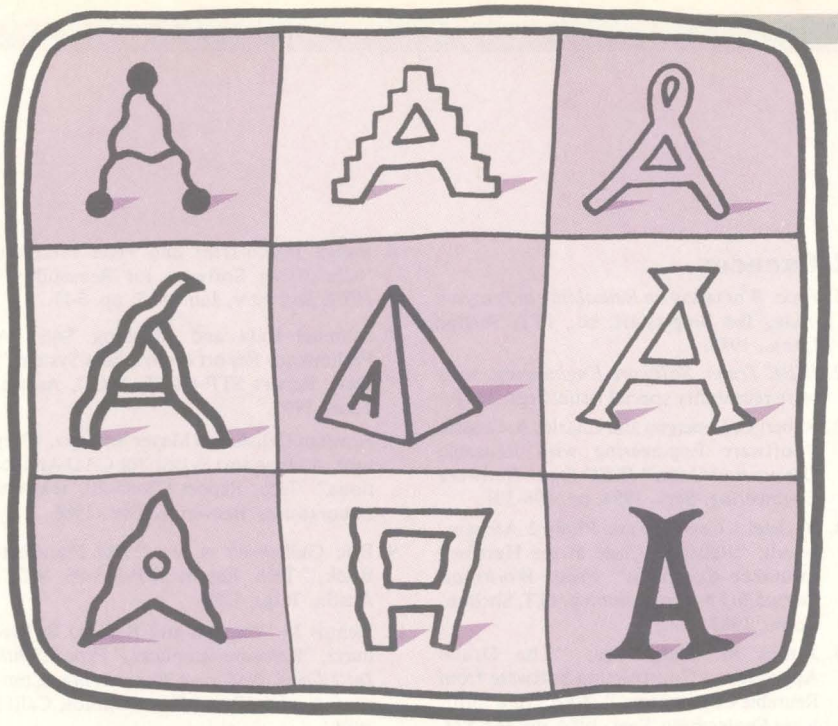
"Neither ever quite the same, nor ever quite another"



Reusability: The Case for Object-Oriented Design

Bertrand Meyer, Interactive Software Engineering

Simply being more organized will not make the reuse problem go away. The issues are technical, not managerial. The answers lie in object-oriented design.

"Why isn't software more like hardware? Why must every new development start from scratch? There should be catalogs of software modules, as there are catalogs of VLSI devices: When we build a new system, we should be ordering components from these catalogs and combining them, rather that reinventing the wheel every time. We would write less software, and perhaps do a better job at that which we do develop. Then wouldn't the problems everyone laments — the high costs, the overruns, the lack of reliability — just go away? Why isn't it so?"

If you are a software developer or manager you have probably heard such remarks before. Perhaps you have uttered them yourself.

The repetitive nature of computer programming is indeed striking. Over and over again, programmers weave a number of basic patterns: sorting, searching, reading, writing, comparing, traversing, allocating, synchronizing. . . Experienced programmers know well the feeling of *déjà vu* that is so characteristic of their trade.

Attempts have been made to measure this phenomenon; one estimate is that less than 15 percent of new code serves an original purpose.[1]

A way to assess this situation less quantitatively but perhaps closer to home is to answer the following question honestly, again assuming you develop software or direct people who do. Consider the problem of table searching: An element of some kind, say x, is given with a set of similar elements, t; the program is to determine if x appears in t. The question is: How many times in the last six months did you or people working for you write some program fragment for table searching?

0740-7459/87/0300/0050/$01.00 ©1987 IEEE

Chances are the answer will be one or more. But what is really remarkable is that, most likely, the fragment or fragments will have been written at the lowest reasonable level of abstraction — as code instead of by calling existing routines.

Yet table searching is one of the best researched areas of computer science. Excellent books describe the fundamental algorithms — it would seem nobody should need to code a searching algorithm in standard cases anymore. After all, electronic engineers don't design standard inverters, they buy them.

This article addresses this fundamental goal of software engineering, reusability, and a companion requirement, extendibility (the ease with which software can be modified to reflect changes in specifications). Progress in one of these areas usually advances the aims of the other as well, so when we discuss reusability, we will be adding *in petto*, ". . .and extendibility."

Our main thesis is that object-oriented design is the most promising technique now known for attaining the goals of extendability and reusability.

Ni tout à fait la même. . .

Why isn't reuse more common? Some of the reasons are nontechnical:

- Economic incentives tend to work against reusability. If you, as a contractor, deliver software that is too general and too reusable, you won't get the next job — your client won't need a next job!
- The famous not-invented-here complex also works against reusability.
- Reusable software must be retrievable, which means we need libraries of reusable modules and good database searching tools so client programmers can find appropriate modules easily. (Some terminology: A client of a module *M* is any module relying on *M*; a client programmer is a person who writes a client module; an implementer of *M* is the programmer who writes *M*.)

In the US, the STARS project is an effort that aims, among other things, to overcome such obstacles.

Tip of the iceberg. In my opinion, these issues are only the tip of the iceberg; the main roadblocks are technical. Reuse is limited because designing reusable software is hard. This article elaborates on what makes it so hard and should dispel any naive hope that software problems would just go away if we were more organized in filing program units.

Let's take a closer look at the repetitive nature of software development. Programmers do tend to do the same kinds of things time and time again, but they are not *exactly* the same things. If they were, the solution would be easy, at least on paper; but in practice, so many details may change as to render moot any simple-minded attempt at capturing commonality.

Such is the software engineer's plight: time and time again composing a new variation that elaborates on the same basic themes: "neither ever quite the same, nor ever quite another. . ."*

Take table searching again. True, the general form of the code is going to look the same each time: Start at some position in the table *t*; explore the table from that position, checking if the element found at the current position is the one being sought; if not, move to another position. The process terminates either when the element has been found or when all positions of interest in the table have been unsucessfully probed.

This paradigm applies to all standard cases of data representation (unsorted or sorted array, unsorted or sorted linked list, sequential file, binary tree, B-tree, hash table, etc.). It may be expressed more precisely as a program schema:

```
Search(x : ELEMENT, t :
TABLE_OF_ELEMENT)
return boolean is
    --Look for element x in table t
  pos: POSITION
begin
  pos:= INITIAL_POSITION (x,t);
  while not EXHAUSTED (pos,t)
  and then not FOUND (pos,x,t) do
    pos := NEXT (pos,x,t);
  end;
  return not EXHAUSTED (pos,t)
end -- Search
```

Too many variants. The difficulty in coming up with a general software element for searching is apparent: Even though the pattern is fixed, the amount of variable information is considerable. Details that may change include the type of table elements (ELEMENT), how the initial position is selected (INITIAL_POSITION), how the algorithm proceeds from one position to the next (NEXT), and all the types and routines in uppercase, which will admit a different refinement for each variant of the algorithm.

Not only is it hard to implement a general-purpose searching module, it is almost as hard to *specify* such a module so that client modules could rely on it without knowing the implementation.

Beyond the basic problem of factoring out the parts that are common to all implementations of table searching, an even tougher challenge is to capture the commonality within some conceptual subset. For example, an implementation using sequential search in arrays is very similar to one based on sequential linked lists; the code will differ only by small (yet crucial) details, shown in Table 1.

Within each group of implementations (all sequential tables, for example), there are similarities. If we really want to write carefully organized libraries of reusable software elements, we must be able to use commonalities at all levels of abstraction.

All these issues are purely technical; solving all the managerial and economical obstacles to reusability that one hears about in executives' meetings will not help a bit here.

Routines

Work on reusability has followed several approaches (see the box on p. 54). The classical technique is to build libraries of routines (we use the word "routine" to cover what is variously called procedure, function, subroutine, or subprogram). Each routine in the library implements a well-defined operation. This approach has been quite successful in scientific computation — excellent libraries exist for numerical applications.

** Et qui n'est chaque fois ni tout à fait la même. Ni tout à fait une autre. . .: And [she] who from one [dream] to the next is neither ever quite the same, nor ever quite another. . . — Gérard de Nerval.*

Table 1.
Implementation variants for sequential search.

	Sequential array	Linked list	Sequential file
Start search at first position	*i* := 1	*l* := *first*	rewind
Move to next position	*i* := *l* + 1	*l* := *l.next*	read
Test for table exhausted	*i* > size	*l* = null	end_ of_ file

Indeed, the routine-library approach seems to work well in areas where a set of individual problems can be identified, provided the following limitations hold:

- Every instance of each problem should be identifiable with a small set of parameters.
- The individual problems should be clearly distinct. Any significant commonality that might exist cannot be put to good use, except by reusing some of the design.
- No complex data structures should be involved because they would have to be distributed among the routines and the conceptual autonomy of modules would be lost.

The table-searching example may be used to show the limitations of this approach. We can either write a single routine or a set of routines, each corresponding to a specific case.

A single routine will have many parameters and will probably be structured like a gigantic set of case instructions. Its complexity and inefficiency will make it unusable. Worse, the addition of any new case will mean modification and recompilation of the whole routine.

A set of routines will be large and contain many routines that look very similar (like the searching routines for sequential arrays and sequential linked lists). But there is no simple way for the implementers to use this similarity. Client programmers will have to find their way through a maze of routines.

Modular languages

Languages like Modula-2 and Ada offer a first step toward a solution. These languages use the notion of module (the Ada term is package), providing a higher level structuring facility than the routine. A module may contain more than one routine, together with declarations of types, constants, and variables. A module may thus be devoted to an entire data structure and its associated operations.

This approach is rooted in the theory of data abstraction, but its basic concepts may be illustrated simply with our table-searching example.

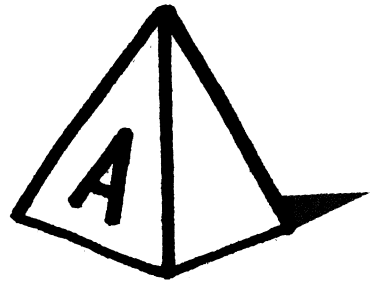
A table-searching routine isn't worth very much by itself; it must be complemented by routines that create and delete tables and insert and delete elements, all governed by a certain representation of the table, given by a type declaration. These routines and the type declaration are closely connected logically, so they might as well be part of the same syntactic unit. Such units are basically what modular languages offer.

This is a significant improvement: We can now keep under one roof a set of related routines that pertain to a specific implementation of a data abstraction. For example, the module for a binary search tree of integers (INT_BINARY_TREE) will contain the declaration of a type intbintree and routines Create, Search, Insert, and so on. The client code might look like:

```
x : integer; b : boolean; p :
INT_BINARY_TREE.intbintree;
INT_BINARY_TREE.Create (t);
INT_BINARY_TREE.Insert (x,b) ;
b := INT_BINARY_TREE.Search(x,p)
```

(Here I use the Ada dot notation: *A.f* means "feature *f*, such as a type or routine, from module *A*." In Ada and other languages, simpler notations are available when a client repeatedly uses features from a given module.)

For reusability, these techniques are useful but limited. They are useful because encapsulating groups of related features helps implementers (in gathering features) as well as clients (in retrieving features), and all of this favors reusability. But they are limited because they do not reduce significantly the amount of software that needs to be written. Specifically, they don't offer any new clue as to how to capture common features.



Overloading and genericity

A further improvement is overloading, as provided in Algol 68 and Ada. Overloading means attaching more than one meaning to a name, such as the name of an operation.

For example, when different representations of tables are each defined by a separate type declaration, you would use overloading to give the same name, say Search, to all associated search procedures. In this way, a search operation will always be invoked as *b* := Search *(x,t)*, regardless of the implementation chosen for *t* and the type of table elements.

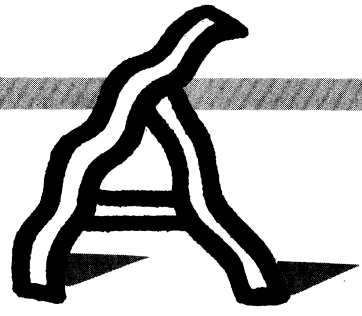
Overloading works well in a strictly typed language where the compiler has enough type information about *x* and *t* to choose the right version of search.

A companion technique is genericity, provided in Ada and Clu. Genericity allows a module to be defined with generic parameters that represent types. Instances of the module are then produced by supplying different types as actual parameters. This is a definite aid to reusability because just one generic module is defined, instead of a group of modules that differ only in the types of objects they manipulate.

For example, instead of having an INT_BINARY_TREE module, a REAL_BINARY_TREE module, and so on, you could define a single generic BINARY_TREE [*T*] module. Any actual type (INTEGER, REAL, etc.) could correspond to the formal generic parameter *T*. The search routine can be defined in the generic module to act on an argument *x* of type *T*. Then every instance of the module automatically has its own version of search.

In summary, overloading and genericity each offer something toward reuse:

- with overloading, the client programmer may write the same code when using different implementations of the same data



abstraction, as provided by different modules;

• with genericity, the implementer may write a single module for all instances of the same implementation of a data abstraction, applied to various types of objects.

These techniques are interesting advancements in reusability. But they do not go far enough. Roughly speaking, they do not provide enough flexibility and they force programmers to decide too much too soon.

Not enough flexibility. They are not flexible enough because they cannot capture fine grains of commonality between groups of implementations of the same general data abstraction. This is because there are only two levels of modules: generic modules, which are parameterized and thus open to variation, but not directly usable; and fully instantiated modules, which are directly usable but not open to refinement. Thus we cannot describe a complex hierarchy of representations that have different levels of parameterizations.

Too much too soon. Neither technique allows a client to use various implementations of a data abstraction (say the table) without knowing which implementation is used in each instance.

On one hand, each generic module refers to a single, explicitly specified instance of that module. Overloading, on the other hand, is essentially a syntactic facility that relieves the programmer of having to invent names for different implementations; the burden is placed on the compiler instead. Nevertheless, each invocation of an overloaded operation name, say Search(x, t), refers to a specific version of the operation — and both the client programmer and compiler know which version that is.

Client programmers do not actually need to know how each version is implemented, since Ada and Modula-2 modules are used by clients through an interface that lists the available routines, independent of their implementation. But they do need to decide explicitly which version is used. For example, if your modules use various kinds of tables, you don't have to know how to implement hash tables, indexed sequential files, and the like — but you must say which representation you want each time you use a table operation.

True representation-independence only happens when a client can write the invocation Search(x, t) and mean, "look for x in t using the appropriate algorithm for whatever kind of table and element x and t happen to be at the time the invocation is executed."

This degree of flexibility, essential for the construction of reusable software elements, can only be achieved with object-oriented design.

Object-oriented design

This fashionable term has been somewhat overused in recent years. The definition used here is fairly dogmatic. Object-oriented design is viewed as a *software decomposition technique*. An overview of some object-oriented languages is given in the box on p. 59.

Object-oriented design may be defined as a technique which, unlike classical (functional) design, bases the modular decomposition of a software system on the classes of objects the system manipulates, not on the functions the system performs. Classical approaches like functional top-down design (even, to a large extent, dataflow analysis methods) require designers to first ask what the system does. Object-oriented design avoids such questions as long as possible, in fact until the system actually is run. Why?

The top-down functional approach is probably adequate if the program you are writing solves a fixed problem once and for all. But the picture changes when you take a long-term view, for what the system will do in its first release is probably going to be a little different from what you think it will do at requirements time, and *very* different from what it will do five years later, if it survives that long.

However, the categories of objects on which the system acts will probably be more or less the same. An operating system will always work on devices, memories, processing units, communication channels, and so on; a document processing system will always work on documents, chapters, sections, paragraphs, and so on.

Thus it is wiser in the long term to rely on categories of objects as a basis for decomposition, but (and this is an important but) only if these categories are viewed at a sufficiently high level of abstraction. This is where abstract data types come in.*

Abstract data types

If we use the physical structure of objects as the basis for decomposition, we won't go very far toward protecting our software's structure against requirement changes. In fact, we will probably be worse off than we would be with functional design. A study by Lientz and Swanson,[3] quoted by Boehm,[4] shows that 17.5 percent of the cost of software maintenance stems from changes in programs that reflect changes in data formats. This emphasizes the need to separate the programs from the physical structure of the objects they handle.

Abstract data types provide a remarkable solution to this problem. An abstract data type describes a class of objects through the external properties of these objects instead of their computer representation. More precisely, an abstract data type is a class of objects characterized by the operations available on them and the abstract properties of these operations.

It turns out that abstract data types, which provide an excellent basis for software specification, are also useful at the design and implementation stage. In fact, they are essential to the object-oriented approach, and enable us to refine the definition of object-oriented design: Object-oriented design is the construction of software systems as structured collections of abstract data-type implementations.

An important aspect of the object-oriented method is that it actually identifies modules with implementations of

* One other design method that does emphasize the motto "look at the data before you look at the functions" is Jackson's method.[2] However, a comparative analysis of Jackson's method and object-oriented design falls beyond the scope of this article.

Reuse in practice

It would be unfair to suggest that reuse never occurs in software. Much research has been published on the issue and various approaches to reusability have gained some acceptance.

- Source-code reusability is common in academic environments. Unix, for example, has spread through universities and research laboratories thanks to the online availability of the source code that allowed users to study, imitate, and extend the system. This is also true of many Lisp environments. It is unlikely that this form of reusability is going to happen in traditional industrial environments, however. Beyond the obvious economical and psychological obstacles to source-code dissemination, this technique does not support information hiding, an essential requirement for large-scale reuse.
- Another form of reusability that is widely practiced in industry (some say the only one) is reusing personnel. By transferring software engineers from project to project, companies maintain know-how and ensure that previous experience is applied to new developments. This nontechnical approach to reusability is obviously limited in scope, if only because of the high turnover in the data processing professions.
- Japanese software factories rely on the approach that designs, not implementations, should be reused. This technique may be viewed as an extension of the personnel approach, if you consider designs as formalized pieces of know-how.

But reuse of *design* does not appear to go much further than this idea. The very notion of designs as independent software products, having lives separate from those of the corresponding implementations, seems dubious. Perpetual consistency between design and code, which software engineering textbooks (rightly) promote as a desirable goal, is notoriously hard to maintain throughout the evolution of a software system. Thus, if only the design is reused there is a strong risk of reusing incorrect or obsolete elements.

Background reading. The first published discussion of reusability was most likely the contribution of M.D. McIlroy, "Mass-Produced Software Components," to the 1968 NATO conference on software engineering.

A particularly good source is the September 1984 issue of the *IEEE Transactions on Software Engineering*, especially the articles by Horowitz and Munso, Standish, Goguen, and Curry and Ayers. An important work not described in that issue is the MIT Programmer's Apprentice project, which relies on the notion of reusable plans and clichés (*IEEE Trans. Software Eng.*, Jan. 1987).

The proceedings of the First DoD-Industry Symposium on the STARS program (Nat'l Security Industry Assoc., 1985) contains several discussions of reusability from an industrial, Ada-oriented perspective.

abstract data types. It is not only that modules *comprise* these implementations (as in Ada and Modula-2, and in Fortran-77, thanks to multiple-entry subroutines); a single program structure *is* both a module and a type. Such a dual-purpose structure was dubbed a "class" by the creators of the pioneer object-oriented language, Simula 67.

Two words should be emphasized in the above definition. The first is "implementation": A module of an object-oriented program is not an abstract data type, but one implementation of an abstract data type. However, the details of the implementation are not normally available to the rest of the world, which only sees the official specification of the abstract data type.

The second is "structured." Collections of classes may indeed be structured using two different relations: client and inheritance. Figure 1 illustrates these two relations. The client relation is represented by horizontal double arrows; inheritance by a single, vertical arrow.

Class A is said to be a client of B if A contains a declaration of the form *bb: B*. (In this and all other object-oriented examples, I use the notations and terminology of the object-oriented language Eiffel. See the box on p. 60 for more about Eiffel.)

In this case, A may manipulate *bb* only through the features defined in the specification of B. Features comprise both attributes (data items associated with objects of type B) and routines (operations for accessing or changing these objects). In Eiffel, features are applied through dot notation, as in *bb.x*, *bb.f* (*u,w,x*).

As an example, consider a client class X of class BINARY_SEARCH_TREE that implements a specific form of tables. Client X may contain elements of the form:

```
bb:BINARY_SEARCH_TREE;
   -- declare bb as binary search tree
bb.Create;
   -- allocate table (routine call)
bb.insert (x);
   -- insert x into bb (routine call)
y:= bb.size ;
   -- (attribute access)
```

The second relation between classes, inheritance, is fundamental to true object-oriented languages. For example, our BINARY_SEARCH_TREE class may be defined as an heir (possibly indirect) to a class TABLE that describes the general properties of tables, independent of the representation.

A class C defined as an heir to a class A has all the features of A, to which it may add its own. Descendants of a class include its heirs, the heirs of its heirs, and so on. The relationship between C and A may be defined from the viewpoint of both the module and the type.

From the module perspective, inheritance allows the programmer to take an existing world (class A) and plunge it as a whole into a new world, C, which will inherit all its properties and add its own. In *multiple* inheritance, as present in Eiffel, more than one world may be used to define a new one.

From the type perspective, C is considered a special case of A: Any object of type C may also be interpreted as an object of type A. In particular, a variable of type A may be assigned an object of type C, although the reverse is not true, at least in a statically typed language like Eiffel. This

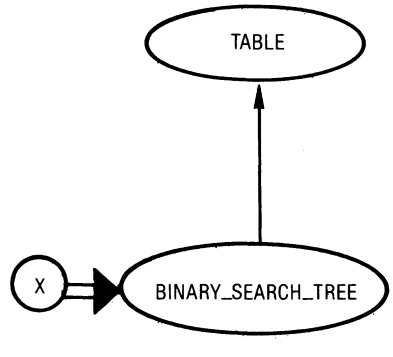


Figure 1. The client and inheritance relations in abstract data types. The client is represented by a horizontal double arrow; inheritance by a single, vertical arrow.

also holds in the case of multiple inheritance, as Figure 2 shows. This property is extremely important because it allows program entities to take different forms at runtime. The relation between *C* and *A* is an instance of the so-called Is-a relation (every lily *is a* flower; every binary search tree *is a* table).

The powerful combination of object-oriented design and these two relations — client and inheritance — is a key element in achieving extendibility and reusability.

An illustrative example

A new example, a full-screen entry system, will help contrast the object-oriented approach with classical functional decomposition. The example, a common data processing problem, should be interesting on its own: The problem is to write an interactive application that guides the user with full-screen panels at each stage.

The problem. Interactive sessions for such systems go through a series of states, each with a well-defined general pattern: A panel is displayed with questions for the user; the user supplies the required answer; the answer is checked for consistency (questions are asked until an acceptable answer is supplied); and the answer is processed somehow (a database is updated, for example). Part of the user's answer is a choice of the next steps; the system translates the user's choice into a transition to another state, and the same process is applied in the new state.

Figure 3 shows a panel for an imaginary airline reservation system. The screen shown is toward the end of a step; the user's answers are in italics.

The process begins in some initial state and ends whenever any among a set of final states is reached. A transition graph, like that in Figure 4, shows the overall structure of a session — the possible states and the transitions between them. The edges of the graph are labeled by numbers that correspond to the user's possible choices for the next step.

Our mission is to come up with a design and implementation for such applications that have as much generality and flexibility as possible.

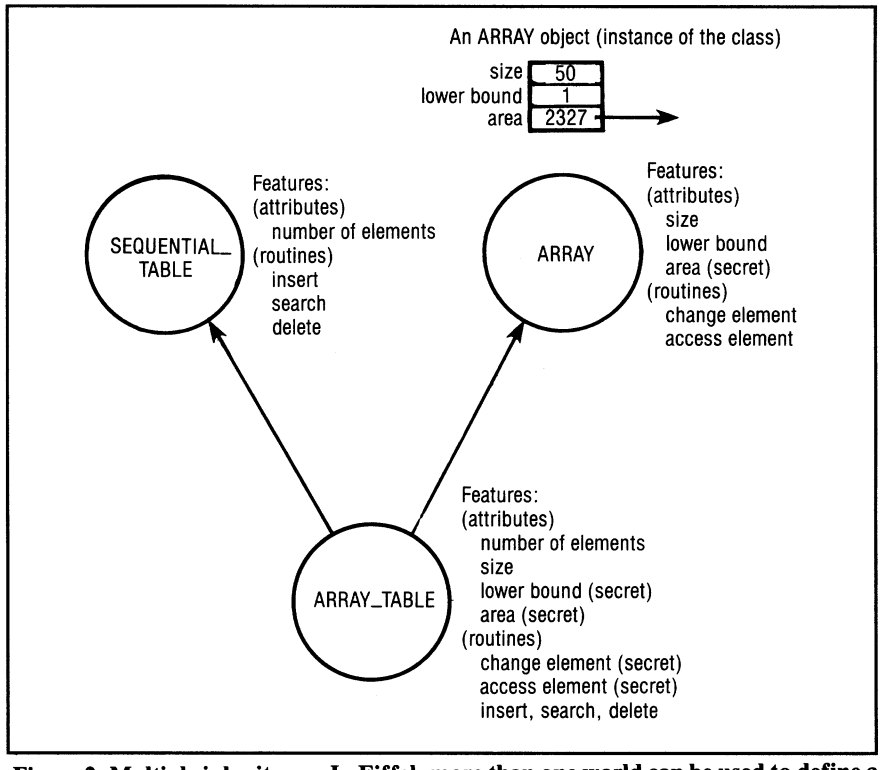


Figure 2. Multiple inheritance. In Eiffel, more than one world can be used to define a new world, which will inherit all the properies and add its own.

Enquiry on Flights

Flight sought from: *Santa Barbara* to: *Paris*
Departure on or after: *Nov 21* on or before: *Nov 21*

Preferred airline(s):
Special requirements:

AVAILABLE FLIGHTS: 1
Flt# AA 42 Dep 8:25 Arr 7:45 Thru Chicago

Choose next action:

0 - Exit
1 - Help
2 - Further enquiry
3 - Reserve a seat

Figure 3. A panel for an interactive airline reservation system. The screen shown is toward the end of a step; the user's answers are in italics.

A simple-minded solution. We'll begin with a straightforward, unsophisticated program scheme. This version is made of a number of blocks, one for each state of the system: $B_{Enquiry}$, $B_{Reservation}$, $B_{Cancellation}$, and so on. A typical block looks like:

```
B_Enquiry:
  output "enquiry on flights" panel;
  repeat
    read user's answers and choice C for
    next step ;
    if error in answer then
    output appropriate message
  end until not error in answer end;
  process answer;
  case C in
    C0:goto Exit,
    C1:goto Bhelp,
    C2:goto BReservation,
    ...
  end
```

(And similarly for each state.)

This structure will do the job, but of course there is much to criticize. The numerous goto instructions give it that famous spaghetti bowl look. This may be viewed as a cosmetic issue, solved by restructuring the program to eliminate jumps. But that would miss the point.

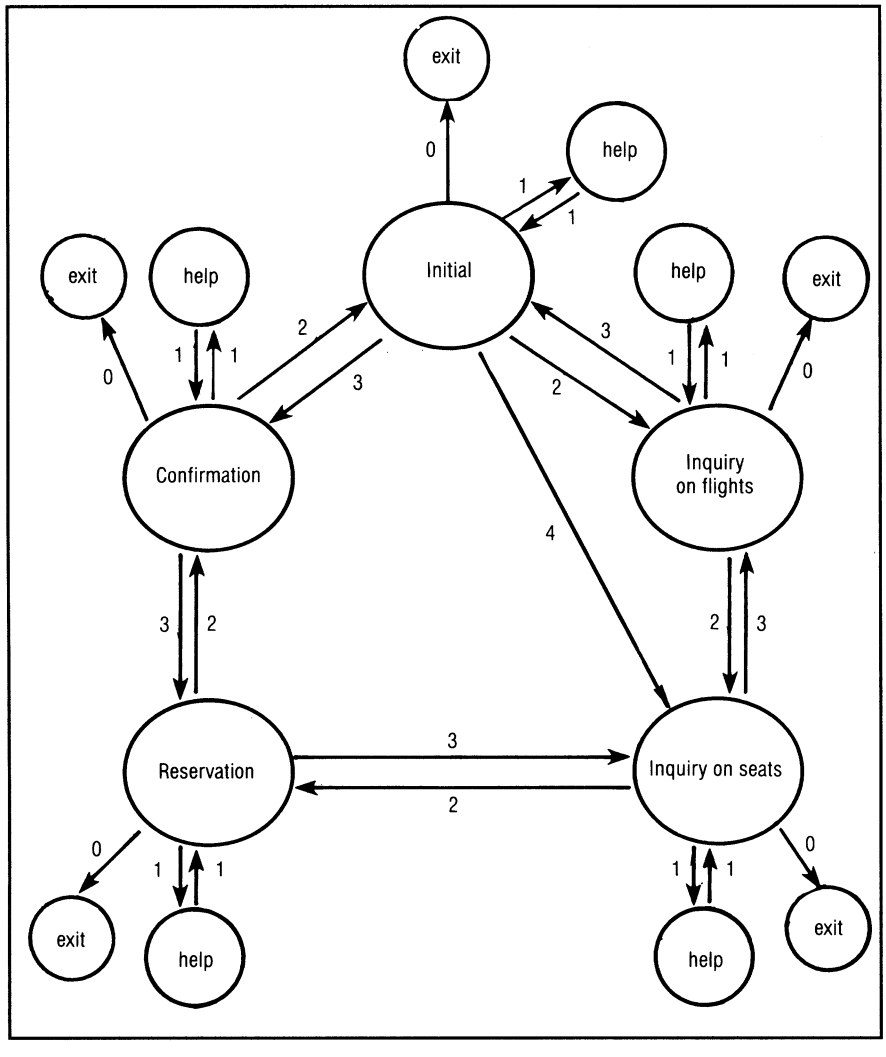


Figure 4. A state transition graph for an interactive application. The edges of the graph are labeled by numbers that correspond to the user's possible choices for the next step.

The problem is deeper. This program is bad not just because it has a lot of explicit branch instructions, but because the physical form of the problem has been wired into it. The branching structure of the program reflects exactly the transition structure of the graph in Figure 4.

This it terrible from a reusability and extendability standpoint. In real-world data-entry systems, the graph of Figure 4 might be quite complex — one study mentions examples with 300 different states.[5]

It is highly unlikely that the transition structure of such a system will be right the first time it is designed. Even after the first version is working, users will inevitably request new transitions, shortcuts, or help states. The prospect of modifying the whole program structure (not just program elements — the overall organization) for any one change is horrendous.

To improve on this solution we must separate the graph structure from the traversal algorithm. This seems appropriate because the structure depends on the particular interactive application (airline reservation), while its traversal is generic. As a side benefit, a functional decomposition will also remove the heretical gotos.

A procedural, "top-down" solution. We may encapsulate the graph structure in a two-argument function, Transition, such that Transition(s,c) is the state obtained when the user chooses c on leaving state s.

We use the word "function" in a mathematical sense: Transition may be represented either by a function in the programming sense (a routine that returns a value) or by a data structure, such as an array. The first solution may be preferable for readability because the transitions will appear in the program code itself. The second is better for flexibility because it is easier to change a data structure than a program. We can afford to postpone this decision.

The function transition is not sufficient to describe the transition graph. We must also define the state, initial, that begins the traversal and a Boolean-valued function isfinal(s) that determines when a state is final. Initial and final states are treated dissymmetrically; while it is reasonable to

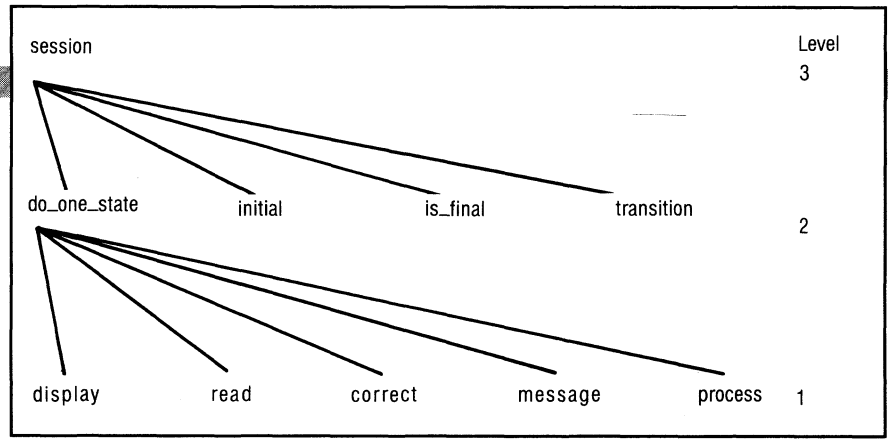


Figure 5. An orthodox, functional architecture for an interactive application. The "top," or main program, is the routine Session.

expect the dialog to always begin in the same state, we cannot expect it to always end in the same state.

Figure 5 shows the orthodox, functional architecture derived from this solution. As the top-down method teaches, this system has a "top," or main program. What else could it be but the routine that describes how to execute a complete interactive session?

This routine may be written to emphasize application-independence. Assume that a suitable representation is found for states (type STATE) and for the user's choice after each state (CHOICE):

```
session is
   -- execute a complete session of
   -- the interactive system
   current: STATE ; next: CHOICE
begin
   current := initial ;
   repeat
      do_one_state (current, next) ;
   -- the value of next is returned by routine
   -- do_one_state
      current := transition (current, next)
   until is_final (current) end
end -- session
```

This procedure does not show direct dependency upon any interactive application. To describe such an application, we must provide three of the elements on level two in Figure 5: a transition function (routine or data structure), an initial state, and an is_final predicate.

To complete the design, we refine the do_one_state routine, which describes the actions to be performed in each state. The body of this routine is essentially an abstracted form of the blocks in our spaghetti version:

```
do_one_state (in s : STATE ; out c : CHOICE) is
      --execute the actions associated with
      --state s,
      --returning into c the user's choice
      --for the next state
   a: ANSWER ; ok: BOOLEAN ;
begin
   repeat
      display(s) ; read(s,a) ;
      ok := correct(s,a) ;
      if not ok then message(s,a) end
   until ok end ;
   process (s,a) ; c := next_choice (a)
end -- do_one_state
```

For the remaining routines, we can only give a specification, because the implementations depend on the details of the application and its various states: display(s) outputs the panel associated with state *s*; read(*s,a*) reads the user's answer to state *s* into *a*; correct(*s,a*) returns true if and only if *a* is an acceptable answer; if it is, process(*s,a*) processes answer *a*; if it isn't, message(*s,a*) outputs the relevant error message.

Type ANSWER is left unspecified. A value of this type, say *a*, globally represents the input entered by the user in a given state, including the user's choice of the next step, next_choice(*a*).

Data transmission. Is this solution satisfactory? Not from the standpoint of reusability.

True, we did separate what is generic and what is specific to a particular application, but this does not buy much flexibility. The main problem is the system's data transmission structure. Consider the functionalities (types of arguments and results) of the routines:

```
do_one_state: (in s: STATE ; out c: CHOICE)
display: (in s: STATE)
read: (in s: STATE ; out a: ANSWER)
correct: (in s: STATE ; a: ANSWER):
BOOLEAN
message: (in s: STATE, a: ANSWER)
process: (in s: STATE, a: ANSWER)
```

All these routines share the state *s* as a parameter, coming from the top module Session (where it is known as Current). The flow of data, illustrated in Figure 6, shows that (as a conservative economist might say) there's far too much state intervention. As a result, all the above routines must perform some form of case discrimination on *s*:

```
case s of
   State1:....,
   .....,
   Staten:....,
end
```

This implies long, complex code (a problem which could be solved with further decomposition) and (more annoying) it means that every routine must deal with, and thus know about, all possible states of the application. This makes it very difficult to implement extensions. Adding a new state, for example, entails modifications throughout. Such a situation is all too common in software development. System evolution becomes a nightmare as simple changes touch off a complex chain reaction in the system.

The situation is even worse than it appears. It would seem desirable to profit from the similar aspects of these types of interactive applications by storing the common parts in library routines. But this is unrealistic in the solution above: On top of the explicit parameters, all routines have an implicit one — the application itself, airline reservations.

A general-purpose version of display, for example, should know about all states of all possible applications in a given environment! The function transition should contain the transition graph for all applications. This is clearly impossible.

The law of inversion. What is wrong? Figure 6 exposes the flaw: there is too much data transmission in the software architecture. The remedy, which leads directly to object-oriented design, may be expressed by the following law: If there is too much data transmission in your routines, then put your routines into your data.

Instead of building modules around operations (session, do_one_state) and distributing data structures between the resulting routines, object-oriented design does the reverse. It uses the most important data structures as the basis for modularization and attaches each routine to the data structure to which it applies most closely.

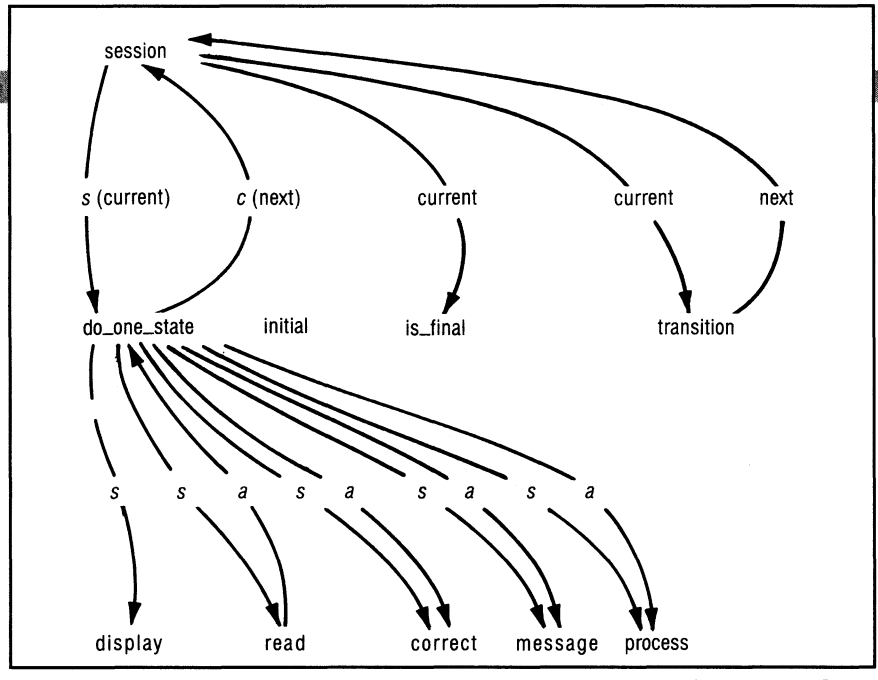


Figure 6. Data transmission in the architecture derived from with the top-down approach.

This law of inversion is the key to turning a functional decomposition into an object-oriented design: reverse the viewpoint and attach the routines to the data structures. To programmers trained in functional approaches, this is as revolutionary as making the Sun orbit Earth.

Of course, it's best to design in an object-oriented fashion from the beginning. However, the process of moving from a functional decomposition to an object-oriented structure is itself interesting. How do we find the most important data structures, around which modules are to be built?

Data transmission provides a clue. The data structures that are constantly transmitted between routines must be important, mustn't they?

Here the first candidate is obvious, the state (current, *s*). So our object-oriented solution will include a class STATE to implement the corresponding abstract data type. Among the features of a state are the five routines of level one in Figure 5 that describe the operations performed in a state (display, read, message, correct, and process), and the routine do_one_state without the state parameter.

In Eiffel notation, the class STATE may be written:

```
class STATE export
    next_choice, display, read, correct,
    message, process, do_one_state
feature
    user_answer: ANSWER ;
    next_choice: INTEGER ;
    do_one_state is
        do
            ...body of the routine...
        end;
    display is ... ;
    read is... ;
    correct: BOOLEAN is... ;
    message: is... ;
    process is... ;
end -- class STATE
```

The features of the class include two attributes, next_choice and user_answer, and six routines. Routines are divided into functions, which return a result (like correct, which returns a Boolean value) and procedures, which don't.

The export clause is used for information hiding: In a client class containing a declaration *s:* STATE, a feature application *s.f* is only correct if *f* is one of the features listed in the clause. Here all features are exported, except for user_answer, which is accessible by STATE routines but not by the outside world. A nonexported feature is said to be secret. As before, we assume that type ANSWER is declared elsewhere, now as a class. Values that represent exit choices are coded as integers.

Unlike its counterpart in a functional decomposition, each routine has no explicit STATE parameter. The state to which routines apply reappears in calls made by clients:

```
s: STATE; b:BOOLEAN;
choicecode: INTEGER;
s.do_one_state ; s.read ;
b := s.correct ;
choicecode := s.next_choice;
etc.
```

We have also replaced the ANSWER parameter in level-one routines with the secret attribute user_answer. Information hiding is the motive — client code doesn't need to look at answers except through the interface provided by the exported features.

Inheritance and deferred features. There's a problem, however. How can we write the class STATE without knowing the properties of a specific state? Routine do_one_state and attribute next_choice are the same for all states, but display is not.

Inheritance is the key to this problem. At the STATE level we know (1) all details of routine do_one_state, (2) the attribute next_choice; (3) the fact that routines like display must exist and (4) what their functionalities are.

So we write the class and define these partially known routines as *deferred*. This means that, while any actual state must have them, their details are postponed to descendant classes that describe specific states. (The notion of deferred routines come from Simula 67, where they are called "virtual.") Thus the class is written:

```
class STATE export
    next_choice, display, read, correct,
    message, process, do_one_state

feature
    user_answer: ANSWER ;
        -- secret attribute
    next_choice: INTEGER ;

    do_one_state is
            -- execute the actions associated
            -- with the current state
            -- and assign to next_choice the
            -- user's choice for the next state
        local
            ok: BOOLEAN
        do
            from
                ok:= false
            until
                ok
            loop
                display ; read ; ok := correct ;
                if not ok then
                    message
                end
            end ; -- loop
            process
        ensure
            correct
        end; -- do_one_state

    display is
        -- display the panel associated
        -- with current state
```

```
  deferred
  end ; -- display
  read is
    -- return the user's answer
    -- into user_answer
    -- and the user's next choice
    -- into next_choice
    deferred
    end ; -- read

  correct: BOOLEAN is
    -- return true if and only if
    -- user_answer is
    -- a correct answer
    deferred
    end; -- correct

  message is
    -- output the error message
    -- associated with user_answer
    require
      not correct
    deferred
    end ; -- message

  process is
    -- process user_answer
    require
      correct
    deferred
    end -- process
end -- class STATE
```

Note the syntax of the Eiffel loop, with initialization in the from clause and the exit test in the until clause. This is equivalent to a while loop, with an exit test rather than a continuation test.

Also note the require clauses that appear at the beginning of routines message and process. These clauses introduce preconditions that must be obeyed whenever a routine is called. Similarly, a postcondition, introduced by the keyword ensure, may be associated with a routine. Preconditions and postconditions express the precise effect of a routine. They can also be monitored at runtime for debugging and control.

The class just described does not by itself describe any actual states — it expresses the pattern common to all states. Specific states are defined by descendants of STATE. It is incumbent on these descendants to provide actual implementations of the deferred routines, such as:

```
class ENQUIRY_ON_FLIGHTS export....
inherit
  STATE
feature
  display is
    do
      ...specific display procedure...
    end ;
      ...and similarly for read, correct,
      message, and process...
end -- class ENQUIRY_ON_FLIGHTS
```

Several important comments are in order:

- We have succeeded in separating — at the exact grain of detail required — the elements common to all states from those specific to individual states. Common elements are concentrated in STATE and need not be redeclared in descendants of STATE like ENQUIRY_ON_FLIGHTS.
- If *s* is an object of type STATE and *d* an object of type DS, where DS is a descendant of STATE, the assignment *s* := *d* is permitted and *d* is acceptable whenever an element of type STATE is required. For example, the array Transition introduced below to represent the transition graph of an application may be declared of type STATE and filled with elements of descendant types.
- This goes beyond Ada-style separation of interface and implementation. First, an Ada interface may contain only bodiless routines; it corresponds to a class where all routines are deferred. In Eiffel, however, you may freely combine nondeferred and deferred routines in the same class. Even more important, Eiffel allows any number of descendant types of STATE to coexist in the same application, whereas Ada allows at most one implementation per interface. This openness of classes (a class may always be extended by new descendants) is a fundamental advantage of object-oriented languages over the closed modules of such languages as Ada and Modula-2.
- The presence of preconditions and postconditions in Eiffel maintains the conceptual integrity of a system. Just as a deferred routine must be defined in descendant classes, so must we define the constraints such a definition must observe. This is why a precondition and postcondition may be associated with a routine even in a deferred declaration. These conditions are then binding on any actual definition of the routine in a descendant of the original class. The technique is paramount in

Object-oriented languages.

Other languages implement the concept of object-oriented programming with inheritance and would allow solutions to our airline reservation system example, in a manner similar to the one given here in Eiffel.

These include Simula, the father of all object-oriented language, object-oriented expressions of C such as Objective C and C++, and an extension of Pascal, Object Pascal. These four languages, however, support only single inheritance. Other object-oriented languages include Smalltalk and extensions of Lisp such as Loops, Flavors, and Ceyx. The Clu language shares some of the properties of these languages, but does not offer inheritance.

In recent years, many languages have been added to the above list, mostly for exploratory programming and artificial intelligence purposes.

Bibliography

Birstwistle, G., et al., *Simula Begin*, Studentliteratur and Auerbach Publishers, Berlin, 1973.

Bobrow, D.G, and M.J. Sefik, "Loops: An Object-Oriented Programming System for Interlisp," Xerox PARC, Palo Alto, Calif., 1982.

Booch, G., "Object-Oriented Software Development," *IEEE Trans. Software Eng.*, Feb. 1986, pp. 211-221.

Cannon, J.I., "Flavors," MIT Artificial Intelligence Lab, Cambridge, Mass., 1980.

Cox, B., *Object-Oriented Programming: An Evolutionary Approach*, Addison-Wesley, Reading, Mass., 1986.

Goldberg, A., and D. Robson, *Smalltalk-80: The Language and Its Implementation*, Addison-Wesley, Reading, Mass., 1983.

Hulot J.-H., "Ceyx, Version 15: 1 — Une Initiation," Tech. Report 44, INRIA, Paris, 1984.

Liskov, et al., *Clu Reference Manual*, Springer Verlag, Berlin-New York, 1981.

Stroustrup, B., *The C++ Programming Language*, Addison-Wesely, Menlo Park, Calif., 1986.

Tesler, L., "Object Pascal Report," *Structured Language World*, 1985.

More on Eiffel

The Eiffel language is part of an environment developed by the author and his colleagues at Interactive Software Engineering. It is accompanied by a design method, a library, and a set of supporting tools. It promotes reusability, extendibility, and software construction by a combination of flexible modules.

The Eiffel library provides the basic building blocks: a set of classes implementing some of the most important data structures and associated operations.

Inheritance plays a central role in this approach. The language supports multiple inheritance, used heavily in the basic library; we have found single inheritance to be insufficient. (Repeated inheritance, not described here, is also supported.) The use of inheritance is made safe and practical with renaming and redefinition techniques. Type parameterization (genericity) is also available.

Inheritance and genericity are powerful techniques for building reusable, extendable software. Their very power entails a risk of misuse. To enhance correctness and reliability, Eiffel includes primitives for systematic software construction: class and loop invariants, and routine preconditions and postconditions, all of which describe semantic constraints imposed on classes and their features.

These constraints (which may be monitored at runtime to help in debugging) must be obeyed by any redefinition of the features in descendant classes, thus preserving the semantic consistency of descendants and helping to control the scope of the inheriance mechanism.

Eiffel is a typed language, where all type checking may be done statically. The language and method are intended for the development of sizable software systems; thus the implementation, which uses C as an intermediate language, emphasizes efficiency. Access to any feature of an object (as in *a.f*) always takes constant time, despite the possiblities for overloading provided by multiple inheritance, renaming, redefinition and genericity (which imply that the version of *f* to be applied depends on the runtime form of *a*). Also, the code for a routine is not duplicated in classes which inherit the routine, even in the presence of multiple inheritance and genericity.

Because the emphasis is on the incremental development of large systems, the Eiffel translator supports separate compilation, class by class. Automatic configuration management is provided, so that each needed module is always used in an up-to-date version (necessary recompilations, and these only, being automatically triggered by the system). The implementation includes a set of supporting tools, in particular for automatic memory management, execution tracing, symbolic debugging, and documentation. The implementation is currently available on Unix systems.

The language and method are described in "Eiffel: A Language and Environment for Software Engineering," to appear in the Journal of Systems and Software, and "Eiffel: Programming for Reusability and Extendibility," SIGPlan Notices, 1987.

using Eiffel as design language: A design module will be written as a class with deferred routines, whose effects are characterized by pre- and postconditions.

A complete system. The final step in our example is to adapt the routine that was at the top of the functional decomposition: session. But we should be a little wiser by now.

The top of the top-down method is mythical. Most real systems have no such thing, especially if the top is meant to be a routine — *the* function of the system. Large software systems perform many functions, all equally important. Again, the abstract data type approach is more appropriate because it considers the system as an abstract entity capable of rendering many services.

In this case the obvious candidate is the notion of application: a specific interactive system like the airline reservation system. It makes sense to associate with this concept a full-fledged abstract data type that will yield a class, say INTERACTIVE_APPLICATION, at the design and implementation stages. For, although INTERACTIVE_APPLICATION will include as one of its features the routine session describing the execution of an applicaton, there are other things we may want to do with an application, all of which may be added incrementally to class APPLICATION.

By renouncing the notion of "main program" and seeing session as just one feature of the class INTERACTIVE_APPLICATION, we have added considerable flexibility.

The class is given in Figure 7. Its principal features include the remaining elements at levels two and three in Figure 5. The following implementations decisions have been made:

- The transition function is represented by a two-dimensional array, Transition, of size $n \times m$, where n is the number of states and m the number of possible exit choices.
- States are numbered 1 to n. An auxiliary, one-dimensional array, associated_state, yields the state corresponding to any integer.
- The number of the initial state is set by the routine Choose_initial and kept in the attribute Initial_number. The convention for final states is a transition to pseudo-state 0; normal states have positive numbers.

The class includes a Create procedure that will be executed on object initialization. As in most object-oriented languages, objects are created dynamically. If *a* is declared of type *C*, the instruction *a*.Create creates an object of type *C* and associates it with *a*.

The Create procedure and its parameters makes it possible to execute specific initialization actions on creation, instead of initializing the new object with standard default values.

The procedure Create of class INTERACTIVE_APPLICATION itself uses the Create procedures of library classes ARRAY and ARRAY2, which allocate arrays dynamically within the bounds given as parameters. For example, a two-dimensional array may be created by *a*.Create (1,25, 1,10).

Classes ARRAY and ARRAY2 also include features Entry and Enter for array access and modification. Other features of an array are its bounds, upper and lower for a one-dimensional array, and so on.

These classes are declared as ARRAY[T] and ARRAY2[T], an example of Eiffel classes with generic parameters, in this case the type of array elements. Many fundamental classes in the Eiffel library (lists, trees, stacks) are generic. With Eiffel a programmer can combine genericity with inheritance in a type-safe manner.[6]

Class INTERACTIVE_APPLICATION uses Eiffel assertions, an aspect of the language designed to emphasize correctness and reliability. Assertions express formal properties of program elements. They may appear in preconditions and postconditions, the loop invariants, and the class invariants.

Such constructs are used primarily to ensure correct program designs and to document the correctness of arguments, but they may also be used as checks at run-time. More profoundly, assertions (especially pre- and postconditions and class invariants) bring the formal properties of abstract data types back into classes.

An interactive application will be represented by an entity, air_reservation, declared of type INTERACTIVE_APPLICATION and initialized by

```
air_reservation.Create (number_of_states,
number_of_possible_choices)
```

The states of the application must be defined separately as entities, declared of descendants STATE, and created. Each state *s* is assigned a number *i*:

```
air_reservation.enter_state(s,i)
```

One state, i_0, is chosen as the initial:

```
air_reservation.choose_initial(i0)
```

Each successive transition (from state number *sn* to state number *tn*, with label *l*) is entered by:

```
air_reservation.enter_transition(sn,tn,l)
```

This includes exits, for which *tn* is 0. The application may now be executed by air_reservation.session.

The same routines can by used during system evolution to add a new state, a new transition, and so on. The class may be extended, of course (either by itself or through descendants).

Multiple inheritance. This example exposes many of the principles in Eiffel, except the concept of multiple inheritance. A previous article on the same example[7,8] relied on Simula 67, which supports only single inheritance. Multiple inheritance is another concept that is essential to a practical use of object-oriented design and programming.

Multiple inheritance makes it possible to define a class as heir to more than one other class, thus combining the features of several previously defined environments. Multiple inheritance would be essential, for example, to implement a satisfactory solution to the table-management problem, detailed in the box on p. 62.

```
classINTERACTIVE_APPLICATION export
  session, first_number, enter_state,
  choose_initial, enter_transition, ...
feature
  transition: ARRAY2 [STATE] ; associated_state: ARRAY [STATE] ;
  -- secret attributes
  first_number; INTEGER ;
  Create (n,m:INTEGER) is
  -- allocate application with n states and m possible choices
    do
      transition.Create (1,n,1,m) ;
      associated_state.Create (1,n)
    end; -- Create
  session is -- execute application
    local
      st: STATE ; st_number: INTEGER ;
    do
      from
        st_number:=first_number ;
      invariant
        0 ≤ next; next ≤ n
      until st_number = 0 loop
        st:=associated_state.entry(st_number) ;
        st.do_one_state ;
        st_number:=transition.entry(st_number, st.next_choice)
      end -- loop
    end; -- session
  enter_state (s: STATE ; number: INTEGER) is
  -- enter state s with index number
    require
      1 ≤ number;
      number ≤ associated_state.upper
    do
      associated_state.enter (number,s)
    end; -- enter_state
  choose_initial (number: INTEGER) is
  -- define state number number as the initial state
    require
      1 ≤ number;
      number ≤ associated_state.upper
    do
      first_number:=number
    end; -- choose_initial
  enter_transition (source: INTEGER ; target:INTEGER ; label: INTEGER) is
  -- enter transition labeled "label" from state number source
  -- to state number target
    require
      1 ≤ source; source ≤ associated_state.upper ;
      0 ≤ target; target ≤ associated_state upper ;
      1 ≤ label; label ≤ transition.upper2 ;
    do
      transition.enter (source, label, target)
    end -- enter_transistion
  ...other features...
  invariant
    0 ≤ st_number ; st_number ≤ n ;
    transition.upper1 = associated_state.upper ;
end -- class INTERACTIVE_APPLICATION
```

Figure 7. The class INTERACTIVE_APPLICATION.

A table-searching module

It is impossible to give, in one article, a satisfactory solution to the problem of designing a general-purpose table-searching module. But we can outline how Eiffel would be applied to that case.

First, it is obvious that we are talking not about a table-searching module, but about a module for table management. In fact, we're talking just about the table as an abstract data type with operations such as search, insert, delete, and so on.

As with STATE, the most general notion of table will be represented by a class with deferred routines. The various kinds of tables are descendants of this class. To obtain them, an in-depth analysis of the notion of table and its possible implementations is required. Such an analysis and the associated design and implementation effort are a considerable endeavor, especially as you realize that there is not a single notion of table, but a network of related notions.

The inheritance mechanism can help express the structure of this network and capture differences and similarities at the exact grain of detail required. For example, we may have a descendant of TABLE, SEQUENTIAL_TABLE, that covers tables stored sequentially in arrays, linked lists, or files, with a version of the function search (*x*):

```
from
   restart
until
   off_limits or else current_value = x
invariant
   -- x does not appear in the table before current postion
loop
   move_forth
end ;
if off_limits ...(etc.)
```

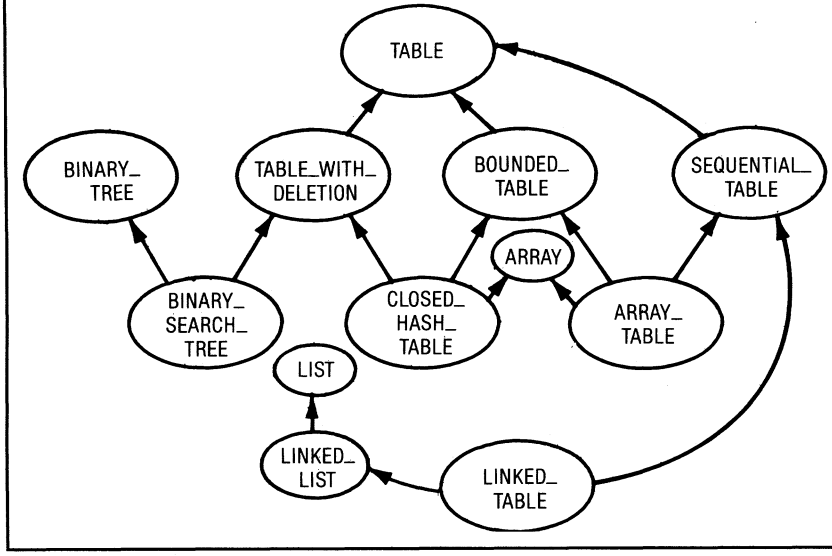
The structure is similar to that of STATE, where the essential routine do_one_state was not deferred, but was expressed in terms of other deferred routines. In this case, search is not deferred but uses deferred routines for which the descendants of TABLE must provide implementations.

What's remarkable is that an entity *t* declared of type TABLE may dynamically refer to an object of any descendant type of TABLE; however the call *t*.search(*x*) may be written without any knowledge of what kind of table implementation *t* will actually be at runtime.

This approach captures — at the exact grain of detail required — the commonality within a family of implementations of the same data abstractions. A family will consist of a header class (SEQUENTIAL_TABLE) and specific descendants (ARRAY_TABLE).

The inheritance graph may span more than one level. Features common to all members of the family (like Search for sequential tables) are concentrated at the header level and shared; features unique to various members are deferred in the header and expanded on in the individual members. The diagram below illustrates this inheritance structure.

Both genericity and multiple inheritance are essential to this problem's solution: All table classes take the type of table elements as a generic parameter, and several will combine two or more parent classes (BINARY_SEARCH_TREE from both BINARY_TREE and TABLE).



Even in the previous example, multiple inheritance is not far away — if we had defined a data abstraction WINDOW to describe screen panels, some descendants of STATE might inherit from this class, too.

In lieu of conclusion

This article has promoted the view that, if one accepts that reusability is essential to better software quality, the object-oriented approach — defined as the construction of software systems as structured collections of abstract data type implementations — provides a promising set of solutions.

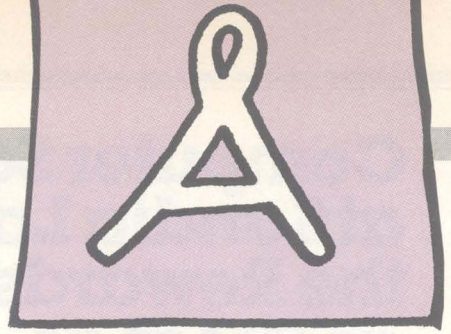
One epithet this approach certainly does not deserve is "top-down." It is puzzling to see this adjective used almost universally as a synonym for "good." Top-down design may be an effective method for developing individual algorithms and routines. But applying it at the system level is inappropriate unless the system can be characterized by a single, frozen, top-level function, a case that is rare in practice.

One epithet this approach certainly does not deserve is "top-down."

More importantly, top-down design goes against the key factor of software reusability because it promotes one-of-a-kind developments, rather than general-pupose, combinable software elements.

It is surprising to see top-down design built in as an essential requirement in the US Dept. of Defense directive MIL-STD-2167, which by the sheer power of its sponsor is bound to have a serious (and, we fear, negative) influence for years to come.

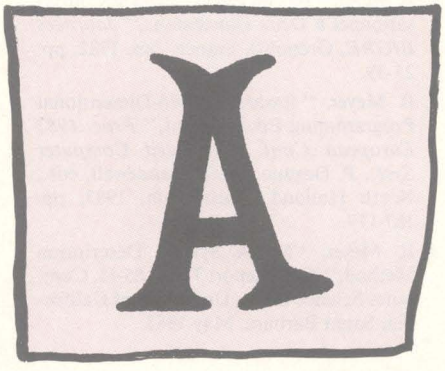
Of course, the bottom-up method promoted here does not mean that system design should start at the lowest possible level. What it implies is construction of systems by reusing and combining existing software. This is a bootstrapping approach in which software elements are progressively combined into more and more ambitious systems.



As a consequence of this approach, there is no notion of main program in Eiffel. Classes are meant to be developed separately. Integrating those classes into an Eiffel "system" is the last, and least binding, decision.

A system is a set of classes with one distinguishing element, the root. The only role of the root is to initiate the execution of the system (by creating an object of the root type and executing its create procedure). A system has no existence as Eiffel construct; it is simply a particular assembly of classes.

Such an approach is viable only if there are adequate facilities to produce flexible software elements and combine them effectively. The concepts of object-oriented design with multiple inheritance and genericity provide such facilities.



Object-oriented design means more than just putting data types into modules; the inheritance concept is essential. This requires an object-oriented language. As structured programming showed a few years ago, you can attempt to implement new methodological concepts without a language that directly supports them — but it will never be quite as good as the real thing: using the right language to implement the right concepts.

Another aspect of the approach promoted here is that it tends to blur the distinction between design and implementation. While this distinction may be unavoidable today, it is undesirable because it tends to introduce an artificial discontinuity in software construction. Design and implementation are essentially the same activity: constructing software to satisfy a certain specification. The only difference is the level of abstraction — during design certain details may be left unspecified, but in an implementation everything should be expressed in full. However, the process of filling in the details should be continuous, from system architecture to working program. Language constructs such as deferred features are particularly helpful in this process. Software development is made much smoother when you use a language that encompasses the traditional area of design and implementation, but that is no more difficult to master than conventional programming languages. Such is the aim of Eiffel.

We do not propose, however, to remove the boundary between design and implementation, on the one hand, and system specification on the other. These activities are of a different nature: specification states problems, design and implementation solve them. (A companion effort, the specification method M,[9] applies similar concepts to formal, nonexecutable specifications.)

One more fundamental theme has been guiding this discussion: the idea that today the essential problems of software engineering are *technical* problems.

Not everybody agrees. There is a large and influential school of thought that sees management, organization, and economic issues as the biggest obstacles to progress in software development. Programming aspects, in this view, are less important. This view is evidenced in many discussions of reusability that consider technical issues, such as the choice of programming language, as less important for reusability than such things as an easily accessible database of components.

As evidence of the limitations of the "managerial" approach to reusability, consider the relations that exist between these components, such as specialization (a hash table is a specialized table, a B-tree is a specialized tree). If an object-oriented language is used, they can be expressed directly by inheritance and recorded within the components themselves. But if the language does not provide direct support for expressing this relation, the information must be entered explicitly into the database, separate from the components.

Give your poor, your huddled projects a decent technical environment in the first place.* Then *worry about whether you are managing them properly.

This immediately raises some difficult issues: how to provide an adequate user interface, check the consistency of the relation information, and maintain the integrity of this information as components are updated. Advanced project-management techniques are required to solve these issues. This is a typical example of an organizational solution to a technical problem, with the resulting complexity and loss of effectiveness.

Overemphasis on management issues is premature. While it is indeed true that many software projects are plagued with management problems, focusing on these problems first confuses the symptom with cause. It's like expecting better hospital management to solve the public hygiene problem 10 years before Pasteur came along!

Give your poor, your huddled projects a decent technical environment in the first place. *Then* worry about whether you are managing them properly. □

Acknowledgments

The comments and suggestions made by the referees, those who agreed with the article's thesis and those who didn't, were much appreciated. The influence of Simula 67, the first object-oriented language and still one of the best, is gratefully acknowledged.

References

1. T.C. Jones, "Reusability in Programming: A Survey of the State of the Art," *IEEE Trans. Software Eng.* Sept. 1984, pp. 488-494.
2. M.A. Jackson, *System Development*, Prentice-Hall, Englewood Cliffs, N.J., 1983.
3. B.P. Lientz and E.B. Swanson, "Software Maintenance: A User/Management Tug of War," *Data Management*, April 1979, pp. 26-30.
4. B.W. Boehm, "Software Engineering — As It Is," *Proc. Fourth Int'l Conf. Software Eng.*, CS Press, Los Alamitos, Calif., Sept. 1979, pp. 11-21.
5. B. Dwyer, "A User-Friendly Algorithm," *Comm. ACM*, Sept. 1981, pp. 556-561.
6. B. Meyer, "Genericity versus Inheritance," *Proc. ACM Conf. Object-Oriented Programming Syst, Languages, and Applications*, ACM, New York, pp. 391-405 (revised version to appear in *J. Pascal, Ada, and Modula-2*).
7. B. Meyer, "Vers un Environnement Conversationnel à Deux Dimensions," *Journées BIGRE*, Grenoble, France, Jan. 1982, pp. 27-39.
8. B. Meyer, "Towards a Two-Dimensional Programming Environment," *Proc. 1982 European Conf. Integrated Computer Syst.*, P. Degano and R. Sandewall, eds., North Holland, Amsterdam, 1983, pp. 167-179.
9. B. Meyer, "M: A System Description Method," Tech. Report TRCS 85-15, Computer Science Dept., University of California, Santa Barbara, May 1985.

Bertrand Meyer is president of Interactive Software Engineering, Inc., a Santa Barbara, California, company specializing in tools and education for improving software quality. The Eiffel environment is one of Interactive's products.

From 1983 to 1986, he was a visiting associate professor at the University of California, Santa Barbara. Previously, he was division head at Electricité de France. He is a member of AFCET, the Computer Society of the IEEE, and ACM. He has engineering degrees from Polytechnique and Suptélécom, an MS from Stanford University, and the Dr. Sc. from the University of Nancy, France.