33

O-O programming and Ada

*I*n the nineteen-seventies, advances in programming methodology brought about a new generation of languages combining the control structures of Algol 60 and the data structuring constructs of Algol W and Pascal with better system structuring facilities and support for information hiding. Although their precise traits differ, these languages share a common spirit and may be collectively called the **encapsulation languages**. (They are also known in the literature as "object-based", a terminology that will be discussed in the next chapter.)

Although a complete list of encapsulation languages would be long, only a few have developed a sizable user community. Five deserve particular attention: **Modula-2**, a successor to Pascal designed at the Swiss Federal Institute of Technology by Niklaus Wirth, creator of Algol W, Pascal itself and (later) Oberon; **CLU**, developed at MIT under the direction of Barbara Liskov, which comes closest to realizing object-oriented concepts but lacks inheritance; **Mesa**, a Xerox effort with particular emphasis on describing inter-module relationships of large systems; **Alphard**, by Mary Shaw, William Wulf and Ralph London of Carnegie-Mellon University, which included an assertion mechanism; and **Ada**.

We will limit our study of how to approach O-O techniques in encapsulation languages to Ada, which, besides having attracted the most attention, is also the most complete (and complex) of these languages, embodying in some form most of the features found in the others. Modula-2, for example, does not offer genericity or overloading.

33.1 A BIT OF CONTEXT

Ada was a response to a crisis perceived in the mid-seventies by the software policy-makers of the US Department of Defense (DoD). They noted in particular that the various branches of the military were using more than 450 programming languages, many of them technically obsolete, gravely hampering contractor management, programmer training, technical progress, software quality and cost control.

Bearing in mind the successful precedent of COBOL (the result, in the late fifties, of a DoD call for a COmmon Business-Oriented Language), they put out successive versions of a Request For Proposals for a modern software engineering language capable of supporting embedded real-time applications. A first winnowing out of the several dozen initial responses led to four candidate designs, sealed and color-coded for fairness. The field was narrowed down to two, finally leading in 1979 to the selection of the Green language designed by Jean D. Ichbiah and his group at CII-Honeywell Bull in France

(today's Bull). Following a few years' experience with the first industrial implementations, the language was revised and made into an ANSI standard in 1983.

Ada (as Green was renamed) began a new era in language design. Never before had a language be subjected to such intense examination before being released. Never before (in spite of some valiant efforts by the PL/I team) had a language been treated like a large-scale engineering project. Working groups comprising the best experts in many countries spent weeks reviewing the proposals and contributed — in those pre-Internet days — reams of comments. Like Algol 60 a generation earlier, Ada redefined not just the language landscape but the very notion of language design.

A recent revision of Ada has yielded a new language, now officially called Ada 95, which will be described at the end of this chapter. In the rest of the discussion, as elsewhere in this book, the name Ada without further qualification refers to the preceding version, Ada 83, by far the most widely used today.

Has Ada been successful? Yes and no. The DoD got what it had commissioned: thanks to a rigorous implementation of the "Ada mandate", Ada became in a few years the dominant technical language in the various branches of the US military, and of the military establishment of some other countries too. It has also achieved significant use in such non-military government agencies as NASA and the European Space Agency. But except for some inroads in computing science education — aided in part by DoD incentives — the language has only had limited success in the rest of the software world. It would probably have spread more widely were it not for the competition of the very ideas described in this book: object technology, which burst into the scene just as Ada and the industry were becoming ripe for each other.

The careful observer of language history can detect two ironies here. The first is that the designers of Ada were well aware of O-O ideas; although this is not widely known, Ichbiah had in fact written one of the first compilers for Simula 67, the original O-O language. As he has since explained when asked why he did not submit an O-O design to the DoD, he estimated that in the competitive bidding context of Ada's genesis such a design would be considered so far off the mainstream as to stand no chance of acceptance. No doubt he was right; indeed one can still marvel at the audacity of the design accepted by the DoD. It would have been reasonable to expect the process to lead to something like an improvement of JOVIAL (a sixties' language for military applications); instead, all four candidate languages were based on Pascal, a language with a distinct academic flavor, and Ada embodied bold new design ideas in many areas such as exceptions, genericity and concurrency. The second irony is that the Ada mandate, meant to force DoD software projects to catch up with progress in software engineering by retiring *older* approaches, has also had in the ensuing years the probably unintended effect of slowing down the adoption of *newer* (post-Ada) technology by the military-aerospace community.

> The lessons of Ada remain irreplaceable, and it is a pity that many of the O-O languages of the eighties and nineties did not pay more attention to its emphasis on software engineering quality. However obvious, this comment is all the more necessary because the occasion for discussing Ada in this book is often to contrast some of its solutions with those of O-O development — as will again happen several times in this chapter. The resulting

critiques of Ada techniques should be viewed less as reproach than as homage to the precursor against which any new solution must naturally be assessed.

33.2 PACKAGES

See "Packages", page 90.

Each of the encapsulation languages offers a modular construct for grouping logically related program elements. Ada calls it a package; corresponding notions are known as modules in Modula-2 and Mesa, and clusters in CLU.

"Modules and types", page 170.

A class was defined as both a structural system component — a module — and a type. In contrast, a package is only a module. An earlier discussion described this difference by noting that packages are a purely *syntactic* notion, whereas classes also have a *semantic* value. Packages provide a way to distribute system elements (variables, routines …) into coherent subsystems; but they are only needed for readability and manageability of the software. The decomposition of a system into packages does not affect its semantics: one can transform a multi-package Ada system into a one-package system, producing exactly the same results, through a purely syntactical operation — removing all package boundaries, expanding generic derivations (as explained below) and resolving name clashes through renaming. Classes, for their part, are also a semantic construct: besides providing a unit of modular decomposition, a class describes the behavior of a set of run-time objects; this semantics is further enriched by polymorphism and dynamic binding.

An Ada package is a free association of program elements and may be used for various purposes. Sensible uses of this notion include writing a package to gather:

"Facility inheritance", page 832.

- A set of related constants (as with facility inheritance).
- A library of routines, for example a mathematical library.
- A set of variables, constants and routines describing the implementation of one abstract object, or a fixed number of abstract objects, accessible only through designated operations (as we will do in Fortran in the next chapter).
- An abstract data type implementation.

The last use is the most interesting for this discussion. We will study it through the example of a stack package, adapted from an example in the Ada reference manual.

33.3 A STACK IMPLEMENTATION

Information hiding is supported in Ada by the two-tier declaration of packages. Every package comes in two parts, officially called "specification" and "body". The former term is too strong for a construct that does not support any formal description of package semantics (in the form of assertions or similar mechanisms), so we will use the more modest word "interface".

The standard Ada term for "routine" is "subprogram". We keep the former for consistency with other chapters.

The interface lists the public properties of the package: exported variables, constants, types and routines. For routines it only gives the headers, listing the formal arguments and their types, plus the result type for a function, as in:

```
function item (s: STACK) return X;
```

The body part of a package provides the routines' implementations, and adds any needed secret elements.

A simple interface

A first version of the interface part of a stack package may be expressed as follows. Note that the keyword **package** by itself introduces a package interface; the body, which will appear later, is introduced by **package body**.

```
package REAL_STACKS is
    type STACK_CONTENTS is array (POSITIVE range <>) of FLOAT;
    type STACK (capacity: POSITIVE) is
        record
            implementation: STACK_CONTENTS (1..capacity);
            count: NATURAL := 0;
        end record;
    procedure put (x: in FLOAT; s: in out STACK);
    procedure remove (s: in out STACK);
    function item (s: STACK) return FLOAT;
    function empty (s: STACK) return BOOLEAN;
    Overflow, Underflow: EXCEPTION;
end REAL_STACKS;
```

This interface lists exported elements: the type *STACK* for declaring stacks, the auxiliary type *STACK_CONTENTS* used by *STACK*, the four basic routines on stacks, and two exceptions. Client packages will only rely on the interface (provided their programmers have some idea of the semantics associated with the routines).

This example suggests several general observations:

- It is surprising to see all the details of stack representation, as given by the declarations of types *STACK* and *STACK_CONTENTS*, appear in what should be a pure interface. We will see shortly the reason for this problem and how to correct it.
- Unlike the classes of object-oriented languages, a package does not by itself define a type. Here you must separately define a type *STACK*. One consequence of this separation, for the programmer who builds a package around an abstract data type implementation, is the need to invent two different names — one for the package and one for the type. Another consequence is that the routines have one more argument than their object-oriented counterparts: here they all act on a stack *s*, implicit in the stack classes given in earlier chapters.
- A declaration may define not only the type of an entity, but also its initial value. Here the declaration of *count* in type *STACK* prescribes an initial value of 0. It obviates

the need for an explicit initialization operation corresponding to creation; this would not be the case, however, if a less straightforward initialization were required.

- A few details of Ada are needed to understand the type declarations: *POSITIVE* and *NATURAL* denote the subtypes of *INTEGER* covering positive and non-negative integers, respectively; a type specification of the form **array** (*TYPE* **range** <>), where <> is known as the Box symbol, describes a template for array types. To derive an actual type from such a template, you choose a finite subrange of *TYPE*; this is done here in *STACK*, which uses the subrange *1..capacity* of *POSITIVE*. *STACK* is an example of a parameterized type; any declaration of an entity of type *STACK* must specify an actual value for *capacity*, as in

  s: *STACK* (*1000*)

- In Ada, every routine argument must be characterized by a mode: **in**, **out** or **in out**, defining the routine's rights on the corresponding actual arguments (read-only, write-only or update). In the absence of an explicit keyword, the default mode is **in**.

- Finally, the interface also specifies two exception names: *Overflow* and *Underflow*. An exception is an error condition that the programmer has decided to treat separately from the normal flow of control. The interface of the package should list any exceptions that may be raised by the package's routines and propagated to clients. More on the Ada exception mechanism below.

Using a package

Client code using the package is based on the interface. Here is an example from some package needing a stack of real numbers:

```
s: REAL_STACKS.STACK (1000);
REAL_STACKS.put (3.5, s); …;
if REAL_STACKS.empty (s) then …;
```

An Ada environment must be able to compile such client code even if only the interface of *REAL_STACKS*, not its body, is available.

Syntactically, note how each use of an entity from this package (where "entities" here include type names such as *STACK* as well as routine names) must repeat the name of package *REAL_STACKS*, using dot notation. This could become tedious, hence the need for a more implicit form of qualification. If you include the directive

```
use REAL_STACKS;
```

at the beginning of the client package, you may write the above extract more simply as

```
s: STACK (1000);
put (3.5, s); …;
if empty (s) then …;
```

You still need the full form, however, for any entity whose name conflicts with the name of another accessible to the client package (that is to say, declared in that package itself or in another supplier listed in a **use** directive).

Some of the Ada literature advises programmers to stay away from the **use** directive altogether on the grounds that it hampers clarity: an unqualified reference such as *empty* (*s*) does not immediately tell the reader what supplier *empty* comes from (*REAL_STACKS* in the example). The equivalent in the object-oriented approach, *s.empty*, unambiguously indicates the supplier through the type of *s*.

A similar problem does arise in the O-O world because of inheritance: when you see a name in a class, it may refer to a feature declared in any ancestor. But we saw a technique that solves this problem at least in part: the notion of flat form.

"FLATTENING THE STRUCTURE", page 541.

Implementation

The body of the *REAL_STACKS* package might be declared along the following lines. Only one routine is shown in full.

```
package body REAL_STACKS is
    procedure put (x: in FLOAT; s: in out REAL_STACK) is
        begin
            if s.count = s.capacity then
                raise Overflow
            end if;
            s.count := s.count + 1;
            s.implementation (count) := x;
        end put;
    procedure remove (s: in out STACK) is
            … Implementation of remove …
        end remove;
    function item (s: STACK) return X is
            … Implementation of item …
        end item;
    function empty (s: STACK) return BOOLEAN is
            … Implementation of empty …
        end empty;
end REAL_STACKS;
```

Two properties apparent in this example will be developed in more detail below: the use of exceptions to handle a run-time error by raising a special condition and treating it separately; and the need for the body to repeat most of the interface information (routine headers) that already appeared in the interface.

Genericity

The package as given is too specific; it should be made applicable to any type, not just *FLOAT*. To turn it into a generic package, use the following syntax:

```
generic
        type G is private;
package STACKS is
        … As before, replacing all occurrences of FLOAT by G …
end STACKS;
```

See appendix B.

The **generic** clause is heavier syntax than our O-O notation for generic classes (**class** *C* [*G*]…) because it offers more options. In particular, the parameters declared in a **generic** clause may represent not just types but also routines. The appendix on genericity *vs*. inheritance will discuss these possibilities.

The **generic** clause is not repeated in the package body, which will be identical to the version given earlier, except for the substitution of *G* for *FLOAT* throughout.

The **is private** specification directs the rest of the package to treat *G* as a private type. This means that entities of the type may only be used in operations applicable to all Ada types: use as source or target of an assignment, as operand of an equality test, as actual argument in a routine, and a few other special operations. This is close to the convention used for unconstrained formal generic parameters in our notation. In Ada, other possibilities are also available. In particular, you can restrict the operations further by declaring the parameter as **limited private**, which essentially bars all uses other than as actual argument to a routine.

Although called a package, a generically parameterized module such as *STACKS* is really a package template, since clients cannot use it directly; they must derive an actual package from it by providing actual generic parameters. We may define a new version of our stack-of-reals package through such a generic derivation:

```
package REAL_STACKS_1 is new STACKS (FLOAT);
```

Generic derivation is the principal Ada mechanism for adapting modules. It is somewhat inflexible, since you can only choose between generic modules (parameterized, but not directly usable) or usable modules (not extendible any more). In contrast, inheritance allows arbitrary extensions to existing modules, according to the Open-Closed principle. Appendix B pursues the comparison further.

33.4 HIDING THE REPRESENTATION: THE PRIVATE STORY

Package *STACKS*, as given, fails to implement the principle of information hiding: the declarations of types *STACK* and *STACK_CONTENTS* are in the interface, allowing clients to access the representation of stacks directly. For example, a client might include code of the form

[1]

```
use REAL_STACKS_1;…
s: STACK; …
s.implementation (3) := 7.0; s.last := 51;
```

grossly violating the underlying abstract data type specification.

Conceptually, the type declarations belong in the body. Why did we not put them there in the first place? The explanation requires that we look, beyond the language, at programming environment issues.

One requirement on the Ada design, already mentioned, was that it should be possible to compile packages separately and, moreover, to compile a client of any package *A* as soon as you have access to the interface of *A*, but not necessarily to its body. This favors top-down design: to proceed with the work on a module, it suffices to know the specification of the facilities it needs; actual implementations may be provided only later.

So if you have access to the interface of *REAL_STACKS_1* (that is to say, the interface of *STACKS*, of which *REAL_STACKS_1* is just a generic derivation) you must be able to compile one of its clients. Such a client will contain declarations of the form

```
use REAL_STACKS_1;…
s1, s2: STACK; …
s2 := s1;
```

which the poor compiler cannot properly handle unless it knows what size is taken up by an object of type *STACK*. But that can only be determined from the type declarations for *STACK* and the auxiliary type *STACK_CONTENTS*.

Hence the dilemma that faced the designers of Ada: conceptually, such declarations belong to the inferno — the body; but implementation concerns seem to require their inclusion in the paradise — the interface.

The solution retained was to create a purgatory: a special section of the package that is physically tied to the interface, and compiled with it, but marked in such a way that clients may not refer to its elements. The purgatory section is called the private part of the interface; it is introduced by the keyword **private** (also used, as we saw above, as a qualifier for protected types). Any declaration appearing in the private part is unavailable to clients. This scheme is illustrated by our final version of the stack package interface:

```
generic
        type G is private;
package STACKS is
        type STACK (capacity: POSITIVE) is private;
        procedure put (x: in G; s: in out STACK);
        procedure remove (s: in out STACK);
        function item (s: STACK) return G;
        function empty (s: STACK) return BOOLEAN;
        Overflow, Underflow: EXCEPTION;
```

```
    private
        type STACK_VALUES is array (POSITIVE range <>) of G;
        type STACK (capacity: POSITIVE) is
            record
                implementation: STACK_VALUES (1..capacity);
                count: NATURAL := 0;
            end record
    end STACKS;
```

Note how type *STACK* must now be declared twice: first in the non-private part of the interface, where it is only specified as **private**; then again in the private part, where the full description is given. Without the first declaration, a line of the form *s: REAL_STACK* would not be legal in a client, since clients only have access to entities declared in the non-private part. This first declaration only specifies the type as **private**, barring clients from accessing any property of stack objects other than universal operations such as assignment, equality test and use as actual argument. This is consistent with the discussion of information hiding.

Type *STACK_VALUES* is purely internal, and irrelevant to clients: so it need only be declared in the package body.

[1] *was on page 1085.*

Make sure to understand that the information in the private part should really be in the package body, and only appears in the package specification for reasons of language implementation. With the new form of *STACKS* client code such as [1], which directly accessed the representation in a client, becomes invalid.

See "common misunderstanding" discussed on page 52.

Authors of clients modules can *see* the internal structure of *STACK* instances, but they cannot take advantage of it in their modules. This can be tantalizing (although one may imagine that a good Ada environment could hide this part from a client author requesting interface information about the class, in the manner of the **short** tool of earlier chapters). While surprising to newcomers, the policy does not contradict the rule of information hiding: as was pointed out during the discussion of that rule, the goal is not physically to prevent client authors from reading about the hidden details, but to prevent them from *using* these details.

Someone who would like to make things sound very complicated could summarize by the following two sentences (to be spoken very quickly to impress friend and foe): The private section of the public part of a package lists the implementation of those conceptually private types which must be declared in the public part although their implementation is not publicly available. In the non-private part, these types are declared private.

33.5 EXCEPTIONS

The *STACKS* generic package lists two exceptions in its interface: *Overflow* and *Underflow*. More generally, you may deal with error conditions by defining arbitrary exception names; Ada also includes predefined exceptions, triggered by the hardware or the operating system, for such cases as arithmetic overflow or exhaustion of memory.

Some elements of the Ada exception mechanism were introduced in the chapter on exceptions, so that we can limit ourselves to a brief examination of how exceptions fit in the Ada approach to software construction.

"How not to do it — an Ada example", page 415.

Simplifying the control structure

Exceptions as they exist in Ada are a technique for dealing with errors without impairing the control structure of normal processing. If a program performs a series of actions, each of which may turn out to be impossible because of some erroneous condition, the resulting control structure may end up looking like

Like others in this chapter, this example follows Ada's use of the semicolon as an instruction terminator.

```
action1;
if error1 then
        error_handling1;
else
        action2;
        if error2 then
                error_handling2;
        else
                action3;
                if error3 then
                        error_handling3;
                else
                        …
```

The Ada exception mechanism is an effort to fight the complexity of such a scheme — where the elements that perform "useful" tasks sometimes look like a small archipelago in an ocean of error-handling code — by separating the handling of errors from their detection. There must still be tests to determine whether a certain erroneous condition has occurred; but the only action to take then is to raise a certain signal, the exception, which will be handled elsewhere.

Raising and handling an exception

To raise exceptions rather than handle errors in place, you may rewrite the extract as:

```
action1;
if error1 then raise exc1; end;
action2;
```

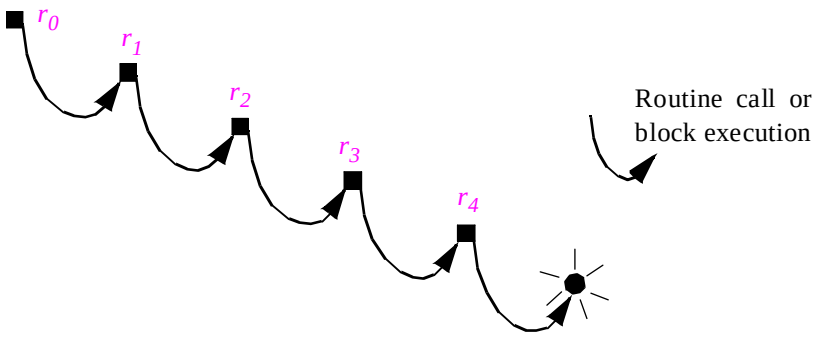
```
if error2 then raise exc2; end;
action3;
if error3 then raise exc3; end;
…
```

When an instruction **raise** *exc* is executed, control does not flow to the instructions that would normally follow, but is transferred to an **exception handler**. This disruption of the normal flow of control explains why the **else**… clauses are no longer necessary here. An exception handler is a special paragraph of a block or routine, of the form

```
exception
    when exc1, … => treatment1;
    when exc2 … => treatment2;
    …
```

The handler that a **raise** *exc* will select is the first one that handles *exc* in the dynamic chain, that is to say the list of units beginning with the routine or block containing the **raise** and continuing with its caller, its caller's caller etc.

The call chain
(*This figure originally appeared on page 418.*)



A handler is said to handle *exc* if *exc* appears in one of its **when** clauses (or it has a clause of the form **when others**). If there is such a handler, the corresponding instructions (after the => symbol) are executed and the enclosing routine returns control to its caller, or terminates if it is the main program. (Ada does have a notion of main program.) If no handler in the dynamic chain handles *exc*, execution terminates and control goes back to the operating system, which presumably will print out an error message.

Discussion

See chapter 12.

It is interesting to compare the Ada exception mechanism with the one developed in the chapter on exceptions earlier in this book. There are technical differences and differences of methodology.

The technical differences, apart from the different ways of discriminating between exceptions (multiple **when** clauses *vs.* inheriting from class *EXCEPTIONS*), involve retrying, which the O-O design considered sufficiently important to warrant a special instruction, whereas Ada has no direct support for it and requires **goto** instructions or similar control structures.

The methodological difference follows from the strong policy that we adopted, leading to the Disciplined Exception Handling principle that requires every exception handler, apart from the rare case of a "false alarm", to end in either **retrying** or official **failure** ("organized panic"). Ada is less strict in this respect, and we saw that as a consequence it is possible to misuse exceptions by executing a seemingly normal return without having handled the problem.

Page 417.

The need to avoid such dangerous situations led us to a basic rule, worth repeating:

Ada exception rule

The execution of any Ada exception handler should end by either executing a **raise** instruction or retrying the enclosing program unit.

Initially on page 416.

More generally, exceptions in the Ada spirit are control structures, helping to separate the handling of abnormal situations from their detection and hence to keep software structure simple. In practice, however, this hope is often disappointed.

When you write **raise** *some_exception*, you may have the impression of freeing yourself from the messy and boring task of taking care of strange cases, and instead concentrate on the core of the algorithm, handling normal cases. But raising an exception does not by itself solve the problem. Exceptions in the *STACKS* package are typical. An attempt to push an element into a full stack raises exception *Overflow*, and an attempt to access an element of an empty stack raises *Underflow*. How will you handle *Underflow*, the exception raised by a call to *remove* or *item* on an empty stack? As we saw in the discussion of Design by Contract, the routines themselves cannot reasonably contain a handler (*item* does not know what to do when applied to an empty stack); so the responsibility lies with the client, which should include code of the form

*The **raise** instructions appeared in REAL_STACKS, the initial variant of STACKS, page 1084.*

[2]

```
use REAL_STACKS;
procedure proc (…) is
        s: STACK; …
begin
        … remove (s); …
exception
        when Underflow => action1;
        …
end proc;
```

So the client must specify exactly what happens in the erroneous case. Omitting the **when** *Underflow* clause would be a design error. Compare this with the usual, non-exception-based form of the call (written in the syntax of the rest of this book):

[3]

```
if not s.empty then s.remove else action1 end
```

(or a variant which detects the error a posteriori). Form [2], using exceptions, differs from [3] in two aspects only:

"The a posteriori scheme", page 801.

- The code for handling the error, *action1*, is textually separate from the calls that may raise the error;
- Error handling is the same for all such calls if more than one.

See "Precondition design: tolerant or demanding?", page 355.

On the first point, although it is desirable to avoid the deeply nested **if**… **then**… **else**… error-handling structures cited at the beginning of this chapter, the place in the algorithm where an error is detected is often the one that has the best information to handle the error; and if you separate the two you may need to use complicated control structures for cases that require restarting or resuming processing.

On the second point, if a routine contains more than one call to *remove*, the way to deal with empty stacks will unlikely be the same in each case.

There are two general styles of exception usage: the **control structure style**, which views exceptions as a normal mechanism to handle all but the most common cases; and the **abnormal case style**, which reserves them for unpredictable situations, when all other mechanisms have failed. The **rescue/retry** approach described earlier in this book tends to favor the abnormal case style, although it can be used for the other style as well. Ada exception handling is more geared towards the control structure style.

You will decide for yourself which of the two styles you prefer; you should in any case remember, from this discussion and the earlier ones, not to place any naïve hope in the use of exceptions. With or without an exception mechanism, run-time errors are a fact of system life, which the software must handle explicitly. A good methodological approach, supported by an effective exception mechanism, can help; but some of the complexity is inherent to the problem of error handling, and no magical wand will make it go away.

33.6 TASKS

On CSP see "Communication-based mechanisms", page 979.

Besides packages, Ada offers another interesting modular construct: the task. Tasks are the basic Ada mechanism for handling concurrency; the underlying concurrency model is close to the CSP approach described in the concurrency chapter. But they also deserve a mention purely for their modular concepts, since they actually come closer than packages to supporting object-oriented concepts.

Syntactically, tasks share many aspects of packages. The main difference is that a task is not just a modular unit but the representation of a process, to be executed in parallel with other processes. So besides making up a syntactical unit it also describes a semantic component — unlike a package, and like a class.

Like a package, a task is declared in two parts, interface and body. Instead of routines, a task specification introduces a number of entries. To the client, entries look like procedures; for example, the interface of a buffer manager task may be

```
task BUFFER_MANAGER is
        entry read (x: out G);
        entry write (x: in G);
end BUFFER_MANAGER;
```

(Tasks may not be generic, so that type *G* has to be globally available, or a generic parameter of an enclosing package.) It is only the implementation of entries that distinguishes them from procedures: in the body, **accept** instructions will specify synchronization and other constraints on execution of the entries; here, for example, we might prescribe that only one *read* or *write* may proceed at any point in time, that *read* must wait until the buffer is not empty, and *write* until it is not full.

Besides individual tasks you may also specify a **task type**, and use it to create as many tasks — instances of the task type — as you need at run time. This makes tasks similar to classes, without inheritance. One can indeed conceive of an Ada realization of O-O concepts which would represent classes by task types and objects by their instances (perhaps even using **accept** instructions with different conditions to emulate dynamic binding). Because in sequential O-O computation we may expect classes to have many instances, this exercise is mostly of academic interest, given the overhead of creating a new process in current operating systems. Perhaps some day, in massively parallel hardware environments…

33.7 FROM ADA TO ADA 95

The Ada 95 version of the language is a major revision intended in particular to add O-O concepts. There is in fact no notion of class in the sense of this book (module *cum* type), but support for inheritance and dynamic binding for record types.

O-O mechanisms of Ada 95: an example

The package text at the top of the facing page illustrates some of the Ada 95 techniques; its meaning should be clear enough to a reader of this book. To derive a **new** type with more fields (the Ada 95 form of inheritance), you must have declared a type, such as *ACCOUNT*, as **tagged**; this of course contradicts the Open-Closed principle, since you must know in advance which types may have descendants and which may not. A **new** type may be derived from only one type; that is to say, there is no multiple inheritance. Note the syntax (**null record**, with, surprisingly, no **end**) for a derived type that adds no attribute.

Tagged types remain declared as records. The basic property of most O-O languages — that operations on a type become part of that type, and in fact, as we saw in the discussion of abstract data types, *define* the type — is not in force here: the routines that apply to a tagged type appear outside of its declaration, and take as argument a value of that type. (In languages generally recognized as object-oriented, *deposit* etc. would be part of the declaration of *ACCOUNT* and *compound* part of *SAVINGS_ACCOUNT*; they would not need their first arguments.) Here the only link between the routines and the type is that they must be declared as part of the same package; they do not even have to appear next to each other. Only the layout conventions, in the above example, indicate to the reader that certain routines are conceptually attached to certain tagged record types.

This is different from the usual view of O-O software construction. Although a tagged record type and the associated routines are, from a theoretical perspective, part of the same abstract data type, they do not form a syntactical unit — contradicting the Linguistic Modular Units principle, which suggested a close association between the modularizing concept and the syntactical structure.

"Linguistic Modular Units", page 53.

An Ada 95 package

The package may be better split off into three, with "child packages" for checking and savings accounts. See next page.

```
package Accounts is
    type MONEY is digits 12 delta 0.01;

    type ACCOUNT is tagged private;
        procedure deposit (a: in out ACCOUNT; amount: in MONEY);
        procedure withdraw (a: in out ACCOUNT; amount: in MONEY);
        function balance (a: in ACCOUNT) return MONEY;

    type CHECKING_ACCOUNT is new ACCOUNT with private;
        function balance (a: in CHECKING_ACCOUNT) return MONEY;

    type SAVINGS_ACCOUNT is new ACCOUNT with private;
        procedure compound (a: in out SAVINGS_ACCOUNT; period: in
Positive);

private
    type ACCOUNT is tagged
        record
            initial_balance: MONEY := 0.0;
            owner: String (1..30);
        end record;

    type CHECKING_ACCOUNT is new ACCOUNT with null record;

    type SAVINGS_ACCOUNT is new ACCOUNT with
        record
            rate: Float;
        end record;
end Accounts;
```

"OVERLOADING AND GENERICITY", 4.8, page 93.

The appearance of a new declaration for *balance* for *SAVINGS_ACCOUNT* signals a redefinition. Procedures *withdraw* and *deposit* are not redefined. As you will have recognized, this means that Ada 95 uses the *overloading* mechanism to obtain the O-O effect of *routine redefinition*. There is no syntactical mark (such as **redefine**) to signal a routine redefinition: to find out that function *balance* differs for *SAVINGS_ACCOUNT* from its base version in *ACCOUNT*, you must scan the text of the entire package. Here, of course, each routine version appears next to the corresponding type, with indentation to highlight the relationship, but this is a style convention, not a language rule.

A tagged type can be declared as **abstract**, corresponding to the notion of deferred class; you may make a routine **abstract** too instead of giving it a body.

A function returning a result of an abstract type must be abstract itself. This rule may seem strange at first, appearing to preclude writing an effective function returning, say, the top of a stack of figures, assuming *FIGURE* is abstract. In Ada, however, the result of such a function will typically be not of type *FIGURE* but of an "access type" describing references to instances of *FIGURE*. Then the function can be effective.

You can apply dynamic binding to entities of a tagged type, as in:

```
procedure print_balance (a: in ACCOUNT'Class) is
        -- Print current balance.
    begin
        Put (balance (a));
        New_Line;
    end print_balance;
```

You must request the dynamic binding explicitly by declaring the routine as a "classwide operation", as represented by the *'Class* qualification to the type of its argument; this is similar to the C++ obligation to declare any dynamically bound function as "virtual", except that here it is the client that must choose between static and dynamic binding.

"The C++ approach to binding", page 514.

Ada 95 allows you to define a "child package" *A.B* of an existing package *A*. This enables the new package to obtain features from *A* and add its own extensions and modifications. (This concept is of course close to inheritance — but distinct.) Instead of declaring the three account types in a single package as on the preceding page, it is indeed probably better to split the package into three, with *Accounts.Checking* introducing *CHECKING_ACCOUNT* and its routines, and *Accounts.Saving* doing the same for *SAVINGS_ACCOUNT*.

Ada 95 and object technology: an assessment

If you come to Ada 95 from a background in object technology, you will probably find the language befuddling at first. After a while, you should be able to master the various language mechanisms enabling you to obtain the effects of single inheritance, polymorphism and dynamic binding.

The price to pay, however, is complexity. To Ada 83, a sophisticated construction, Ada 95 has added a whole new set of constructs with many potential interactions both between themselves and with the old constructs. If you come from the O-O side and are used to the pristine simplicity of the notion of class, you will find that you have to learn the intricacies of at least five concepts, each covering some of the aspects of classes:

- Packages, which are modules but not types, can be generic, and offer something resembling inheritance: child packages (as well as a number of other options not detailed above, such as the possibility of declaring a child package as **private**).
- Tagged record types, which are types but not modules, and have a form of inheritance, although unlike classes they do not allow the syntactical inclusion of routines into a type declaration.
- Tasks, which are modules but not types and have no inheritance.
- Task types, which are modules and types, but cannot be generic (although they can be included in generic packages) and have no inheritance.
- "Protected types" (a notion we have not yet encountered), which are types and **may** include routines, as in

```
protected type ANOTHER_ACCOUNT_TYPE is
    procedure deposit (amount: in MONEY);
    function balance return MONEY;
private
    deposit_list: …; …
end ANOTHER_ACCOUNT_TYPE;
```

making them at first similar to classes — but with no inheritance.

The combination of interacting possibilities is mind-boggling. Packages, for example, still have, in addition to the notion of child package, the Ada mechanisms of **use** and **with**, with explanations such as this one from a tutorial text:

From [Wheeler-Web].

> *Private children are intended for "internal" packages that should only be "with'ed" by a restricted number of packages. A private child can only be "with'ed" by the body of its parent or by descendants of the private child's parent. In exchange for such a restrictive requirement, a private child gets a new authority: a private child's specification automatically sees both the public and private parts of all of its ancestors' specifications.*

No doubt it is possible to make sense of such explanations. But is the result worth the trouble?

> It is interesting to note that Jean Ichbiah, the creator of Ada, resigned publicly from the Ada 95 reviewing group after trying in vain for several years to keep the extensions simple. His long resignation letter includes comments such as: *A massive increase in complexity will result from 9X* [later renamed Ada 95] *adding one or more additional possibilities where Ada now offers two. For example, 9X adds*: […] *access parameters, to* **in**, **out**, *and* **in out**; *tagged types, to normal types*; *dispatched subprogram calls, to normal subprogram calls*; *use type clause, to use package clauses*; [Other examples skipped; overall 12 were included.] *With 9X, the number of interactions to consider is close to 60,000 since we have 3 or more possibilities in each case* (*that is,* 3^{10}).

The basic concepts of object technology, for all their power, are strikingly simple. Ada 95 may be the most ambitious effort so far to make them appear complicated.

Discussion: module and type inheritance

As a side observation following from this study of Ada 95, it is interesting to note that the Ada 95 design has found it necessary, along with the "inheritance" mechanism for tagged record types, to introduce the notion of child package. Ada, of course, has always kept module and type concepts separate, whereas classes are both. But then Ada 95 methodologists will suggest that when you introduce a descendant type such as *SAVINGS_ACCOUNT* you should declare it, for clarity and modularity, not in the original package (*Accounts*) but in a child package. If you generalize this advice, you will end up creating, along with the type hierarchy, a module hierarchy which follows it faithfully.

With the classes of object technology, such questions do not arise; classes being modules, there is by construction only one hierarchy.

The choices of Ada 95 show yet another example of the popular view that "*one should separate type inheritance from code reuse*". Instead the insight of object technology since Simula has been to *unify* concepts: module and type, subtyping and module extension. Like any other bold unification of notions theretofore considered completely distinct, this idea can be scary at times, hence the attempts to reintroduce the distinction. But they would deprive us of the remarkable simplification that O-O ideas have brought to the understanding of software architecture.

"ONE MECHANISM, OR MORE?", 24.6, page 833; *"The two styles", page 609*.

Towards an O-O Ada

That Ada 95 seems hard to teach and to manage does not mean the idea of making Ada more O-O is doomed; one should simply set reasonable goals and keep a constant concern for simplicity and consistency. The Ada community might try again to develop an object-oriented extension, which should be accompanied by the removal of a few facilities to keep the language size palatable. Two general directions are possible:

- The first idea, close in spirit to what the design of Ada 95 has attempted to achieve, is to keep the package structure and introduce a notion of class that would generalize Ada's record types, with support for inheritance and dynamic binding. But these should be true classes, including the applicable routines. Such an extension would be similar in principle to that which led from C to C++. It should strive for minimalism, trying to reuse as much as possible of the existing mechanisms (such as **with** and **use** for packages), rather than introducing new facilities which would then cause the interaction problems mentioned by Ichbiah.

- The other approach would build on an observation made in the presentation of tasks earlier in this chapter. It was noted then that task types are close in spirit to classes, since they may have instances created at run time; but structurally they have most of the properties of packages (visibility and information hiding rules, separate compilation). This suggests adding a modular unit that, roughly, has the syntax of packages and the semantics of classes; think of it as a package-class, or as a task type that does not need to be concurrent. The notion of "protected type" may be a starting point; but of course, it should be integrated into the existing mechanism.

A thesis by Mats Weber explores the idea of package types. See the link in www.adahome.com/Resources/Research/Research.html.

Exercises at the end of this chapter ask you (if, like many software people, you like dabbling in language design experiments, if only to gain a better understanding of existing languages and, through them, of software issues) to explore these possibilities further.

Exercises 33.4 and E33.5, page 1098.

33.8 KEY CONCEPTS INTRODUCED IN THIS CHAPTER

- Ada, studied as a representative of the class of "encapsulation languages" which also includes Modula-2, offers modular decomposition constructs: packages (and tasks).
- The emphasis is on information hiding: interface and implementation are declared separately.
- Genericity increases the flexibility of packages.
- Conflicts between methodological requirements and language implementation concerns give rise to the "private" section, a conceptually secret element that is syntactically included in the interface..
- The package is a purely syntactic mechanism. Modules remain distinct from types. No inheritance mechanism is possible.
- Exceptions separate error detection from error handling, but provide no magic solution to the problem of run-time errors.
- The Ada exception mechanism should only be used in a disciplined fashion; any execution of an exception handler should terminate by either retrying the operation or signaling failure to the caller.
- Task types could in principle be used to implement classes without inheritance, but this solution is not practical in most current environments.
- Ada 95 enables the definition of a new record type as being derived from an existing type, with support for routine redefinition, polymorphism and dynamic binding.

33.9 BIBLIOGRAPHICAL NOTES

[Booch 1986a] discusses (under the label "object-oriented design", but not using classes, inheritance, polymorphism etc.) how to obtain some of the benefits of object orientation in Ada.

The official reference on Ada is [ANSI 1983], recommended neither as bedtime reading nor as introductory material. Numerous books are available to fulfill the latter need.

References on the other modular languages mentioned at the beginning of this chapter are [Mitchell 1979] for Mesa, [Wirth 1982] for Modula-2, and [Liskov 1981] for CLU. See also [Liskov 1986] on programming methodology, based on CLU. The reference on Alphard is [Shaw 1981].

The Ada 95 reference manual is available on-line at [Ada 95-Web]. [Wheeler-Web] is an on-line tutorial (prelude to an announced book). For a commented list of Ada 95 textbooks, see [Feldman-Web]. I am greatly indebted to Richard Riehle and Magnus Kempe for clarifying a number of points about Ada 95; the views expressed are as usual my own. Magnus Kempe is the source of the reference to Mats Weber's thesis.

EXERCISES

E33.1 Winning the battle without privates

The Ada compilation problem that gives rise to the **private** construct might appear to affect object-oriented languages as well if the underlying environment supports separate compilation of classes. In fact, the problem seems to be worse because of inheritance: a variable declared of type *C* may at run time refer to instances not only of *C* but of any descendant class; since any descendant may add its own attributes, the size of these instances is variable. If *C* is a deferred class, it is not even possible to assign a default size to its instances. Explain why, in spite of these remarks, the object-oriented notation of this book does not need a language construct similar to the **private** mechanism of Ada. (**Hint**: your discussion should consider in particular the following notions: expanded *vs.* reference types; deferred classes; and the techniques used in our O-O framework to produce abstract class specifications without requiring class authors to write two separate module separate parts.) Discuss the tradeoffs involved in both solutions. Can you suggest other approaches to the problem in the Ada framework?

"HIDING THE REPRESENTATION: THE PRIVATE STORY", 33.4, page 1085.

E33.2 Generic routine parameters

Generic parameters to Ada packages may be not just types but also routines. Explain the relevance of this possibility to the implementation of object-oriented concepts, and its limitations. (See also appendix B.)

E33.3 Classes as tasks (for Ada programmers)

Rewrite class *COMPLEX* as an Ada task type. Show examples using the resulting type.

"Legitimate side effects: an example", page 759.

E33.4 Adding classes to Ada

(This language design exercise assumes a good knowledge of Ada 83.) Devise an adaptation of Ada (83) that keeps the notion of package but extends records to classes with polymorphism, dynamic binding and inheritance (single or multiple?), in line with general O-O principles.

E33.5 Package-classes

(This language design exercise assumes a good knowledge of Ada 83.) Using task types as inspiration, devise an adaptation of Ada (83) supporting packages that can be instantiated at run time and hence can play the role of classes, with polymorphism, dynamic binding and inheritance.