Some Desirable Properties of Data Abstraction Facilities

J. J. Horning
Computer Systems Research Group
University of Toronto
Toronto, CANADA M5S 1A4
(416) 928-6321 or (416) 445-8246

It is currently popular to say that programming languages need "data abstraction facilities," and to assert that the provision of such facilities would provide conceptual and practical advantages in the domain of data structures akin to the advantages provided by procedures in the domain of computational structures. This note explores some of the implications of this metaphor, without attempting to make it precise. I shall use the term capsule to refer to the data analog of procedure. [Those who are familiar with the SIMULA class, the CLU cluster, or the ALPHARD form, may use any of these as an approximation to capsule; I use a neutral term to avoid implying the details of any particular language.]

First, what are the advantages provided by procedures (subroutines, functions, macros)? I can think of at least eight (highly interrelated) categories: 1) avoidance of repetition, 2) modular program structure, 3) a basis for structured programming, 4) conceptual units for understanding and reasoning about programs, 5) clearly defined interfaces that may be precisely specified, 6) units of maintenance and improvement, 7) a language extension mechanism, and 8) units for separate compilation. Let us consider each of these in turn.

Repetition: A procedure allows us to write (and check) a collection of related statements once and then use it in many places, perhaps in several programs. This reduces the amount of time and effort expended in problem analysis and programming, and under some circumstances may also reduce compilation time. Similarly, capsules should allow us to write a collection of declarations once and then use them in many places.

Modularity: Procedures give us a way to break an algorithm up into meaningful components (as opposed to an arbitrary division, say into pages). All of the advantages following in the list depend in some way on this modularity. Just as a complex algorithm can be expressed in terms of a few "abstract operations" implemented by procedures (perhaps involving further "abstract operations" and procedures), a complex data structure can be decomposed into a few "abstract objects" implemented by capsules. Just as a procedure hides its internal structure from "the outside world," a capsule must hide its internal structure. [This is one of the major reasons why the modes of Algol 68 fail to provide an adequate data abstraction facility.]

Structured programming: Not every modularization of a program is an improvement. (Consider, for example, turning each statement into a procedure body.) Great skill is required to decompose complex programs into suitable structures of simple elements. The various forms of structured programming (stepwise refinement, modular decomposition) provide guidelines for making programming decisions in a sensible order, isolating the consequences of decisions, and explicitly postponing certain decisions until further information is available. Procedures make it easy to record the algorithmic structure (and even the sequence of development): procedure names denote abstract operations, and the procedure bodies that provide their implementation can be written separately. Capsules should make it easy to record the data structure (and the sequence in which decisions are taken): capsule names would denote abstract objects, with the capsule bodies that provide their implementation being written separately. [In particular, the notation by which an abstract object is referenced cannot be allowed to depend on the implementation, any more than the notation for procedure call can depend on the procedure body. (Ross has called this the uniform referent property.)]

Conceptual units: When we try to understand a large program, it is essential that we be able to understand what the procedures do without worrying about how they do it and to separately understand how they work without worrying about why they are invoked. Similarly, it will be necessary to understand programs in terms of what capsules represent, without

worrying about how they do it, and to separately understand how they represent abstract objects without worrying about why they were created. This is particularly clear when our "understanding" takes the form of a proof that the program has some property. We must be able to separate the proof that the program has the property (if its procedures and capsules have certain properties) and the proofs that the procedures and capsules have the assumed properties. The cost of a proof (either manual or mechanical) grows combinatorially with program length; our only hope of proving large programs comes from our ability to factor them into enough small modules. [Thus our language must be such that the combination of these small proofs always remains valid.]

Specification: It must be possible to precisely specify the properties that may be assumed of a module (its interface) — either for informal understanding or formal proof. Several techniques are known for specifying the properties of procedures (e.g., pre- and post-assertions, predicate transformers). Loosely speaking, a procedure may be characterized by the relationships that it imposes on the objects existing at given points in time (termination of its executions). Similarly, a capsule can be characterized by the relationships that it imposes on operations relating to given points in space (objects that are instances of the capsule). We need simple and rigourous specification techniques for capsules. The algebraic techniques used by Zilles and Guttag appear to provide the most promising approach.

Maintenance: Useful programs are continually modified. Generally, the maintenance process begins even before the program is complete, as its authors respond to changed requirements, new insights, and detected errors or inefficiencies. Maintenance itself must not introduce too many new errors. It is crucial that module boundaries isolate the consequences of a change to within a known (and preferably small) region of the source program. A change should have consequences outside a capsule or procedure only if it changes the specification of the interface. The "narrower" the interface, the greater the freedom of the implementor to seek an efficient implementation. [The ability to reference arbitrary local variables of a capsule from outside would cause the same disastrous widening of its interface caused by the ability to access arbitrary global variables from within a procedure.]

Language extension: Procedures allow the programmer to "extend" his language by adding new kinds of operations; it is often useful to build a library of procedures to augment a language for a particular application. Depending on the language, there will probably be some notational differences between "built-in" and "programmer-defined" operations (and almost certainly differences in efficiency), but there is no fundamental conceptual distinction. Similarly, capsules should allow the programmer to "extend" his language with new kinds of objects (i.e., with new types), either for a particular program or for some application area. Differences in notation and efficiency between "built-in" and "programmer-defined" objects should be minimized, and there should be no conceptual distinction. [In particular, programmer-defined objects should be "first-class citizens," with all the rights of built-in objects, such as the rights to be elements of arrays and to be passed as parameters, and the type-checking system of the language should extend to programmer-defined types.]

Separate compilation: Because we can separate the interface from the body of a procedure, it is possible to compile a procedure and the program(s) that use it separately, reserving a fixed-length space in the calling program into which the address of the procedure can be placed when it is bound (e.g., by the linkage editor or loader). However, if the procedure body is to be expanded "in-line" as an open subroutine, the calling program cannot be compiled independently of the size of the code for the procedure body. Similarly, compilation of programs invoking capsules can only be independent of the size of the data for the capsule body if the data is referenced indirectly, by means of a pointer. By analogy with "in-line" implementation of procedures, we say that a capsule is implemented "in-line" if its local data is accessed directly, rather than via a pointer. [This decision is entirely separate from whether the local procedures of a capsule are to be implemented "in-line."] "In-line" implementation of either type of module seems to be incompatible with independent compilation and with recursion [cf. Hoare on "recursive data types"].

The preceding remarks should have made the general characteristics of capsules clear. In form, they will look much like procedures, with a name, an interface specification, and a body. Within the scope of a capsule declaration, the name (optionally followed by parameters) can be used to create new instances (particular objects), primarily for use in the declaration of variables. Part of the interface will be the specification of the operations available on each instance. The body will contain declarations of the local variables that collectively implement the abstract object, the local procedures that implement the operations required by the interface, and initialization statements to "start up" the abstract object in a consistent state.

We resist the temptation to "unify" the twin concepts of procedure and capsule into a single concept, for two reasons: they are abstractions of quite different things (operations and objects), and the lifetimes of their instances are quite different — in a sequential program, each operation must terminate before the next is started, but an object must persist throughout the scope in which it can be referenced. Hoare has suggested an elegant approach to the implementation of capsules (based on SIMULA) that emphasizes both the similarities and the distinction: Each capsule invocation is treated precisely like a procedure invocation in the creation of an "activation record" for its local variables and the execution of its initialization statements. However, the activation record is not freed upon return from the initialization statements; rather, its address is stored in the (fixed-length) space reserved for the variable associated with the instance. Operations on the abstract object are then simply invocations of the associated procedures, with this address providing the "environment." The activation record can be freed with the variables in the scope containing the declaration. [In sequential programs, a stack discipline is still preserved for storage management.]

The details of capsules are much less clear than the broad outlines. The issues of parameter-passing (e.g., are procedures and capsules acceptable parameters?), result values (e.g., can procedures return procedures or capsules?), inheritance of environment, programmer-supplied specifications and other forms of checkable redundancy, proof techniques, and even appropriate notations are still matters for heated debate. Decisions taken on these issues will have major effects on the convenience, power, security, and efficiency of any data abstraction facility.

Finally, I should like to mention one thing that capsules are not: They are not mechanisms for defining new data structures. Procedures are abstractions that hide the details of the control structures (do, if-then-else, case, etc.) that are used in their implementation, but do not extend the class of control structures available in the language (i.e., they provide new kinds of operations, rather than new structures). Similarly, capsules are abstractions that hide the details of the data structures (array, record, list) that are used in their implementation; they do not extend the class of data structures in the language (i.e., they provide new types, rather than new structures).

Acknowledgement: Most of the ideas in this paper are not new. I have acquired them largely through extensive discussion with the members of IFIP Working Groups 2.3 (Programming Methodology) and 2.4 (Machine-Oriented Higher-Level Languages); many have previously appeared in print. What I have attempted to do is juxtapose them within a meaningful framework for discussion and evaluation.