On the Subject of Objects

Peter J. Denning

DRAFT v3 5/22/06

The notion of an abstract data type is quite simple. It is a set of objects and the operations on those objects. The specification of those operations defines an interface between the abstract data type and the rest of the program. The interface defines the behavior of the operations -- what they do but not how they do it.

-- John V. Guttag
Inventor of the term
"abstract data type"

Most mainstream programming languages today support the concept of "objects". Many programmers swear by "object oriented programming" (OOP). Software designers say that "object oriented design" (OOD) is the most resilient way to construct large software systems. The Educational Testing Service has changed the Computer Science Advanced Placement curriculum in high schools, focusing on "object orientation" with Java as its language. Since the mid 1990s most computer science departments have made object orientation the centerpiece of their first courses in computing, first with C++ as the language of instruction and later with Java.

John Guttag, who coined the phrase "abstract data type" in 1975, says that the idea of objects is easy. Despite the simplicity of the idea, I have found that many students struggle with objects. The reasons vary, but include (1) objects rely on a complex web of foundational concepts, (2) the language syntax is complex, and (3) the libraries are complex. I also found that when I explained the history of the object principles, my students were able to make sense of this subject and to appreciate the reasons behind the concepts. This short article records the story I tell them. Others may find it useful.

This then is a short tutorial on the history of object principles, aimed to expose the component concepts and the reasons for them. It is not a survey of the current state of the art. I have included only as many citations as needed to identify the historical sources of ideas.

Object oriented design and programming are means to deal with complex systems; the simple examples students encounter during their introductions to the subject leave them mystified why such a heavy aircraft is needed for puddle jumping. One student told me he felt like novice pilot who came to class for a Cessna license and found a flight instructor who wanted only to talk about 747s.

At first glance, it looks like "object" is a different name for "module". It is certainly true that objects support modularity. So why not call all this modular programming and modular design, in the best traditions of engineering? Is there something new that justifies a new word?

The answer is yes. The principle of objects is different from the principle of modules. In fact, the principle of objects grew out of attempts to overcome some difficult problems with modules. These notes explain what happened and how we settled on objects as a powerful way to organize systems.

Abstraction and Information Hiding

Two fundamental principles of software design are abstraction and information hiding. Abstraction means a simplified representation of a phenomenon, retaining only essential features. Abstraction is a powerful tool for dealing with complexity. Here is an example:

> The file, an abstraction in operating systems, is a sequence of bytes with a name. Users are allowed to do only two things with a file: read and write. Read means to copy the file's bytes into the user's workspace. Write means to replace the contents of the file with a new sequence of bytes from the user's workspace.

The actual implementation of files requires a complex subsystem including a disk storage unit, a driver for the disk, an interrupt routine for disk completion signals, division of files into fixed size records, index tables or trees that lead to the disk tracks containing each file's records, buffers to hold portions of files as they are transferred between disk and main memory, pointers to track the current read positions in opened files, and identifier numbers that uniquely distinguish files from each other. With abstraction, the operating system programmer builds this subsystem once and each user interacts with the file system as if the files were simple objects that can be read or written as described above. Users of a file do not have to follow links through the index table, arrange buffers, start the disk, wait for interrupt signals, manage the buffers, etc.

Information hiding means to limit the knowledge of, and access to, details of an implementation. In the file system example, none of the details about disks, drivers, index tables, buffers, pointers, etc. is visible to the user of files. Information hiding is thus a way to enforce the abstraction. If any of the details is subsequently changed without changing the abstraction itself -- for example, changing to a new, more efficient record size on the disk -- then the modification can be installed without affecting any users.

One of the most common ways to think about this is with modules and interfaces. All the programming code of the file system is partitioned among a set of modules. Users can access files by making calls on the simple interface to these modules. Any internal change to a module is allowed as long as it does not change the way the interface works.

Many people use the term "encapsulation" to refer to an implementation of an abstract object that enforces information hiding by allowing access only through a well-defined interface and by placing all internal functions and data

structures in a memory region inaccessible to other programs. Thus all the code, disk layouts, tables, and internal code of a file system are completely inaccessible to anyone outside the interface.

The terms decomposition and abstraction used above go hand in hand but do not mean the same thing. Decomposition means to break a complex operation into simple components, which are implemented separately and then combined to make the more complex operation. Abstraction means to define a simple conceptual superstructure for a complex operation. Decomposition has a "top down" flavor -- continuing to subdivide operations until very simple ones are reached; abstraction has "bottom up" flavor -- continuing to define more general and simpler operations. Many system designers think both top down and bottom up at the same time: they define careful and precise abstractions for each module so that they can be sure the combinations of the modules produce a system with the desired behavior.

While abstraction and information hiding are often used together synergistically, they are distinct principles. An example of abstraction without information hiding is a file system whose internal source code is public and all internal subroutines are callable by anyone who learns their names. Suppose then that a sophisticated file system user reads the source code and discovers that he can make all reads and writes of his files twice as fast by storing them in a particular way and making his own call on the internal subroutine that starts the disk. A file system patch that changes either of these details would invalidate that user's way of using the file system; moreover, it would likely cause the file system to crash whenever that user tries to read a file.

An example of information hiding without abstraction is a file system that makes all its internal modules accessible to the users. Although the internals of the modules would be hidden, users could substitute their own versions of disk drivers, record sizes, buffer control schemes, etc., as long as they don't change the interfaces. There is no abstraction, just a maze of details. And in practice no real guarantee that the "file system" -- which can consist of anyone's modules -- has been thoroughly tested and is reliabile.

Even if designers are careful to honor both principles, they still have a long way to go before coming to a sound design. Let me tell a story illustrating why. In the middle 1960s I took a class on system programming. The instructor proposed that we build a file system. He led a discussion in which we agreed on the abstractions that the file system would present to its users, and then on the division of the system into a set of modules with precisely defined interfaces. He divided us into teams, one for each module. He told us we would next meet in the middle of term, link our modules together, and -- voila! -- there would be the file system. We thought this was great because we would get the second half of the semester off after earning our A's. He made us promise that we would give top priority to thoroughly testing our modules and being 100% certain they would work when we linked them together. Cleverness, elegance, and efficiency were secondary for this project. We spent considerable time agreeing on the interfaces; then off we went as individual teams building modules.

We came together on that midterm day in the computer laboratory and loaded our modules into the computer. The instructor linked them. Then he went around the table asking the team leaders to certify again that their teams were 100% certain that their modules met the specifications. We did. He thanked us all and started the system. The first time someone tried to read a file, it crashed. In fact, every operation we tried crashed the system. It didn't even work a little bit. There was considerable finger-pointing and wringing of hands as we saw our easy A's going out the window. The instructor then said this is why he had us meet in the middle of the term. He said that the modules-interface design approach hardly ever works, but no one believes that until they try it. He said we would need the second half of the term to figure out why it wasn't working and fix it. Over the next two months we uncovered many subtle bugs. Some bugs resulted from different teams' slightly different interpretations of the specifications. Others resulted from unforeseen interactions among the modules. For example, two teams used what they thought was an unused region of memory for temporary storage; their modules overwrote each other's critical temporary values. In another example, three modules got into an infinite loop calling each other. In those days we programmed in assembly language and did not have the benefit of modern languages with block structure and type checking to prevent such errors.

The question of how to decompose a complex system into manageable modules and then get them to work properly has plagued designers since the beginning of computer science. The principle of objects is one of the long-term results of that struggle. It is not perfect, but it is a lot better than the modules-interface approach we tried in the 1960s. In the remainder of these notes we will explore some of the main problems that system designers encountered and how they evolved the concepts of objects to overcome those problems.

In the beginning

In the 1950s the new craft of programming appeared and good programmers discovered certain practices that helped them reduce complexity and errors. They realized very early that some form of reusable code module would be very useful. For example, it was inconvenient and error prone to manually copy the "sine" code to every place in a program that needed to compute the sine of an angle. One solution was the macro, a way to define a block of instructions and have the assembler automatically copy the block to each site using it.

A more elegant solution was the subroutine (also called procedure). The reusable block of code was placed in a region of memory and the processor used call/return instructions to transfer control to that block when needed and return control when done. One of the first machines to incorporate call/return instructions was the Atlas at University of Manchester around 1950. The programmer simply stated the subroutine name and its parameters at any point where a value from the subroutine was needed; the assembler would convert that to a call on the subroutine. This meant that in effect the subroutine name became a new instruction -- a programmer-crafted abstraction. The subroutine

call looked like a single step at the point of call, even though the implementation of the subroutine might involve many instructions and loops.

At the same time, various domain experts crafted libraries of standard mathematical functions, implemented as subroutines. A new program called linking loader collected all the library subroutines needed by one's main program and linked them together into an executable module (McCarthy 1963). With the linking loader, the new "abstract instructions" created by subroutines became available to all users of the system.

The important fact is that the first generation of computer architects, programmers, and programming language designers realized that "abstract instructions" created by subroutines would be fundamental to reducing code complexity and errors. They provided for subroutines in the design of computers.

Data abstraction

By the late 1950s software designers considered it good practice to create a *data layout* before specifying the program code (Yourdon 1979). A data layout was the pattern by which the data were arranged in main and secondary memory.

The data layout had an enormous influence on the speed of algorithms. For example, sorted data allows binary search (order log n) whereas random data requires the much slower linear search (order n). Good programming thought through data layout before working on the details of algorithms.

By the middle 1960s, many programmers described their programs in terms of "packages" containing the data structure and a set of subroutines that performed operations on it. They used the term "data abstraction" for the practice of organizing programs this way. Figure 1 is a conceptual picture.

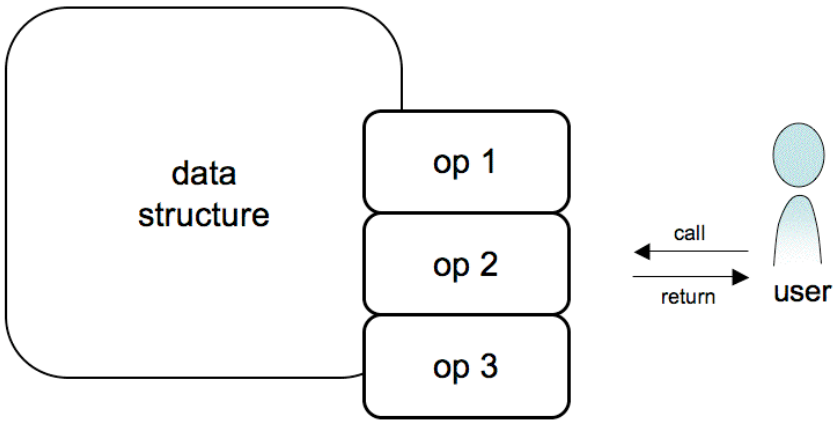

Figure 1. The practice of data abstraction organized a program module as a package consisting of the data structure and a series of operations. The user called operations as needed to read or modify the data state, and agreed not to access the data directly. Many modern systems refer to the operations as "methods" for reading or modifying the data structure.

The "bounded buffer" is a simple example of a program that can be organized this way. The abstract object is a buffer that holds up to `N` items in a first-in-first-out (FIFO) list. The operation `insert(x)` adds item `x` to the end of the list, except if the buffer is full when it has no effect; and `x=remove()` returns (and deletes) the item at the head of the list, except if the buffer is empty when it has no effect. (There are some synchronization issues of preventing the user from trying to remove from an empty buffer or insert into a full one. We will not discuss them here.)

How might the buffer itself be implemented? One possibility is to use a linear array with in- and out- pointers that cycle through the array.[1] Another is to use a linked list data structure.[2] The programmer of `insert` and `remove` organizes the code to work with the buffer data structure. Which of these two (or another) buffer data structures is actually used is invisible to the user. The user interacts with the abstract version of the buffer only with the insert and remove operations.

When programming language designers in the middle 1960s began to talk about adding "package syntax" to programming languages, the first reaction of experienced programmers was, "We already do it this way. What's the point? The added structure of the language will only slow us down and will add nothing." Of course, the language designers prevailed because they wanted to accommodate many new programmers, who weren't as experienced and who could begin using the package structure right away. They turned the package practice into a programming tool enforced by the language's syntax.

Data Abstraction with classes

The package idea was very powerful. Operating system designers used it for important subsystems like the file system. But they had to deal with a new complication. These subsystems did not manage a single object, but many objects all of the same kind. Figure 2 illustrates how the package idea could be extended for classes of objects of like kind.

---

[1] Define an array `B[0..N-1]` and two pointers `in` and `out`, initially `0`. Operation `insert` places `x` in `B[in]` and advances the pointer `in = in+1 mod N`. Operation `remove` returns `B[out]` and advances the pointer `out = out+1 mod N`.

[2] A list element `E` contains two parts, the value (`E.v`) and the link (`E.l`). Initially the `N` elements are chained together on a free list, with their links connecting one to the next. The free list descriptor `FREE` contains two parts, the head element (`FREE.h`) and the tail element (`FREE.t`). The buffer descriptor `BUFF` also has two parts, the head (`BUFF.h`) and the tail (`BUFF.t`) of the FIFO list. The buffer contents are the series of elements linked from `BUFF.h` to `BUFF.t`. The `insert` operation unlinks the first free element (`FREE.h`), puts `x` in its value field, and links it to the buffer tail (`BUFF.t`). The `remove` operation returns the value from the buffer head (`BUFF.h`) and relinks the element to the free-list tail (`FREE.t`).

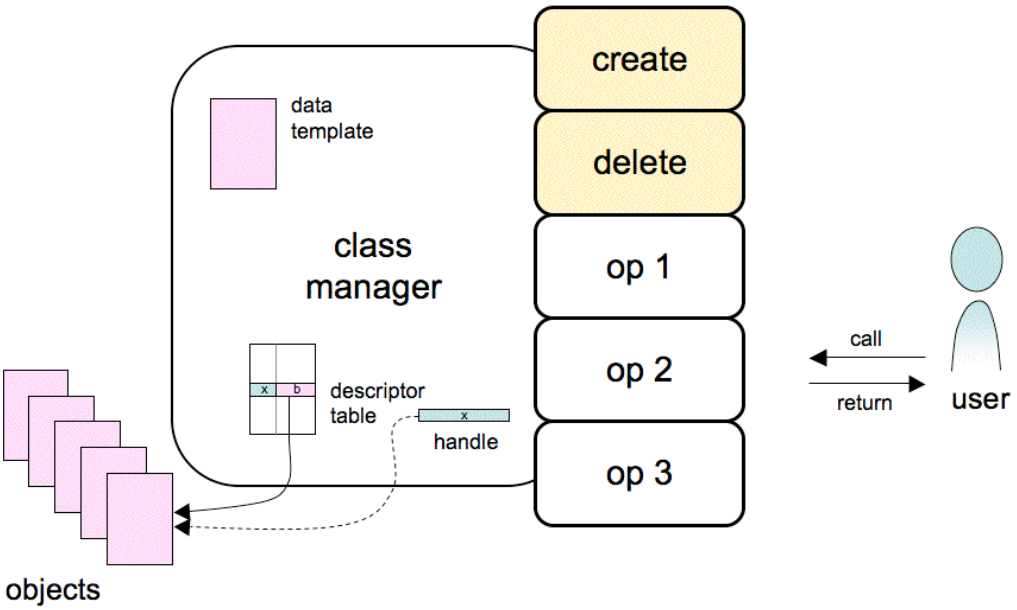

Figure 2. When data abstraction is extended to classes of objects of the same kind (e.g., files), the class manager contains only a template of the data layout of an object. The create operation allocates storage for a new object, following the template, and returns a pointer to it (the handle). The handle contains a unique identifier (x) for the object. The delete operation removes an object designated by a handle and releases its storage. Once an object is created, the user can apply an operation to it by passing its handle to the operation's subroutine. The class manager maintains an internal descriptor table mapping handles to physical storage addresses (b).

The data structure of Figure 1 is replaced by a template for the data, and two new operations are added: create and delete. In a file system, the template says that a file is of the form

(header, series of bytes, EOF),

where EOF is an end-of-file marker. The create operation allocates a new file on the disk with format following the template, and returns a handle containing the disk address. The delete operation removes the file designated by a handle and releases the disk storage it occupied. A read operation might take the form

```
(x,n) = read(h),
```

meaning to copy the `n` bytes of the file with handle `h` into the workspace beginning at location `x`. A write operation might take the form

```
write(h,x,n),
```

meaning to copy the series of bytes from location `x` to `x+n-1` in the workspace to the disk, replacing the old version of the file with handle `h`.

A handle is a string that class manager interprets as a unique identifier for an object. The class manager maintains a hidden, internal table that maps handles

to the physical memory regions containing objects. The class manager can therefore relocate the objects in storage without having to notify the user.

As you can see, the structure is now starting to get complicated and there is clear benefit to designing programming languages that manage all the complications for the user. The first such language was Simula (1967) designed by Ole-Johan Dahl and Kristen Nygaard. Simula was designed for discrete event simulation. For example, a simulation of traffic flow in a city might have classes for cars, streets, and traffic lights. Simula was the first language to use classes of objects. (As we will see shortly, however, it was not the beginning of object oriented programming.)

As described, the addition of classes creates an important problem for the user: keeping track of handles. Figure 3 illustrates. The user must assign new variables in his workspace for each object created. Errors in the user's program can inadvertently change handle values or destroy them. Thereafter objects may be lost or have the wrong operations applied to them. Even worse, when the user's program completes, quits, or crashes, all the handles are lost.[3] This would be a disaster with a file system, where files are intended to survive user programs and remain in existence until the user deletes them explicitly.

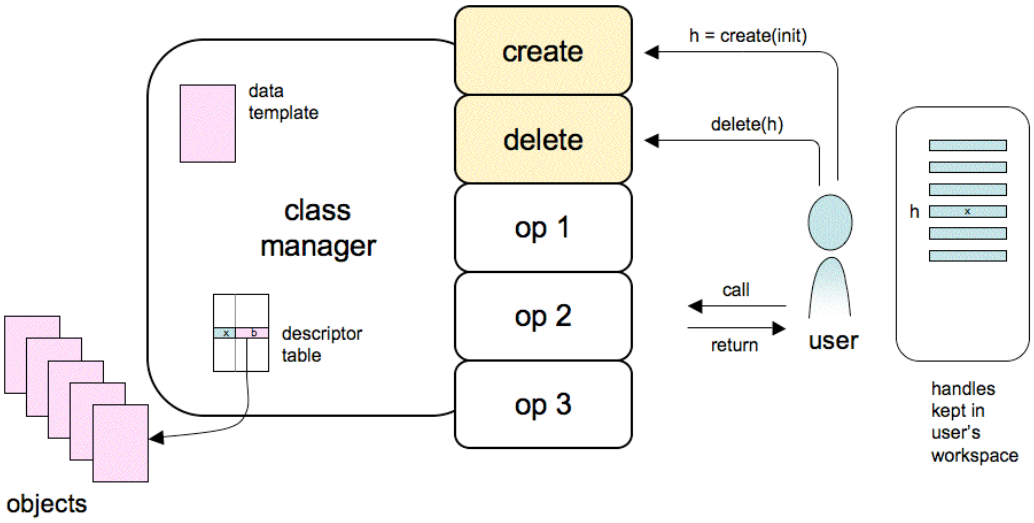

Figure 3. Handles create numerous problems if the user is responsible to store and manage them in the user's workspace. Program errors can modify or destroy handles. Program termination can lose all objects, which will no longer exist the next time the user starts the program. An object can be "orphaned" if no user has a handle.

[3] Actually, because the class manager's internal handle table survives user program terminations, it could be used to reconstruct a particular program's handles. If no user has a handle for an object, the object is "orphaned" and can be deleted; however, there is no way for the class manager to know which objects are orphans.

The solution to this problem is to associate an "object list" with each user, maintained by the operating system. New handles are stored in the list and the user refers to them by index numbers. Thus `h = create()` in a file system would mean to create an empty file and place its handle at (empty) position `h` in the object list. A file read operation `read(h)` or write operation `write(h,x,n)` applies to the file whose handle is a position `h` in the object list. A `delete(h)` operation frees the storage held by file `h` and erases the entry `h` from the object list. Figure 4 illustrates the principle.

With this arrangement, the user's object list survives program terminations. Objects will continue to exist until the user explicitly deletes them.

This arrangement also has important benefits for error control. A user cannot refer to any object not mentioned in the object list; therefore errors are confined to the user's workspace. A user cannot change any handles; therefore errors cannot be propagated to other objects.

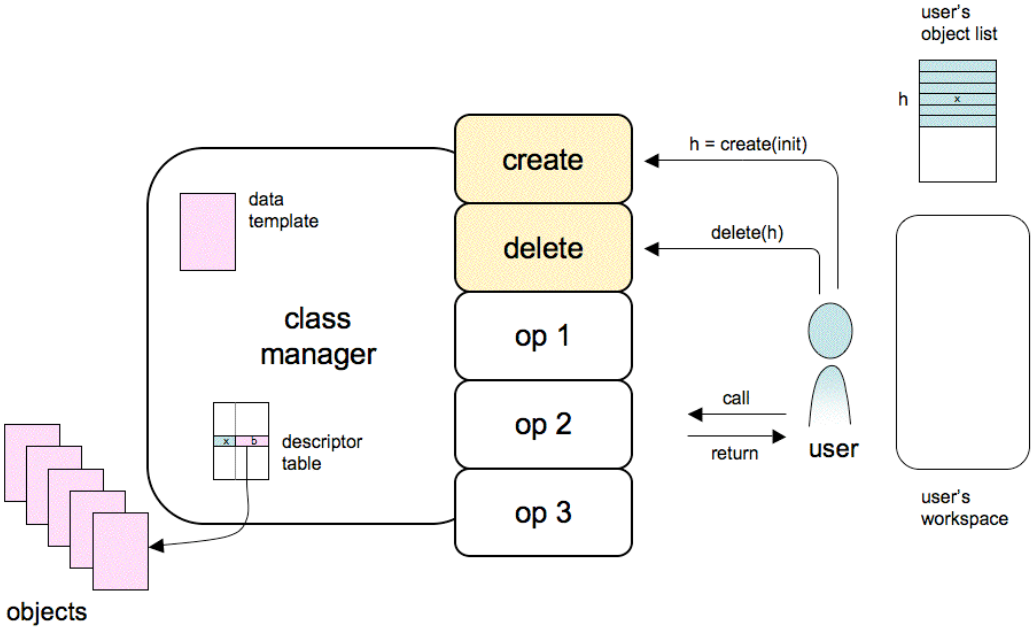

Figure 4. Storing all handles in a user's private, tamper-proof object list, which survives program terminations and session logouts, solves the problem of handle management. The separate object list also protects against program errors that mistakenly modify or delete handles. Because the object list is outside the workspace in a different region of the storage system, internal program errors cannot affect it.

Class Hierarchies

Many designers found it useful to create hierarchies of classes and subclasses. In other words, the data template could be a tree describing a set related templates.

In a file system, for example, the highest level abstraction is that a file is a sequence of bytes with a name. This declaration might also state the block size used to implement files on the disk. Several kinds of specialized files can be defined:

- A directory is a file with specially formatted entries that associate a user-chosen name string with a file handle.
- An index is file containing of specially formatted entries that enumerate the disk tracks on which a file is stored.
- An access control list is a specially formatted file that lists users allowed to read or write a given file.

We can simplify the descriptions of directories, indexes, and access control lists by declaring them to be subtypes of "file". In other words, they have all the attributes of file plus specific attributes of their own. A programmer can change global properties of files (e.g., the size of blocks used to implement a file) at the top level, and the changes automatically propagate into all the specialized types of files including directories, indexes, and access control lists.

The idea that a specialization of an object gets some of its properties from its "parent" object is called inheritance. Early versions of this idea appeared in 1958 in the Algol programming language block structure. In Algol a procedure that includes definitions for other procedures is called the "parent" of the others; the child procedures are allowed to refer to private variables of their parents. The Burroughs B5700 computer system offered an elegant method of implementing block structures with inheritance of parent variables (Organick 1972).

An early example of class hierarchies associated directly with data types appeared in the Hydra operating system (1974). Important subsystems such as file system, records system (database), and directory system were defined as classes of objects. Each class had a class manager that created and deleted objects, and attached to objects the procedures implementing the class's operations. A user could invoke any operation attached to an object or its parents in the hierarchy. Hydra used some amusingly self-referential terminology: the generic description of class managers was called the "class 'class'".

In modern object oriented languages, class hierarchies can be defined around a number of relationships. The examples above illustrate the relation "inherits", meaning that a subclass gets some attributes from its parent class. Other relationships include "extends", "is an abstraction of", and "interface definition".

Data Abstraction with multiple classes

Users almost always deal with multiple classes of objects. Figure 5 illustrates with a user who has access to both a records manager (database) and file manager system. There is now a need to tag each handle with a mark for the type of object it points to. The subroutines of a class manager automatically check incoming handles to make sure they are of the right type.

In Figure 5, we have added open and close commands to the file system. Open is used to establish an efficient communication path to the file, and close to terminate it. An open-file handle points to a control block that manages the transfers of file segments between disk and main memory and caches them to eliminate extra disk accesses. Read and write work only on open files. This illustrates how even a single class manager can deal with more than one handle type.

Data Abstraction with multiple users

Object sharing and reuse is an important objective in most systems. Two users can share an object by having the same handle in their object lists. (The copies of the handle need not be stored at the same position in the two object lists.) To prevent the users from conflicting in their use of a shared object, the class manager needs some sort of locking mechanism with a queue of users who are waiting for access.

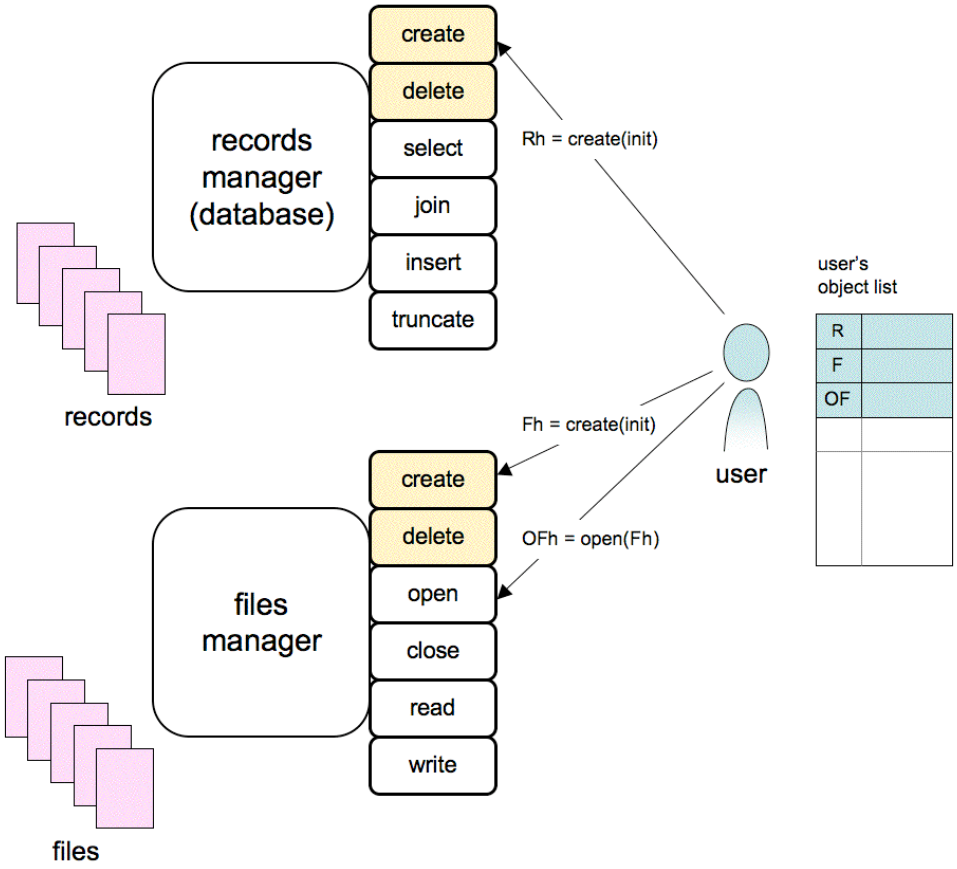

Figure 5. This example shows a user who can interact with both a database system and a file system. To prevent confusion, handles are tagged to indicate the type of objects they point to. The first handle points to a database record (tag R), the second to a file (tag F), and the third to an open file (tag OF). The record manager's create command returns an R-tagged handle, the file system's create command an F-

tagged handle, and the file system's open command an OF-tagged handle. The class manager's subroutines check that the handle supplied by the user is of the correct type. Thus the records manager subroutines all require R-handles, the file system subroutines delete and open require F-handles, and the file system subroutines close, read, and write require OF-handles.

One approach to the locking is to lock up the entire class manager (Figure 6). Race conditions between two users working on the same object cannot happen since only one user at a time is able to perform any operation on the object. This is the essence of the locking mechanism called monitors, proposed by C A R Hoare in 1974. One of Hoare's motivations for a language structure was that the details of locking a class were quite tricky; an inexperienced user could get them wrong and wind up allowing two users to update the same object at the same time. By providing an abstraction for the monitor, Hoare delegated to the compiler the work of laying out the locking semaphores in the right way.

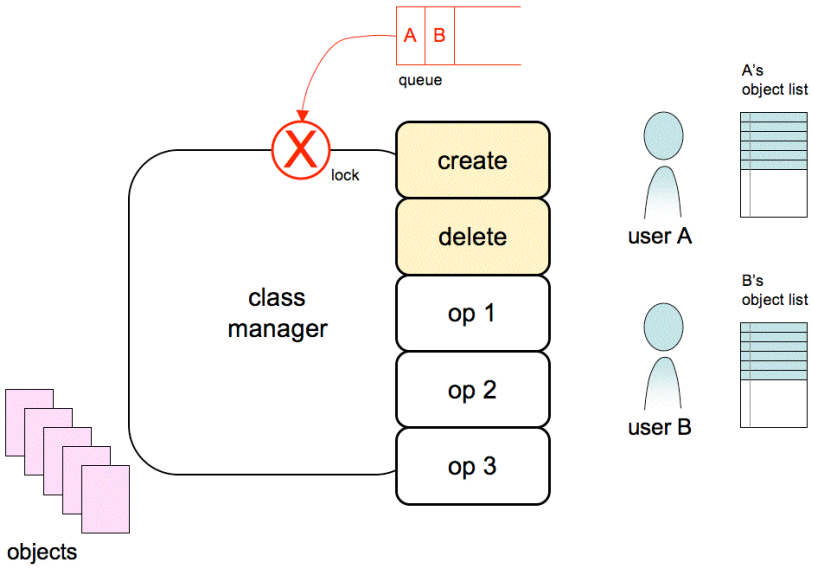

Figure 6. In this example, the entire class manager is locked when any user is accessing an object. All other users seeking access to an object are queued up. The first user in the queue gets access next.

The monitor does not scale up well. If used with a few users and a relatively small number of objects, the likelihood of a queue building up is small. But with many users, a queue is likely; users can face significant delays in accessing objects. It makes little sense to lock up an entire file system (or database) system to prevent the rather unlikely even that two users want to update the same file (record) at the same time.

The solution to this problem is to move all the locks and their queues to the individual objects (Figure 7). Now the user locks up just those objects needed for a particular transaction. In this case, the locking protocol needs to be carefully designed to avoid deadlock. Two users can generate a deadlock by requesting

the same two locks in the opposite order and timing their requests so that their first locks are obtained before the second locks are requested. Two methods to prevent deadlock are the two-phase locking protocol[4] and the ordered lock protocol[5]. (Both are studied in as part of deadlocks under concurrency control, and will not be considered further here.)

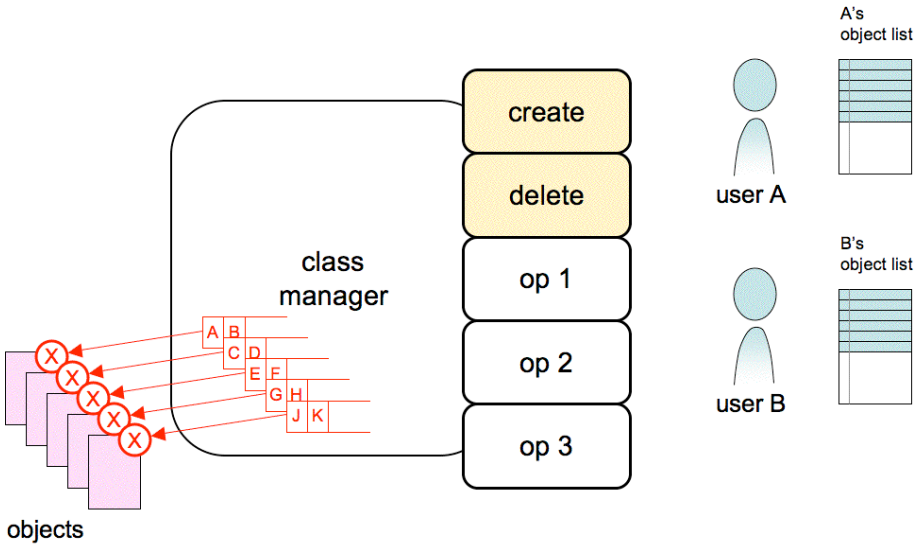

Figure 7. Much greater parallelism is possible if the class operations can be performed concurrently by different users. This is possible if the locks (and their queues) are associated with individual objects. Operations on multiple objects must follow special locking protocols to avoid deadlocks between users attempting to lock some of the same objects.

Fine-grained locking (lock on each object) is now commonly used in object languages (e.g., Java, Smalltalk) and in database systems.

Autonomous Objects

The first language to incorporate everything outlined above -- multiple users with shared access to multiple classes each containing many objects -- was Smalltalk, first implemented in 1971 and standardized in 1980. Smalltalk does this by presenting the user with a different operational view of the language. In Smalltalk, every object is an autonomous process that contains a data structure

---

[4] A process moves down the list of locks, locking them; but if it encounters one already locked, it releases all locks it just acquired and starts over. It is advisable to insert a random delay before starting over to prevent two processes with identical lists from going into an endless loop trying to do the identical thing.

[5] All locks have global serial numbers. The list of locks requested by a process is reordered by serial number, and the locks requested and held in that order.

and a set of operations that can be performed on it. This view enables a high degree of concurrency in systems with many users and many objects. The user activates an operation by sending a message to the object specifying the desired operation and its parameters. The object performs the operation and sends the result back to the user in a return message.[6] The class manager is itself an object. Figure 8 illustrates.

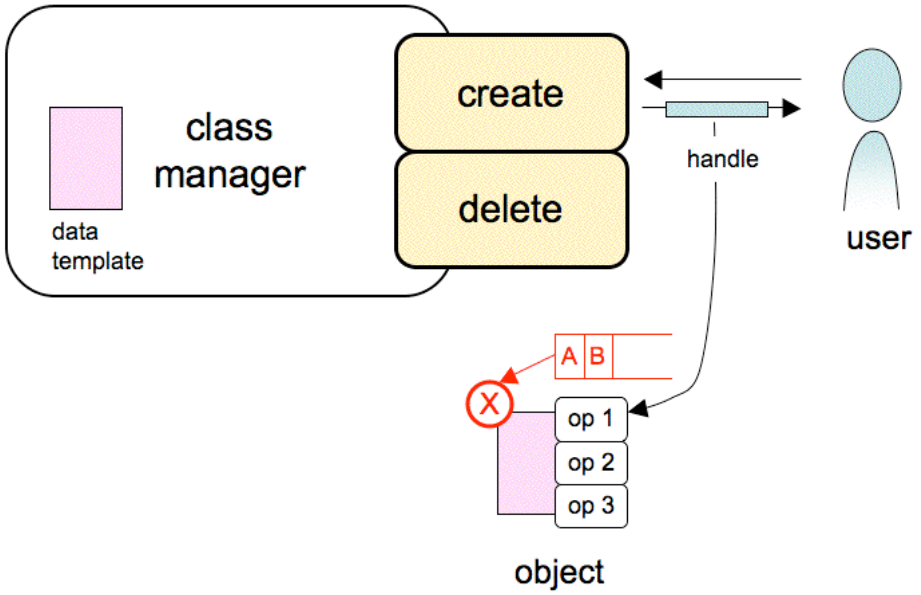

Figure 8. Smalltalk treats all objects as autonomous processes containing data and a set of allowable operations. A user wanting an operation on an object sends a message to the object telling it which operation to perform; the object responds with a return message containing the result. The class manager is itself an object; it job is to create and delete objects of the class.

Shared objects

One of the complicating factors in object systems is sharing. Most classes of objects are shared by many users; for example, the file system operations can be invoked by any user. Individual objects can also be shared; for example, a file containing a web page is shared among all web users. Sharing is a very

[6] In the procedure-based view, a file operation like "`a=read(h)`" is compiled as a call on a `read` procedure in the file system. In the autonomous-object view, it is compiled as "`send(file_manager, read, h); a=receive(file_manager)`", using message-sending operations in the underlying operating system.

important consideration and needs to be approached carefully. The two main ground rules for sharing are:

- A user can share an object by giving a copy of a handle to another user. Users make these exchanges by mutual agreement without coordinating with a central authority.
- On receiving a new handle, a user puts it in any unused object-list slot. The index of that slot becomes a local name for the handle. There is no global control on what local names users choose for new handles.

With these assumptions, we cannot put the object's memory address into a handle. For if we did, any relocation of the object would entail an impossible search and update for all outstanding handles. Figure 9 illustrates.

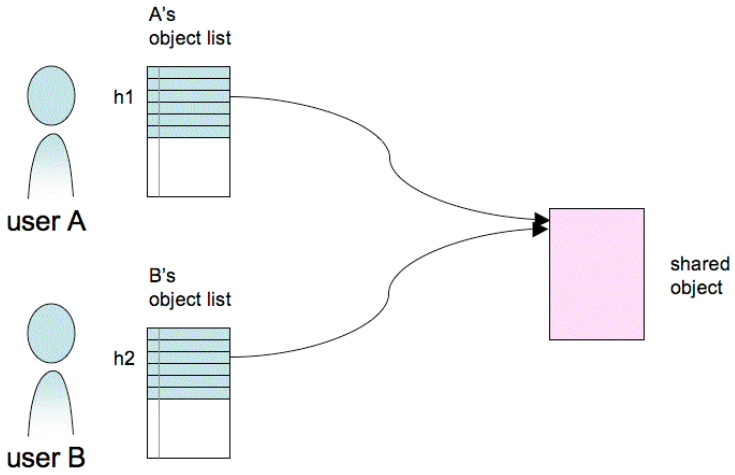

Figure 9. One method of sharing objects is to embed the storage address of the object into its handles and to allow users to distribute copies of their handles to other users. Here User A has a copy of a handle at position h1 in its object list, and User B has a copy in position h2 of its object list. This method has an insurmountable difficulty because it is impossible to find all holders of the handle and notify them of any modification to the shared object's location.

To solve the problem of object sharing, we need to separate two aspects of a handle does: (1) uniquely identify the object, distinguishing it from all other objects for all time, and (2) map from the unique name to the physical location. If the unique identifier is independent of location, we can store the object location in a single variable, called the "descriptor", and divide the mapping of handle to object into two parts:

- Map the unique name to an object descriptor. Any number of copies of the handle can point to the one descriptor.
- Map the descriptor to the object itself.

As shown in Figure 10, the operating system can relocate or delete the object simply by updating the object's descriptor. Any users seeking access will be

mapped to the descriptor and will immediately pick up the current location of the object. Figure 11 is a more detailed view of the two-level mapping.

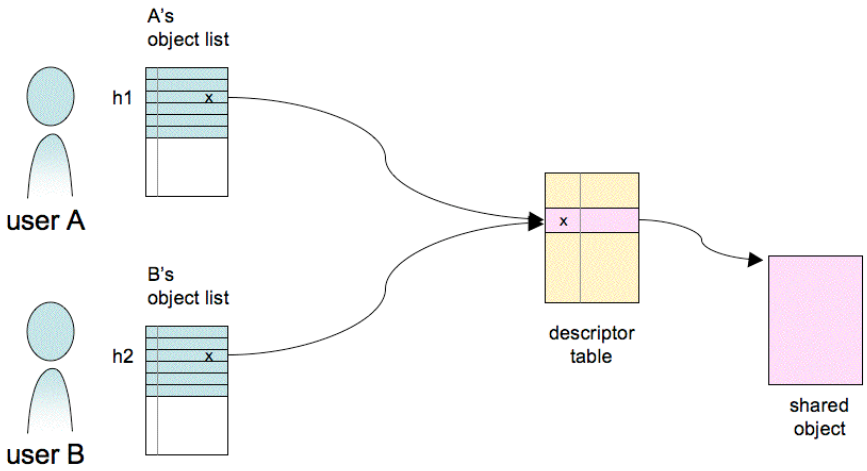

Figure 10. In a two level mapping scheme, handles contain unique identifiers (here shown as x), which are long bit strings that are assigned to an object when it is created. Once a string x has been used to name an object, x is never reused. These unique identifiers are keys to the internal *descriptor table* table by which a class manager maps to object physical locations. The first step in mapping is to search the descriptor table for an entry that matches its unique id. That entry is the address of the current physical location of the object. If an object is deleted, its descriptor is deleted; a future attempt to access the object will fail because no descriptor will be found.

An object's owner (the user who originally created it) can regulate access in two ways: first, by determining who gets, or does not get, handles, and, second, by saying which operations a particular user can apply to an object. The second way is implemented with an access code embedded into a handle:

```
handle = (tag, access code, unique identifier)
```

The tag and unique identifier are generated by the create operation in the class manager. Thus the file system creates handles with the tag `F`, full access, and identifiers that distinguish files over all time.[7] The owner can turn off access bits when passing copies of a handle to other users, thereby restricting their access.

Thus the file system gives the owner a handle of the form `(F,rw,x)`. The owner can give you a handle `(F,rw,x)` enabling you to read or write the file; or the owner can give you a read-only handle `(F,r,x)`. The write operation of the file system would accept the first handle but not the second.

[7] The unique-for-all-time requirement implies that identifiers have sufficient bits to allow every file ever created to have its own identifier. The identifiers may be 64 or even 128 bits long. A simple way to generate an identifier is to combine a computer identifier with a clock reading on that computer.

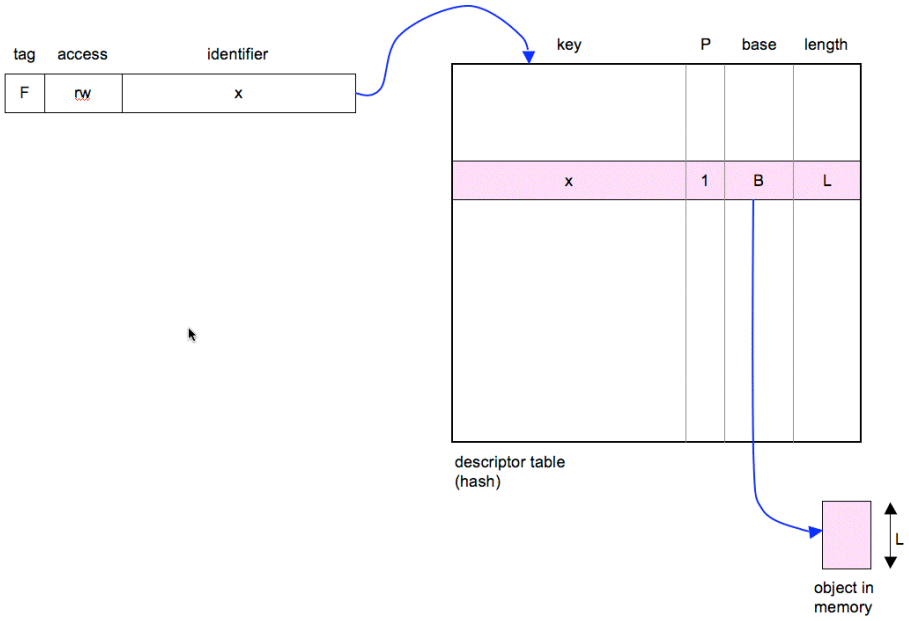

Figure 11. This is a more detailed view of the two-level mapping, used inside a class manager for its internal handle-to-location mapping. The descriptor table entries have a key field (the unique identifier) and a descriptor consisting of P (presence bit), B (base), and L (Length). P=1 means the object is stored in L contiguous bytes of memory beginning at address B. P=0 means the object is not in memory and must be located by an additional search in the secondary memory; when it is found, P can be set to 1, B to the base address, and L to the length.

Two level mapping is pervasive in all forms of sharing systems, be they virtual memories on individual computers or the entire Internet. In the Internet, a service called handle.net provides a way that publishers can choose unique identifiers, called digital object identifiers (DOI), for their works and register those identifiers together with their current URLs in a handle server. A browser plugin allows a user to refer to objects as "handle://DOI". The handle protocol contacts the handle server to translate the DOI into a URL, which is then passed to the HTTP protocol for final access.

Data Abstraction and Operating Systems

Most of the implementation structure described above originated in a seminal paper by Jack Dennis and Earl Van Horn in 1966. They recognized that operating systems would have to provide many users with many classes of objects, control access to them, and allow any degree of sharing the users want to arrange. They sought a uniform way to deal with these issues. Their proposal was much the same as described above: a set of class managers, each with a set of subroutines implementing operations on objects within the class. Each class manager had synchronization mechanisms that prevented two users from conflicting while updating a shared object. Objects were identified with handles

containing globally unique identifiers. Two level mapping was used to get a physical object address from a handle. Access was regulated with access codes embedded into handles.

In their proposal, Dennis and Van Horn used the term *capability* instead of "handle". A capability gave the holder the right of access. Capabilities were seen as tickets: no capability, no access. The integrity of the entire system rested on the integrity of capabilities. Dennis and van Horn therefore argued that the hardware should be designed to protect capabilities from tampering. To facilitate this, they put capabilities into "C-lists" attached to user processes, but outside user workspaces. Their C-lists are the same as the object lists specified above.

Dennis and Van Horn called the set of objects visible through a C-list a "sphere" or "domain" of protection. In normal operation, a process calls many local procedures, all operating from one C-list. Occasionally, however, the user needs to call a program that operates with a different C-list; for example, a disk controller or login program. They approached this by treating a domain as an object consisting of a C-list and an entry procedure as the first entry in the C-list. A capability pointing to a domain object was called an entry capability and could only be used with an enter instruction. If a user executes "`enter h`", where `h` designates an entry capability, the user's C-list is changed to the one in the domain `h` and control is passed to the entry point of `h`'s entry procedure. This elaborate mechanism has been largely superceded in modern systems by message passing: the user of Smalltalk or Java does not transfer control to a new domain; the user simply sends it a message.

Capabilities conferred an extraordinarily fine grain of access. A user had access only to the objects listed in his C-list, and nothing else. It was impossible to access anything else, not even accidentally. This offered major benefits for error control. A user's error could affect at most the objects in his own C-list; errors could not propagate to other protection domains. Capability architectures were therefore of immediate interest to designers of high-reliability systems.

The first working capability system was prototyped by Robert Fabry at the University of Chicago in 1969. He called his experimental system the "magnum" machine, short for "magic number". He viewed capabilities as magic numbers that facilitated safe sharing of all objects in the system.

The most well known capability machines and systems were built between 1972 and 1981: Plessey 250, IBM system 38, Hydra, Cambridge CAP, and Intel 432. The first two were special-purpose: Plessey for telephone circuit switching and the IBM for records processing. The other three were for general purpose interactive computing.

The Hydra system (1974 at Carnegie-Mellon University) was mentioned earlier to illustrate class hierarchies. This system implemented the capability architecture completely in software and connected it strongly with object oriented programming.

The Cambridge CAP system (completed 1979 at University of Cambridge) was a concerted attempt to build a capability system "right". They designed the

hardware to protect capabilities and efficiently implement the two level mappings and synchronizations. They designed the operating system to provide general purpose interactive computing with comprehensive support for objects. In the end, they were disappointed at their inability to keep the operating system simple. They concluded that the need to protect capabilities from tampering forced them to implement two "parallel universe" images of the operating system, one for capabilities only and the other for data only. Capability based operating systems wedded to the absolute prevention of tampering with capabilities were too complex to attract commercial interest.

Intel sought to create the iAPX 432, a new generation of "micro mainframe" chips that would replace their x86 series during the 1980s. They built many of the operations that manage objects into the 432 hardware. The resulting CPU was very complex and Intel was unable to field efficient compilers. The 432 wound up being slower than software-based object systems on conventional hardware. Intel had to abandon it. Paradoxically, object management was not the downfall. Instead it was more mundane things such as the decisions to use variable-length instructions (slower to decode), context-switches to call procedures (much slower than normal procedure call), and insufficient caches.

All of this was overtaken by events in early 1980s with arrival of RISC, the Reduced Instruction Set Computers. The idea of RISC was to use ultra simple instruction sets that could be made to run very fast on chips. More complex operations were implemented as library programs, which often ran faster on the RISC chips than the corresponding instructions on mainframes. RISC became a revolution that wiped out complex hardware platforms such as Cambridge CAP or Intel 432. Capabilities fell into disrepute and capability machines were remembered as extinct dinosaurs.

However, the architectural structure first recommended by Dennis and Van Horn became the template for object oriented software systems. The term capability, which has acquired a strong hardware meaning, was replaced by handle. Although the machines are gone, their architecture is alive and well. It is found in the run time systems of object-oriented languages including Java and Smalltalk, and in many operating systems. The original goal of protecting handles from tampering has been realized in these systems through highly structured, type-checking languages.

An Abstract World

The foregoing discussion shows that the simple idea of abstract objects invokes considerable machinery in the underlying computer structures. The complexities experienced by users arise mainly from the mechanisms of sharing.

One of the great contributions of the years of study and experience with abstract objects is that most of this complexity can be hidden if the abstract operations are precisely specified. The secret is to define how abstract operations read or modify abstract states and leave it to the implementer to make sure these abstract state-transitions are implemented correctly. Figure 12 illustrates.

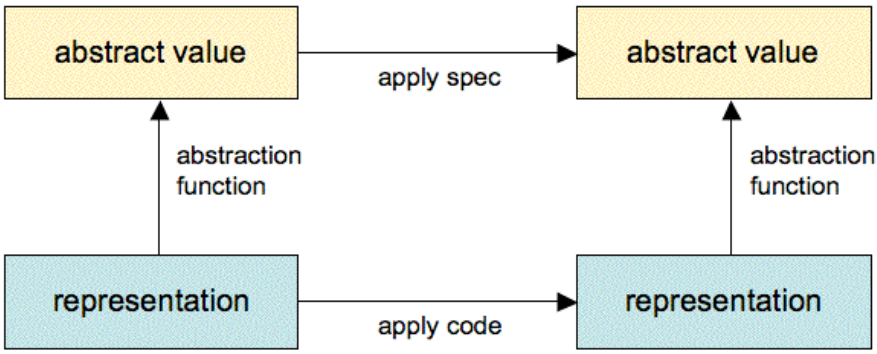

Figure 12. Representations are mapped to new representations by applying the code of an abstract operation. The corresponding abstract values are mapped to new abstract values by applying the operation's specification. The abstraction function tells how to construct the abstract value from the representation. If the implementer applies these rules consistently, the user will never have to be concerned with representations, abstraction functions, or operational codes. (Source: Guttag (2002).)

Consider the file system example. The file is an abstract object. The state of a file is the string of bytes it contains. A read operation copies the state to the user but does not change it. A write operation copies a new state from the user, replacing the old state. A large number of lower-level internal file operations are needed to read or write, but as long as they have the overall effect just described, the user need never be concerned with them.

The designer of the class manager needs to decide how the abstract state is represented and ensure that all internal operations preserve that representation. In the file system example, a file is represented as a set of equal size blocks pointed to by an index tree. A read operation, therefore, must locate all the blocks via the index table and then concatenate their contained bit-strings into one long bit-string returned to the user. A write operation must delete the old file representation and replace it with a new one, constructed as an index tree and set of blocks.

Some forms of abstraction consistent with Figure 12 involve generalizing an operation so that it works with different types of objects. This is called "polymorphism", an allusion to multiple forms of similar kind. A beautiful early example was in the Multics operating system around 1968 and moved into the Unix file system around 1972. Someone noticed that users tended to do similar things with files, devices, and pipes. Devices were external input devices such as keyboards and scanners, and output devices such as displays and printers. Pipes were objects used to stream the output of one computing process into the input of another. The common operations were `open, close, read,` and `write`. The Multics designers created a new abstraction: data stream, a sequence of bytes. They said that files contain snapshots of streams, input devices generate streams, output devices consume streams, and pipes convey streams between processes. Thus `open, close, read,` and `write` all applied to streams; a

call on the stream manager would be instantly converted to a call on the file system, the input-output system, or the interprocess communication system depending on the type field of the incoming handle. Thus `open(h)` would open a file if `h` were a file handle, device if `h` were a device handle, or pipe if `h` were a pipe handle.

Conclusions

Object Orientation is really the name of a pattern of data abstraction that allows an entire class of objects to be managed by a single manager and allows multiple users to share the objects. Information hiding and abstraction come together in these systems. These systems provide additional structure not immediately apparent from these two principles -- such as handles, two level mapping, class hierarchies, and object locking.

The original idea of using hardware to protect handles from tampering eventually proved to be too cumbersome. Modern object oriented languages give almost the same degree of protection against tampering by using sophisticated compiling methods such as type checking in class hierarchies.

Synchronization among users of multiple objects is achieved by putting locks on individual objects and using special locking protocols that avoid deadlock.

The principle of two-level mapping was discovered during the design of capability systems and has been exploited in programming languages, applications, operating systems, networking, and digital object publishing.

Bibliography

Brinch Hansen, Per. Distributed processes: a concurrent programming concept. ACM *Communications 21*, 11 (November 1978), 934-941.

Cardelli, Luca, and Peter Wegner. On understanding types, data abstraction, and polymorphism. ACM *Computing Surveys 17*, 4 (Sep 1985), 471-523.

Cox, Brad, and Andrew Novobilski. *Object-Oriented Programming: An Evolutionary Approach.* Addison Wesley (1991).

Dahl, Ole-Johan and Kristen Nygaard. SIMULA: An ALGOL-based simulation language. ACM *Communications 9*, 9 (Sept 1966), 671-678.

Dahl, Ole-Johan and Kristen Nygaard, and B. Myhrhaug. *The SIMULA 67 common base language.* Norwegian Computing Center, Forskningsveien 1B, Oslo (1968).

Dennis, Jack B., and Earl C. Van Horn. Programming semantics for multiprogrammed computations. ACM *Communications 9*, 3 (March 1966), 143-155.

Fabry, Robert S. Capability based addressing. ACM *Communications 17*, 7 (July 1974), 403-412.

Goldberg, Adele. *SMALLTALK-80: The Interactive Programming Environment.* Addision Wesley (1984).

Guttag, John V. Abstract data types and the development of data structures. ACM *Communications 20*, 6 (June 1977), 396-404.

Guttag, John V. Abstract data types, then and now. In *Software Pioneers* (M. Broy and E Denert, eds), Spinger-Verlag (2002), 443-452.

Hoare, C A R. Monitors: an operating system structuring concept. ACM *Communications 17*, 10 (October 1974), 549-557.

McCarthy, John, Fernando Corbato, Marjory Daggett. The linking segment subprogram language and linking loader. ACM Communications 6, 7 (July 1963), 391-395.

Organick, Elliott I. *Computer System Organization: The B5700/B6700 Series.* ACM Monograph Series, Academic Press (1973).

Parnas, David. A technique for software module specification with examples. ACM *Communications 15*, 5 (May 1972), 330-336.

Yourdon, Edward. *Structured Design: Fundamentals of a Discipline of Computer Program and System Design.* Prentice-Hall Facsimile Edition (1979).

Wilkes, Maurice, and Roger Needham. *The Cambridge CAP Computer and Its Operating System.* Elsevier (1979).

Wulf, William, E. Cohen, W. Corwin, Anita Jones, Roy Levin, C. Pierson, F. Pollack. HYDRA: The kernel of a multiprocessor operating system. ACM *Communications 17*, 6 (June 1974), 337-345.

