M. J. Flynn, J. N. Gray, A. K. Jones, K. Lagally
H. Opderbeck, G. J. Popek, B. Randell
J. H. Saltzer, H. R. Wiedle

Operating Systems
An Advanced Course

Edited by
R. Bayer, R. M. Graham, and G. Seegmüller

Springer-Verlag
Berlin Heidelberg New York 1978
CHAPTER 2.A.

Anita K. Jones
Carnegie-Mellon University
Pittsburgh, Pa., USA

The Object Model:
A Conceptual Tool for Structuring Software
THE OBJECT MODEL:
A CONCEPTUAL TOOL FOR STRUCTURING SOFTWARE

Anita K. Jones
Department of Computer Science
Carnegie-Mellon University
Pittsburgh, PA 15213 USA

Computers are programmed to simulate complex physical and abstract systems. Design, construct, and communicate these programmed systems to others, human beings need appropriate conceptual tools. The object model is both a concept and a tool. It provides guidelines for characterizing the abstract entities in terms of which think. In particular, use of the object model can lead to clear and explicit expression of the dependency relations between these entities in a way that conducive to rendering them as programs. An example benefit is that different programmers can be assigned different parts of a design to program, and the products can be integrated with a minimum of inconsistencies. The object model provides a framework in terms of which to think about and communicate designs for programmed systems; it is implicitly and explicitly used in other papers in this volume. Thus, it is appropriate to explore the model itself.

The notion of the object model has evolved over the past decade or so. It has roots at least as far back as the Simula language design [Dahl68]. Researchers in the area of programming methodology are investigating the object model and the kind of abstractions it enables [Liskov76]. Some recently designed programming languages incorporate constructs to assist the programmer thinking in the framework of the object model [Wulf77, Liskov77]. In this paper I will not develop the arguments for and against use of the object model, nor will I explore its many nuances. I will explain the model generally, and consider some of its ramifications with respect to operating systems.

1. The Object Model

In the object model emphasis is placed on crisply characterizing the component of the physical or abstract system to be modeled by a programmed system. These components, that are thought of as being "passive", are called objects. Objects have a certain "integrity" which should not—in fact, cannot—be violated. An object can only change state, behave, be manipulated, or stand in relation to other objects in ways appropriate to that object. Stated differently, there exist invariant properties that characterize an object and its behavior. An elevator, for example, is characterized by invariant properties including: it only travels up and down inside its shaft; it cannot be moving and stopped at the same time; it can stop on only one floor at a time; its maximum capacity, measured in volume and weight cannot be exceeded. Any elevator simulation must incorporate these invariants, for they are integral to the notion of an elevator.
The object model dictates that these invariant properties are preserved by a set of operations that are the only means by which an object can be directly manipulated. To alter or even to determine the state of the object, an appropriate operation must be invoked. Thus, the set of operations for an object collectively define its behavior. In practice, the number of operations required for an object is relatively small (say, from three to twelve).

The behavior of any elevator object could be defined using three operations. The first one would be used only once to 'install' the elevator, initializing its state. For example, the Install operation would fix the relevant parameters of the building, such as the number of floors, in which the elevator exists. Once an elevator is installed, the other two operations, Up and Down, can be invoked by passengers who wish to change floors. In a programmed simulation of the elevator, only the procedures implementing these three operations would be able to alter the state of the elevator. For example, synchronization of actions necessary to preserve elevator invariants are found in the code bodies of the procedures implementing the Up and Down operations.

Because many objects essentially have the same behavioral characteristics, it is convenient to define a single set of operations, perhaps parameterized, that are equally applicable to many objects. Two objects are said to be of the same type if they share the same set of operations. The literature on programming methodology contains numerous articles exploiting the notion of type. For our purposes it is not necessary to delve into the theology that surrounds the issue of precisely what constitutes a type definition. I will rely on the reader's intuitions.

In a programmed implementation of a type, the programmed operations are collected together in what is called a type module. Some recently designed languages provide syntactic constructs designed to permit and encourage a programmer to build his program as a set of independent type modules. Alphard [Wulf77] includes the form construct; Clu [Liskov77] includes the cluster. Within a type module definition appears a description of the representation, if any, that is created when an object is instantiated, as well as the procedures that implement the operations of the type. Scope rules are defined so that only the code that is part of the type module can directly manipulate the representation of an object of that type. As a result, only the code in the type module implementation must be considered to determine the invariant properties that hold for objects of the type. Every type has a specification that expresses all that is known or can be assumed by programs outside the type module. Details of implementation, both of an object's representation and the exact algorithm followed in the implementation of an operation, are hidden behind the type module boundary. The intent is that from the type specifications a user can understand enough about the type to use it, but cannot make use of type module implementation details.

To express a new abstraction, a designer specifies a new type. New types are
defined using existing types. First, one assumes the existence of some primitive types provided by a language or a machine. Objects may be—in fact, usually are represented in terms of other component objects. Operations for the new type are implemented assuming the existence of the specified operations for manipulati component objects. To implement an entire system a programmer constructs a set type module definitions related by dependence; a second type module depends upon first, if operations in the first are assumed for the implementation of the second.

The object model is merely a structuring tool; it does not imply a particular design technique. It is amenable to use with both the "top down" and "bottom up" design techniques. Using the "top down" technique, a programmer designs the main program in terms of whatever, as yet, nonexistent, types seem convenient, to implement the types found necessary for designing the main program. This process is repeated until all types are defined, either by the programmer or as primitives. Alternatively, a designer using the "bottom up" design technique constructs types that express low level abstractions that are deemed to be useful building blocks and successively builds up to the higher level abstractions, and eventually to the entire system. In either case, at each step in the design process a programmer implements a type and ignores unnecessary detail. He focuses only on the specifications and implementation of the new type he is currently defining and on the specifications the types he is using to construct the new type.

To illustrate the object model and the corollary notion of type modules consider an example of a customer of a telephone service as seen by those who provide the service. Relevant operations that need to be performed for the customer include:

- **Lookup** — given a customer's name, determine customer's primary telephone number
- **ChangeService** — alter the current service provided to a customer, e.g., remove a phone, or install a new extension
- **Credit** — credit the customer's account by a certain amount
- **Debit** — debit the customer's account by a specified amount
- **WriteBill** — output a formatted copy of a customer's bill suitable for sending him for payment

Each customer can be represented in the computer by an object called telephone-service-customer. Each customer is characterized by a name and address, the kind of telephone service presently provided, as well as billing and service information. There are various groups of people that cooperate to provide telephone service; each group has a need to reference telephone-service-customer objects. The telephone operator needs to lookup telephone numbers upon request. Likewise, service representatives of the company should be able to assign new numbers otherwise alter (ChangeService) the current service that is provided to a custom Business office employees need to be able to print bills, inspect billing and serv data, and to credit and debit a user's account (WriteBill, Debit, Credit). Each
the above sees the customer from a different perspective and has available operations which support that perspective.

As part of the type definition, a customer might be represented by a record containing at least the following component objects (of types not specified here):

name
address
current service (an array, one entry for each installed phone number)
assigned phone number
location of phone
number of extensions
color/type of phone
billing data (an array, one entry for each installed phone number)
rate schedule
local call charges
itemized long distance charges
credit or debit carried from previous month
billing address

As stated earlier, the representation of the telephone-service-customer object is not available for manipulation except by code that implements the telephone-service-customer type module, in particular the operations sketched above. Thus, details of implementation, such as record formats, are not available outside the type module.

2. The Object Model Applied to Operating Systems

An operating system provides a variety of services—address space management, i/o support, and process management including synchronization and interprocess communication. Following the object model, an operating system can be described as a set of types, each of which can be thought of as a kind of resource. Some resources have a direct physical realization, such as i/o devices. Others are further removed from the hardware, such as processes, semaphores, mailboxes (for communication of messages between processes), and files. Each resource is an object. As an example, intuitively described operations for two types, processes and mailboxes, are listed below:

**Process operations:**

- Create -- create a new process capable of executing code
- Destroy -- destroy an existing process
- Fork -- create a new process to execute in parallel with the invoker of the Fork operation
- Join -- two processes are joined together so that one is destroyed and the other continues execution
- Schedule -- cause a process to compete for CPU resources
- Unschedule -- remove a process from competition for CPU resources

**Mailbox operations:**

- Create -- create empty mailbox
- Destroy -- destroy an existing mailbox
- Send -- place a particular message into a specified mailbox
- Receive -- take a message from a specific mailbox, waiting until a message appears, if necessary
- ConditionalReceive -- take a message from a specific mailbox, but do not wait if the mailbox is empty
Any dependency between operating system types may or may not be of interest to the user. For example, implementation of the mailbox type relies on the existence of a message type. A user must be able to create and initialize a message, otherwise mailboxes will not be very useful. In addition the mailbox type module relies upon some synchronization module. Note that the user need not be concerned with whether or not code implementing a particular type executes in privileged mode. The object model paradigm is a basis for designing an operating system in which different facilities are provided in different domains of privilege. In particular, the code of each type module has the privilege to manipulate the components of an object of the type; no other code executes with that privilege. Note also that when both operating systems and application programs can be designed using the object model the boundary between two modules is the same; no artificial boundary separates the operating system and the application.

Next I consider three operating system features—naming, protection, and synchronization—from the perspective of the object model. My objective is to discuss how these features common to operating systems manifest themselves as views in the object model framework. First, consider naming. When thinking or programming in the terms of the object model, it is appropriate to be able to name objects. In contrast, most extant systems provide to users a space of names for one or more memory segments. From the perspective of the object model such systems provide for naming of only a single type of object—segments. Let us consider the ramifications of segment naming.

In systems that restrict the space of names to segments only, other types of objects are usually named in one of the two following ways. In one case, the name of an object is the address of the segment used for representing the object. For example, in many systems a process context block is maintained by the operating system to represent a process object. The address of the first word in that block is used as the name for the process. This necessitates either validating an address each time it is used (e.g., presented to the operating system as a parameter), or, possible, restricting its use to trusted programs. It is very sad to read code in which a check of whether an address is on a double word boundary is made, as a futile attempt to determine whether a parameter is a process name or not. Such a naming scheme is inadequate.

The second technique for naming an object is to introduce a new name interpreter for each new type of object. For example, process objects may be named, say using integers that are interpreted as indices into a table of process representation. Only code in the process type module can access this table, so the interpreter process names is part of the process type module. But this means only that the process type module maps integers to processes, not that it can determine from a submitted parameter whether that integer name designates a legitimate process, that the caller should be able to access it in any way. This second naming scheme also inadequate.
It would seem that a facility to name objects—not just segments—is desirable. Such a facility would make programming more convenient, and would free the programmer from the burden of mentally translating from the object to the details of that object's representation. It is unclear how such a facility should be implemented. Such naming of objects can be supported dynamically by the operating system or, applications programmers can be constrained to write programs only in languages that provide object naming syntax and a compiler to map objects to their representations.

Closely related to naming is protection, a facility provided by an operating system to constrain the way information is used and changed. Because logically separate pieces of information are encoded in different objects, it is appropriate to provide protection for each object individually. Manipulation of an object is requested by specifying an operation to be performed on that object. A straightforward technique for constraining arbitrary manipulation of an object is to constrain the ability to perform operations on that object. Rights to perform certain operations defined for an object are distributed only to those who should be able to manipulate the object. A protection mechanism permits an operation to be successfully invoked only if the invoker possesses the right to do so. Controlling the use of an object based on the operations defined for it is desirable. Certainly, it is more meaningful to users than protecting on the basis of read/write access to the memory cells used for representing objects. Such protection mechanisms enable fine distinctions between the manipulations allowed to various users. In the telephone-service-customer example the operator can be granted only the right to Lookup telephone numbers, while the telephone service office can be granted the right to perform both the Lookup and ChangeService operations, yet not be permitted to perform the billing operations. Thus, the service office can cause the customer object to be altered, but only in constrained ways related to the responsibilities of the service office.

Our conclusion is that both naming and protection can profitably be provided on the basis of objects. In an operating system in which both naming and protection are provided for all objects—not just segments, there exist implementations in which protection and naming are integrated. For now, an implementation will be sketched. It will be investigated in more detail in the paper on protection. Let the set of objects that are accessible during the execution of a program, in particular, an operation, be called the \textit{domain}. A domain can be expressed as a set of \textit{descriptors}, sometimes called \textit{capabilities} [Dennis65]. Each descriptor is an unforgeable token that identifies a particular object. The name of an object is a local name, say an integer offset into a list of descriptors. The system name interpreter locates the unique object specified by information in the descriptor. Using this naming mechanism, code is restricted to use of only those objects in its domain. For the naming mechanism and the protection mechanism to be well defined, the alteration of domains, i.e., the acquisition and dispersion of descriptors via execution in a domain must be controlled in a disciplined manner.
This naming mechanism can be extended to support protection if a domain is redefined to be not just a set of objects, but a set of rights to objects. We extend the descriptor to encode rights to an object in addition to the information needed to find a unique object. An operation can be successfully performed in a domain only if the right to do so is in that domain.

There are a number of extant systems which support the naming of object types besides memory segments [Burroughs51, Lampson76, Wulf74, Needham77]. It remains a research issue to determine how to provide object naming and protection cost-effectively. If the operating system supports generalized object naming, interesting issues is what hardware support, if any, should be provided. Indeed, inexpensive can object naming be made? Another alternative is to provide a lang system as the "front end" to the operating system and have the language support object naming. As exemplified by the Burroughs 5000 system, the compiler and a minimal run-time system would support the mapping between a virtual object and (virtual) memory used to represent it. The supporting operating system need not provide more modest naming and protection mechanisms. A disadvantage of this is that support for individual naming and protection of objects for debugging and runtime reliability checks.

Synchronization is yet another facility which is affected by adoption of an object model paradigm. According to the object model, each different manipulation of an object is performed by a different operation. It is frequently the case that synchronization is naturally expressed at the level of operations, i.e., that certain sequences of operations are allowed. For example, one invariant property: a mailbox is that the number of messages removed cannot exceed the number of messages sent to that mailbox. This can be expressed by saying that the Send operation cannot be performed more times than the Receive operation.

Habermann [77] has developed a notation called path expressions to express permissible operation sequences. The advantage of expressing synchronization restrictions as relations among operations is that synchronization constraints can be meaningfully stated as part of the specification of the type module. Thus, synchronization constraints are expressed to the user in natural terms--i.e., in terms of permissible operation sequences--of the object. One can view a path expression as a declarative statement of how synchronization constraints are to be observed. The code actually realizing the synchronization may not even be written by the author of the type module, but may be provided statically by the language system or dynamically by the operating system.

In this section I have tried to argue that three of the features that an operating system provides all have a natural expression, given the object model paradigm. In particular, each one can be phrased in terms of the objects and operations that are meaningful to the user. It is my opinion that some model, perhaps the object model is the correct one, is needed to raise operating system designers and implementors above the level of that common denominator, the new word, and all the extraneous, debilitating detail it forces us to think about.
3. Mechanics of Supporting Type Modules

Consider the invocation of operations defined as part of a type. The operations are implemented as procedures in hardware, firmware, and more often, in software. Provisions must be made to invoke these procedures in a well defined manner, and to provide a domain containing the objects that are to be accessible for the duration of the procedure's execution. To support the notion of a type module there must exist an invocation mechanism that, at a minimum, locates the procedure that implements the desired operation, acquires or constructs a domain to make available to the procedure those objects required for its correct execution, and causes execution of the procedure to begin at the procedure entry point.

Because objects are specified as parameters to operation invocations, a question arises: does the ability to perform operations on an object change as a result of its being passed as a parameter? If one program passes an object as a parameter to a slave program that is to perform a task that the caller could conceivably perform, the second program should not have any rights to manipulate the parameter object that the caller program does not have. In fact, the second program may have less.

In contrast, if an object is passed as a parameter to an operation defined as part of the object's type, the code implementing that operation will require the ability to manipulate the object's representation. Thus, some means for amplification, i.e., for obtaining additional rights to manipulate an object is required [Jones75]. Most extant hardware provides only an extremely primitive amplification mechanism. When a user program invokes an operation that happens to be provided by a module of the operating system, the hardware state changes so that when the operating system code is entered, it has access to all of main memory. In particular, it has all necessary access to the representation of the parameter object, but it also has much, much more.

Such a mechanism does not support the object model very well. It places an undue burden on the implementor of the operating system, because that programmer has no means to restrict the objects, or memory, that are accessible to his code, making debugging more difficult. Such mechanisms inadequately support the concept of software reliability. More selective amplification mechanisms can be designed. The Multics hardware permits domains of execution to be ordered so that segments, the nameable objects in the Multics system, that are available to one domain are available both to it and to domains lower in the ordering [Organick72]. The Multics hardware can be augmented so that each domain can be treated independently eliminating the ordering constraint [Schroeder72]. Other systems, such as Hydra, that lack hardware to perform amplification provide such support in software [Wulf74]. Programming languages that support the concept of abstract data types provide such amplification mechanisms [Jones76].

So, to support the object model requires support for the notion of a domain. Ideally, domains are small; only the rights and objects necessary to perform the task
at hand are available. Domain support must include a facility for suspend execution in one domain in order to enter another, and subsequently to return to first. Some provision for amplification is required. Domain management needs to be efficient for domain entry and exit and occur often. Current operating system research and some programming language research is addressing these issues.

4. Observation

The fidelity with which a particular system adheres to the object model varies widely. Some operating systems, such as Multics, define a single type of object, a segment, and permit users to create segments at will. Other systems, such as Hydra, permit users to dynamically create new object types, as well as new objects. Hydra, in particular, provides naming and protection of objects of user-defined types, as well as operating system types, as was sketched above. However, even in cases where the operating system design does not closely adhere to the object model, the model often provides a convenient vehicle for describing system components. Consequently, in the other papers in this volume, authors have used the notion with greater or lesser fidelity, as suited their needs and their taste.

5. References


Dennis, J. B. and Van Horn, E. C., Programming Semantics for Multiprogram Computations, CACM 9, 3 (March 1966) 143-155.


Lampson, B. W. and H. Sturgis, Reflections on an Operating System Design. CACM 19, (May 76), 251-266.


