13

Supporting mechanisms

Except for one crucial set of mechanisms, we have now seen the basic techniques of object-oriented software construction. The major missing piece is inheritance and all that goes with it. Before moving to that last component of the approach, we should review a few mechanisms that will be important to the writing of actual systems: external routines and the encapsulation of non-O-O software; argument passing; control structures; expressions; string manipulation; input and output.

These are technical aspects, not essential to the understanding of the method; but we will need them for some later examples, and they blend well with the fundamental concepts. So even on your first reading you should spend some time getting at least generally familiar with them.

13.1 INTERFACING WITH NON-O-O SOFTWARE

So far, we have expressed software elements entirely in the object-oriented notation. But the software field grew up long before object technology became popular, and you will often need to interface your software with non-O-O elements, written in such languages as C, Fortran or Pascal. The notation should support this process.

We will first look at the language mechanism, then reflect on its broader significance as a part of the object-oriented software development process.

External routines

Our object-oriented systems are made of classes, consisting of features, particularly routines, that contain instructions. What is, among these three, the right level of granularity for integrating external software?

The construct must be common to both sides; this excludes classes, which exist only in object-oriented languages. (They may, however, be the right level of integration between two different O-O languages.) Instructions are too low-level; a sequence in which two object-oriented instructions bracket a C instruction:

Warning: this style is neither supported nor recommended. For purposes of discussion only.

```
!! x.make (clone (a))
(struct A) *x = &y; /* A piece of C */
x.display
```

would be very hard to understand, validate and maintain.

This leaves the feature level, the right one since encapsulating features is compatible with O-O principles: a class is an implementation of a data type protected by information hiding; features are the unit of interaction of the class with the rest of the software; since clients rely on the features' official specification (the short form) independently of their implementation, it does not matter to the outside world whether a feature is internally written in the object-oriented notation or in another language.

Hence the notion of external routine. An external routine will have most of the trappings of a normal routine: name, argument list, result type if it is a function, precondition and postcondition if appropriate. Instead of a **do** clause it will have an **external** clause stating the language used for the implementation. Here is an example, extracted from a class describing character files:

```
put (c: CHARACTER) is
            -- Add c to end of file.
        require
            write_open: open_for_write
        external
            "C" alias "_char_write";
        ensure
            one_more: count = old count + 1
        end
```

The **alias** clause is optional, useful only if the name of the external routine, in its language of origin, is different from the name given in the class. This happens for example when the external name would not be legal in the object-oriented notation, as here with a name beginning with an underscore (legal in C).

Advanced variants

The mechanism just described covers most cases and will suffice for the purposes of this book. In practice some refinements are useful:

- Some external software elements may be *macros* rather than routines. They will appear to the O-O world as routines, but any call will be expanded in-line. This may be achieved by varying the language name (as in "*C*:[**macro**]… ").
- It is also necessary to permit calls to routines of "Dynamic Link Libraries" (DLL) available on Windows and other platforms. Instead of being a static part of the system, a DLL routine is loaded at run time, on the first call. It is even possible to define the routine and library names at run time. DLL support should include both a way to specify the names statically (as in **external** "*C*:[**dll**]…") and a completely dynamic approach using library classes *DYNAMIC_LIBRARY* and *DYNAMIC_ROUTINE* which you can instantiate at run time, to create objects representing dynamically determined libraries and routines.
- You may also need communication in the reverse direction, letting non-O-O software create objects and call features on them. For example you may want the *callback mechanism* of a non-O-O graphical toolkit to call certain class features.

All these facilities are present in the O-O environment described in the last chapter. Their detailed presentation, however, falls beyond the scope of this discussion.

Uses of external routines

External routines are an integral part of the method, fulfilling the need to combine old software with new. Any software design method emphasizing reusability must allow accessing code written in other languages. It would be hard to convince potential users that reusability begins this minute and that all existing software must be discarded.

Openness to the rest of the world is a requirement for most software. This might be termed the **Principle of Modesty**: authors of new tools should make sure that users can still access previously available facilities.

External routines are also necessary to provide access to machine-dependent or operating system capabilities. The file class is a typical example. Another is class *ARRAY*, whose interface was presented in earlier chapters but whose implementation will rely on external routines: the creation procedure *make* use a memory allocation routine, the access function *item* will use an external mechanism for fast access to array elements, and so on.

"Polite society" is not classless.

This technique ensures a clean interface between the object-oriented world and other approaches. To clients, an external routine is just a routine. In the example, the C routine *_char_write* has been elevated to the status of a feature of a class, complete with precondition and postcondition, and the standard name *put*. So even facilities which internally rely on non-O-O mechanisms get repackaged in data abstractions; the rest of the object-oriented software will see them as legitimate members of the group, their lowly origins never to be mentioned in polite society.

Object-oriented re-architecturing

The notion of external routine fits well with the rest of the approach. The method's core contribution is architectural: object technology tells us how to devise the structure of our systems to ensure extendibility, reliability and reusability. It also tells us how to fill that structure, but what fundamentally determines whether a system is object-oriented is its modular organization. It is often appropriate, then, to use an O-O architecture — what is sometimes called a **wrapper** — around internal elements that are not all O-O.

One extreme but not altogether absurd way to use the notation would rely solely on external routines, written in some other language, for all actual computation. Object technology would then serve as a pure packaging tool, using its powerful encapsulation mechanisms: classes, assertions, information hiding, client, inheritance.

In general there is no reason to go that far, since the notation is perfectly adequate to express computations of all kinds and execute them as efficiently as older languages such as Fortran or C. But object-oriented encapsulation of external software is useful in several cases. We have seen one of them: providing access to platform-specific operations. Another is to address a problem that faces many organizations: managing so-called *legacy software*. During the sixties, seventies and eighties, companies have accumulated a legacy

of Cobol, Fortran, PL/I and C code, which is becoming harder and harder to maintain, and not just because the original developers are gone or going. Object technology offers an opportunity to re-engineer such systems by re-architecturing them, without having to rewrite them completely.

> Think of this process as the reverse of turkey stuffing: instead of keeping the structure and changing the internals, you keep the entrails and replace the skeleton, as if repackaging the content of a turkey into the bones of a zebra or a mouse. It must be noted, however, that such non-software applications of the idea appear neither useful nor appetizing.

This technique, which we may call **object-oriented re-architecturing**, offers an interesting solution for preserving the value of existing software assets while readying them for future extension and evolution.

It will only work, however, under specific conditions:

- You must be able to identify good abstractions in the existing software. Since you are not dealing with object-oriented software, they will typically be function abstractions, not data abstractions; but that is normal: it is your task to find the underlying data abstractions and repackage the old software's routines into the new software's classes. If you cannot identify proper abstractions already packaged in routines, you are out of luck, and no amount of object-oriented re-architecturing attempts will help.
- The legacy software must be of good quality. Re-architectured junk is still junk — possibly worse than the original, in fact, as the junkiness will be hidden under more layers of abstraction.

These two requirements are partly the same, since quality in software, O-O or not, is largely determined by quality of structure.

When they are satisfied, it is possible to use the **external** mechanism to build some very interesting object-oriented software based on earlier efforts. Here are two examples, both part of the environment described in the last chapter.

- The *Vision* library provides portable graphics and user interface mechanisms, enabling developers to write graphical applications that will run on many different platforms, with the native look-and-feel, for the price of a recompilation. Internally, it relies on the native mechanisms, used through external routines. More precisely, its lower level — WEL for Windows, MEL for Motif, PEL for OS/2 Presentation Manager — encapsulates the mechanisms of the corresponding platforms. WEL, MEL, PEL and consorts are also usable directly, providing developers who do not care about portability with object-oriented encapsulations of the Windows, Motif and Presentation Manager Application Programming Interfaces.

  On these libraries see "PORTABILITY AND PLATFORM ADAPTATION",

- Another library, *Math*, provides an extensive set of facilities for numerical computation in such areas as probability, statistics, numerical integration, linear and non-linear equations, ordinary differential equations, eigenproblems, fitting and interpolation, orthogonal factorizations, linear least squares, optimization,

special functions, Fast Fourier Transforms and time series analysis. Internally, it is based on a commercial subroutine library, the NAG library from Nag Ltd. of Oxford, but it provides a completely object-oriented interface to its users. The library hides the underlying routines and instead is organized around such abstract concepts as integrator, matrix, discrete function, exponential distribution and many others; each describes "objects" readily understandable to a mathematician, physicist or economist, and is represented in the library by a class: *INTEGRATOR*, *BASIC_MATRIX*, *DISCRETE_FUNCTION*, *EXPONENTIAL_DISTRIBUTION*. The result builds on the quality of the external routines — NAG is the product of hundreds of person-years of devising and implementing numerical algorithms — and adds the benefits of O-O ideas: classes, information hiding, multiple inheritance, assertions, systematic error handling through exceptions, simple routines with short argument lists, consistent naming conventions.

These examples are typical of how one can combine the best of traditional software and object technology.

The compatibility issue: hybrid software or hybrid languages?

Few people would theoretically disagree with the principle of modesty and deny the need for some integration mechanism between O-O developments and older software. The matter becomes more controversial when it comes to deciding on the level of integration.

See chapter 35.

A whole set of languages — the best known are Objective-C, C++, Java, Object Pascal and Ada 95 — have taken the approach of adding O-O constructs to an existing non-O-O language (respectively C in the first three cases, Pascal and Ada). Known as *hybrid languages*, they are discussed in varying degree of detail in a later chapter.

The integration technique described above, relying on external routines and object-oriented re-architecturing, follows from a different principle: that the need for *software* compatibility does not mean that we should burden the *language* with mechanisms that may be at odds with the principles of object technology. In particular:

- A hybrid adds a new language level to the weight of an existing language such as C. The result can be quite complex, limiting one of the principal attractions of object technology — the essential simplicity of the ideas.
- Beginners as a result often have trouble mastering a hybrid language, since they do not clearly see what is truly O-O and what comes from the legacy.
- Some of the older mechanisms may be incompatible with at least some aspects of object-oriented ideas. We have seen how the type concepts inherited from C make it hard to equip C++ environments with garbage collection, even though automatic memory management is part of the appeal of object technology. There are many other examples of clashes between the C or Pascal type system and the O-O view.

- The non-O-O mechanisms are still present, often in apparent competition with their higher-level object-oriented counterparts. For example C++ offers, along with dynamic binding, the ability to choose a function at run time through arithmetic on function pointers. This is disconcerting for the non-expert who lacks guidance on which approach to choose in a particular case. The resulting software, although compiled by an O-O environment, is still, deep-down, C code, and does not yield the expected quality and productivity benefits — giving object technology a bad name through no fault of its own.

If the aim is to obtain the best possible software process and products, compromising at the language level does not seem the right approach. *Interfacing* object-oriented tools and techniques with previous achievements is not the same thing as *mixing* widely different levels of technology.

> With the usual precautions about attaching too much weight to a metaphor, we can think of the precedent of electronics. It is definitely useful to combine different technology levels in a single system, as in an audio amplifier which still includes a few diodes together with transistors and integrated circuits. But the levels remain separate: there is little use for a basic component that would be half-diode, half-transistor.

O-O development should provide compatibility with software built with other approaches, but not at the expense of the method's power and integrity. This is what the **external** mechanism achieves: separate worlds, each with its own consistency and benefits, and clear interfaces between these worlds.

13.2 ARGUMENT PASSING

One aspect of the notation may require some clarification: what may happen to values passed as arguments to routines?

Consider a routine call of the form

$r\ (a_1, a_2, \ldots, a_n)$

corresponding to a routine

$r\ (x_1: T_1, x_2: T_2, \ldots, x_n: T_n)$ **is** …

where the routine could be a function as well as a procedure, and the call could be qualified, as in $b.r\ (\ldots)$. The expressions a_1, a_2, …, a_n are called actual arguments, and the x_i are called formal arguments. (Recall that we reserve the term "parameter" for generic type parameters.)

The relevant questions are: what is the correspondence between actual and formal arguments? What operations are permitted on formal arguments? What effect will they have on the corresponding actuals? For all three we should stick to simple and safe rules.

We already know the answer to the first question: the effect of actual-formal argument association is the same as that of a corresponding assignment. Both operations are called **attachments**. For the above call we can consider that the routine's execution starts by executing instructions informally equivalent to the assignments

$x_1 := a_1; x_2 := a_2; \ldots x_n := a_n$

On the second question: within the routine body, any formal argument x is protected. The routine may not apply to it any direct modification, such as:

- An assignment to x, of the form $x := \ldots$
- A creation instruction with x as its target: !! x.*make* (…)

Readers familiar with the passing mechanism known as *call by value* will note that the restriction is harsher here: with call by value, formals are initialized to actuals but may then be the target of arbitrary operations.

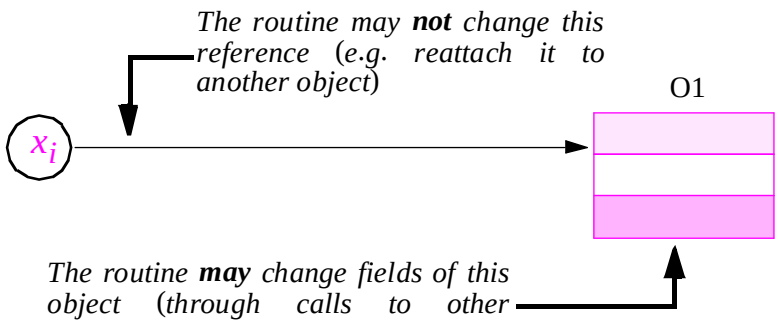
"ATTACHMENT: REFERENCE AND VALUE SEMANTICS", 8.8, page 261, in particular table on page 264.

The answer to the third question — what can the routine actually do to the actuals? — follows from the use of attachment to define the semantics of actual-formal association Attachment means copying either a reference or an object. As you will remember from the discussion of attachment, this depends on whether the types involved are expanded:

- For reference types (the more common case), argument passing will copy a reference, either void or attached to an object.
- For expanded types (which include in particular the basic types: *INTEGER*, *REAL* and the like), argument passing will actually copy an object.

In the first case, the prohibition of direct modification operations means that you cannot modify the **reference** through reattachment or creation; but if the reference is not void you can modify the attached **object** through appropriate routines.

Permissible operations on a reference argument



If x_i is one of the formal arguments to routine r, the body of the routine could contain a call of the form

$x_i.p\ (\ldots)$

where p is a procedure applicable to x_i, meaning a procedure declared in the base class of x_i's type T_i. This routine may modify the fields of the object attached to x_i at execution time, which is the object attached to the corresponding actual argument a_i.

So although a call *q* (*a*) can never change the value of *a* — the corresponding object if *a* is expanded, the reference otherwise — it can, in the reference case, change the attached object.

There are many reasons for not permitting routines to modify their arguments directly. One of the most striking is the *Conflicting Assignments To Actual* trick. Assume a language that permits assignments to arguments, and a procedure

WARNING: invalid routine text. For purposes of illustration only.

```
dont_I_look_innocuous (a, b: INTEGER) is
        -- But do not trust me too much.
    do
        a := 0; b := 1
    end
```

Then consider the call *dont_I_look_innocuous* (*x*, *x*) for some entity *x*. What is the value of *x* on return: 0 or 1? The answer depends on how the compiler implements formal-to-actual update on routine exit. This has fooled more than a few Fortran programmers, among others.

On constant attributes see chapter 18.

Permitting argument-modifying routines would also force us to impose restrictions on actual arguments: the actual corresponding to a modifiable formal must be an element that can change its value (a writable entity); this allows variable attributes, but not constant attributes, *Current*, or general expressions such as *a* + *b*. By precluding argument-modifying routines we can avoid imposing such restrictions and accept any expression as actual argument.

As a consequence of these rules, there are only three ways to modify the value of a reference *x*: through a creation instruction !! *x*…; through an assignment *x* := *y*; and through a variant of assignment, assignment attempt *x* ?= *y*, studied in a later chapter. Passing *x* as actual argument to a routine will never modify *x*.

This also means that a routine returns at most one result: none if it is a procedure; the official result (represented in the routine's body by the entity *Result*) if it is a function. To achieve the effect of multiple results, you can either:

- Use a function that returns an object with several fields (or more commonly a reference to such an object).
- Use a procedure that sets several fields of an object, corresponding to attributes that the client may then query.

See chapter 23, especially "The a posteriori scheme", page 800.

The first technique is appropriate when the result is truly made of several components; a function may not for example return two values corresponding to the title and publication year of a book, but it may return a single value of type *BOOK*, with attributes *title* and *publication_year*. The second technique is applicable for a routine that, besides its principal job, sets some status indicators. We will study it, as well as the more general question of *side effects*, in the discussion of module design principles.

13.3 INSTRUCTIONS

The object-oriented notation developed in this book is imperative: we specify computations through commands, also called instructions. (The word "statement" is commonly used in this sense but we will steadfastly avoid it since it is misleading: a statement is an expression of facts, not a command.)

Except for some specific properties of loops, intended to make their verification easier, instructions will look familiar to anyone who has had some experience with a modern language of the Algol line such as Pascal, Ada or Modula, or even just with C or a derivative. They include: Procedure call; Assignment; Creation; Conditional; Multi_branch; Loop; Check; Debug; Retry; Assignment attempt.

Procedure call

A routine call involves a routine, possibly with actual arguments. In a call instruction, the routine must be a procedure; if it is a function, the call is an expression. Although for the moment we are interested in instructions, the following rules apply to both cases.

A call is either qualified or unqualified. An unqualified call to a routine of the enclosing class uses the current instance as target; it appears under the form

r (without arguments), or

r (*x*, *y*, …) (with arguments)

A qualified call explicitly names its target, denoted by an expression: if *a* is an expression of a certain type, *C* is the base class of that type, and *q* is one of the routines of *C*, then a qualified call is of the form *a*.*q*. Again, *q* may be followed by a list of actual arguments; *a* may be an unqualified function call with arguments, as in *p* (*m*).*q* (*n*) where the target is *p* (*m*). You may also use as target a more complex expression, provided you enclose it in parentheses, as in (*vector1* + *vector2*).*count*.

Multidot qualified calls, of the form $a.q_1.q_2 \ldots .q_n$ are also permitted, where *a* as well as any of the q_i may include a list of actual arguments.

Export controls apply to qualified calls. Recall that a feature *f* declared in a class *B* is **available** to a class *A* if the feature clause declaring *f* begins with **feature** (without further qualification) or **feature** {*X*, *Y*, …} where one of *X*, *Y*, … is *A* or an ancestor of *A*. Then:

Qualified Call rule

A qualified call of the form $b.q_1.q_2 \ldots .q_n$ appearing in a class *C* is valid only if it satisfies the following conditions:

R1 • The feature appearing after the first dot, q_1, must be available to *C*.

R2 • In a multidot call, every feature after the second dot, that is to say every q_i for $i > 1$, must also be available to *C*.

To understand the reason for the second rule, note that *a*.*q*.*r*.*s* is a shorthand for

```
b := a.q; c := b.r; c.s
```

which is only valid if *q*, *r* and *s* are all available to *C*, the class where this fragment appears. Whether *r* is available to the base class of *q*'s type, and *s* available to the base class of *r*'s type, is irrelevant.

As you will remember it is also possible to express calls in infix or prefix form; an expression such as *a + b* is a different syntax for a call that would otherwise be written *a.plus (b)*. The same validity rules apply to such expressions as to the dot form.

See "Operator features", page 187.

Assignment

The assignment instruction is written

```
x := e
```

where *x* is a writable entity and *e* an expression of compatible type. A writable entity is either:

- A non-constant attribute of the enclosing class.
- A local entity of the enclosing routine, including *Result* for a function.

Other, non-writable kinds of entity include constant attributes (introduced in declarations such as *Zero*: *INTEGER* **is** *0*) and formal arguments of a routine — to which, as we just saw, the routine may not assign a new value.

Chapter 18 discusses constant attributes.

Creation instruction

The creation instruction was studied in an earlier chapter in its two forms: without a creation procedure, as in !! *x*, and with a creation procedure, as in !! *x.p (…)*. In both cases, *x* must be a writable entity.

See "The creation instruction", page 232 and "CREATION PROCEDURES", 8.4, page 236. A variant will be seen in "Polymorphic creation", page 479.

Conditional

A conditional instruction serves to specify that different forms of processing should be applied depending on certain conditions. The basic form is

```
if boolean_expression then
        instruction; instruction; …
else
        instruction; instruction; …
end
```

where each branch may have an arbitrary number of instructions (possibly none).

This will execute the instructions in the first branch if the *boolean_expression* evaluates to true, and those in the second branch otherwise. You may omit the **else** part if the second instruction list is empty, giving:

```
if boolean_expression then
        instruction; instruction; …
end
```

When there are more than two relevant cases, you can avoid nesting conditional instructions in **else** parts by using one or more **elseif** branches, as in

```
if c1 then
        instruction; instruction; …
elseif c2 then
        instruction; instruction; …
elseif c3 then
        instruction; instruction; …
…
else
        instruction; instruction; …
end
```

where the **else** part remains optional. This avoids the repeated nesting of

```
if c1 then
        instruction; instruction; …
else
        if c2 then
                instruction; instruction; …
        else
                if c3 then
                        instruction; instruction; …
                        …
                else
                        instruction; instruction; …
                end
        end
end
```

For handling a set of cases defined by the possible values of a certain expression, the multi-branch **inspect**, studied next, may be more convenient than the plain conditional.

The object-oriented method, in particular through polymorphism and dynamic binding, tends to reduce the need for explicit conditional and multi-branch instructions by supporting an implicit form of choice: you apply a feature to an object, and if the feature has several variants the right one automatically gets selected at run time on the basis of the object's type. When applicable, this implicit style is usually preferable. But of course some of your algorithms will still require explicit choice instructions.

Multi-branch

The multi-branch (also known as a Case instruction because of the corresponding keyword in Pascal, where it was first introduced based on a design by Tony Hoare) discriminates between a set of conditions that are all of the form $e = v_i$ where x is an expression and the v_i are constants of the same type. Although a conditional instruction (**if** $e = v_1$ **then** …

elseif $e = v_2$ **then**…) would do the job, two reasons justify a special instruction, departing from the usual rule that if the notation offers one good way to do something it does not need to offer two:

- This case is so common as to justify specific syntax, which will enhance clarity by avoiding the useless repetition of "$e =$".
- Compilers can use a particularly efficient implementation technique, the *jump table*, not applicable to general conditional instructions and avoiding explicit tests.

For the type of the discriminated values (the type of e and the v_i), the multi-branch instruction only needs to support two possibilities: integers and booleans. The rule will indeed be that e and the v_i must be declared as either all *INTEGER* or all *CHARACTER*. The general form of the instruction is:

```
inspect
        e
when v1 then
        instruction; instruction; …
when v2 then
        instruction; instruction; …
…
else
        instruction; instruction; …
end
```

All the v_i values must be different. The **else**… part is optional. Each of the branches may have an arbitrary number of instructions, possibly none.

The effect of the instruction is the following: if the value of e is equal to one of the v_i (this can be the case for at most one of them), execute the instructions in the corresponding branch; otherwise, execute the instructions in the **else** branch if any.

If there is no **else** branch and the value of e does not match any of the v_i, the effect is to raise an exception (of code *Incorrect_inspect_value*). This policy may seem surprising, since the corresponding conditional instruction would simply do nothing in this case. But it highlights the specificity of the multi-branch. When you write an **inspect** with a set of v_i values, you should include an **else** branch, empty or not, if you are prepared for run-time values of e that match none of the v_i. If you do not include an **else**, you are making an explicit statement: that you expect the value of e always to be one of the v_i. By checking this expectation and raising an exception if it is not met, the implementation is providing a service. Doing nothing would be the worst possible response, since this case usually reflects a bug (forgetting a possible case to be handled in its own specific way), which in any case should be fixed as early as possible.

A typical application of the multi-branch is to decode a single-character user input:

This is an elementary scheme. See chapter 21 for more sophisticated user command processing techniques.

```
inspect
        first_input_letter
when 'D' then
        "Delete line"
when 'I' then
        "Insert line"
…
else
        message ("Unrecognized command; type H for help")
end
```

"UNIQUE VALUES", 18.6, page 654. Do and Si are also known as Ut and Ti.

In the integer case, the v_i can be Unique values, a concept detailed in a later chapter. This makes it possible to define a number of abstract constants, in a declaration such as *Do, Re, Mi, Fa, Sol, La, Si*: *INTEGER* **is unique**, and then discriminate among them in an instruction such as **inspect** *note* **when** *Do* **then**… **when** *Re* **then**… **end**.

The Discrimination principle appears on page 655.

Like conditionals, multi-branch instructions should not be used as a substitute for the implicit discrimination techniques of object technology, based on dynamic binding. The restriction to integer and character values helps avoid misuse; the Discrimination principle, introduced together with unique values, will provide further guidance.

Loop

See "LOOP INVARIANTS AND VARIANTS", 11.12, page 381.

The syntax of loops was introduced in the presentation of Design by Contract:

```
from
        initialization_instructions
invariant
        invariant
variant
        variant
until
        exit_condition
loop
        loop_instructions
end
```

The **invariant** and **variant** clauses are optional. The **from** clause is required (but may be empty); it specifies the loop initialization instructions. Leaving aside the optional clauses, the execution of such a loop consists of executing the *initialization_instructions* followed by the "loop process", itself defined as follows: if the *exit_condition* is true, the loop process is a null instruction; if it is false, the loop process is the execution of the *loop_instructions* followed (recursively) by a new loop process.

Check

The **check** instruction was also seen in the discussion of assertions. It serves to express that certain assertions must be satisfied at certain points:

See "LOOP INVARIANTS AND VARIANTS", 11.12, page 381.

```
check
    assertion -- One or more clauses
end
```

Debug

The ***debug*** instruction is a facility for conditional compilation. It is written

debug *instruction*; *instruction*; … **end**

For every class, you may turn on or off the corresponding *debug* option of the control file (the Ace). If on, any debug instruction in the class is equivalent to the instructions it contains; if off, it has no effect on the execution.

You can use this instruction to include special actions that should only be executed in debugging mode, for example instructions to print out some values of interest.

Retry

The last instruction is **retry**, introduced in the discussion of exceptions. It may only appear in a **rescue** clause, and will restart the body of a routine that was interrupted by an exception.

See "AN EXCEPTION MECHANISM", 12.3, page 419.

13.4 EXPRESSIONS

An expression serves to denote a computation that yields a value — an object, or a reference to an object. Expressions include the following varieties:

- Manifest constants.
- Entities (attributes, local routine entities, formal routine arguments, *Result*).
- Function calls.
- Expressions with operators (technically are a special case of function calls).
- *Current*.

Entities were defined on page 213.

Manifest constants

A manifest constant is a value that denotes itself (such as the integer value written *0*) — as opposed to a symbolic constant, whose name is independent of the denotation of the value.

There are two boolean manifest constants, written *True* and *False*. Integer constants follow the usual form and may be preceded by a sign. Examples are

453 –678 +66623

Real constants use a decimal point. Either the integer part or the fractional part may be absent; you may include a sign, and specify an integer power of 10 by *e* followed by the exponent value. Examples are:

52.5 –54.44 +45.01 .983 –897. 999.e12

Character constants consist of a single character written in quotes, as in *'A'*; they describe single characters. For strings of more than one character we will use the library class *STRING*, discussed later in this chapter.

Function calls

Function calls follow the same syntax as procedure calls studied earlier in this chapter. They may be qualified or unqualified; in the qualified case, multidot notation is available. Assuming the proper class and function declarations, examples are:

b.f
b.g (x, y, …)
b.h (u, v).i.j (x, y, …)

The Qualified Call rule introduced for procedures applies to function calls as well.

Current object

The reserved word *Current* denotes the current instance of the class and may be used in an expression. Note that *Current* itself is an expression, not a writable entity; thus an assignment to *Current*, such as *Current := some_value*, would be syntactically illegal.

When referring to a feature (attribute or routine) of the current instance, it is not necessary to write *Current.f*; just *f* suffices. Because of this rule, we will use *Current* less frequently than in object-oriented languages where every feature reference must be explicitly qualified. (In Smalltalk, for example, there is no such convention; a feature is always qualified, even when it applies to the current instance, written *self*.) Cases in which you will need to name *Current* explicitly include:

- Passing the current instance as argument to a routine, as in *a.f (Current)*. A common application is to create a duplicate of the current instance, as in *x := clone (Current)*.
- Testing whether a reference is attached to the current instance, as in the test *x = Current*.
- Using *Current* as anchor in an "anchored declaration" of the form **like** *Current*, as will be seen in the study of inheritance.

"ANCHORED DECLARATION", 16.7, page 599.

Expressions with operators

Operators are available to construct composite expressions.

Unary operators are + and –, applicable to integer and real expressions, and **not**, applicable to boolean expressions.

Binary operators, which take exactly two operands, include the relational operators

= /= < > <= >=

where /= is "not equal". The relational operators yield boolean results.

Multiary expressions involve one or more operands, combined with operators. Numerical operands may be combined using the following operators:

+ – * / ^ // \\

where // is integer division, \\ is integer remainder and ^ is power (exponentiation).

Boolean operands may be combined with the operators **and**, **or**, **xor**, **and then**, **or else**, **implies**. The last three are explained in the next section; **xor** is exclusive or.

The precedence of operators, based on the conventions of ordinary mathematics, has been devised according to the "Principle of Least Surprise". To avoid any uncertainty or confusion, this book makes generous use of parentheses even where they are not needed, as in the examples of the next section.

Non-strict boolean operators

The operators **and then** and **or else** (whose names have been borrowed from Ada) as well as **implies** are not commutative, and are called *non-strict* boolean operators. Here is their semantics:

For an explanation of the word "non-strict" see e.g. [M 1990] *or* [M 1992].

Non-strict boolean operators

- *a* **and then** *b* has value false if *a* has value false, and otherwise has the value of *b*.
- *a* **or else** *b* has value true if *a* has value true, and otherwise has the value of *b*.
- *a* **implies** *b* has the same value as: (**not** *a*) **or else** *b*.

The boolean values from mathematics are written in regular font: true and false; *True* and *False* are predefined language constants and hence written in color italics.

The first two definitions at first seem to yield the same semantics as **and** and **or**. But the difference is what happens when *b* is not defined. In that case the expressions using the standard boolean operators are mathematically undefined, but the above definitions may still yield a result: if *a* is false, *a* **and then** *b* is false regardless of *b*; and if *a* is true, *a* **or else** *b* is true regardless of *b*. Similarly, *a* **implies** *b* is true if *a* is false, even if *b* is undefined. So the non-strict operators may yield a result when the standard ones do not.

A typical application is the boolean expression (using integer division //)

(*i* /= *0*) **and then** (*j* // *i* = *k*)

which, from the above definition, has value false if *i* is equal to zero (as the first operand is then false). If the expression had been written using **and** rather than **and then**, then its second operand would be undefined when *i* is zero, so that the status of the whole expression is unclear in this case. This uncertainty is reflected in what may happen at run time:

B1 • If the compiler generates code that evaluates both operands and then takes their boolean "and", a division by zero will result at run time, producing an exception.

B2 • If, on the other hand, the generated code only evaluates the second operand when the first is true, otherwise returning false as the value of the expression, then the expression will indeed evaluate to false.

To guarantee interpretation B2, use **and then**. Similarly,

```
(i = 0) or else (j // i /= k)
```

will evaluate to true if *i* is zero, whereas the **or** variant could produce a run-time error.

An expression using **and then** always yields the same value as the corresponding expression written using **and** if both are defined. But the **and then** form may yield a value (false) in cases when the **and** form does not. The same holds with **or else** (and the value true) with respect to **or**. In this sense, the non-commutative operators may be said to be "more defined than or equal to" their respective counterparts. This also means that the non-strict interpretation — strategy B2 — is a correct implementation for the ordinary operators: a compiler writer may decide to implement **and** as **and then** and **or** as **or else**. But he does not have to, so the software developer may not rely on the assumption that **and** and **or** will be non-strict; only **and then** and **or else** guarantee the correct behavior in cases such as the last two examples.

One might wonder why two new operators are needed; would it not be simpler and safer to just keep the standard operators **and** and **or** and take them to mean **and then** and **or else**? This would not change the value of any boolean expression when both operands are defined, but would extend the set of cases in which expressions may be given a consistent value. This is indeed the way some programming languages, notably ALGOL W and C, interpret boolean operators. There are, however, both theoretical and practical reasons for keeping two sets of distinct operators:

- On the theoretical side, the standard mathematical boolean operators are commutative: *a* **and** *b* always has the same value as *b* **and** *a*, whereas *a* **and then** *b* may be defined when *b* **and then** *a* is not. When the order of operands does not matter it is preferable to use a commutative operator.
- In practice, some compiler optimizations become impossible if we require the compiler to evaluate the operands in a given sequence, as is the case with the non-commutative operators. So it is better to use the standard operators if both operands are known to be defined.

Note that it is possible to simulate the non-strict operators through conditional instructions in a language that does not include such operators. For example, instead of

```
b := ((i /= 0) and then (j // i = k))
```

one may write

```
if i = 0 then b := false else b := (j // i = k) end
```

The non-strict form is of course simpler. This is particularly clear when it is used as the exit condition of a loop, such as the following iteration on an array:

```
from
        i := a.lower
invariant
        -- For all elements in the interval [a.lower .. i –- 1], (a @ i) /= x
variant
        a.upper — i
until
        i > a.upper or else (a @ i = x)
loop
        i := i + 1
end;
Result := (i <= a.upper)
```

whose purpose is to make *Result* true if and only if the value *x* appears in the array *a*. The use of **or** would be incorrect here: a compiler may generate code that will always evaluate both operands, so that for the last index examined (*i* > *a*.*upper*) if no array value equals *x*, there will be an erroneous attempt at run time to access the non-existent array item *a* @ (*a*.*upper* + 1), causing a run-time error (a precondition violation if assertion checking is on).

It is possible to program this example safely without non-strict operators, but the result is heavy and inelegant (try it).

Another example is an assertion — appearing for example in a class invariant — expressing that the first value of a certain list *l* of integers is non-negative — provided, of course, that the list is not empty. You may express this as

```
l.empty or else l.first >= 0
```

Using **or** would have been incorrect. Here there is no way to write the condition without non-strict operators (except by writing a special function and calling it in the assertion). The Base libraries of algorithms and data structures contain many such cases.

The **implies** operator, describing implication, is also non-strict. Mathematical logic defines "*a* implies *b*" as "**not** *a* **or** *b*"; but in practical uses property *b* is often meaningless for false *a*, so that it is appropriate to use **or else** rather than **or**; this is the official definition given above. In this case there is no need for a strict variant.

The **implies** form does not always come first to mind when you are not used to it, but it is often clearer; for example you might like the last example better under the form

```
(not l.empty) implies (l.first >= 0)
```

13.5 STRINGS

Class *STRING* describes character strings. It enjoys a special status since the notation permits manifest string constants, understood as denoting instances of *STRING*.

A string constant is written enclosed in double quotes, as in

```
"ABcd Ef ~*_ 01"
```

The double quote character must be preceded by a percent % if it appears as one of the characters of the string.

Non-constant character strings are also instances of class *STRING*, whose creation procedure *make* takes as argument the expected initial length of the string, so that

```
text1, text2: STRING; n: INTEGER;
...
!! text1.make (n)
```

will dynamically allocate a string *text1*, reserving the space for *n* characters. Note that *n* is only an initial size, not a maximum; any string can grow or shrink to an arbitrary size.

Numerous features are available on instances of *STRING*: concatenation, character or substring extraction, comparison etc. (They may change the size of the string, automatically triggering re-allocation if it becomes greater than the currently allocated size.)

Assignment of a *STRING* to another implies sharing: after *text2 := text1*, any modification to the contents of *text1* will also affect the contents of *text2* and conversely. To duplicate rather than share, use by *text2 := clone (text1)*.

You can declare a constant string attribute:

```
message: STRING is "Your message here"
```

13.6 INPUT AND OUTPUT

Two Kernel Library classes provide basic input and output facilities: *FILE* and *STD_FILES*.

Among the operations defined on an object *f* declared of type *FILE* are the following:

```
!! f.make ("name")          --Associate f with a file of name name.
f.open_write                -- Open f for writing
f.open_read                 -- Open f for reading
f.put_string ("A_STRING")   --Write the given string on f
```

For I/O operations on the standard input, output and error files, you can inherit from *STD_FILES*, which defines the features *input*, *output* and *error*. Alternatively you can use the predefined value *io*, as in *io.put_string ("ABC")*, bypassing inheritance.

13.7 LEXICAL CONVENTIONS

Identifiers are sequences of characters, all of which must be letters, digits or underscore characters (_); the first character of an identifier must be a letter. There is no limit to the length of identifiers, and all the characters of identifiers are significant. This can be used to make both feature names and class names as clear as possible.

Letter case is not significant in identifiers, so that *Hi*, *hi*, *HI* and *hI* all denote the same identifier. The reason is that it would be dangerous to allow two identifiers that differ

from each other by just one character, say *Structure* and *structure*, to denote different elements. Better ask developers to use some imagination than risk mistakes.

The notation, however, comes with a set of precise standard style conventions, detailed in a later chapter entirely devoted to style: classes (*INTEGER*, *POINT*…) and formal generic parameters (*G* in *LIST* [*G*]) in all upper case; predefined entities and expressions (*Result*, *Current*…) and constant attributes (*Pi*) start with an upper-case letter and continue in lower case; all other identifiers (non-constant attributes, formal routine arguments, local entities) in all lower case. Although compilers do not enforce them since they are not part of the notation's specification, these rules are essential to the readability of software texts; the libraries and this book apply them consistently.

Chapter 26.

13.8 KEY CONCEPTS INTRODUCED IN THIS CHAPTER

- External routines are accessible through a well-defined interface.
- Object technology can serve as a wrapping mechanism for legacy software.
- Routines may not directly modify their arguments, although they may change the *objects* associated with these arguments.
- The notation includes a small set of instructions: assignment, conditional, loop, call, debug, check.
- Expressions follow common usage. *Current* is an expression denoting the current instance. Not being an entity, *Current* may not be the target of an assignment.
- Non-strict boolean operators yield the same values as the standard boolean operators when both operands are defined, but are defined in some cases when the standard operators are not.
- Strings, input and output are covered by simple library classes.
- Letter case is not significant in identifiers, although the style rules include recommended conventions

EXERCISES

E13.1 External classes

The discussion of how to integrate external software mentioned that although features are the right level of integration for non-O-O software elements, interaction with another object-oriented language might take place at the class level. Discuss a notion of "external class" meant for that purpose, and its addition to the notation of this book.

E13.2 Avoiding non-strict operators

Write a loop that determines if an element *x* appears in an array *a*, similar to the algorithm given in this chapter but not using any of the non-strict operators.

Page 456.