26

A sense of style

Implementing the object-oriented method requires paying attention to many details of style, which a less ambitious approach might consider trifles.

26.1 COSMETICS MATTERS!

Although the rules appearing hereafter are not as fundamental as the principles of object-oriented software construction covered in earlier chapters, it would be foolish to dismiss them as just "cosmetics". Good software is good in the large *and* in the small, in its high-level architecture and in its low-level details. True, quality in the details does not guarantee quality of the whole; but sloppiness in the details usually indicates that something more serious is wrong too. (If you cannot get the cosmetics right, why should your customers believe that you can master the truly difficult aspects?) A serious engineering process requires doing *everything* right: the grandiose and the mundane.

So you should not neglect the relevance of such seemingly humble details as text layout and choice of names. True, it may seem surprising to move on, without lowering our level of attention, from the mathematical notion of sufficient completeness in formal specifications (in the chapter on abstract data types) to whether a semicolon should be preceded by a space (in the present chapter). The explanation is simply that both issues deserve our care, in the same way that when you write quality O-O software both the design and the realization will require your attention.

We can take a cue from the notion of style in its literary sense. Although the first determinant of good writing is the author's basic ability to tell a story and devise a coherent structure, no text is successful until everything works: every paragraph, every sentence and every word.

Applying the rules in practice

Some of the rules of this chapter can be checked or, better yet, enforced from the start by software tools. Tools will not do everything, however, and there is no substitute for care in writing every piece of the software.

There is often a temptation to postpone the application of the rules, writing things casually at first and thinking "I will clean up everything later on; I do not even know how much of this will eventually be discarded". This is not the recommended way. Once you get used to the rules, they do not add any significant delay to the initial writing of the software; even without special tools, it is always more costly to fix the text later than to write it properly from the start. And given the pressure on software developers, there is ever a risk that you will forget or not find the time to clean things up. Then someone who is asked later to take up your work will waste more time than it would have cost you to write the proper header comments, devise the right feature names, apply the proper layout. That someone may be you.

Terseness and explicitness

Software styles have oscillated between the terse and the verbose. In programming languages, the two extremes are perhaps APL and Cobol. The contrast between the Fortran-C-C++ line and the Algol-Pascal-Ada tradition — not just the languages themselves, but the styles they have bred — is almost as stark.

What matters for us is clarity and, more generally, quality. Extreme forms of terseness and verbosity can both work against these goals. Cryptic C programs are unfortunately not limited to the famous "obfuscated C" and "Obfuscated C++" contests; but the almost equally famous *DIVIDE DAYS BY 7 GIVING WEEKS* of Cobol is a waste of everyone's attention.

The style that follows from this chapter's rules is a particular mix of Algol-like explicitness (although not, it is hoped, verbosity) and telegram-style terseness. It never begrudges keystrokes, even lines, when they truly help make the software readable; for example, you will find rules that enjoin using clear identifiers based on full words, not abbreviations, as it is foolish to save a few letters by calling a feature *disp* (ambiguous) rather than *display* (clear and precise), or a class *ACCNT* (unpronounceable) rather than *ACCOUNT*. There is no tax on keystrokes. But at the same time when it comes to eliminating waste and unneeded redundancies the rules below are as pitiless as the recommendations of a General Accounting Office Special Commission on Improving Government. They limit header comments to indispensable words, getting rid of all the non-essential "the" and other such amenities; they proscribe over-qualification of feature names (as in *account_balance* in a class *ACCOUNT*, where *balance* is perfectly sufficient); against dominant mores, they permit the grouping of related components of a complex construct on a single line, as in **from** *i* := *1* **invariant** *i* <= *n* **until** *i* = *n* **loop**; and so on.

This combination of terseness and explicitness is what you should seek in your own texts. Do not waste space, as exaggerated size will in the end mean exaggerated complexity; but do not hesitate to use space when it is necessary to enhance clarity.

Also remember, if like many people you are concerned about how much smaller the text of an object-oriented implementation will be than the text of a comparable C, Pascal, Ada or Fortran program, that the only interesting answer will appear at the level of a significant system or subsystem. If you express a basic algorithm — at the level of

Quicksort, say, or Euclid's algorithm — in C and in the notation of this book, expect the O-O version to be at least as large. In many cases, if you apply the principles thoroughly, it will be larger, since it will include assertions and more type information. Yet in ISE's experience of looking at medium-scale systems we have sometimes found (without being able to give a general law, as the circumstances vary considerably) the object-oriented solution to be several times smaller. Why? This is not due to terseness at the "micro" level but to systemwide application of the architectural techniques of the O-O method:

- Genericity is one of the key factors. We have found C programs that repeated essentially the same C code many times to handle different types. With a generic class — or for that matter a generic Ada package — you immediately get rid of that redundancy. It is disturbing in this respect to see that Java, a recent O-O language based on C, does not support genericity.

- Inheritance is also fundamental in gathering commonalities and removing duplications.

- Dynamic binding replaces many complex decision structures by much shorter calls.

- Assertions and the associated idea of Design by Contract avoid redundant error checking, a principal source of bloat.

- The exception mechanism gets rid of some error code.

If you are concerned with source size, make sure to concentrate on these architectural aspects. You should also be terse in expressing algorithms, but never skimp on keystrokes at the expense of clarity.

The role of convention

Most rules define a single permissible form, with no variants. The few exceptions include font use, which is governed by external considerations (what looks good in a book may not be visible on overhead transparencies), and semicolons, for which there exist two opposite schools with equally forceful arguments (although we will have a few universal rules anyway). In all other cases, in line with the introductory methodology chapter's exhortations against wishy-washiness, the rules leave about as much room to doubt as a past due reminder from the Internal Revenue Service.

The rules are rooted in a careful analysis of what works and what works less well, resulting from many years of observation; some of the rationale will appear in the discussion. Even so, some rules may appear arbitrary at first, and indeed in a few cases the decision is a matter of taste, so that reasonable persons working from the same assumptions may disagree. If you object to one of the recommended conventions, you should define your own, provided you explain it in detail and document it explicitly; but do think carefully before making such a decision, so obvious are the advantages of abiding by a universal set of rules that have been systematically applied to thousands of classes over more than ten years, and that many people know and understand.

As noted in an earlier chapter (in the more general context of design principles), many of the style rules were originally developed for libraries, and then found their way into ordinary software development. In object technology, of course, all software is developed under the assumption that even if it is not reusable yet it *might* eventually be made reusable, so it is natural to apply the same style rules right from the start.

The comment was in the introduction to chapter 23.

Self-practice

Like the design rules of the preceding chapters, the style rules which follow have been carefully applied to the many examples of this book. The reasons are obvious: one should practice what one preaches; and, more fundamentally, the rules do support clarity of thought and expression, which can only be good for a detailed presentation of the object-oriented method.

The only exceptions are a few occasional departures from the rules on software text layout. These rules do not hesitate to spread texts over many lines, for example by requiring that every assertion clause have its own label. Lines are not a scarce resource on computer screens; it has been observed that with the computer age we are reversing the direction of the next-to-last revolution in written communication, the switch from papyrus rolls to page-structured books. But this text is definitely a book, structured into pages, and a constant application of the layout-related rules would have made it even bigger than it is.

The cases of self-dispensation affect only two or three layout-related rules, and will be noted in their presentation below. Any exception only occurs after the first few examples of a construct in the book have applied the rules scrupulously.

Such exceptions are only justified for a paper presentation. Actual software texts should apply the rules literally.

Discipline and creativity

It would be a mistake to protest against the rules of this chapter (and others) on the grounds that they limit developer creativity. A consistent style favors rather than hampers creativity by channeling it to where it matters. A large part of the effort of producing software is spent reading existing software and making others read what is being written. Individual vagaries benefit no one; common conventions help everyone.

Some of the software engineering literature of the nineteen-seventies propounded the idea of "egoless programming": developing software so that it does not reflect anything of its authors' personality, thereby making developers interchangeable. Applied to system design, this goal is clearly undesirable, even if some managers may sometimes long for it (as in this extract of a programming management book quoted by Barry Boehm: "*…the programmer*['*s*] *creative instincts should be totally dulled to insure uniform and understandable programming*", to which Boehm comments: "Given what we know about programmers and their growth motivation, such advice is a clear recipe for disaster").

Sentence in italics from D.H. Brandon, "Data Processing Organization and Manpower Planning", Petrocelli, 1974, emphasis in original. Quoted in [Boehm 1981], p. 674.

What quality software requires is **egoful design** with **egoless expression**.

More than style standards, what would seem to require justification is the current situation of software development, with its almost total lack of style standards. In no other discipline that demands to be called "engineering" is there such room for such broad personal variations of whim and fancy. To become more professional, software development needs to regulate itself.

26.2 CHOOSING THE RIGHT NAMES

The first aspect that we need to regulate is the choice of names. Feature names, in particular, will be strictly controlled for everyone's benefit.

General rules

What matters most is the names of **classes** and **features** which will be used extensively by the authors of classes that rely on yours.

For feature and class names, use full words, not abbreviations, unless the abbreviations are widely accepted in the application domain. In a class *PART* describing parts in an inventory control system, call *number*, not *num*, the feature (query) giving the part number. Typing is cheap; software maintenance is expensive. An abbreviation such as *usa* in a Geographical Information System or *copter* in a flight control system, having gained an independent status as a word of its own, is of course acceptable. In addition, a few standard abbreviations have gained recognition over the years, such as *PART* for *PARTIAL* in class names such as *PART_COMPARABLE* describing objects equipped with a partial order relation.

In choosing names, aim for clarity. Do not hesitate to use several words connected by underscores, as in *ANNUAL_RATE*, a class name, or *yearly_premium*, a feature name.

Although modern languages do not place any limit on the length of identifiers, and treat all letters as significant, name length should remain reasonable. Here the rule is not the same for classes and for features. Class names are input only occasionally (in class headers, type declarations, inheritance clauses and a few other cases) and should describe an abstraction as completely as possible, so *PRODUCT_QUANTITY_INDEX_EVALUATOR* may be fine. For features, there is seldom a need for more than two or possibly three underscore-connected words. In particular, *do not overqualify feature names*. If a feature name appears too long, it is usually because it is overqualified:

Composite Feature Name rule

Do not include in a feature name the name of the underlying data abstraction (which should serve as the class name).

The feature giving the part number in class *PART* should be called just *number*, not *part_number*. Such over-qualification is a typical beginner's mistake; the resulting names obscure rather than illuminate the text. Remember that every use of the feature will unambiguously indicate the class, as in *part1*.*number* where *part1* must have been declared with a certain type, *PART* or a descendant.

For composite names, it is better to avoid the style, popularized by Smalltalk and also used in such libraries as the X Window System, of joining several words together and starting the internal ones with an upper-case letter, as in *yearlyPremium*. Instead, separate components with underscores, as in *yearly_premium*. The use of internal upper-case letters is ugly; it conflicts with the conventions of ordinary language; and it leads to cryptic names, hence to possible errors (compare *aLongAndRatherUnreadableIdentifier* with *an_even_longer_but_perfectly_clear_choice_of_name*).

Sometimes, every instance of a certain class contains a field representing an instance of another class. This suggests using the class name also as attribute name. You may for example have defined a class *RATE* and, in class *ACCOUNT*, need one attribute of type *RATE*, for which it seems natural to use the name *rate* — in lower case, according to the rules on letter case stated below. Although you should try to find a more specific name, you may, if this fails, just declare the feature as *rate*: *RATE*. The rules on identifier choice explicitly permit assigning the same name to a feature and a class. Avoid the style of prefixing the name with *the*, as in *the_rate*, which only adds noise.

Local entities and routine arguments

The emphasis on clear, spelled-out names applies to features and classes. Local entities and arguments of a routine only have a local scope, so they do not need to be as evocative. Names that carry too much meaning might almost decrease the software's readability by giving undue weight to ancillary elements. So it is appropriate to declare local entities (here in routines of *TWO_WAY_LIST* in the Base libraries) as

```
move (i: INTEGER) is
        -- Move cursor i positions, or after if i is too large.
    local
        c: CURSOR; counter: INTEGER; p: like FIRST_ELEMENT
    …
remove is
        -- Remove current item; move cursor to right neighbor (of after if none).
    local
        succ, pred, removed: like first_element
    …
```

If *succ* and *pred* had been features they would have been called *successor* and *predecessor*. It is also common to use the names *new* for a local entity representing a new object to be created by a routine, and *other* for an argument representing an object of the same type as the current one, as in the declaration for *clone* in *GENERAL*:

```
frozen clone (other: GENERAL): like other is…
```

Letter case

Letter case is not significant in our notation, as it is too dangerous to let two almost identical identifiers denote different things. But strongly recommended guidelines help make class texts consistent and readable:

- Class names appear in all upper case: *POINT*, *LINKED_LIST*, *PRICING_MODEL*. Formal generic parameters too, usually with just one letter: *G*.
- Names of non-constant attributes, routines other than once functions, local entities and routine arguments appear in all lower case: *balance*, *deposit*, *succ*, *i*.
- Constant attributes have their first letter in upper case and the rest in lower case: *Pi*: *INTEGER* **is** *3.1415926524*; *Welcome_message*: *STRING* **is** "*Welcome*!". This applies to unique values, which are constant integers.
- The same convention applies to once functions, the equivalent of constants for non-basic types: *Error_window*, *Io*. Our first example, the complex number *i*, remained in lower case for compatibility with mathematical conventions.

The example of i was on page 648.

This takes care of developer-chosen names. For reserved words, we distinguish two categories. *Keywords* such as **do** and **class** play a strictly syntactic role; they are written in lower case, and will appear in boldface (see below) in printed texts. A few *reserved words* are not keywords because they carry an associated semantics; written with an initial upper case since they are similar to constants, they include *Current*, *Result*, *Precursor*, *True* and *False*.

Grammatical categories

Precise rules also govern the grammatical category of the words from which identifiers are derived. In some languages, these rules can be applied without any hesitation; in English, as noted in an earlier chapter, they will leave more flexibility.

See the Class Name rule on page 727.

The rule for class names has already been given: you should always use a noun, as in *ACCOUNT*, possibly qualified as in *LONG_TERM_SAVINGS_ACCOUNT*, except for the case of deferred classes describing a structural property, which may use an adjective as in *NUMERIC* or *REDEEMABLE*.

Routine names should faithfully reflect the Command-Query separation principle:

- Procedures (commands) should be verbs in the infinitive or imperative, possibly with complements: *make, move*, *deposit*, *set_color*.
- Attributes and functions (queries) should never be imperative or infinitive verbs; never call a query *get_value*, but just *value*. Non-boolean query names should be nouns, such as *number*, possibly qualified as in *last_month_balance*. Boolean queries should use adjectives, as in *full*. In English, because of possible confusions between adjectives and verbs (*empty*, for example, could mean “is this empty?” or “empty this!”), a frequent convention for boolean queries is the *is_* form, as in *is_empty*.

Standard names

You will have noted, throughout this book, the recurrence of a few basic names, such as *put* and *item*. They are an important part of the method.

Many classes will need features representing operations of a few basic kinds: insert an element into a structure, replace the value of an element, access a designated element… Rather than devising specific names for the variants of these operations in every class, it is preferable to apply a standard terminology throughout.

Here are the principal standard names. We can start with creation procedures, for which the recommended is *make* for the most common creation procedure of a class. Non-vanilla creation procedures may be called *make*_some_qualification, for example *make_polar* and *make_cartesian* for a *POINT* or *COMPLEX* class.

For commands the most common names are:

Standard command names

extend	Add an element.
replace	Replace an element.
force	Like *put* but may work in more cases; for example *put* for arrays has a precondition to require the index to be within bounds, but *force* has no precondition and will resize the array if necessary.
remove	Remove an (unspecified) element.
prune	Remove a specific element.
wipe_out	Remove all elements.

For non-boolean queries (attributes or functions):

Standard names for non-boolean queries

item	The basic query for accessing an element: in *ARRAY*, the element at a given index; in *STACK* classes, the stack top; in *QUEUE* classes, the oldest element; and so on.
infix "@"	A synonym for *item* in a few cases, notably *ARRAY*.
count	Number of usable elements in a structure.
capacity	Physical size allocated to a bounded structure, measured in number of potential elements. The invariant should include *0 <= count* **and** *count <= capacity*.

For boolean queries:

Standard names for boolean queries

empty	Is the structure devoid of elements?
full	Is there no more room in the representation to add elements? (Normally the same as *count* = *capacity*.)
has	Is a certain element present? (The basic membership test.)
extendible	Can an element be added? (May serve as a precondition to *extend*.)
prunable	Can an element be removed? (May serve as a precondition to *remove* and *prune*.)
readable	Is there an accessible element? (May serve as precondition to *item* and *remove*.)
writable	Is it possible to change an element? (May variously serve as precondition to *extend*, *replace*, *put* etc.)

A few name choices which may seem strange at first are justified by considerations of clarity and consistency. For example *prune* goes with *prunable* and *extend* with *extendible*; *delete* and *add* might seem more natural, but then *s*●*deletable* and *s*●*addable* would carry the wrong connotation, since the question is not whether *s* can be deleted or added but whether we can add elements to it or delete elements from it. The verbs *prune* and *extend*, with the associated queries, convey the intended meaning.

The benefits of consistent naming

The set of names sketched above is one of the elements that most visibly contribute to the distinctive style of software construction developed from the principles of this book.

Is the concern for consistency going too far? One could fear that confusion could result from routines that bear the same name but internally do something different. For example *item* for a stack will return the top element, and for an array will return an element corresponding to the index specified by the client.

With a systematic approach to O-O software construction, using static typing and Design by Contract, this fear is not justified. To learn about a feature, a client author can rely on four kinds of property, all present in the short form of the enclosing class:

F1 • Its name.

F2 • Its signature (number and type of arguments if a routine, type of result if a query).

F3 • Its precondition and postcondition if any.

F4 • Its header comment.

A routine also has a body, but that is not part of what client authors are supposed to use.

Three of these elements will differ for the variants of a basic operation. For example in the short form of class *STACK* you may find the feature

```
put (x: G)
        -- Push x on top.
    require
        writable: not full
    ensure
        not_empty: not empty
        pushed: item = x
```

whereas its namesake will appear in *ARRAY* as

```
put (x: G; i: INTEGER)
        -- Replace by x the entry of index i
    require
        not_too_small: i >= lower
        not_too_large: i <= upper
    ensure
        replaced: item (i) = x
```

The signatures are different (one variant takes an index, the other does not); the preconditions are different; the postconditions are different; and the header comments are different. Using the same name *put*, far from creating confusion, draws the reader's attention to the common role of these routines: both provide the basic element change mechanism.

This consistency has turned out to be one of the most attractive aspects of the method and in particular of the libraries. New users take to it quickly; then, when exploring a new class which follows the standard style, they feel immediately at home and can zero in on the features that they need.

26.3 USING CONSTANTS

Many algorithms will rely on constants. As was noted in an early chapter of this book, constants are widely known for the detestable practice of changing their values; we should prepare ourselves against the consequences of such fickleness.

Manifest and symbolic constants

The basic rule is that uses of constants should not explicitly rely on the value:

Symbolic Constant principle

Do not use a manifest constant, other than the zero elements of basic operations, in any construct other than a symbolic constant declaration.

In this principle, a **manifest constant** is a constant given explicitly by its value, as in *50* (integer constant) or "*Cannot find file*" (string constant). The principle bars using instructions of the form

```
population_array.make (1, 50)
```

or

```
print ("Cannot find file")          -- See mitigating comment below about this case
```

Instead, you should declare the corresponding constant attributes, and then, in the bodies of the routines that need the values, denote them through the attribute names:

```
US_state_count: INTEGER is 50
File_not_found: STRING is "Cannot find file"
…
population_array.make (1, state_count)
…
print (file_not_found)
```

The advantage is obvious: if a new state is added, or the message needs to be changed, you have only have to update one easy-to-locate declaration.

The use of *1* together with *state_count* in the first instruction is not a violation of the principle, since its prohibition applies to manifest constants "*other than zero elements of basic operations*". These zero elements, which you may use in manifest form, include the integers *0* and *1* (zero elements of addition and multiplication), the real number *0.0*, the null character written '*%0*', the empty string "". Using a symbolic constant *One* every time you need to refer to the lower bound of an array (1 using the default convention) would lead to an unsustainable style — pedantic, and in fact less readable because of its verbosity. Sometimes, Freud is supposed to have said, a cigar is just a cigar; sometimes *One* is just 1.

Some other times *1* is just a system parameter that happens to have the value one today but could become 4,652 later — its role as addition's zero element being irrelevant. Then it should be declared as a symbolic constant, as in *Processor_count*: *INTEGER* **is** *1* in a system that supports multiple processors and is initially applied to one processor.

The Symbolic Constant principle may be judged too harsh in the case of simple manifest strings used just once, such as "*Cannot find file*" above. Some readers may want to add this case to the exception already stated in the principle (replacing the qualification by "*other than manifest string constants used only once in the same class, and zero elements of basic operations*"). This book has indeed employed a few manifest constants in simple examples. Such a relaxation of the rule is acceptable, but in the long run it is probably preferable to stick to the rule as originally given even if the result for string constants looks a little pedantic at times. One of the principal uses of string constants, after all, is for messages to be output to users; when a successful system initially written for the home market undergoes internationalization, it will be that much less translation work if all the user-visible message strings (at least any of them that actually appear in the software text) have been put in symbolic constant declarations.

Where to put constant declarations

If you need more than a handful of local constant attributes in a class, you have probably uncovered a data abstraction — a certain concept characterized by a number of numeric or character parameters.

It is desirable, then, to group the constant declarations into a class, which can serve as ancestor to any class needing the constants (although some O-O designers prefer to use the client relation in this case). An example in the Base libraries is the class *ASCII*, which declares constant attributes for the different characters in the ASCII character set and associated properties.

See "Facility inheritance", page 832.

26.4 HEADER COMMENTS AND INDEXING CLAUSES

Although the formal elements of a class text should give as much as possible of the information about a class, they must be accompanied by informal explanations. Header comments of routines and feature clause answer this need together with the indexing clause of each class.

Routine header comments: an exercise in corporate downsizing

Like those New York street signs that read "Don't even *think* of parking here!", the sign at the entrance of your software department should warn "Don't even think of writing a routine without a header comment". The header comment, coming just after the **is** for a routine, expresses its purpose concisely; it will be kept by the short and flat-short forms:

```
distance_to_origin: REAL is
            -- Distance to point (0, 0)
        local
            origin: POINT
        do
            !! origin
            Result := distance (origin)
        end
```

Note the indentation: one step further than the start of the routine body, so that the comment stands out.

Header comments should be informative, clear, and terse. They have a whole style of their own, which we can learn by looking at an initially imperfect example and improve it step by step. In a class *CIRCLE* we might start with

Warning: not the recommended style for header comments!

```
tangent_from (p: POINT): LINE is
            -- Return the tangent line to the current circle going through the point p,
            -- if the point is outside of the current circle.
      require
            outside_circle: not has (p)
      …
```

There are many things wrong here. First, the comment for a query, as here, should not start with "Return the…" or "Compute the…", or in general use a verbal form; this would go against the Command-Query Separation principle. Simply name what the query returns, typically using a qualified noun for a non-boolean query (we will see below what to use for a boolean query and a command). Here we get:

Not the recommended style.

```
            -- The tangent line to the current circle going through the point p,
            -- if the point p is outside of the current circle
```

Since the comment is not a sentence but simply a qualified noun, the final period disappears. Next we can get rid of the auxiliary words, especially *the*, where they are not required for understandability. Telegram-like style is desirable for comments. (Remember that readers in search of literary frills can always choose Proust novels instead.)

Not yet…

```
            --Tangent line to current circle from point p,
            -- if point p is outside current circle
```

The next mistake is to have included, in the second line, the condition for the routine's applicability; the precondition, **not** *has* (*p*), which will be retained in the short form where it appears just after the header comment, expresses this condition clearly and unambiguously. There is no need to paraphrase it: this could lead to confusion, if the informal phrasing seems to contradict the formal precondition, or even to errors (a common oversight is a precondition of the form *x >= 0* with a comment stating "applicable only to positive *x*", rather than "non-negative"); and there is always a risk that during the software's evolution the precondition will be updated but not the comment. Our example becomes:

Still not it…

```
            -- Tangent line to current circle from point p.
```

Yet another mistake is to have used the words line to refer to the result and point to refer to the argument: this information is immediately obvious from the declared types, *LINE* and *POINT*. With a typed notation we can rely on the formal type declarations — which again will appear in the short form — to express such properties; repeating them in the informal text brings nothing. So:

Almost there, but not quite…

```
            -- Tangent to current circle from p.
```

The mistakes of repeating type information and of duplicating the precondition's requirements point to the same general rule: in writing header comments, *assume the reader is competent in the fundamentals of the technology*; do not include information that is obvious from the immediately adjacent short form text. This does not mean, of course, that you should never specify a type; the earlier example, -- Distance to point (0,0), could be ambiguous without the word point.

When you need to refer to the current object represented by a class, use phrasing such as current circle, current number and so on as above, rather than referring explicitly to the entity *Current*. In many cases, however, you can avoid mentioning the current object altogether, since it is clear to everyone who can read a class text that features apply to the current object. Here, for example, we just need

```
            -- Tangent from p.
```

This is it.

At this stage — three words, starting from twenty-two, an 87% reduction that would make the toughest Wall Street exponent of corporate downsizing jealous — it seems hard to get terser and we can leave our comment alone.

A few more general guidelines. We have noted the uselessness of "Return the …" in queries; other noise words and phrases to be avoided in routines of all kinds include "This routine computes…", "This routine returns…"; just say what the routine does, not that it does it. Instead of

```
    -- This routine records the last outgoing call.
```

write

```
    -- Record outgoing call.
```

As illustrated by this example, header comments for commands (procedures) should be in the imperative or infinitive (the same in English), in the style of marching orders. They should end with a period. For boolean-valued queries, the comment should always be in the form of a question, terminated by a question mark:

```
    has (v: G): BOOLEAN is
            -- Does v appear in list?
        …
```

A convention governs the use of software entities — attributes, arguments — appearing in comments. In typeset texts such as the above they will appear in italics (more on font conventions below); in the source text they should always appear between an opening quote ("backquote") and a closing quote; the original text for the example is then:

```
            -- Does ‘v’ appear in list?
```

Tools such as the **short** class abstracter will recognize this convention when generating typeset output. Note that the two quotes should be different: ‘v’, not ’v’.

Be consistent. If a function of a class has the comment Length of string, a routine of the same class should not say Update width of string if it affects the same property.

All these guidelines apply to routines. Because an exported attribute should be externally indistinguishable from argumentless functions — remember the Uniform Access principle — it should also have a comment, which will appear on the line following the attribute's declaration, with the same indentation as for functions:

```
    count: INTEGER
                -- Number of students in course
```

For secret attributes a comment is desirable too but the rule is less strict.

Feature clause header comments

As you will remember, a class may have any number of feature clauses:

```
indexing
        …
class LINKED_LIST [G] inherit … creation
        …
feature -- Initialization
        make is …
feature -- Access
        item: G is …
        …
feature -- Status report
        before: BOOLEAN is …
        …
feature -- Status setting
        …
feature -- Element change
        put_left (v: G) is …
        …
feature -- Removal
        remove is …
        …
feature {NONE} -- Implementation
        first_element: LINKABLE [G].
        …
end -- class LINKED_LIST
```

One of the purposes of having several feature clauses is to allow different features to have different export privileges; in this example everything is generally available except the secret features in the last clause. But another consequence of this convention is that you could, and should, group features by categories. A comment on the same line as the keyword **feature** should characterize the category. Such comments are, like header comments of routines, recognized an preserved by documentation tools such as **short**.

"Operands and options", page 766.

Eighteen categories and the corresponding comments have been standardized for the Base libraries, so that every feature (out of about 2000 in all) belongs to one of them. The example above illustrates some of the most important categories. Status report corresponds to options (set by features in the Status setting category, not included in the example). Secret and selectively exported features appear in the Implementation category. These standard categories always appear in the same order, which the tools know (through a user-editable list) and will preserve or reinstate in their output. Within each category, the tools list the features alphabetically for ease of retrieval.

The categories cover a wide range of application domains, although for special areas you may need to add your own categories.

Indexing clauses

Similar to header comments but slightly more formal are indexing clauses, appearing at the beginning of a class:

Indexing clauses were previewed in "A note about component indexing", page 78.

```
indexing
    description: "Sequential lists, in chained representation"
    names: "Sequence", "List"
    contents: GENERIC
    representation: chained
    date: "$Date: 96/10/20 12:21:03 $"
    revision: "$Revision: 2.4$"
    …
class LINKED_LIST [G] inherit
    …
```

Indexing clauses proceed from the same Self-Documentation principle that has led to built-in assertions and header comments: include as much as possible of the documentation in the software itself. For properties that do not directly appear in the formal text, you may include indexing entries, all of the form

"Self-Documentation", page 54.

```
indexing_term: indexing_value, indexing_value, …
```

where the *indexing_term* is an identifier and each *indexing_value* is some basic element such as a string, an integer and so on. Entries can indicate alternative names under which potential client authors might search for the class (*names*), contents type (*contents*), implementation choices (*representation*), revision control information, author information, and anything else that may facilitate understanding the class and retrieving it through keyword-based search tools — tools that support reuse and enable software developers to find their way through a potentially rich set of reusable components.

Both the indexing terms and the indexing values are free-form, but the possible choices should be standardized for each project. A set of standard choices has been used throughout the Base libraries; the above example illustrates six of the most common entry kinds. Every class must have a *description* entry, introducing as *index_value* a string describing the role of the class, always expressed in terms of the instances (as *Sequential lists…*, not "this class describes sequential lists", or "sequential list", or "the notion of sequential list" etc.). Most significant class texts in this book — but not short examples illustrating a specific point — include the *description* entry.

More details in [M 1994a].

Non-header comments

The preceding rules on comments applied to standardized comments, appearing at specific places — feature declarations and beginning of feature clauses — and playing a special role for class documentation.

As in all forms of software development, there is also a need for comments within routine bodies, to provide further explanations

Another use of comments, although frequent in the practice of software development, does not figure much in software engineering and programming methodology textbooks. I am referring here to the technique of transforming some part of the code into comments, either because it does not work, or because it is not ready yet. This practice is clearly a substitute for better tools and techniques of configuration management. It has enriched the language with a new verb form, *comment out*, whose potential, surprisingly enough, has not yet been picked up by hip journalists, even though the non-technical applications seem attractive and indeed endless: "The last elections have enabled Congress to *comment out* the President", "Letterman was *commented out* of the Academy Awards", and so on.

Every comment should be of a level of abstraction higher than the code it documents. A famous counter-example is -- Increase *i* by 1 commenting the instruction *i* := *i* + *1*. Although not always that extreme, the practice of writing comments that paraphrase the code instead of summarizing its effect is still common.

Low-level languages cry for ample commenting. It is a good rule of thumb, for example, that for each line of C there should be a comment line; not a negative reflection on C, but a consequence that in modern software development the role of C is to encapsulate machine-oriented and operating-system-level operations, which are tricky by nature and require a heavy explanatory apparatus. In the O-O part, non-header comments will appear much more sparsely; they remain useful when you need to explain some delicate operation or foresee possible confusion. In its constant effort to favor prevention over cure, the method decreases the need for comments through a modular style that yields small, understandable routines, and through its assertion mechanisms: preconditions and postconditions of routines, to express their semantics formally; class invariants; **check** instructions to express properties expected to hold at certain stages; the systematic naming conventions introduced earlier in this chapter. More generally, the secret of clear, understandable software is not adding comments after the fact but devising coherent and stable system structures right from the start.

26.5 TEXT LAYOUT AND PRESENTATION

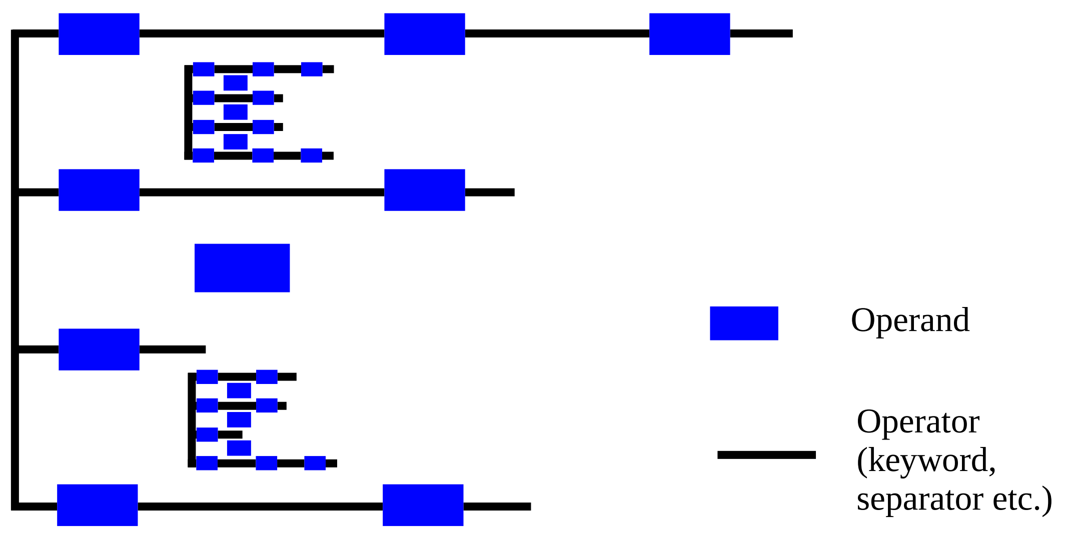
The next set of rules affects how we should physically write our software texts on paper — real, or simulated on a screen. More than any others, they prompt cries of "Cosmetics!"; but such cosmetics should be as important to software developers as Christian Dior's are to his customers. They play no little role in determining how quickly and accurately your software will be understood by its readers — maintainers, reusers, customers.

Layout

The recommended layout of texts results from the general form of the syntax of our notation, which is roughly what is known as an "operator grammar", meaning that a class text is a sequence of symbols alternating between "operators" and "operands". An operator is a fixed language symbol, such as a keyword (**do** etc.) or a separator (semicolon, comma …); an operand is a programmer-chosen symbol (identifier or constant).

Based on this property, the textual layout of the notation follows the **comb-like structure** introduced by Ada; the idea is that a syntactically meaningful part of a class, such as an instruction or an expression, should either:

- Fit on a line together with a preceding and succeeding operators.
- Be indented just by itself on one or more lines — organized so as to observe the same rules recursively.



The comb-like structure of software texts

Each branch of the comb is a sequence of alternating operators and operands; it should normally begin and end with an operator. In the space between two branches you find either a single operand or, recursively, a similar comb-like structure.

As an example, depending on the size of its constituents *a*, *b* and *c*, you may spread out a conditional instruction as

if *c* **then** *a* **else** *b* **end**

or

```
if
    c
then
    a
else
    b
end
```

or

```
if c then
    a
else b end
```

You would not, however, use a line containing just **if** *c* or *c* **end**, since they include an operand together with something else, and are missing an ending operator in the first case and a starting operator in the second.

Similarly, you may start a class, after the **indexing** clause, with

```
class C inherit                -- [1]
```

or

```
class C feature                -- [2]
```

or

```
class                          -- [3]
        C
feature
```

but not

```
class C                        -- [4]
feature
```

because the first line would violate the rule. Forms [1] and [2] are used in this book for small illustrative classes; since most practical classes have one or more labeled **feature** clauses, they should in the absence of an **inherit** clause use form [3] (rather than [2]):

```
class
        C
feature -- Initialization
        …
feature -- Access
        etc.
```

Height and width

Like most modern languages, our notation does not attach any particular significance to line separations except to terminate comments, so that you can include two or more instructions (or two or more declarations) on a single line, separated by semicolons:

```
count := count + 1; forth
```

This style is for some reason not very popular (and many tools for estimating software size still measure *lines* rather than syntactical units); most developers seem to prefer having one instruction per line. It is indeed not desirable to pack texts very tightly; but in some cases a group of two or three short, closely related instructions can be more readable if they all appear on one line.

In this area it is best to defer to your judgment and good taste. If you do apply intra-line grouping, make sure that it remains moderate, and consistent with the logical relations between instructions. The Semicolon Style principle seen later in this chapter requires any same-line instructions to be separated by a semicolon.

For obvious reasons of space, this book makes a fair use of intra-line grouping, consistent with these guidelines. It also avoids splitting multi-line instructions into more lines than necessary; on this point one can recommend the book's style for general use: there is really no reason to split **from** *i*:= *1* **invariant** *i* <= *n* **until** *i* = *n* **loop** or **if** *a* = *b* **then**. Whatever your personal taste, do observe the Comb structure.

Indenting details

The comb structure uses indentation, achieved through tab characters (**not** spaces, which are messy, error-prone, and not reader-parameterizable).

Here are the indentation levels for the basic kinds of construct, illustrated by the example on the facing page:

- Level 0: the keywords introducing the primitive clauses of a class. This includes **indexing** (beginning of an indexing clause), **class** (beginning of the class body), **feature** (beginning of a feature clause, except if on the same line as **class**), **invariant** (beginning of an invariant clause, not yet seen) and the final **end** of a class.
- Level 1: beginning of a feature declaration; indexing entries; invariant clauses.
- Level 2: the keywords starting the successive clauses of a routine. This includes **require**, **local**, **do**, **once**, **ensure**, **rescue**, **end**.
- Level 3: the header comment for a routine or (for consistency) attribute; declarations of local entities in a routine; first-level instructions of a routine.

Within a routine body there may be further indentation due to the nesting of control structures. For example the earlier **if** *a* **then** … instruction contains two branches, each of them indented. These branches could themselves contain loops or conditional instructions, leading to further nesting (although the style of object-oriented software construction developed in this book leads to simple routines, seldom reaching high levels of nesting).

A **check** instruction is indented, together with the justifying comment that normally follows it, one level to the right of the instruction that it guards.

A layout example

Note: this class has no useful semantics!

```
indexing
	description: "Example for formating"
class EXAMPLE inherit
	MY_PARENT
		redefine f1, f2 end
	MY_OTHER_PARENT
		rename
			g1 as old_g1, g2 as old_g2
		redefine
			g1
		select
			g2
		end
creation
	make
```

```
feature -- Initialization
    make is
            -- Do something.
        require
            some_condition: correct (x)
        local
            my_entity: MY_TYPE
        do
            if a then
                b; c
            else
                other_routine
                    check max2 > max1 + x ^ 2 end
                    -- Because of the postcondition of other_routine.
                new_value := old_value / (max2 – max1)
            end
        end
feature -- Access
    my_attribute: SOME_TYPE
            -- Explanation of its role (aligned with comment for make)

    … Other feature declarations and feature clauses …
invariant
    upper_bound: x <= y
end -- class EXAMPLE
```

Note the trailer comment after the **end** of the class, a systematic convention.

Spaces

White space contributes as much to the effect produced by a software text as silence to the effect of a musical piece.

The general rule, for simplicity and ease of remembering, is to follow as closely as possible the practice of standard written language. By default we will assume this language to be English, although it may be appropriate to adapt the conventions to the slightly different rules of other languages.

Here are some of the consequences. You will use a space:

- Before an opening parenthesis, but not after: *f* (*x*) (not *f*(*x*), the C style, or *f*(*x*)).
- After a closing parenthesis *unless* the next character is a punctuation sign such as a period or semicolon; but not before. Hence: *proc1* (*x*); *x* := *f1* (*x*) + *f2* (*y*)
- After a comma but not before: *g* (*x*, *y*, *z*).
- After the two dash signs that start a comment: **--** A comment.

Similarly, the default rule for semicolons is to use a space after but not before:

p1; *p2* (*x*); *p3* (*y*, *z*)

Here, however, some people prefer, even for English-based software texts, the French style of including a space both before and after, which makes the semicolon stand out and emphasizes the symmetry between the components before and after it:

p1 ; *p2* (*x*) ; *p3* (*y*, *z*)

Choose either style, but then use it consistently. (This book uses the English style.) English and French styles have the same difference for colons as for semicolons; since, however, the software notation only uses colons for declarations, in which the two parts — the entity being declared and its type — do *not* play a symmetric role, it seems preferable to stick to the English style, as in *your_entity*: *YOUR_TYPE*.

Spaces should appear before and after arithmetic operators, as in $a + b$. (For space reasons, this book has omitted the spaces in a few cases, all of the form $n+1$.)

For periods the notation departs from the conventions of ordinary written language since it uses periods for a special construct, as originally introduced by Simula. As you know, *a*•*r* means: apply feature *r* to the object attached to *a*. In this case there is a space neither before nor after the period. To avoid any confusion, this book makes the period bigger, as illustrated: • rather than just .

There is another use of the period: as decimal point in real numbers, such as *3.14*. Here, to avoid any confusion, the period is not made any bigger.

Some European languages use a comma rather than a period as the separator between integral and fractional parts of numbers. Here the conflict is irreconcilable, as in English the comma serves to separate parts of big numbers, as in “300,000 dollars”, where other languages would use a space. The committee discussions for Algol 60 almost collapsed when some continental members refused to bow to the majority’s choice of the period; the stalemate was resolved when someone suggested distinguishing between a reference language, fixed, and representation languages, parameterizable. (In retrospect, not such a great idea, at least not if you ever have to compile the same program in two different countries!) Today, few people would make this a point of contention, as the spread of digital watches and calculators built for world markets have accustomed almost everyone to alternate between competing conventions.

Precedence and parentheses

The precedence conventions of the notation conform to tradition and to the “Principle of Least Surprise” to avoid errors and ambiguities.

Do not hesitate, however, to add parentheses for clarity; for example you may write (*a* = (*b* + *c*)) **implies** (*u* /= *v*) even though the meaning of that expression would be the same if all parentheses were removed. The examples in this book have systematically over-parenthesized expressions, in particular assertions, risking heaviness to avert uncertainty.

The War of the Semicolons

Two clans inhabit the computing world, and the hatred between them is as ferocious as it is ancient. The Separatists, following Algol 60 and Pascal, fight for the recognition of the semicolon as a separator between instructions; the Terminatists, rallied behind the contrasting flags of PL/I, C and Ada, want to put a semicolon behind every instruction.

Each side's arguments are endlessly relayed by its propaganda machine. The Terminatists worship uniformity: if every instruction is terminated by the same marker, no one ever has to ask the question "do I need a semicolon here?" (the answer in Terminatist languages is always yes, and anyone who forgets a semicolon is immediately beheaded for high treason). They do not want to have to add or remove a semicolon because an instruction has been moved from one syntactical location to another, for example if it has been brought into a conditional instruction or taken out of it.

The Separatists praise the elegance of their convention and its compatibility with mathematical practices. They see **do** *instruction1; instruction2; instruction3* **end** as the natural counterpart of *f (argument1, argument2, argument3)*. Who in his right mind, they ask, would prefer *f (argument1, argument2, argument3,)* with a superfluous final comma? They contend, furthermore, that the Terminatists are just a front for the Compilists, a cruel people whose only goal is to make life easy for compiler writers, even if that means making it hard for application developers.

The Separatists constantly have to fight against innuendo, for example the contention that Separatist languages will *prevent* you from including extra semicolons. Again and again they must repeat the truth: that every Separatist language worthy of the name, beginning with the venerated Patriarch of the tribe, Algol 60, has supported the notion of empty instruction, permitting all of

```
a; b; c
a; b; c;
; a ;; b ;;; c;
```

to be equally valid, and to mean exactly the same thing, as they only differ by the extra empty instructions of the last two variants, which any decent compiler will discard anyway. They like to point out how much more tolerant this convention makes them: whereas their fanatical neighbors will use any missing semicolon as an excuse for renewed attacks, the Separatists will gladly accept as many extra semicolons as a Terminatist transfuge may still, out of habit, drop into an outwardly Separatist text.

The article is a study by Gannon and Horning [Gannon 1975].

Modern propaganda needs science and statistics, so the Terminatists have their own experimental study, cited everywhere (in particular as the justification for the Terminatist convention of the Ada language): a 1975 measurement of the errors made by two groups of 25 programmers each, using languages that, among other distinguishing traits, treated semicolons differently. The results show the Separatist style causing almost ten times as many errors! Starting to feel the heat of incessant enemy broadcasts, the Separatist leadership turned for help to the author of the present book, who remembered a long-forgotten principle: *quoting is good, but reading is better.* So he fearlessly went back to the original article and discovered that the Separatist language used in the comparison — a mini-language meant only for "teaching students the concepts of asynchronous processes" — treats an extra semicolon after the final instruction of a compound, as in **begin** *a*; *b*; **end**, as an error! No real Separatist language, as noted above, has ever had such a rule, which would be absurd in any circumstance (as an extra semicolon is obviously harmless), and is even more so in the context of the article's experiment since some of the subjects apparently had Terminatist experience from PL/I and so would have been naturally prone to add a few semicolons here and there. It then seems likely, although the article gives no data on this point, that many of the semicolon errors were a result of such normally harmless additions — enough to disqualify the experiment, once and for all, as a meaningful basis for defending Terminatism over Separatism.

On some of the other issues it studies, the article is not marred by such flaws in its test languages, so that it still makes good reading for people interested in language design.

All this shows, however, that it is dangerous to take sides in such a sensitive debate, especially for someone who takes pride in having friends in both camps. The solution adopted by the notation of this book is radical:

Semicolon Syntax rule

Semicolons, as markers to delimit instructions, declarations or assertion clauses, are optional in almost all the positions where they may appear in the notation of this book.

"Almost" because of a few rare cases, not encountered in this book, in which omitting the semicolon would cause a syntactical ambiguity.

Exercise E26.2, page 902.

The Semicolon Syntax rule means you can choose your style:

- Terminatist: every instruction, declaration or assertion clause ends with a semicolon.
- Separatist: semicolons appear between successive elements but not, for example, after the last declaration of a **feature** or **local** clause.
- Moderately Separatist: like the Separatist style, but not worrying about extra semicolons that may appear as a result of habit or of elements being moved from one context to another.
- Discardist: no semicolons at all (except as per the Semicolon Style principle below).

This is one of the areas where it is preferable to let each user of the notation follow his own inclination, as the choice cannot cause serious damage. But do stick, at least across a class and preferably across an entire library or application, to the style that you have chosen (although this will not mean much for the Moderately Separatist style, which is by definition lax), and observe the following principle:

Semicolon Style principle

If you elect to include semicolons as terminators (Terminatist style), do so for all applicable elements.

If you elect to forego semicolons, use them only when syntactically unavoidable, or to separate elements that appear on the same line.

The second clause governs elements that appear two or more to a line, as in

```
found := found + 1; forth
```

which should always include the semicolon; omitting it would make the line quite confusing.

Just for once, this discussion has **no advice** here, letting you decide which of the four styles you prefer. Since the earliest version of the notation required semicolons — in other words, it had not yet been tuned to support the Semicolon Syntax rule — the first edition of this book used a Separatist style. For the present one I dabbled into a few experiments; after polling a sizable group of co-workers and experienced users of the notation, I found (apart from a handful of Terminatists) an almost equal number of Discardists and Separatists. Some of the Discardists were very forceful, in particular a university professor who said that the main reason his students loved the notation is that they do not need semicolons — a comment which any future language designer, with or without grandiose plans, should find instructive or at least sobering.

You should defer to your own taste as long as it is consistent and respects the Semicolon Style principle. (As to this book: for a while I stuck to the original Separatist style, more out of habit than of real commitment; then, hearing the approach of the third millenium and its call to start a new life free of antique superstitions, I removed all the semicolons over a single night of utter debauchery.)

Assertions

You should label assertion clauses to make the text more readable:

```
require
    not_too_small: index >= lower
```

See "Monitoring assertions at run time", page 392.

This convention also helps produce useful information during testing and debugging since, as you will remember, the assertion label will be included in the run-time message produced if you have enabled monitoring of assertions and one of them gets violated.

This convention will spread an assertion across as many lines as it has clauses. As a consequence, it is one of the rules to which the present book has made a few exceptions, again in the interest of saving space. When collapsing several clauses on one line, you should actually remove the labels for readability:

```
require
    index >= lower; index <= upper
```

In normal circumstances, that is to say for software texts rather than a printed textbook, better stick to the official rule and have one labeled clause per line.

26.6 FONTS

In typeset software texts, the following conventions, used throughout this book and related publications, are recommended.

Basic font rules

Print software elements (class names, feature names, entities…) in *italics* to distinguish them from non-software text elements. This facilitates their inclusion in sentences of the non-software text, such as "We can see that the feature *number* is a query, not an attribute". (The word *number* denotes the name of the feature; you do not want to mislead your reader into believing that you are talking about the number of features!)

Keywords, such as **class**, **feature**, **invariant** and the like, appear in **boldface**.

This was also the convention of the first edition of this book. At some stage it seemed preferable to use ***boldface italics*** which blends more nicely with italics. What was esthetically pleasing, however, turned out to hamper quality; some readers complained that the keywords did not stand out clearly enough, hence the return to the original convention. This is a regrettable case of fickleness. [M 1994a] and a handful of books by other authors show the intermediate convention.

Keywords play a purely syntactic role: they have no semantics of their own but delimit those elements, such as feature and class names, that do carry a semantic value. As noted earlier in this chapter, there are also a few non-keyword reserved words, such as *Current* and *Result*, which have a denotation of their own — expressions or entities. They are written in non-bold italics, with an initial upper-case letter.

Following the tradition of mathematics, symbols — colons and semicolons :;, brackets [], parentheses (), braces { }, question and exclamation marks ?! and so on — should always appear in roman (straight), even when they separate text in italics. Like keywords, they are purely syntactic elements.

Comments appear in roman. This avoids any ambiguity when a feature name — which, according to the principles seen earlier, will normally be a word from ordinary language — or an argument name appears in a comment; the feature name will be in italics and hence will stand out. For example:

```
accelerate (s: SPEED; t: REAL) is
        -- Bring speed to s in at most t seconds.
…
set_number (n: INTEGER) is
        -- Make n the new value of number.
…
```

In the software text itself, where no font variations are possible, such occurrences of formal elements in comments should follow a specific convention already mentioned earlier: they will appear preceded by a back quote ‘ and followed by a normal quote ’ , as in

```
-- Make ‘n’ the new value of ‘number’.
```

(Remember that you must use two different quote characters for opening and closing.) Tools that process class texts and can produce typeset output, such as **short** and **flat**, know this convention and so can make sure the quoted elements are printed in italics.

Other font conventions

The preceding font conventions work well for a book, an article or a Web page. Some contexts, however, may call for different approaches. In particular, elements in plain italics, and sometimes even bold italics, are not always readable when projected on a projection screen, especially if what you are projecting is the output of a laptop computer with a relatively small display.

In such cases I have come to using the following conventions:

- Use non-italics boldface for everything, as this projects best.
- Choose a wide enough font, such as **Bookman** (for which boldface may be called “demibold”).
- Instead of italics versus roman versus bold, use color to distinguish the various elements: keywords in black; comments in red; the rest (entities, feature names, expressions…) in blue. More colors can be used to highlight special elements.

These conventions seem to work well, although there is always room for improvement, and new media will undoubtedly prompt new conventions.

Color

The particularly attentive reader may by now have come to notice another convention used by this book: for added clarity, all formal elements — software texts or text extracts, but also mathematical elements — appear in color. This technique, which of course cannot be presented as a general requirement, enhances the effect of the rules seen so far on font usage.

26.7 BIBLIOGRAPHICAL NOTES

[Waldén 1995] is the source of the idea of showing by example that even a longer *separated_by_underscores* identifier is easier to read than an *internalUpperCase* identifier.

[Gannon 1975] is an experimental study of the effect of various language design choices on error rates.

The rules on standard feature names were first presented in [M 1990b] and are developed in detail in [M 1994a].

I received important comments from Richard Wiener on students’ appreciation of the optionality of semicolons, and from Kim Waldén on the respective merits of bold italics and plain bold.

EXERCISES

E26.1 Header comment style

Rewrite the following header comments in the proper style:

```
reorder (s: SUPPLIER; t: TIME) is
            -- Reorders the current part from supplier s, to be delivered
            -- on time t; this routine will only work if t is a time in the future.
        require
            not_in_past: t >= Now
        …
next_reorder_date: TIME is
            -- Yields the next time at which the current part is scheduled
            -- to be reordered.
```

E26.2 Semicolon ambiguity

Can you think of a case in which omitting a semicolon between two instructions or assertions could cause syntactic ambiguity, or at least confuse a simple-minded parser? (**Hint**: a feature call can have as its target a parenthesized expression, as in *(vector1 + vector2)*•*count*.)