects can have the same current state, yet one can change state and the other remain unchanged. For such objects, object identification is a requirement; object description (in terms of attribute values) is inadequate. To use object description requires the introduction of an explicit *identity* attribute, which is disadvantageous because it introduces additional integrity constraints of immutability and uniqueness.

Benefit: Clients can depend on their ability to refer to objects. They do not have to worry that an attempt to refer to an object might access the wrong object or fail entirely (unless the object no longer exists).

Example: An iconic user interface allows the user to identify objects by pointing at visual representations on the screen. The user interface software maintains the mapping

Classical and Generalized Object Models

Most current object systems have what we call a *classical object model*. In a classical object model, each request contains a distinguished parameter that identifies a target object. The target object controls the interpretation of the request. A request in a classical object model is often called a *message*, and is viewed as being delivered to the target object where it is processed. For example, a request to print a document on a printer might be expressed as a request to the printer object to print the document object.

A *generalized object model* does not require a request to identify a target object. A request contains zero or more parameters, any of which might identify an object. The interpretation of a request can be based on any or all of the parameters; there is no distinguished parameter. The service might be provided by one of the identified objects, or by a third party. For example, a request to print a specific document on a specific printer might be fulfilled by a procedure that embodies specialized knowledge about printing that one kind of document on that one kind of printer; this procedure would be invoked only for requests that identify a document *and* a printer of those specific kinds.

The generalized object model encompasses the classical object model as a special case. The classical sending of a message to an object is equivalent to a generalized request where the service is provided by the object identified by a specific parameter corresponding to the distinguished parameter in the classical model. The generalized object model also encompasses ordinary procedures as a special case, where the interpretation of a request is based strictly on the identified operation.

Generalized object models are used in the Common Lisp Object System and the Iris database. While the full ramifications of the generalized model are not yet known, it is clear that the more general model is needed to solve problems where application knowledge cannot be neatly partitioned among individual objects.

from that visual representation to the underlying data. Because the mapping from icon to data is reliable, the user can think of the icon as *being* the data.

Contrasting example: The standard Unix user interface uses textual names to refer to files. A textual name is mapped to the actual file using a directory hierarchy. This mapping is unreliable because files can be moved or the directory structure rearranged at any time. For example, if the user lists a file directory and the display shows a file foo, the user cannot be sure that later performing the command more foo will display the same file, or any file at all. (Within the Unix kernel itself, *inode* numbers are often used to reliably identify files.)

Example: In contrast to Unix directory listings, where a displayed file name does not reliably identify a file, in Dynamic Windows, when the user lists a file directory, the system remembers which file (object) corresponds to each text string on the screen. If the listing shows a file foo, the user can click on the text foo in the listing to bring up a menu of actions that can be performed on that file. This action works even if the file has since been renamed.

Example: In the HP NewWave environment, an application can create a link from a document object to a chart object. This link will persist even if the user moves the chart object to a new folder. If the document is copied to another system, the chart object will also be copied.

Contrasting example: In Unix, a C source file refers to included files by file name. If the directory structure is rearranged, the "link" to an included file can be broken. The file name might refer to the wrong file, or more likely to no file.

Example: In C++, objects are identified using pointers or references. A pointer or reference remains valid as long as the referenced object exists.

Contrasting example: In a relational database, a tuple is accessed (in a query) by a description that specifies the values of attributes of the tuple. Different tuples may be accessed by the same description at different times, if tuple attributes are modified. (Reliable reference to a tuple can be obtained only if the tuple contains key attributes. Key attributes are immutable attributes that uniquely identify the tuples in a relation.)

Contrasting example: In the HP SoftBench environment, the target of a request message is specified by a description consisting of a tool class, a message name, and a context that identifies a particular data file. A tool activation is selected to perform the requested service by matching these values against corresponding values specified by each tool. The requester cannot assume that the same tool or tool activation will respond each time a given specification is used in a request message. In fact, the requester cannot assume that any tool activation will respond, or that at most one tool activation will respond.

Benefit: Direct object identification is normally more efficient than designating an object by its description.

New objects can be created.

A client can request the creation of new objects that are distinct from existing objects.

Benefit: Objects with time-varying behavior allow multiple clients to interact. The ability to create new objects is used to ensure that clients do not inadvertently conflict by sharing the same object. Clients are not responsible for ensuring the distinctness of newly created objects.

Example: A client program creates a new, empty dictionary object. The new dictionary is distinct from all other dictionaries, even empty ones. If the client changes the state of this dictionary object by entering a name and associated value, other dictionary objects will not be affected.

Example: A ViewPoint user creates a new document object, for example, by copying an existing document. The new document is visibly distinct from other documents as a new icon on the display. The uniqueness of the document is ensured by the system. The user is not required to assign a unique name to the document.

Operations can be generic.

A service can have different implementations (different code) for different objects, which can produce observably different behavior (although sharing some common intent). A client can uniformly issue requests for the service (the requests identify a common operation); an appropriate implementation is selected for each request. There is no limit to the number of different implementations of a given service.

An operation with multiple implementations is a *generic operation*. Clients that request generic operations may themselves be generic in the sense that they can perform a common activity on different kinds of objects.

The selection of code to perform a service (*binding*) is based on the objects identified in the request. In general, the identification of the objects occurs when the request is actually issued, so the selection of code would happen at that time (*dynamic binding*). Code selection is sometimes based on factors that are known prior to execution, so that the code can be selected during program compilation or linking (*static binding*).

Benefit: One benefit of generic operations is that they allow a system to transparently provide multiple implementations of a service. In this particular case, the implementations produce observably equivalent effects. A client that requests a service need not know that different code may execute for different objects.

Example: A heterogeneous distributed system could provide different versions of an application that execute on different hardware architectures. A client application simply issues requests, not knowing that different code will execute based on the location of the target object.

Example: A network management application that manages modems simply issues requests to modems, not knowing that different code might be run in different locations for each modem in a system.

Example: A new implementation of dictionary objects is added to the object class library under a different name, so that the old implementation remains available. The new implementation provides better performance in most cases, but the old implementation is known to give better performance under certain patterns of use. A client program using dictionary objects is changed to select the more appropriate implementation for each dictionary it creates, based on expected usage. The parts of the program that use dictionary objects are not changed; they work with both implementations.

Benefit: Another benefit of generic operations is that code can be more general, which implies that the code is more reusable. In this case, the different implementations of a generic operation may produce effects that are observably different.

Example: One can provide a single sorting module that will sort any set of objects that supply a comparison service.

Example: A graphical display module can display any object that supplies the appropriate services (such as services to draw the object or return its bounding polygon).

Benefit: In the case of user interfaces, users benefit by being able to apply a standard mental model in many cases.

Example: ViewPoint provides a small set of generic commands that can be invoked on an object of any type using dedicated function keys.

Example: Users of the HP NewWave environment need remember only that to print any kind of object, simply drop it on the printer icon.

Benefit: Generic operations facilitate the realization of open systems. An *open system* is one in which new objects can be introduced dynamically, such that the new objects can be operated upon by existing clients without modifying those clients. Existing clients are able to use the new objects because the new objects support the operations that the clients request. An open system allows new software components to be created and installed in the system while it is operational. Open systems are easier to enhance and evolve.

Objects can be classified in terms of their services.

The services associated with an object can be described in the form of *interfaces*. An interface describes how an object can be used by a client, by describing a set of potential requests that identify the object as a parameter. This specification may include information about the legitimate values of the other request parameters and the possible results associated with each request, and may also describe the behavior of the requests. Objects can be classified in terms of their services, i.e., in terms of the interfaces they support.

Benefit: A classification of objects based on their services is a way of organizing objects to make them easier to understand. A classification of object services can also be used to describe the services expected of an object by a client, and to perform compile-time checks for certain erroneous uses of objects (i.e., type errors, such as requesting an object to perform an operation it does not support).

Example: A user of the HP NewWave environment can visually determine a set of applicable operations on an object by consulting menus. Operations that are not applicable to the selected object are shaded to distinguish them from the operations that are applicable.

Contrasting example: A Unix user must remember which commands can be applied to each kind of file. If the user makes a mistake and invokes a command on the wrong kind of file, the best that can be expected is that the command will detect an improper file format and issue an error message. There is no direct way for the user to discover the set of commands appropriate to a given file.

Example: A C++ class describes the services provided by the instances of the class, by listing the member functions of each instance. It also serves as a type for the purpose of static type checking. (A C++ class can do more than describe the services provided by instances of the class; it can also define their implementation.)

Example: In the draft ISO standard for the management of open systems, a systems management object is described by a class definition listing the attributes and actions of the object. Developers use the class to write applications that access the object.

One object could provide a subset of the services provided by another object, leading to a hierarchical classification. (The term *hierarchy* is often used informally, even though many object systems support more general classification structures.) This *in-*

Associated Concepts

In our survey of object systems, several concepts appeared with sufficient frequency to suggest that they are associated with the notion of objects, although not essential to it. Perhaps in the future, as our ideas evolve, the associations will grow stronger and these concepts will also be considered part of the essence of objects.

Event notification. A client can register interest in an event or condition associated with an object, such as a state change, and be notified when that event or condition occurs.

Event-driven control structure. A program structure characterized by a top-level event loop that waits for incoming events and dispatches them to appropriate handlers.

Presentation-semantic split. A program structure characterized by the use of separate objects to model abstract information (a semantic object) and present such information to a user (a presentation object). A single semantic object can have many different associated presentation objects, as in financial data that is presented both as a pie chart and a bar chart.

Composite objects. A composite object is formed by combining or linking together a number of different objects, which can then be manipulated (e.g., moved or copied) as a unit. For example, a document object might be linked to a chart object that provides a graphic for a figure in the document. A *hot link* allows the document to be updated whenever the data underlying the chart object is changed.

Active objects. An active object can initiate computation spontaneously, without being requested to do so by a client. An active object is a concurrent process; it has its own activity, or thread of control.

Relationships. A relationship is information associated with multiple objects, and not with the objects individually. For example, the employee relationship describes an association between organizations and persons.

terface hierarchy can be used as a type hierarchy, for example, to describe the legitimate values of request parameters.

Benefit: An interface hierarchy illustrates the ability of a client to operate on multiple kinds of objects by issuing requests for generic operations.

Example: C++ defines a type hierarchy based on class derivation: an instance of a C++ class can be used (via a reference or a pointer) in any context where an instance of a (public) base class is expected. Class derivation in C++ is defined such that an instance of a class is guaranteed (syntactically, not semantically) to provide all the services defined by the base class, if not more.

Example: The Dynamic Windows user interface defines a hierarchy of presentation types, which characterize to the user the set of objects that are acceptable in a given context, for example, as an operand to a command.

Example: The HP NewWave environment defines a conceptual hierarchy of objects based on common services, such as the distinction between tools and user objects. Users benefit from understanding these distinctions, as it allows a simpler mental model.

Objects can share implementations.

Objects can share a common implementation.

The implementation of services associated with an object generally specifies both the format of the data used to represent the relevant information and the code used to perform the services. Mechanisms are generally provided to allow multiple objects to share a common implementation. Objects that share a common implementation have identical data formats and share the executable code; however, each object typically has its own copy of the data. Each object can be thought of as an *instance* of the common implementation.

Benefit: Sharing one implementation among many objects has the obvious benefits of reduced source code duplication (which eases maintenance by avoiding the need for manual propagation of changes) and reduced executable code size (where sharing of executable code is possible). Allowing multiple instances of an implementation makes the implementation more useful.

Example: In Smalltalk and C++, an object is an instance of a *class*; a class defines the data format for each instance as well as the procedures that implement the services provided by each instance.

Contrasting example: A single data abstraction can be implemented in C by a source file that defines a set of external functions with private static data. However, this definition cannot be instantiated to support multiple entities with the same behavior.

Objects can share partial implementations.

Mechanisms are often provided that allow objects with similar behavior to share parts of their implementations. For example, *implementation inheritance* supports the incremental construction of an object implementation by extending or refining other object implementations. *Delegation* is a similar mechanism that operates during

Examples

The following examples illustrate requests. A request is shown as an operation followed by parameters, along with a description of its effect. The name s identifies a stack object. The name stack-factory identifies an object that creates new stack objects.

(new stack-factory)	Returns a new, empty stack identified here by s.
(push s 3)	Modifies s (extends it with 3).
(push s 4)	Modifies s (extends it with 4).
(pop s)	Modifies s (removes 4). Returns 4.
(pop s)	Modifies s (removes 3). Returns 3.
(pop s)	Returns empty, an exception.

The following examples illustrate interfaces. An interface describes a set of requests applicable to an object. Each interface is shown as a set of request schemas, where a request schema is a pattern that describes a set of potential requests that identify a particular operation. The symbol ● in a request schema denotes a parameter position where an object satisfying the interface can legitimately appear. In the stack interface, it indicates a parameter where a stack object can appear. Other parameters are described using types, in this case, the type Integer.

The stack interface describes the services that characterize a stack object:

(push ● Integer)	No result.
(pop ●)	Returns an integer or empty, an exception.

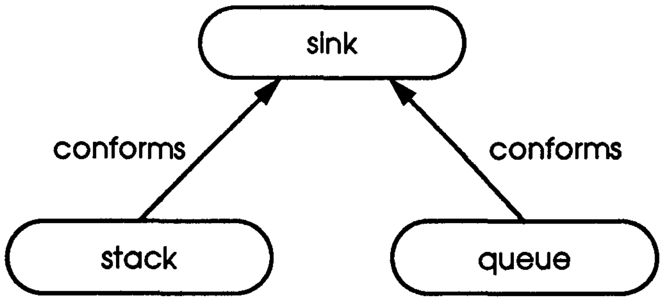
The queue interface describes the services that characterize a queue object:

(push ● Integer)	No result.
(pull ●)	Returns an integer or empty, an exception.

The sink interface describes services common to stacks and queues:

(push ● Integer)	No result.

The following figure shows an interface hierarchy based on the conformance relations among these three interfaces:



execution: an object may *delegate* a request to other objects, which then perform (appropriate parts of) the originally requested service on its behalf. (A delegated request is special because it carries information about the original request, allowing the subsequent service provider to issue further requests using the parameters of the original request.)

Benefit: In addition to the maintenance and size benefits listed above, sharing of partial implementations extends the benefits of software reuse to cases where the requirements are similar but not identical. Sharing of partial implementations is a useful technique for encouraging consistent behavior among related objects.

Example: In Smalltalk and C++, a class can be defined by inheriting from (deriving from) an existing class (the superclass or base class). The new class can extend the definition of the existing class by adding data declarations (instance variables or data members) and defining procedures that implement additional operations (methods or member functions). The new class can refine the definition of the existing class by replacing or refining individual methods or member functions. A class can be designed to support customization using inheritance: the class is customized by inheriting from it and replacing or refining specific methods. For example, a generic graphics object class that defines common behavior for graphics objects would be designed with the expectation that the definition of the draw method would be replaced in each inheriting class that defines a specific graphics object.

3 Terminology

In this section, we define terms for the essential concepts introduced above and other terms that, in our experience, cause communication problems. We relate the terms to other terms in common usage in specific areas, such as object-oriented programming, as well as terms used in specific systems, such as the C++ programming language. The terms are shown in the accompanying table, along with a selection of synonyms and other related terms in existing usage. The definitions are grouped into three sections: terms related to abstraction, terms related to requesting services, and terms related to providing services.

3.1 Terms related to abstraction

Object

Definition: An *object* is an entity that plays a visible role in providing services that can be requested by clients (people or programs). An object explicitly embodies an abstraction characterized by the behavior of certain requests. The services may access or modify data associated with objects. The services are described without dependencies on the form of the data or the algorithms used to implement the services; in particular, there could be several possible implementations of the same behavior. An identified service is called an *operation*.

Recommended terms and related terms in existing usage

object	instance, class instance, surrogate, entity
encapsulated object	information hiding
embedded object	encapsulated tool, proxy, foreign function, integrated application
protected object	access control
object reference	handle, object identifier, object name
request	message, method invocation, function invocation
generic operation	message selector, method, generic function, overloaded function, virtual member function, polymorphism, dynamic binding
interface	protocol, type, abstract class, virtual class, signature
interface hierarchy	inheritance, specification hierarchy, type hierarchy, class hierarchy, subtyping, conformance
type	class
subtype	subclass, derived class
dynamic binding	late binding
static binding	early binding
object implementation	class, type, template, manager, server
state variable	instance variable, data member, attribute, field, slot
method	member function
implementation inheritance	subclassing, derivation, prefixing

Example: A simple example of an object is a stack whose behavior is characterized by the operations push and pop. These operations are specified and used without reference to the underlying data representation, which might be an array or a linked list. The code for the operations push and pop would access the data through private operations, such as array-index or list-head, that are not part of the abstract stack behavior.

Synonyms and Related Terms: Objects are sometimes called *class instances*. The term *surrogate* is sometimes used to indicate that an object models something else.

Encapsulated Object

Definition: An *encapsulated object* is an object that can be accessed by clients only by issuing requests.

Example: In an object-oriented programming language like C++, the data structure and associated operations used to implement an object are hidden from clients of the object. For example, a stack implemented as a linked list cannot be accessed by clients using the linked list operations.

Synonyms and Related Terms: Encapsulation is also called *information hiding*. The term encapsulation is sometimes used in a stronger sense to mean that an object is *self-contained*, i.e., it shares no information with any other object and is physically represented in a single location.

Rationale: This concept is one of three common meanings of the term *encapsulation*. The purpose of this definition is to specify the preferred usage. The next two definitions provide alternate terms for the other meanings.

Embedded Object

Definition: An *embedded object* is an object that has been created by wrapping an existing structure or process (e.g., a non-object) with appropriate interface code.

Example: A terminal-oriented application is made to conform to the standards of an object-oriented user interface by wrapping it with code that maps the requests generated by the user interface into appropriate character sequences in the input stream and maps character sequences in the output stream into appropriate requests on presentation objects.

Synonyms and Related Terms: Creating embedded objects is often called *encapsulation* or *tool encapsulation*. This use of the term *encapsulation* is different from that in *encapsulated object*. Related terms include *tool integration* and *application integration*. Embedded objects in network management are called *proxies*.

Rationale: This concept is one of the three concepts to which the term *encapsulation* is commonly applied. The term *embedding* was chosen as an alternative to distinguish this concept from *encapsulated objects*.

Ambiguous Terms

In developing the terminology, we identified several terms with multiple meanings that are frequent sources of confusion and miscommunication.

The term that causes the most confusion is *encapsulation*. In existing usage, encapsulation has three meanings: the enforcement of abstraction barriers, the act of integrating foreign components into a system, and controlling access to services by different users. (The recommended terms for these concepts are *encapsulation*, *embedding*, and *protection*.)

Another confusing term is *inheritance*. In existing usage, inheritance has two primary meanings: a mechanism by which object implementations can be organized to share descriptions, and a classification of objects based on common behavior or common external interfaces. (The recommended terms for these concepts are *implementation inheritance* and *interface hierarchy*.)

Other confusing terms are *type* and *class*, whose multiple meanings refer to either the external interfaces of objects or the implementations of objects.

The distinctions between these multiple meanings are subtle, even to people familiar with the basic concepts.

Protected Object

Definition: A *protected object* is one that restricts the ability of specific clients to request its services.

Example: A mailbox object may permit a read request only for a particular person.

Synonyms and Related Terms: Protected objects are said to provide *access control.*

Rationale: This concept is often confused with the encapsulation provided by *encapsulated objects*. In fact, some systems use similar mechanisms to implement both encapsulation and protection. (One would expect protected objects to also be encapsulated.) The term *protected* was chosen based on common usage in the field of operating systems.

3.2 Terms related to requesting services

Object Reference

Definition: An *object reference* is a value that reliably identifies a specific object.

Example: In C++, a pointer value serves as an object reference. In the HP NewWave environment, a link from a document object to a chart object is an object reference. In the Unix file system, a file name is not a reference, because it may refer to different files over time if the file or a containing file directory is renamed.

Synonyms and Related Terms: An object reference is sometimes called a *handle,* an *object identifier*, or an *object name.*

Request

Definition: A *request* is an action performed by a client to cause a service to be performed. A request identifies an operation, which denotes the requested service. A request includes parameters, which may identify objects. A request does not by itself determine how the service will be performed. When a request is issued, a *binding* process determines the actual code to be executed and the data that will be accessed by the code. One outcome of performing a request may be to return results to the client; the results may include values as well as status information indicating that exceptional conditions arose in attempting to perform the requested service.

Example: In an object-oriented programming language, a request is issued by executing an invocation form specific to the particular language. In an object-oriented user interface, a request is issued by user gestures, such as clicking a mouse button while the mouse sprite is over the visual image of an object.

Synonyms and Related Terms: In Smalltalk, issuing a request is called *message sending* or *message passing*. In C++, issuing a request is called *member function invocation*. In CLOS, issuing a request is called *generic function invocation*.

Rationale: We avoid the traditional term *message* for two reasons: One is the common misconception that *message sending* implies concurrent execution by the client and the service performer. The other is the implication that a message is sent to a single location at which it is processed. As described in the sidebar on classical and generalized object models, this implication may be inaccurate for generalized object models.

Generic Operation

Definition: A *generic operation* is an operation that has different implementations for different objects, with observably different behavior, that can be uniformly requested by clients.

Example: The print operation can be requested of any printable object, e.g., a document or a spreadsheet.

Synonyms and Related Terms: A generic operation is called a *virtual member function* in C++. It is called a *message selector* in Smalltalk. The ability to support generic operations is also called *polymorphism* and *function overloading*. (See also *static binding* and *dynamic binding*.)

Rationale: The word *generic* highlights the fact that the service is common to many kinds of objects.

Interface

Definition: An *interface* is a description of a set of possible uses of an object. In particular, an interface describes a set of potential requests that identify the object as a parameter. An object *satisfies* an interface if it is *meaningful* in each potential use described by the interface. An interface may specify the required types of other parameter values and the corresponding result values, and may also describe the behavior of the requests. The *principal interface* of an object is an interface that describes *all* possible uses of the object.

Example: A stack interface might consist of the operations push and pop. It might require that only integer values be pushed on stacks. (See the examples sidebar on page 12 for more detailed examples of interfaces.)

Synonyms and Related Terms: The term *protocol* is often used in object-oriented programming to refer to a set of messages that may be sent to an object. An interface that describes parameter and result types is called a *signature*. In many object-oriented programming languages, an interface is described by an *abstract class* or *virtual class*, which is a class that omits purely implementation-oriented information (such as procedure bodies) and thus is not instantiable. An interface used as a type is called an *interface type*.

Interface Hierarchy

Definition: An *interface hierarchy* is classification of interfaces, and therefore of objects, in terms of *interface conformance*. One interface conforms to another if any object that satisfies the first interface necessarily satisfies the second interface.

Example: Both stacks and queues can be thought of as data sinks where data can be "pushed", one value at a time. The stack interface consists of the requests push and pop. The queue interface consists of the requests push and pull. The shared service is described by the sink interface, which consists of just the push request. The stack interface and the queue interface both conform to the sink interface, because stack objects and queue objects both satisfy the sink interface. A program that operates on objects assuming they satisfy the sink interface (i.e., issues only push requests on those objects) can be used on both stack objects and queue objects. (See the examples sidebar on page 12.)

Synonyms and Related Terms: Interface hierarchies are also called *specification hierarchies.* In systems where interface conformance is defined by the use of an inheritance mechanism, an interface hierarchy is sometimes called an *inheritance hierarchy*; however, we discourage using the term *inheritance* for this concept (see *implementation inheritance*, below). In systems where interfaces are used as *types*, an interface hierarchy may be called a *type hierarchy* or a *subtype hierarchy.* In systems where interfaces are not distinguished from implementations, such as C++, an interface hierarchy may be called a *class hierarchy.*

Rationale: One of the many uses of the term *inheritance* is to refer to interface hierarchies. We have chosen to restrict the word inheritance to mean an incremental definition mechanism, particularly implementation inheritance, as distinguished from a classification. The term *interface hierarchy* emphasizes the classification of objects based on their client-visible behavior (their services). Classification based on interface conformance need not produce a strict hierarchy; however, the term *hierarchy* has the advantage of being more commonly used and understood than other more technical terms, such as *lattice.*

Type

Definition: A *type* is a classification of values that characterizes their meaningful use. Specifically, a type is a predicate that defines the set of values that *satisfy* the type (called the *extension* of the type). Each object system defines its own notion of meaningful use and its own notion of type. In a typed object-oriented system, the legal values of the parameters and results of a request are characterized by types.

Example: A typed object-oriented language provides type declarations for variables. A variable can denote only values that satisfy the declared type. Object types may be defined in different ways, for example, in terms of interfaces (e.g., the type of all objects that support the print operation), or in terms of implementations (e.g., the type of all objects with the stack-as-list implementation). In many typed object-oriented programming languages, a single *class* construct is used to define both interface types and implementation types.

Synonyms and Related Terms: The term *class* is often used instead of type, especially in systems like Smalltalk where object implementations are the only kind of type.

Subtype

Definition: A *subtype* of a given type (called the *supertype*) is a type with the property that each value that satisfies the subtype necessarily satisfies the supertype. In systems where interfaces are types, interface conformance is an example of a subtype relationship.

Example: Employee is a subtype of person: each employee object is also a person object. A generic operation is often defined to apply to objects of a specific type; it is applicable to objects of all of the subtypes of the type.

Synonyms and Related Terms: In systems where implementation inheritance is the only way to create a subtype, a subtype is called a *subclass* or a *derived class.*

Dynamic Binding

Definition: Dynamic binding is the selection of code to perform a requested service made at the time the service is actually requested.

Example: In Smalltalk, each request is processed by examining the class hierarchy to locate the appropriate method for the target object. Any change to the class hierarchy, such as the addition or redefinition of a method, will immediately be effective for all subsequent requests.

Synonyms and Related Terms: Dynamic binding is also called *late binding.*

Static Binding

Definition: Static binding is the selection of code to perform a requested service made *before* the service is actually requested. Systems that support static binding can offer improved performance and early error checking, at the cost of reduced generality and extensibility.

Example: In C++, a variable that is not a pointer or a reference is known to denote an object of a specific class (and *not* an object of a class derived from that class); an invocation of a virtual member function on that variable can be compiled to a direct call on a specific procedure.

Synonyms and Related Terms: Static binding is also called *early binding.*

3.3 Terms related to providing services

Object Implementation

Definition: An *object implementation* is an executable description of how to perform the set of services associated with an object. It defines the format of the data associated with the object *and* how the services manipulate that data. Multiple objects may share (parts of) a common implementation; although the executable code is shared among the objects, each object typically has its own copy of the data.

Example: The implementation of a circle object might define the state of a circle object to consist of two state variables, center and radius, and define two methods implementing the expand and move operations:

```
expand (factor) radius := radius * factor;
move (displacement) center := center + displacement;
```

Synonyms and Related Terms: The data format given by an object implementation is sometimes called a *template*. An implementation of multiple objects is sometimes called a *manager* or a *server*. A *class* is an object implementation that can be *instantiated* to create multiple objects sharing a common implementation; the objects are called *instances* of the class. A *factory* is an object that provides a service for creating objects; a class *object* is therefore a factory.

State Variable

Definition: A *state variable* is a variable that serves as the concrete realization of data associated with objects. A state variable may be associated with a single object or with multiple objects. State variables are frequently defined in object implementations along with the methods that can read and write those state variables.

Synonyms and Related Terms: State variables also called *instance variables*, *data members*, *attributes*, *fields*, or *slots*.

Rationale: The term *state variable* emphasizes that state variables are a device for *implementing* time-varying behavior, as opposed to a client-visible feature of objects. The term *instance variable* is inappropriate for objects that are not instances of a class.

Method

Definition: A *method* is a procedure that performs services. Typically, an object has one method for each operation it supports. Methods are frequently defined in object implementations along with the state variables they can read and write.

Synonyms and Related Terms: In C++, methods are called *member function definitions*.

Implementation inheritance

Definition: Implementation inheritance (called *inheritance* here for convenience) is a mechanism for creating object implementations incrementally: one object implementation is defined in terms of other object implementations. The new implementation can extend the existing ones by adding data to the object representation (data format), adding new operations, and changing or extending the definition of existing opera-

Other Definitions of Object

Other authors have attempted to identify the fundamental concepts of object-oriented programming languages:

- Stefik and Bobrow[11] define an object to be an entity that performs computation and saves state. While admitting to great variety in object-oriented programming languages, they describe two concepts as fundamental: message sending (requests for generic operations) and specialization (implementation inheritance), and one as common: classes (instantiable object implementations). Message sending is described as supporting data abstraction, although the explicit nature of the abstraction is not mentioned, and encapsulation is mentioned only briefly. Object behavior is defined in terms of protocols (interfaces), although interface conformance is not described. Object identification is not addressed explicitly. Stefik and Bobrow also discuss method specialization and combination, composite objects, perspectives, annotations, active values, and metaclasses.
- Wegner[9] defines an object-oriented data abstraction language as one that supports objects (a set of operations sharing a local state), classes (an instantiable implementation that defines one or more interfaces), class inheritance, and information hiding (encapsulation). Omitted are the notion of generic operations, the explicit nature of the abstraction, object identification, and the distinction between interface conformance and class inheritance. (Subtyping is described in the context of a class being a type.) Wegner also discusses strong typing, concurrency, distribution, and persistence as factors that distinguish object-oriented programming languages.
- Thomas[10] defines four key concepts of object-orientation: encapsulation, message passing (requests for generic operations), class inheritance, and binding (static and dynamic). He describes objects as analogs of real-world entities that are self-describing (explicit abstraction). He describes classes (instantiable implementations) and a logical class hierarchy, which relates classes based on external behavior (just as an interface hierarchy relates interfaces). Interfaces are not mentioned. Classes are identified with types, and subtyping is described as behavior-preserving class inheritance (implementation inheritance). Object identification is not addressed. Thomas also mentions metaclasses, composite objects, and private messages (a form of access control).
- Wand[12] proposes a formal model of objects in which the concepts of services and requests are replaced by a more general concept of object interaction described by laws constraining legal object states and state transitions. In Wand's view, requests and operations are an implementation of object interactions. Wand's model also omits object creation.

Our definition of object differs from most existing definitions in the following ways: We do not require that an object have state. We do not require that every computational entity be an object. We permit state that is not associated with individual objects, or that is shared by multiple objects. We permit requests to name multiple objects, and to be performed by third parties. These generalizations are intended to capture a wider range of object system. Also, our definition emphasizes object identification, the concept of generic operation, and the distinctions between services (interfaces) and implementations and between interface hierarchy (classification) and implementation inheritance (construction). Perhaps most importantly, we attempt to use terms and descriptions that are appropriate to many areas, not just programming languages.

tions. *Single inheritance* permits an object implementation to be defined in terms of a single existing object implementation. *Multiple inheritance* allows more than one object implementation to be used in defining a new one. Inheritance often takes the form of *class inheritance*, in which one class inherits from other classes.

Inheritance gives the effect of copying and editing the textual definition of an object or class to produce a new definition, except that changes to the old definition (eventually) propagate to the new definition. It also gives a way of organizing object implementations in an *inheritance hierarchy*.

Example: A class titled-window could be defined to inherit from the class window. The titled-window class would add a definition for a title instance variable and procedures to implement the operations title (to return the title) and set-title (to change the title).

Synonyms and Related Terms: We define *inheritance* to be any mechanism for incremental definition. A class that is defined by inheritance is called a *subclass* of those classes used in defining it, which are called its *superclasses*. In C++, a subclass is called a *derived class*, and a superclass is called a *base class*. An inheritance hierarchy is also called an *implementation hierarchy* or a *class hierarchy*.

Rationale: Inheritance is a common term used to refer to many kinds of hierarchies. In object-oriented systems, it is primarily used to refer to interface hierarchies and to incremental definition mechanisms. We restrict the term inheritance to the latter concept, which is consistent with the everyday usage of the word inheritance to refer to a process, rather than a relationship.

4 Acknowledgments

We thank the members of the HP object task force for their contributions to this effort, and Walter Olthoff and Walt Hill for their assistance. We are also grateful to the employees of Hewlett-Packard and members of the Object Management Group technical committee who reviewed earlier documents and whose suggestions have hopefully resulted in improvements. We are particularly grateful to Bill Kent for his contributions.

5 References

1. D. G. Bobrow, L. G. DeMichel, R. P. Gabriel, S. E. Keene, G. Kiczales, D. A. Moon. Common Lisp Object System Specification X3J13. *SIGPLAN Notices* 23, 9 (1988).
2. I. J. Fuller, et al. An Overview of the HP NewWave Environment. *HP Journal* 40, 4 (Aug. 1989), 6-23.
3. J. Johnson, et al. The Xerox Star: A Retrospective. *Computer* 22, 9 (Sept. 1989), 11-29.
4. M. R. Cagan. The HP SoftBench Environment: An Architecture for a New Generation of Software Tools. *HP Journal* 41, 3 (June 1990), 36-47.
5. D. H. Fishman, et al. Iris: An Object-Oriented Data Base System. *ACM Transactions on Office Information Systems* 5, 1 (1987), 48-69.
6. S. McKay, W. York, and M. McMahon. A Presentation Manager Based on Application Semantics. *Proc. ACM Symposium on User Interface Software and Technology*, Nov. 1989, 141-148.
7. ISO. *Information Technology – Open Systems Interconnection – Management Information Services – Structure of Management Information – Part 1: Management Information Model.* Draft International Standard 10165-1. ISO/IEC JTC1/SC21 N5252, June 1990.
8. R. M. Soley, ed. *Object Management Architecture Guide.* Object Management Group, Inc. Framingham, Ma., October 1990.
9. P. Wegner. Learning the Language. *Byte*, March 1989, 245-253.
10. D. Thomas. What's in an Object? *Byte*, March 1989, 231-240.
11. M. Stefik & D. G. Bobrow. Object-Oriented Programming: Themes and Variations. *AI Magazine* 6, 4 (Winter 1986), 40-62.
12. Y. Wand. A Proposal for a Formal Model of Objects. In *Object-Oriented Concepts, Databases, and Applications*. W. Kim, F. H. Lochovsky, eds. ACM Press, 1989, 537-559.