SHARED CLASSES

PER BRINCH HANSEN

(1973)

The author discusses the close relationship between data and operations and suggests that a compiler should be able to check that data structures are accessed by meaningful procedures only. This idea leads to the introduction of shared classes—a programming notation for the monitor concept. The notation is illustrated by a message buffer for concurrent processes.

We will discuss the close relationship between data and operations and use it to define a very important form of resource protection.

If we consider variables of *primitive types* such as *integer* and *boolean*, it is quite possible that values of different types will be represented by identical bit strings at the machine level. For example both the *boolean* value *true* and the *integer* value 1 might be represented by the bit string

000...001

in single machine words.

So data of different types are distinguished not only by the representation of their values, but also by the operations associated with the types. An *integer*, for example, is a datum subject only to arithmetic operations, comparisons, and assignments involving other data subject to the same restrictions.

Now consider *structured types*. Take for example a variable that represents a message buffer which contains a sequences of messages sent, but not yet received. A *static* picture of process communication can be defined by

P. Brinch Hansen, *Operating System Principles*, Section 7.2 Class Concept, Prentice Hall, Englewood Cliffs, NJ, (July 1973), 226–232.

assertions about the relationships of the components of the message buffer. But to understand how and when messages are exchanged *dynamically*, one must also study the *send* and *receive* procedures defined for a message buffer. These operations in turn are only meaningful for the particular representation of the message buffer chosen and can only be understood precisely by studying its type definition.

These examples illustrate the point made by Dahl (1972): "Data and operations on data seem to be so closely connected in our minds, that it takes elements of both kinds to make any concept useful for understanding computing processes."

Simon (1962) has pointed out that the search for state and process descriptions of the same phenomenon is characteristic of problem solving: "These two modes of apprehending structure are the warp and weft of our experience. Pictures, blueprints, most diagrams, chemical structural formulae are state descriptions. Recipes, differential equations, equations for chemical reactions are process descriptions. The former characterize the world as sensed; they provide the criteria for identifying objects, often by modeling the objects themselves. The latter characterize the world as acted upon; they provide the means for producing or generating objects having the desired characteristics."

"The distinction between the world as sensed and the world as acted upon defines the basic condition for the survival of adaptive organisms. The organism must develop correlations between goals in the sensed world and actions in the world of process."

In Section 2.6 on program construction, I have illustrated this alternation between a refinement of data (representing states) and program (representing processes). The essence of this form of problem solving is the following:

When a programmer needs a concept such as process communication, he first postulates a set of operations (in this case, *send* and *receive*) that have the desired effect at his present level of thinking. Later, he chooses a specific representation of a data structure (a message buffer), that enables him to implement the operations efficiently on the available machine.

When the programmer is trying to convince himself of the correctness of a program (by formal proof or testing), he will tacitly assume that these operations (*send* and *receive*) are the only ones carried out on data structures of this type (*message buffers*).

If other statements in his program are able to operate on message buffers, he cannot make this assumption. The most extreme case is unstructured ma-

chine language, which potentially permits each statement to influence any other statement, intentionally or by mistake. This makes program verification an endless task since one can never be sure, when a new component is added to a large program, how this will influence previously tested components.

If, on the other hand, the previous assumption is justified, the programmer can convince himself of the correctness of process communication by studying only the type definition of a message buffer and the procedures *send* and *receive*. Once this program component has been shown to be correct, the designer can be confident that subsequent addition of other components will not invalidate this proof. This makes the task of verification grow linearly with the number and size of components—an essential requirement for the design of large, reliable programs.

According to the previous definition, it is an obvious protection problem to check that data are accessed by operations consistent with their type. To what extent do the structures of present high-level languages enable a compiler to do this?

A decent compiler for an algorithmic language such as *Fortran*, *Algol 60*, or *Pascal* will check the compatibility of data and operations on them for *primitive types* (Naur 1963). The compiler can do this because the permissible operations on primitive types are part of the language definition.

But in the case of *structured types*, only the most rudimentary kind of checking is possible with these languages. All the compiler can check is that data in assignment statements and comparisons for equality are of the same type. But, since the languages mentioned do not enable the programmer to associate a set of procedures with a type definition, the compiler cannot check whether the operations on a message buffer are restricted to *send* and *receive* procedures as intended by the programmer. This is a serious deficiency of most programming languages available today.

An exception is the *Simula 67* language (Dahl 1968), an extension of Algol 60 originally designed for simulation. In Simula 67, the definition of a structured data type and the meaningful operations on it form a single, syntactical unit called a class.[1]

I will briefly describe a simplified, restricted form of the Simula 67 class concept in a Pascal-inspired notation.

The notation

[1]Readers of the Pascal report by Wirth (1971) should notice that the Simula class concept is completely unrelated to the Pascal class concept.

class $T = v_1$: T_1; v_2: T_2; ...; v_m: T_m;

procedure $P_1(\ldots)$ **begin** S_1 **end**
...
procedure $P_n(\ldots)$ **begin** S_n **end**

begin S_0 **end**

defines: (1) a data structure of type T consisting of the components v_1, v_2, ..., v_m of types T_1, T_2, ..., T_m; (2) a set of procedures (or functions), P_1, P_2, ..., P_n that operate on the data structure; and (3) a statement S_0 that can define its initial value.

A variable v of type T is declared as usual:

var v: T

Upon entry to the context in which the variable v is declared, storage is allocated for its components v_1, v_2, ..., v_m, and the initial statement S_0 is carried out for this variable.

A call of a procedure P_i on the variable v is denoted:

$v.P_i(\ldots)$

Procedure P_i can refer to the components v_1, v_2, ..., v_m of v, to its own local variables, and to the parameters of the given call. The operations P_1, P_2, ..., P_n are the only ones permitted on the variable v.

An obvious idea is to represent critical regions by the concept *shared class*, implying that the operations P_1, P_2, ..., P_n on a given variable v of type T exclude one another in time.

The concept *message buffering* is defined as a shared class in Algorithm 1. A buffer variable b and a message variable t are declared and accessed as follows:

var b: B; t: T;

b.send(t) b.receive(t)

Strictly speaking, assignment to a message parameter m can only be made within the class B if its type T is primitive. But it seems reasonable to retain the simple type definition

type T = <type>

```
shared class B =
  buffer: array 0..max−1 of T;
  p, c: 0..max−1;
  full: 0..max;

procedure send(m: T);
begin
  await full < max;
  buffer[p] := m;
  p := (p + 1) mod max;
  full := full + 1;
end

procedure receive(var m: T);
begin
  await full > 0;
  m := buffer[c];
  c := (c + 1) mod max;
  full := full − 1;
end

begin p := 0; c := 0; full := 0 end
```

Algorithm 1 Representation of a message buffer by a shared class.

to indicate that variables of this type can be accessed directly.

The class concept in Simula 67 has several other aspects, among them a mechanism for defining a hierarchy of classes (Dahl 1972). My main purpose here is to show a notation which explicitly restricts operations on data and enables a compiler to check that these restrictions are obeyed. Although such restrictions are not enforced by Simula 67, this would seem to be essential for effective protection.

Many computers support a restricted form of shared class at the machine level of programming. I am referring to the *basic monitor* procedures and data structures which control the sharing of processors, storage, and peripherals at the lowest level of programming. This class concept enforced at run time is implemented as follows: The address mapping performed by a central processor prevents computations from referring directly to data structures belonging to the basic monitor, but permits them to call a well-

defined set of monitor procedures. Mutual exclusion in time of such calls is achieved by means of an arbiter and by delaying interrupt response. To prevent computations from bypassing the monitor and referring directly to physical resources, the central processor recognizes two states of execution: the *privileged state*, in which all machine instructions can be executed; and the *user state*, in which certain instructions cannot be executed (those that control program interruption, input/output, and address mapping). The privileged state is entered after a monitor call; the user state is entered after a monitor return.

In Chapter 1 I said "It is now recognized that it is desirable to be able to distinguish in a more flexible manner between many levels of protection (and not just two)." We have seen that it is indeed desirable to be able to enforce a separate set of access rules for each data type used. The class concept is a general structuring tool applicable at all levels of programming, sequential as well as concurrent.

The class concept was introduced here to protect *local* data structures within a program against inconsistent operations. But the concept is applicable also to data structures which are *retained* within the computer after the termination of computations.

One example of retained data structures are those used within an *operating system* to control resource sharing among unrelated computations. These data structures must be accessed only through well-defined procedures; otherwise, the operating system might crash. So an operating system defines a set of standard procedures which can be called by computations. Since these procedures remain unchanged over reasonable periods of time, a compiler should be able to use a description of them to perform type checking of calls of them within user programs in advance of their execution.

We are thus lead to the idea of maintaining *data structures defining environments of compilation and execution*. An environment defines a set of retained data structures and procedures accessible to a given computation.

Another example of retained data structures are files stored semipermanently on backing stores. In most present *file systems*, a computation can either be denied access to a given file or be permitted to *read*, *write*, or *execute* it. This seems a rather crude distinction. In most cases, a data file is intended to be used only in a particular manner; for example, a source text of a program is intended to be edited or compiled by a particular compiler; most other operations on it may be entirely meaningless from the user's point of view. To maintain the integrity of a file, its creator should therefore be

able to associate it with a set of procedures through which it can be accessed in a meaningful manner. This is possible, for example, in the file system for the *B5500* computer (McKeag 1971).

Assuming that this set of procedures remains unchanged over reasonable periods of time, it would again be possible to check the consistency of references to files within user programs at compile time. The basic requirement is that the access rules remain fixed between compilation and execution of programs.

Such a system differs from the present ones in two aspects: (1) a program is compiled to be executed in a particular environment; and (2) a compiled program may become invalid if its environment changes. This is acceptable only if most programs are compiled shortly before execution or if they operate in a fairly constant environment. The benefits of this approach would be an early detection of program errors and a more efficient execution because fewer protection rules would have to be checked dynamically.

References

Dahl, O.-J., Myhrhaug, B., and Nygaard, K. 1968. Simula 67—common base language. Norsk Regnesentral, Oslo, Norway, (May).

Dahl, O.-J., and Hoare, C.A.R. 1972. Hierarchical program structures. In *Structured Programming*, O.-J. Dahl, E.W. Dijkstra, and C.A.R. Hoare, Eds., Academic Press, New York, 175–220.

McKeag, R.M. 1971. Burroughs B5500 master control program. In *Studies in Operating Systems*, R.M. McKeag and R. Wilson, Academic Press, New York, (1976), 1–66.

Naur, P. 1963. The design of the GIER Algol compiler. *BIT 3*, 2–3, 124–140 and 145–166.

Simon, H.A. 1962. The architecture of complexity. *Proceedings of the American Philosophical Society 106*, 6, 468–482.

Wirth, N. 1971. The programming language Pascal. *Acta Informatica 1*, 1, 35–63.