23

Principles of class design

Experienced software developers know that few issues are more critical than the proper design of module interfaces. In a multi-person, or just multi-week software project, many of the decisions, discussions, disputes and confusions tend to revolve around matters of module interface specification: “Who takes care of making sure that…?”, “But I thought you only passed me normalized input…”, “Why are you processing this since I already took care of it?”.

If there were just one advantage to expect from object technology, this would have to be it. From the outset of this presentation, object-oriented development has been described as an architectural technique for producing systems made of coherent, properly interfaced modules. We have now accumulated enough technical background to review the design principles through which you can take advantage of the best O-O mechanisms to develop modules with attractive interfaces.

In the following pages we will explore a set of class design principles which extensive practice has shown to yield quality and durability. Because what determines the success of a class is how it will look to its clients, the emphasis here is not on the internal implementation of a class but on how to make its interface simple, easy to learn, easy to remember, and able to withstand the test of time and change.

We will successively examine: whether functions should be permitted to have side effects; how many arguments a feature should reasonably have, and the associated notions of operand and option; whether you should be concerned about the size of your classes; making abstract structures active; the role of selective exports; how to document a class; how to deal with abnormal cases.

From this discussion will emerge an image of the class designer as a patient craftsman who chisels out and polishes each class to make it as attractive as possible to clients. This spirit of treating classes as carefully engineered products, aiming at perfection from the start and yet always perfectible, is a pervasive quality of well-applied object technology. For obvious reasons it is particularly visible in the construction of library classes, and indeed many of the design principles reviewed in this chapter originated in library design; in the same way that successful ideas first tried in Formula 1 racing eventually trickle down to the engineering of cars for the rest of us, a technique that has shown its value by surviving the toughest possible test — being applied to the development of a successful library of reusable components — will eventually benefit all object-oriented software, whether or not initially intended for reuse.

23.1 SIDE EFFECTS IN FUNCTIONS

The first question that we must address will have a deep effect on the style of our designs. Is it legitimate for functions — routines that return a result — also to produce a side effect, that is to say, to change something in their environment?

The gist of the answer is no, but we must first understand the role of side effects, and distinguish between good and potentially bad side effects. We must also discuss the question in light of all we now know about classes: their filiation from abstract data types, the notion of abstraction function, and the role of class invariants.

Commands and queries

A few reminders on terminology will be useful. The features that characterize a class are divided into *commands* and *queries*. A command serves to modify objects, a query to return information about objects. A command is implemented as a procedure. A query may be implemented either as an attribute, that is to say by reserving a field in each run-time instance of the class to hold the corresponding value, or as a function, that is to say through an algorithm that computes the value when needed. Procedures (which also have an associated algorithm) and functions are together called routines.

"Attributes and routines", page 173.

The definition of queries does not specify whether in the course of producing its result a query may change objects. For commands, the answer is obviously yes, since it is the role of commands (procedures) to change things. Among queries, the question only makes sense for functions, since accessing an attribute cannot change anything. A change performed by a function is known as a *side effect* to indicate that it is ancillary to the function's official purpose of answering a query. Should we permit side effects?

Forms of side effect

Let us define precisely what constructs may cause side effects. The basic operation that changes an object is an assignment *a* := *b* (or an assignment attempt *a* ?= *b*, or a creation instruction !! *a*) where the target *a* is an attribute; execution of this operation will assign a new value to the field of the corresponding object (the target of the current routine call).

We only care about such assignments when *a* is an attribute: if *a* is a local entity, its value is only used during an execution of the routine and assignments to it have no permanent effect; if *a* is the entity *Result* denoting the result of the routine, assignments to it help compute that result but have no effect on objects.

Also note that as a result of information hiding principles we have been careful, in the design of the object-oriented notation, to avoid any indirect form of object modification. In particular, the syntax excludes assignments of the form *obj.attr* := *b*, whose aim has to be achieved through a call *obj.set_attr* (*b*), where the procedure *set_attr* (*x*:…) performs the attribute assignment *attr* := *x*.

"The client's privileges on an attribute", page 206.

The attribute assignment that causes a function to produce a side effect may be in the function itself, or in another routine that the function calls. Hence the full definition:

Definition: concrete side effect

A function produces a concrete side effect if its body contains any of the following:

- An assignment, assignment attempt or creation instruction whose target is an attribute.
- A procedure call.

(The term "concrete" will be explained below.) In a more fine-tuned definition we would replace the second clause by "A call to a routine that (recursively) produces a concrete side effect", the definition of side effects being extended to arbitrary routines rather than just functions. But the above form is preferable in practice even though it may be considered both too strong and too weak:

- The definition seems too strong because any procedure call is considered to produce a side effect whereas it is possible to write a procedure that changes nothing. Such procedures, however, are rarely useful — except if their role is to change something in the software's environment, for example printing a page, sending a message to the network or moving a robot arm; but then we do want to consider this a side effect even if it does not directly affect an object of the software itself.

- The definition seems too weak because it ignores the case of a function f that calls a side-effect-producing function g. The convention will simply be that f can still be considered side-effect-free. This is acceptable because the rule at which we will arrive in this discussion will prohibit *all* side effects of a certain kind, so we will need to certify each function separately.

The advantage of these conventions is that to determine the side-effect status of a function you only need to look at the body of the function itself. It is in fact trivial, if you have a parser for the language, to write a simple tool that will analyze a function and tell you whether it produces a concrete side effect according to the definition.

Referential transparency

Why should we be concerned about side effects in functions? After all it is in the nature of software execution to change things.

"Introducing a more imperative view", page 145.

The problem is that if we allow functions to change things as well as commands, we lose many of the simple mathematical properties that enable us to reason about our software. As noted in the discussion of abstract data types, when we first encountered the distinction between the applicative and the imperative, mathematics is change-free: it talks about abstract objects and defines operations on these objects, but the operations do not change the objects. (Computing $\sqrt{2}$ does not change the number two.) This immutability is the principal difference between the worlds of mathematics and computer software.

Some approaches to programming seek to retain the immutability of mathematics: Lisp in its so-called "pure" form, "Functional Programming" languages such as Backus's FP, and other *applicative* languages shun change. But they have not caught on for practical software development, suggesting that change is a fundamental property of software.

The object immutability of mathematics has an important practical consequence known as *referential transparency*, a property defined as follows:

Definition: referential transparency

An expression *e* is referentially transparent if it is possible to exchange any subexpression with its value without changing the value of *e*.

Definition from "The Free On-Line Dictionary of Computing", http://wombat.

If *x* has value three, we can use *x* instead of *3*, or conversely, in any part of a referentially transparent expression. (Only Swift's Laputa academicians were willing to pay the true price of renouncing referential transparency: always carrying around all the things you will ever want to talk about.) As a consequence of the definition, if we know that *x* and *y* have the same value, we can use one interchangeably with the other. For that reason referential transparency is also called "substitutivity of equals for equals".

The Swift quotation was on page 672.

With side-effect-producing functions, referential transparency disappears. Assume a class contains the attribute and the function

```
attr: INTEGER
sneaky: INTEGER is do attr := attr + 1 end
```

Remember that Result in an integer function is initialized to zero.

Then the value of *sneaky* (meaning: of a call to that function) is always 0; but you cannot use *0* and *sneaky* interchangeably, since an extract of the form

```
attr := 0; if attr /= 0 then print ("Something bizarre!") end
```

will print nothing, but would print *Something bizarre!* if you replaced *0* by *sneaky*.

See [Dijkstra 1968].

Maintaining referential transparency in expressions is important to enable us to reason about our software. One of the central issues of software construction, analyzed clearly by Dijkstra many years ago, is the difficulty of getting a clear picture of the dynamic behavior (the myriad possible executions of even a simple software element) from its static description (the text of the element). In this effort it is essential to be able to rely on the proven form of reasoning, provided by mathematics. With the demise of referential transparency, however, we lose basic properties of mathematics, so deeply rooted in our practice that we may not even be aware of them. For example, it is no longer true that $n + n$ is the same thing as $2 * n$ if n is the *sneaky*-like function

```
n: INTEGER is do attr := attr + 1; Result := attr end
```

since, with *attr* initially zero, $2 * n$ will return *2* whereas $n + n$ will return *3*.

By limiting ourselves to functions that do not produce side effects, we will ensure that talking about "functions" in software ceases to betray the meaning of this term in ordinary mathematics. We will maintain a clear distinction between commands, which

change objects but do not directly return results, and queries, which provide information about objects but do not change them.

Another way to express this rule informally is to state that *asking a question should not change the answer*.

Objects as machines

The following principle expresses the prohibition in more precise terms:

Command-Query Separation principle

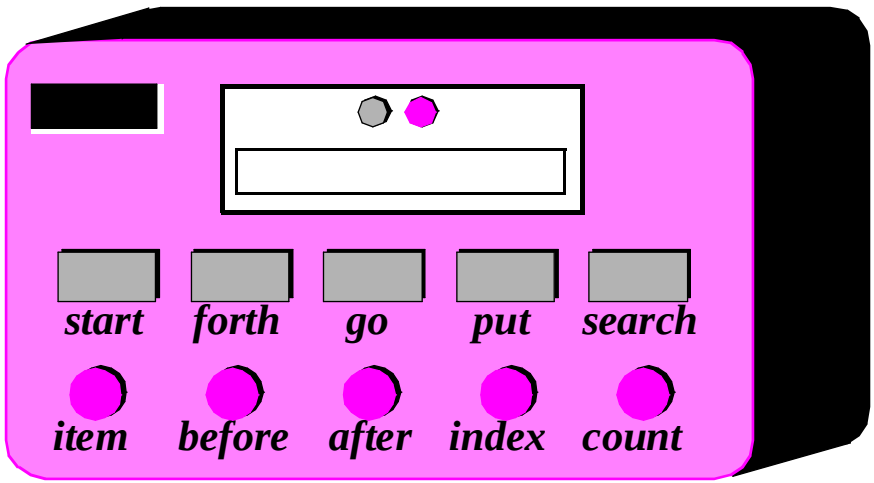
Functions should not produce abstract side effects.

The definition of abstract side effects appears on page 757.

Note that we have only defined *concrete* side effects so far; for the moment you can ignore the difference.

As a result of the principle, only commands (procedures) will be permitted to produce side effects. (In fact, as noted, we not only permit but expect them to change objects — unlike in applicative, completely side-effect-free approaches.)

A list object as list machine



The view of objects that emerges from this discussion (a *metaphor*, to be treated with care as usual) is that of a machine, with an internal state that is not directly observable, and two kinds of button: command buttons, rectangular on the picture, and query buttons, round.

Object lifecycle picture: page 366.

Pressing a command button is a way to make the machine change state: it starts moving and clicking, then comes back to a new stable state (one of the states shown in the earlier picture of object lifecycle). You cannot directly see the state — open the machine — but you can press a query button. This does not change the state (remember: asking a question does not change the answer) but yields a response in the form of a message appearing in the display panel at the top; for boolean queries one of the two indicators in the display panel,

representing true and false, will light up. If you press the button several times in a row, without touching the command buttons, you will get the same result each time. If, on the other hand, you push a command button and then a query button, the answer that you get will usually be different from what you would have obtained before the command.

Commands as well as queries may take arguments; these are figuratively entered in the slot at the top left.

The figure is based on the example of a list object with the kind of interface hinted at in earlier chapters and studied in more detail later in the present one. Commands include *start* (move the cursor to the first element), *forth* (advance the cursor one position), *search* (move the cursor to the next occurrence of the element entered into the top-left slot); queries include *item* (show in the display panel the value of the element at cursor position) and *index* (show the current cursor position). Note the difference between a notion such as "cursor", relative to the internal state and hence not directly visible, and *item* or *index* which provide more abstract, officially exported information about the state.

Functions that create objects

A technical point needs to be clarified before we examine further consequences of the Command-Query Separation principle: should we treat object creation as a side effect?

The answer is yes, as we have seen, if the target of the creation is an attribute *a*: in this case, the instruction !! *a* changes the value of an object's field. The answer is no if the target is a local entity of the routine. But what if the target is the result of the function itself, as in !! *Result* or the more general form !! *Result*.*make* (…)?

Such a creation instruction need not be considered a side effect. It does not change any existing object and so does not endanger referential transparency (at least if we assume that there is enough memory to allocate all the objects we need). From a mathematical perspective we may pretend that all of the objects of interest, for all times past, present and future, are already inscribed in the Great Book of Objects; a creation instruction is just a way to obtain one of them, but it does not by itself change anything in the environment. It is common, and legitimate, for a function to create, initialize and return such an object.

> These observations assume that in the second form the creation procedure *make* does not produce side effects on any object other than the one being created.

A clean style for class interfaces

From the Command-Query Separation principle follows a style of design that yields simple and readable software, and tremendously helps reliability, reusability and extendibility.

As you may have realized, this style is very different from the dominant practices of today, as fostered in particular by the C programming language. The predilection of C for side effects — for ignoring the difference between an action and a value — is not just a feature of the common C style (it sometimes seems just psychologically impossible for a C programmer to resist the temptation, when accessing a value, also to modify it a little in

passing); it is embedded deeply into the language, with such constructs as *x++*, meaning: return the value of *x*, then increase it by one — saving a few keystrokes in *y = x++* compared to *y = x; x := x+1*, and not to be confused with *++x* which increments *before* computing the value. A whole civilization, in fact, is built on side effects.

It would be foolish to dismiss this side-effect-full style as thoughtless; its widespread use shows that many people have found it convenient, and it may even be part of the reason for the amazing success of C and its derivatives. But what was attractive in the nineteen-seventies and eighties — when the software development population was growing by an order of magnitude every few years, and the emphasis was on getting some kind of job done rather than on long-term quality — may not be appropriate for the software technology of the twenty-first century. There we want software that will grow with us, software that we can understand, explain, maintain, reuse and trust. The Command-Query Separation principle is one of the required conditions for these goals.

Applying a strict separation between commands and queries by prohibiting abstract side effects in functions is particularly appropriate for the development of large systems, where the key to success is to exert full control on every inter-module interaction.

If you have been used to the converse style, you may at first, like many people, find the new one too extreme. But after starting to practice it I think you will quickly realize its benefits.

Quietly, the preceding chapters have already applied Command-Query Separation to its full extent. You may remember for example that the interface for all our stack classes included a procedure *remove* describing the operation of popping a stack (removing the top element), and a function or attribute *item* which yields the top element. The first is a command, the second a query. In other approaches you might have seen a routine *pop* which both removes the element and returns it — a side-effect-producing function. This example has, I hope, been studied in enough depth to show the gains of clarity and simplicity that we achieve by keeping the two aspects cleanly separated.

Other consequences of the principles may seem more alarming at first. For reading input, many people are used to the style of using functions such as *getint* — the C name, but its equivalent exists in many other languages — whose effect is to read a new input element and return its value. This is a side-effect-producing function in all its splendor: a call to the function, written *getint ()* — with the empty parentheses so unmistakably characteristic of the C look-and-feel — does not just return a value but affects the context ("asking a question changes the answer"); as typical consequences, excluding the chance case in which the input has two identical consecutive values:

- If you call *getint ()* twice you will get different answers.
- *getint () + getint ()* and *2 * getint ()* will not yield the same value. (If an overzealous "optimizing" compiler treats the first expression like the second, you will report a bug to the compiler vendor, and you will be right.)

In other words, we lose the benefits of referential transparency — of reasoning about software functions as if they were mathematical functions, with a crystal-clear view of how we can build expressions from them and what values these expressions will denote.

The Command-Query Separation principle brings referential transparency back. Here this means that we will distinguish between the procedure that advances the input cursor to the next item and the function or attribute that yields the item last read. Assume *input* is of type *FILE*; the instructions to read the next integer from file *input* will be something like

```
input.advance
n := input.last_integer
```

If you call *last_integer* ten times in a row you will, unlike with *getint*, get ten times the same result. If you are new to this style, it may take some getting used to; but the resulting simplicity and clarity will soon remove any temptation to go back to side effects.

> In this example as in the *x++* case seen earlier, the traditional form beats the object-oriented one if the goal of the game is to minimize keystrokes. This illustrates a general observation: the productivity gains of object technology will not derive from trying to be as terse as possible on a microscopic scale (a game at which APL or modern "scripting languages" such as Perl will always win against a good O-O language). The achievements are on the global structure of a system: through reuse, through such mechanisms as genericity and garbage collection, through the use of assertions, you can decrease the size of your software by amounts far higher than anything you can achieve by skimping by a character here or a line there. Keystroke-wise is often system-foolish.

Pseudo-random number generators: a design exercise

An example sometimes quoted in favor of functions with side effects is that of pseudo-random number generators, which return successive values from a sequence enjoying adequate statistical properties. The sequence is initialized by a call of the form

```
random_seed (seed)
```

where *seed* is a seed value provided by the client. A common way to get the successive pseudo-random values is by calling a function:

```
xx := next_random ()
```

But here too there is no reason to make an exception to the command/query dichotomy. Before looking at the solution let us just forget that we have seen the above and restart from scratch by asking the question: how should we handle random generation in an object-oriented context? This will provide the opportunity of a little design exercise, and will enable us, if the need arises, to explain the results to someone whose view has not been unduly influenced by pre-O-O approaches.

As always in object technology, the relevant question — often the only one — is:

What are the data abstractions?

The relevant abstraction here is not "random number generation" or "random number generator", both of them quite functional in nature, focusing on *what the system does* rather than *what it does it to*.

Probing further, we might think "random number", but that is not the right answer yet. Remember, a data abstraction is characterized by features — commands and queries; it is hard to think of features applicable to "random number".

"Discovery and rejection", page 725.

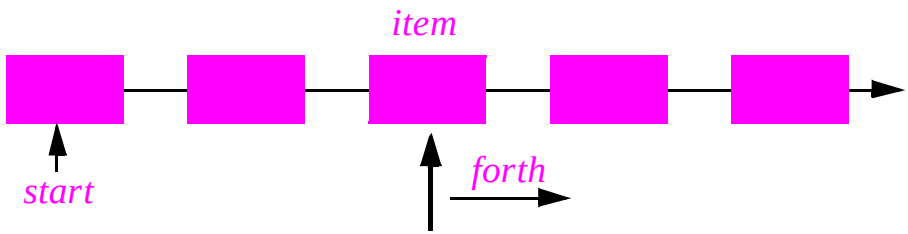
That "random number" leads to a dead end illustrates the Class Elicitation principle encountered when we studied the general rules for finding the classes: a key step may be to *reject* inappropriate candidates. And once again we see that not all promising nouns yield classes: were a "requirements document" written for this problem, the noun *random number* would certainly figure prominently in it.

A random number does not mean much by itself; it must be understood in relation to its predecessors and successors in the sequence.

Wait a minute — here we have it: *sequence*, more precisely pseudo-random number sequence. This is the abstraction we have been looking for; a perfectly legitimate data abstraction, similar to the cursor lists we have seen on a number of occasions, only infinite (do not look for an *after* boolean query!). Features will include:

- Commands: *make* — initialize with a certain seed; *forth* — advance to next element.
- Queries: *item* — return the element at cursor position.

An infinite list as a machine



To get a new random number sequence *rand*, clients will use `!! rand.make (seed)`; to advance to the next value, they will call `rand.forth`; and they will obtain the current value by `xx := rand.item`.

There is really nothing specific to *random number* sequences in the interface, except for the *seed* argument to the creation procedure. Adding a *start* procedure which brings the cursor to the first item (and which *make* may call for random number sequences), what we have is the framework for a deferred class *COUNTABLE_SEQUENCE* describing arbitrary infinite sequences. Think for example of how to model prime numbers in an object-oriented way; the answer is the same: define a class *PRIMES*, an heir to *COUNTABLE_SEQUENCE*, whose successive elements are the prime numbers. Other sequences — Fibonacci numbers and the like — will be modeled in the same way.

These examples illustrate in passing that contrary to popular belief it is quite possible, and even trivial, to represent infinite structures on a computer. Abstract data types provide the key: a structure is entirely defined by the applicable operations, of which there is of course a finite number, three in this case — *start*, *forth*, *item* — plus any auxiliary features we may want to add. The trick, of course, is that any execution will only try to evaluate a finite number of elements of an infinite structure.

[M 1994a].

COUNTABLE_SEQUENCE and its heirs such as *PRIMES* are part of the universal computing science hierarchy described in the companion guide to reusable components.

Abstract state, concrete state

From the discussion of referential transparency it would seem desirable to bar all concrete side effects from functions. Such a rule would have the advantage that — in line with one of our methodology precepts — we could build it into the language, since a compiler can easily detect concrete side effects (as we saw just after the definition of this notion).

"If it is baroque, fix it", page 670.

Unfortunately, this would be unacceptably restrictive, explaining why the Command-Query Separation principle only prohibits *abstract* side effects, a notion that will now be defined. The problem is that some concrete side effects are not only harmless but necessary. They are of two kinds.

The first category includes functions which, in the course of their execution, modify the state, sometimes drastically, and affecting very visible features; but then they restore the original state. Consider for example a class describing integer lists with cursor, and the following function for computing the maximum of a list:

```
max is
            -- The highest value of items in the list
        require
            not empty
        local
            original_index: INTEGER
        do
            original_index := index
            from
                start; Result := item
            until is_last loop
                forth; Result := Result.max (item)
            end
            go (original_index)
        end
```

To traverse the list, the algorithm needs to move the cursor over all elements. The function, calling such procedures as *start*, *forth* and *go*, is indeed full of concrete side effects on the cursor position; but it begins by noting the cursor position into *original_index* and ends by returning the cursor to that position through a call to *go*. All is well that ends well: the function leaves the list in exactly the state in which it found it. But no compiler in the world is going to detect that the side effect is only apparent.

Side effects of the second acceptable category may change the state of the object, but only affecting properties that are not visible to clients. To understand the concepts in depth, it will be useful to make sure that you are familiar with the discussion of "abstraction function" and "implementation invariants" in the presentation of Design by Contract. (In particular, take a look at the accompanying figures to refresh your memory.)

Pages 376 to 378.

We saw then that an object of our software (a *concrete* object) is the representation of an abstract object, and that two concrete objects may represent the same abstract object. For example two different stack representations, each made of an array and a top marker *count*, represent the same stack if they have the same value for *count* and the same array elements up to index *count*. They may differ in other properties, such as the array sizes and the values stored at indices above *count*. In mathematical terms, every concrete object belongs to the domain of the abstraction function *a*, and we can have $c1 \neq c2$ even with $a\ (c1) = a\ (c2)$.

What this means for us is that a function that modifies a concrete object is harmless if the result of this modification still represents the same abstract object — yields the same *a* value. For example assume in a function on stacks contains the operation

representation.*put* (*some_value*, *count* + *1*)

(with the guarantee that the array's capacity is at least *count* + *1*). This side effect changes a value above the stack-significant section of the array; it can do no ill.

More generally, a concrete side effect which changes the concrete state of an object *c* is an *abstract side effect* if it also changes its *abstract state*, that is to say the value of $a\ (c)$ (a more directly usable definition of abstract side effects will appear shortly). If a side effect is only concrete — does not affect the abstract state — it is harmless.

Figure page 751.

In the object-as-machine metaphor, functions producing concrete-only side effects correspond to query buttons that may produce an internal state change having absolutely no effect on the answers given by any query button. For example the machine might save energy by automatically switching off some internal circuits if nobody presses a button for some time, and turning them on again whenever someone presses any button, queries included. Such an internal state change is unnoticeable from the outside and hence legitimate.

The object-oriented approach is particularly favorable to clever implementations which, when computing a function, may change the concrete state behind the scenes without producing any visible side effect. The example of a stack function that changes array elements above the top is somewhat academic, but we will see below a practical and useful design that relies on this technique.

Since not every class definition is accompanied by a full-fledged specification of the underlying abstract data type, we need a more directly usable definition of "abstract side effect". This is not difficult. In practice, the abstract data type is defined by the interface offered by a class to its clients (expressed for example as the short form of the class). A side effect will affect the abstract object if it changes the result of any query accessible to these clients. Hence the definition:

Definition: abstract side effect

An abstract side effect is a concrete side effect that can change the value of a non-secret query.

This is the notion used by the Command-Query Separation principle — the principle that prohibits abstract side effects in functions.

The principle appears on page 751.

The definition refers to "non-secret" rather than exported queries. The reason is that in-between generally exported and fully secret status, we must permit a query to be selectively exported to a set of clients. As soon as a query is non-secret — exported to any client other than *NONE* — we consider that changing its result is an abstract side effect, since the change will be visible to at least some clients.

The policy

As announced at the beginning of this discussion, abstract side effects are (unlike concrete side effects) not easily detectable by a compiler. In particular it does not suffice to check that a function preserves the values of all non-secret attributes: the effect on other queries might be indirect, or (as in the *max* example) several concrete side effects might in the end cancel out. The most a compiler can do would be to issue a warning if a function modifies an exported attribute.

So the Command-Query Separation principle is a methodological precept, not a language constraint. This does not, however, diminish its importance.

Past what for some people will be an initial shock, every object-oriented developer should apply the principle without exception. I have followed it for years, and would never write a side-effect-producing function. ISE applies it in all its O-O software (for the C part we have of course to adapt to the dominant style, although even here we try to apply the principle whenever we can). It has helped us produce much better results — tools and libraries that we can reuse, explain to others, extend and scale up.

Objections

It is important here two deal with two common objections to the side-effect-free style.

The first has to do with error handling. Sometimes a function with side effects is really a procedure, which in addition to doing its job returns a status code indicating how things went. But there are better ways to do this; roughly speaking, the proper O-O technique is to enable the client, after an operation on an object, to perform a query on the status, represented for example by an attribute of the object, as in

```
target.some_operation (…)
how_did_it_go := target.status
```

Note that the technique of returning a status as function result is lame anyway. It transforms a procedure into a function by adding the status as a result; but it does not work if the routine was already a function, which already has a result of its own. It is also problematic if you need more than one status indicator. In such cases the C approach is either to return a "structure" (the equivalent of an object) with several components, which is getting close to the above scheme, or to use global variables — which raises a whole set of new problems, especially in a large system where many modules can trigger errors.

The second objection is a common misconception: the impression that Command-Query Separation, for example the list-with-cursor type of interface, is incompatible with concurrent access to objects. That belief is remarkably widespread (this is one of the places where I know that, if I am lecturing on these topics, someone in the audience will raise his hand, and the question will be the same whether we are in Santa Barbara, Seattle, Singapore, Sydney, Stockholm or Saint-Petersburg); but it is incorrect nonetheless.

Chapter 30.

The misconception is that in a concurrent context it is essential to have atomic access-cum-modification operations, for example *get* on a buffer — the concurrent equivalent of a first-in, first out queue. Such a *get* function non-interruptibly performs, in our terminology, both a call to *item* (obtain the oldest element) and *remove* (remove that element), returning the result of *item* as the result of *get*. But using such an example as an argument for *get*-style functions with side effects is confusing two notions. What we need in a concurrent context is a way to offer a client exclusive access to a supplier object for certain operations. With such a mechanism, we can protect a client extract of the form

x := buffer.item; buffer.remove

thereby guaranteeing that the buffer element returned by the call to *item* is indeed the same one removed by the following call to *remove*. Whether or not we permit functions to have side effects, we will have to provide a mechanism to ensure such exclusive access; for example a client may need to dequeue two elements

buffer.remove; buffer.remove

with the guarantee that the removed elements will be consecutive; this requires exclusive access, and is unrelated to the question of side effects in functions.

Chapter 30. See in particular "Support for command-query separation", page 1029.

Later in this book we will have an extensive discussion of concurrency, where we will study a simple and elegant approach to concurrent and distributed computation, fully compatible with the Command-Query Separation principle — which in fact will help us arrive at it.

Legitimate side effects: an example

To conclude this discussion of side effects let us examine a typical case of legitimate side effects — functions that do not change the abstract state, but can change the concrete state, and for good reason. The example is representative of a useful design pattern.

Consider the implementation of complex numbers. As with points, discussed in an earlier chapter, two representations are possible: cartesian (by axis coordinates *x* and *y*) and polar (by distance to the origin ρ and angle θ). Which one do we choose? There is no easy answer. If we take, as usual, the abstract data type approach, we will note that what counts is the applicable operations — addition, subtraction, multiplication and division among others, as well as queries to access *x*, *y*, ρ and θ — and that for each of them one of the representations is definitely better: cartesian for addition, subtraction and such, polar for multiplication and division. (Try expressing division in cartesian coordinates!)

We could let the client decide what representation to use. But this would make our classes difficult to use, and violate information hiding: for the client author, the representation should not matter.

Alternatively, we could keep *both* representations up to date at all times. But this may cause unnecessary performance penalties. Assume for example that a client only performs multiplications and divisions. The operations use polar representations, but after each one of them we must recompute *x* and *y*, a useless but expensive computation involving trigonometric functions.

A better solution is to refuse to choose between the representations *a priori*, but update each of them only when we need it. As compared to the preceding approach, we do not gain anything in space (since we will still need attributes for each of *x*, *y*, ρ and θ, plus two boolean attributes to tell us which of the representations are up to date); but we avoid wasting computation time.

We may assume the following public operations, among others:

```
class COMPLEX feature
    … Feature declarations for:
            infix "+", infix "–", infix "*", infix "/",
            add, subtract, multiply, divide,
            x, y, rho, theta, …
end
```

The queries *x*, *y*, *rho* and *theta* are exported functions returning real values. They are always defined (except *theta* for the complex number 0) since a client may request the *x* and *y* of a complex number even if the number is internally represented in polar, and its ρ and θ even if it is in cartesian. In addition to the functions "+" etc., we assume procedures *add* etc. which modify an object: *z1 + z2* is a new complex number equal to the sum of *z1* and *z2*, whereas the procedure call *z1.add (z2)* changes *z1* to represent that sum. In practice, we might need only the functions or only the procedures.

Internally, the class includes the following secret attributes for the representation:

```
cartesian_ready: BOOLEAN
polar_ready: BOOLEAN
private_x, private_y, private_rho, private_theta: REAL
```

Not all of the four real attributes are necessarily up to date at all times; in fact only two need be up to date. More precisely, the following implementation invariant should be included in the class:

```
invariant
    cartesian_ready or polar_ready
    polar_ready implies (0 <= private_theta and private_theta <= Two_pi)
        -- cartesian_ready implies (private_x and private_y are up to date)
        -- polar_ready implies (private_rho and private_theta are up to date)
```

The value of *Two_pi* is assumed to be 2π. The last two clauses may only be expressed informally, in the form of comments.

At any time at least one of the representations is up to date, although both may be. Any operation requested by a client will be carried out in the most appropriate representation; this may require computing that representation if it was not up to date. If the operation produces a (concrete) side effect, the other representation will cease to be up to date.

Two secret procedures are available for carrying out representation changes:

```
prepare_cartesian is
                -- Make cartesian representation available
        do
                if not cartesian_ready then
                                check polar_ready end
                                -- (Because the invariant requires at least one of the
                                -- two representations to be up to date)
                        private_x := private_rho * cos (private_theta)
                        private_y := private_rho * sin (private_theta)
                        cartesian_ready := True
                                -- Here both cartesian_ready and polar_ready are true:
                                -- Both representations are available
                end
        ensure
                cartesian_ready
        end

prepare_polar is
                -- Make polar representation available
        do
                if not polar_ready then
                                check cartesian_ready end
                        private_rho := sqrt (private_x ^ 2 + private_y ^ 2)
                        private_theta := atan2 (private_y, private_x)
                        polar_ready := True
                                -- Here both cartesian_ready and polar_ready are true:
                                -- Both representations are available
                end
        ensure
                polar_ready
        end
```

Functions *cos*, *sin*, *sqrt* and *atan2* are assumed to be taken from a standard mathematical library; *atan2* (*y*, *x*) should compute the arc tangent of *y* / *x*.

We will also need creation procedures *make_cartesian* and *make_polar*:

```
make_cartesian (a, b: REAL) is
        -- Initialize with abscissa a, ordinate b.
    do
        private_x := a; private_y := b
        cartesian_ready := True; polar_ready := False
    ensure
        cartesian_ready; not polar_ready
    end
```

and symmetrically for *make_polar*.

The exported operations are easy to write; we can start for example with the procedure variants (we will see the function variants such as **infix** "+" next):

```
add (other: COMPLEX) is
        -- Add the value of other.
    do
        prepare_cartesian; polar_ready := False
        private_x := x + other.x; private_y = y + other.y
    ensure
        x = old x + other.x; y = old y + other.y
        cartesian_ready; not polar_ready
    end
```

(Note the importance in the postcondition of using *x* and *y*, not *private_x* and *private_y* which might not have been up to date before the call.)

```
divide (z: COMPLEX) is
        -- Divide by z.
    require
        z.rho /= 0
            -- (To be replaced by a numerically more realistic precondition)
    do
        prepare_polar; cartesian_ready := False
        private_rho := rho / other.rho
        private_theta = (theta – other.theta) \\ Two_pi
                                        -- Using \\ as remainder operation
    ensure
         rho = old rho / other.rho
         theta = (old theta — other.theta) \\ Two_pi
        polar_ready; not cartesian_ready
    end
```

and similarly for *subtract* and *multiply*. (The precondition and postcondition may need some adaptation to reflect the realities of floating-point computations on computers.) The function variants follow the same pattern:

```
infix "+" (other: COMPLEX): COMPLEX is
        -- Sum of current complex and other
    do
        !! Result.make_cartesian (x + other.x, y + other.y)
    ensure
        Result.x = x + other.x; Result.y = y + other.y
        Result.cartesian_ready
    end

infix "/" (z: COMPLEX): COMPLEX is
        -- Quotient of current complex by z.
    require
        z.rho /= 0
                -- (To be replaced by a numerically more realistic condition)
    do
        !! Result.make_polar (rho / other.rho, (theta – other.theta) \\ Two_pi)
    ensure
        Result.rho = rho / other.rho
        Result.theta = (old theta — other.theta) \\ Two_pi
        Result.polar_ready
    end
```

and similarly for **infix** "–" and **infix** "*".

Note that for the last postcondition clauses of these functions to be valid, *cartesian_ready* and *polar_ready* must be exported to the class itself, by appearing in a clause of the form **feature** {*COMPLEX*}; they are not exported to any other class.

But where are the side effects? In the last two functions, they are not directly visible; this is because *x*, *y*, *rho* and *theta*, behind their innocent looks, are sneaky little side-effectors! Computing *x* or *y* will cause a secret change of representation (a call to *prepare_cartesian*) if the cartesian representation was not ready, and symmetrically for *rho* and *theta*. Here for example are *x* and *theta*:

```
x: REAL is
        -- Abscissa
    do
        prepare_cartesian; Result := private_x
    end

theta: REAL is
        -- Angle
    do
        prepare_polar; Result := private_theta
    end
```

Functions *y* and *rho* are similar. All these functions call a procedure which may trigger a change of state. Unlike *add* and consorts, however, they do not invalidate the previous representation when a new one is computed. For example, if *x* is called in a state where *cartesian_ready* is false, both representations (all four real attributes) will be up to date afterwards. This is because the functions may produce side effects on the concrete objects only, not on the associated abstract objects. To express this property more formally: computing $z.x$ or one of the other functions may change the concrete object associated with z, say from c_1 to c_2, but always with the guarantee that

$$a\ (c_1) = a\ (c_2)$$

where a is the abstraction function. The computer objects c_1 and c_2 may be different, but they represent the same mathematical object, a complex number.

Such side effects are harmless, as they only affect secret attributes and hence cannot be detected by clients.

The object-oriented approach encourages such flexible, self-adapting schemes, which internally choose the best implementation according to the needs of the moment. As long as the resulting implementation changes affect the concrete state but not the abstract state, they can appear in functions without violating the Command-Query Separation principle or endangering referential transparency for clients.

23.2 HOW MANY ARGUMENTS FOR A FEATURE?

In trying to make classes — especially reusable classes — easy to use, you should devote special attention to the number of arguments of features. As we will see, well-understood object technology yields a style of feature interface radically different from what you typically get with traditional approaches; there will, in particular, be far fewer arguments.

The importance of argument counts

When your development relies on a supplier class, features are your day-to-day channel to it. The simplicity of the feature interfaces fundamentally determines the class's ease of use. Various factors influence this, in particular the consistency of the conventions; but in the end a simple numerical criterion dominates everything else: how many arguments do features have? The more arguments, the more you have to remember.

This is particularly true of library classes. The criterion for success there is simple: after a potential library user has taken the (preferably short) time to understand what a class is about and, if he decides to use it, selected the set of features that he needs for the moment, he should be able to learn these features quickly and, after as few uses as possible, remember them without having to go back to the documentation. This will only work if features — aside from all other qualities of consistency, proper naming conventions and general quality of the design — have very short argument lists.

If you examine a typical subroutine library you will commonly encounter subroutines with many arguments. Here for example is an integration routine from a mathematical library justly renowned for the excellence of its algorithms, but constrained in its interface by the use of traditional subroutine techniques:

Warning: this is not an object-oriented interface!

```
nonlinear_ode
        (equation_count: in INTEGER;
         epsilon: in out DOUBLE;
        func: procedure      (eq_count: INTEGER; a: DOUBLE; eps: DOUBLE;
                              b: ARRAY [DOUBLE]; cm: pointer Libtype)
        left_count, coupled_count: in INTEGER;
        …)
        [And so on. Altogether 19 arguments, including:
                - 4 in out values;
                - 3 arrays, used both as input and output;
                - 6 functions, each with 6 or 7 arguments of which 2 or 3 are arrays!]
```

Since the purpose of this example is not to criticize one particular numerical library but to emphasize the difference between O-O and traditional interfaces, the routine and arguments names have been changed and the syntax (in C in the original) has been adapted. The resulting notation resembles the notation of this book, which, however, would of course exclude such non-O-O mechanisms as **in out** arguments, explicit **pointer** manipulation, and arguments (such as *func* and 5 others) that are themselves routines.

Several properties make this scheme particularly complex to use:

- Many arguments are **in out**, that is to say must be initialized by the caller to pass a certain value and are updated by the routine to return some information. For example *epsilon* specifies on input whether continuation is required (yes if less than 0; if between 0 and 1, continuation is required unless *epsilon* < $\sqrt{precision}$, etc.). On output, it provides an estimate of the increment.
- Many arguments, both to the routine itself and to its own routine arguments, are arrays, which again serve to pass certain values on input and return others on output.
- Some arguments serve to specify the many possibilities for error processing (stop processing, write message to a file, continue anyway…).

Even though high-quality numerical libraries have been in existence for many years and, as mentioned in an earlier chapter, provide some of the most concrete evidence of real reuse, they are still not as widely used in scientific computation as they should be. The complexity of their interfaces, and in particular the large number of arguments illustrated by *nonlinear_ode*, are clearly a big part of the reason.

On the Math library and techniques of scientific object-oriented computing, see [Dubois 1997]. *The earlier mention was in* *"Object-oriented re-architecturing", page 441*.

Part of the complexity comes from the problems handled by these routines. But one can do better. An object-oriented numerical library, *Math*, offers a completely different approach, consistent with object technology concepts and with the principles of this book. An earlier discussion cited the Math library as an example of using object technology to re-architecture older software, and the library indeed uses an existing non-O-O library as its core engine, since it would have been absurd to duplicate the basic algorithmic work; but it provides a modern, O-O client interface. The basic non-linear ODE routine has the form

```
solve
        -- Solve the problem, recording the answer in x and y.
```

In other words it takes no argument at all! You simply create an instance of the class *GENERAL_BOUNDARY_VALUE_PROBLEM* to represent the mathematical problem to be solved, set its non-default properties through calls to the appropriate procedures, attach it to a "problem solver" object (an instance of the class in which the above routine appears: *GENERAL_BOUNDARY_VALUE_PROBLEM_SOLVER*), and call *solve* on that object. Attributes of the class, *x* and *y*, will provide the handle to the computed answer.

More generally, the thorough application of O-O techniques has a dramatic effect on argument counts. Measures on the ISE libraries, published in more detailed elsewhere, show an average number of arguments ranging from 0.4 for the Base libraries to 0.7 for the *Vision* graphical library. For the purposes of comparison with non-O-O libraries we should add 1 to all these figures, since we count two arguments for $x.f\ (a, b)$ versus three for its non-O-O counterpart $f\ (x, a, b)$; but even so these averages are strikingly low when compared with the counts for non-O-O routines which, even when not reaching 19 as in the above numerical example, often have 5, 10 or 15 arguments.

See [M 1994a] *for detailed library measurements.*

These numbers are not a goal by themselves — and of course not by themselves an indicator of quality. Instead, they are largely the result of a deeper design principle that we will now examine.

Operands and options

An argument to a routine may be of two different kinds: *operands* and *options*.

To understand the difference, consider the example of a class *DOCUMENT* and a procedure *print*. Assume — just to make the example more concrete — that printing will rely on Postscript. A typical call illustrating a possible interface (*not* compatible with the principle stated below) would be

```
my_document.print (printer_name, paper_size, color_or_not,
                   postscript_level, print_resolution)
```

Warning: this is not the recommended style!

Of the five arguments, which ones are truly indispensable? If you do not provide a Postscript level, the routine can use as a default the most commonly available option. The same applies to paper size: you can use LTR (8.5 by 11 inches) in the US, A4 (21 by 29.7 centimeters) elsewhere. 600 dots per square inch may be a reasonable default for the print resolution, and most printers are non-color. In all these cases, you might have a mechanism supporting installation-level or user-level defaults to override the universal ones (for example if your site has standardized on 1200 dpi resolution). The only argument that remains is the printer name; but here too you might have defined a default printer.

This example illustrates the difference between operands and options:

Definition: operand and option arguments

An operand argument to a routine represents an object on which the routine will operate.

An option argument represents a mode of operation.

This definition is too general to tell us unambiguously whether a proposed argument is an operand or an option, but here are two directly applicable criteria:

How to distinguish options from operands

- An argument is an option if, assuming the client had not supplied its value, it would have been possible to find a reasonable default.
- In the evolution of a class, arguments tend to remain the same, but options may be added and removed.

According to the first criterion, all the arguments to *print* are options (with the possible exception of *printer_name* if you have not defined a default printer). Note, however, that the target of the call, an implicit argument (*my_document* in the example) is, as all targets should be, an operand: if you do not say what document you want to print, no one is going to choose a default for you.

The second criterion is less obvious since it requires some foresight, but it reflects the software engineering concerns that underlie all our discussions since the first chapters of this book. We know that a class is not an immutable product; like all software, it may change over its lifetime. Some properties of a class, however, change more often than others. Operands are there for the long term: adding or removing a operand is a major, incompatible change. Options, on the other hand, may come and go. For example one may imagine that support for colors was not part of the first version of the *print* procedure, a few years back, and was only added later. This is typical of an option.

The principle

The definition of operands and options yields the rule on arguments:

Operand principle

The arguments of a routine should only include operands (no options).

Two cases for loosening the rule, not quite qualifying as exceptions, are mentioned below.

In the style that this principle promotes, options to an operation are set not in calls to the operation but in calls to specific option-setting procedures:

```
my_document.set_printing_size ("A4")
my_document.set_color
my_document.print          -- No argument at all.
```

Once set, each option remains in force for the target object until reset by a new call. In the absence of any call to the corresponding procedures, and of any explicit setting at the time of object creation, options will have the default values.

For any type other than boolean, the option-setting procedure will take one argument of the appropriate type, as illustrated by *set_printing_size*; the standard name is of the form *set*_property_name. Note that the argument to a procedure such as *set_printing_size* itself satisfies the Operand principle: the page size, which was an option for the original *print*, is an operand for *set_printing_size* which by definition operates on page sizes.

For a boolean procedure, the same technique would yield a procedure taking either *True* or *False* as argument; since this is confusing (as users of the procedure may forget which ones of the two possibilities *True* represents), it is better to use a pair of procedures, with conventional names of the form *set*_property_name and *set_no*_property_name, for example *set_color* and *set_no_color*, although in this case it is probably just as well to call the second variant *set_black_and_white*.

Application of the Operand principle yields several benefits:

- You only specify what differs from the defaults. Any property for which you do not need any special setting will be handled with the settings that have proved to be most commonly appropriate.
- Novices need only learn the essentials and can ignore any advanced properties.
- As you get to know the class better and move on to sophisticated uses, you learn more properties; but you only have to remember what you use.
- Perhaps most importantly, the technique preserves extendibility and the Open-Closed principle: as you add more options to a certain facility, you do not need to change the interface of a routine and hence invalidate all existing callers. If the default value corresponds to the previous implicit setting, existing clients will not need to be changed.

Against the Operand principle, a possible objection comes to mind: does it not just trade argument complexity for call complexity (calls will be much simpler, but we will have more of them since we must include calls to option-setting procedures)? This is, however, not accurate. The only new calls will be for options that you want to set to values other than the default. Here the complexity is the same as with option arguments. (You may have a few more keystrokes to type, but what counts is the number of pieces of information you have to provide, and it is the same with both approaches.) The big difference is that you need only pay attention to the options that are relevant for your own use, whereas option arguments force you to specify *all* options explicitly.

Also note that frequently a certain option will apply to many successive calls. In that case, using option arguments forces you to specify it each time. With the style recommended here, you gain even if the value is not the default: you set it the first time around, and it stays in place until explicitly changed. The gain is particularly significant in cases such as the numerical library mentioned above where *every* call must include arguments indicating the desired error processing mode, the name of the file for error output and other general properties, which tend to remain applicable through many calls.

See exercise E23.3, page 807.

Some languages support the notion of *optional argument*, achieving some of the benefits of the Operand principle but not all. The comparison has been left as an exercise, but you may already note that the last point mentioned would not apply: any non-default argument must be specified each time.

Benefiting from the Operand principle

Comments made about the Command-Query Separation principle apply to the Operand principle too: it goes against today's dominant practices, and some readers will undoubtedly balk at it initially; but I can recommend it without any reservation, having applied it for many years and greatly benefited from it. It yields a simple, clear and elegant style, fostering clarity and extendibility.

That style soon becomes a natural one for developers who try it. (Predictably, we have made it part of our standard at ISE.) You create the required objects; set up any of their properties that differ from the defaults; then apply the operations that you need. This is the scheme sketched above for *solve* in the Math library. It certainly beats passing 19 arguments.

Exceptions to the Operand principle?

The Operand principle is of universal applicability. Rather than true exceptions, it requires adaptation in two specific cases.

First, we can take advantage of the flexibility of multiple creation procedures. Since a class can provide more than one way to initialize an object, through creation calls of the form !! *x*.*make_*specific (*argument, …*) where *make_*specific is any of the creation procedures, we can relax the Operand principle for such creation procedures, facilitating the client's task by offering various ways to set up objects with values other than the default. Two constraints, however:

- Remember that, as always, every creation procedure must ensure the class invariant.
- The set of creation procedures must include a minimal procedure (called *make* in the recommended style) which includes no option arguments and sets all option values to their defaults.

The other case for loosening the Operand principle follows from the last observation. If you have applied the principle, you may find that some operations (other than creation procedures) are often used with option-setting procedures according to a standard pattern; for example

```
my_document.set_printing_size ("…")
my_document.set_printer_name ("…")
my_document.print
```

In such a case, it may be convenient, in the name of encapsulation and reusability, and in conformity to the Shopping List principle studied next, to provide an extra routine as a convenience for clients:

```
print_with_size_and_printer (printer_name: STRING; size: SIZE_SPECIFICATION)
```

This assumes, of course, that the basic minimal routine (*print* in the example) remains available, and that the new routine is just a supplementary facility meant to simplify client text in cases that have been recognized as truly frequent.

This is not really a violation of the principle, since the very nature of the new routine requires the arguments (printer and size in the example) to be present, making them operands.

A checklist

The Operand principle and its recognition of the need to pay attention to options suggest a technique that helps get a class right. For each class, list the supported options and produce a table with one row for each option, illustrated here by one of the rows for the *DOCUMENT* class:

Option	Initialized	Queried	Set
Paper size	default:A4 (international) *make_LTR*: LTR (US)	*size*	*set_size* *set_A4* *set_LTR*

The successive columns list: the role of the option; how it is initialized by the various creation procedures; how it can be accessed by clients; how it can be set to various values. This provides a useful checklist for frequent deficiencies:

- **Initialized** entries help spot a wrong initialization, especially when you rely on the defaults. (A boolean option, for example, is initialized to false; you should choose the corresponding attribute accordingly, so that the option for color support is *Black_and_white_only* if you wish the default, false, to represent full color support.)
- The **Queried** entries help spot the mistake of providing clients ways to set an option but not to access it. Note in particular that a routine that takes an object in a certain state may need to change some options for its own purposes, but then restore the initial state; this is only possible if the routine can query the initial value.
- The **Set** entries help spot missing option-setting procedures. For example if the default value for a boolean option is the usual false, and you provide a procedure to change it to true, you should not forget to provide another to reset it to false.

None of the rules suggested here is absolute; for example some options may never need to be returned to false. But they do apply in most cases, so it is important to check that the table's entries indicate the behavior that you expect from the class. The table, or extracts from it, can also help document the class.

23.3 CLASS SIZE: THE SHOPPING LIST APPROACH

We have learned to be paranoid about limiting the external size of features, as measured by the number of arguments, because it fundamentally affects the features' ease of use and hence the quality of a class interface. (We care less about the *internal* size of a feature,

measured for example by the number of its instructions, since it simply reflects the complexity of the algorithm. But as you will certainly have noted most routine bodies in good O-O design will remain small anyway.)

Should we be similarly concerned about the size of each class as a whole? Here the answer will be much less drastic.

Class size definition

We must define how to measure the size of a class. It is possible to count the number of lines (or, preferably, the number of declarations and instructions, which is less subject to individual variations of textual layout, and just requires a simple parser). Although interesting for some applications, this is a supplier-side measure. If we are more interested in how much functionality a class provides to its clients, the appropriate criterion is the number of features.

This still leaves two questions:

- Information hiding: do we count all features (*internal* size) or only exported ones (*external*) size?
- Inheritance: do we count only the immediate features, that is to say those introduced in the class itself (*immediate* size), all the features of the class including those inherited from any proper ancestor (*flat* size, so called in reference to the notion of flat form of a class), or the immediate features plus those which the class inherits but somehow modifies through redefinition or effecting, although renaming does not count (*incremental* size)?

Various combinations may be interesting. For the present discussion the most interesting measure will be *external* and *incremental*: external size means that we take the client's view of the class, regardless of anything that is useful for internal purposes only; and incremental size means that we focus on the class's added value. With immediate size we would ignore the often important part of the functionality that is inherited; but with flat size we would be counting the same features again in every class and its descendants.

Maintaining consistency

Some authors, such as Paul Johnson, have argued for strong restraints on class size:

[Johnson 1995].

> *Class designers are often tempted to include lots of features* (*in both the language sense and system design sense of the word*). *The result is an interface where the few commonly used features are lost in a long list of strange routines. Worse yet, the list of possible features is infinite.*

ISE's experience suggests a different view. We have found that class size is not by itself a problem. Although most classes remain relatively small (a few features to a couple dozen), there is occasionally a need for bigger classes (up to 60 or even 80 features), and they do not raise any particular problem if they are otherwise well designed.

This experience leads to the **shopping list** approach: the realization that it does not hurt to add features to a class if they are conceptually relevant to it. If you hesitate to include an exported feature because you are not sure it is absolutely necessary, you should not worry about its effect on class size. The only criteria that matter involve whether the class fits in with the rest. These criteria can be expressed as a general guideline:

Shopping List advice

When considering the addition of a new exported feature to a class, observe the following rules:

S1 • The feature must be relevant to the data abstraction represented by the class.

S2 • It must be compatible with the other features of the class.

S3 • It must not address exactly the same goal as another feature of the class.

S4 • It must maintain the invariant of the class.

The first two requirements are related to the Class Consistency principle, which stated that all the features of a class must pertain to a single, well-identified abstraction The counter-example given there was that of a string class (from the original NEXTSTEP library) which actually covered several abstractions and, as a result, was eventually split into several classes. What is at issue here, however, is not size per se but design quality.

Page 730.

It is interesting to note that the same example, string, is also one of the larger classes in ISE's libraries and has been criticized by Paul Johnson. But in fact the reaction from library users over the years has been the reverse: asking for more features. The class, although rich, is not particularly difficult to use because all the features clearly apply to the same abstraction, character string, and it is in the nature of that abstraction that many operations are applicable, from substring extraction and replacement to concatenation and global character substitution.

Class *STRING* shows that big does not mean complex. Some abstractions are just naturally endowed with many features. Quoting Waldén and Nerson:

> *A document handling class that contains 100 separate operations to set various font operations … may in fact only be dealing with one or a few underlying concepts which are quite familiar and easy to grasp. Ease of selecting the right operation is then reduced to having nicely organized manual pages.*

[Waldén 1995], *page 187.*

In such a case splitting the class would probably decrease rather than improve its ease of use.

An extreme "minimalist" view holds that a class should only include atomic features — those which cannot be expressed in terms of others. This would preclude some of the fundamental schemes of successful object-oriented software construction, in particular **behavior classes** in which an effective feature, for example a routine describing an iteration on a data structure, relies on other lower-level features of the class, often including some deferred ones.

"Don't call us, we'll call you", page 505.

Minimalism would also prohibit including two theoretically redundant but practically complementary features. Consider a class *COMPLEX* to describe complex numbers, as developed earlier in this chapter. For arithmetic operations, some clients may need the function versions:

infix "+", **infix** "–", **infix** "∗", **infix** "/"

so that evaluating the expression *z1 + z2* will create a new object representing the sum of *z1* and *z2*, and similarly for the other functions. Other clients, or the same client in other contexts, may prefer the procedure versions, where the call *z1.add (z2)* will update the *z1* object to represent the result of the addition, and similarly for *subtract*, *multiply* and *divide*. In theory, it is redundant to include both the functions and the procedures, and either set can in fact be expressed in terms of the other. In practice, it is convenient to have both, for at least three reasons: client convenience; efficiency; and reusability.

Laxity and restrictiveness

In the last example the two sets of features, although theoretically redundant, are practically different. You should not, of course, introduce a feature if another already fills exactly the same need; this is covered by clause S3 of the Shopping List advice. That clause is more restrictive than it may seem at first. In particular:

"CLASS EVOLUTION: THE OBSOLETE CLAUSE", 23.7, page 802.

- Assume that you want to change the order of arguments of a routine, for compatibility with others in the same class or different ones. But you are concerned about compatibility with existing software. The solution in this case is *not* to keep both features with the same status; this would violate the advice. Instead, use the **obsolete** library evolution mechanism described later in this chapter.
- The same applies if you want to provide a default for an argument that used to be required for a certain routine. *Do not* provide two versions, one with the extra argument for compatibility, the other relying on a default along the lines discussed earlier in this chapter. Make one interface the official one; the other will be covered by the **obsolete** mechanism.
- If you hesitate between two possible names for a feature, you should almost always resist the temptation to provide both as synonyms. The only exceptions in ISE's libraries concern a handful of fundamental features for which it is convenient to have both an infix name and an identifier, for example array access which can be used as *my_array.item (some_index)* as well as *my_array @ some_index*, each form being preferable in some contexts. But this is a rare situation. As a general rule the class designer should choose a name, rather than passing the buck to client authors — penalizing them with the consequences of his indecision.

As you will have noted, the policy resulting from this discussion is a mix of laxity and restrictiveness. The policy seems lax because it explicitly encourages you to include acceptable features even if they have not yet proved to be essential. But it is in fact systematic and restrictive because it defines strong conditions for a feature to be considered acceptable. The features of a class should cover as many needs as possible; but they should only cover relevant needs and, for each distinct need, there should be just one feature.

> The Shopping List policy is only possible because we follow a systematic policy of keeping the *language* small. A minimalist attitude to language design — ensuring that we stick to a small number of extremely powerful constructs, and avoid redundancies — enables us to let class designers be non-minimalists. Every developer needs to learn the language and, if the language is minimalist enough, will know *all* of it. Classes, however, are only used by client authors, and they can skip what they do not use.

You should also relate the Shopping List advice to the preceding discussion of feature size. What might make a class difficult to use is not the number of its features but their individual complexity of use. More precisely, class size can only be a significant problem initially, by facilitating or hampering quick comprehension of the purpose and scope of a potentially reusable class which an application developer approaches for the first time. Even there, we have seen that size per se is less relevant than coherence (the Class Consistency principle). Past that stage, the reuser will, day in and day out, deal with the features of the class, or more commonly with a subset of these features. Feature size issues take precedence: a feature with many arguments to remember will make the task difficult. But class size has by then ceased to be relevant. Were you to rely on some arbitrary numerical criterion ("no class shall have more than *m* lines or *n* features"), the result could have been to split the class into several, in some cases making it *more* difficult to use.

The lesson for class developers, embodied in the Shopping List advice, is to worry about the quality of a class, in particular its conceptual integrity and the size of its features, but not about its size.

23.4 ACTIVE DATA STRUCTURES

Examples of this chapter and preceding ones have frequently relied on a notion of list or sequence characterized at any time by a "cursor position" indicating where accesses, insertions and deletions take place. This view of data structures, although different from most presentation in "algorithms and data structures" textbooks, is of broad applicability and deserves a more detailed explanation.

To understand the merits of this approach it will be useful to start with the more common one and assess its limitations.

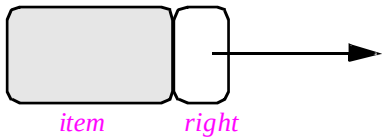
Linked list representation

The discussion will be based on the example of lists. Although its results are independent of the choice of implementation, we need a specific representation to express the algorithms and illustrate the issues. Let us use a popular choice: linked lists. Our general-purpose library must have list classes and, among them, a class *LINKED_LIST*.

Here are a few basics about linked lists, applicable to all the interface styles discussed next — with and without cursors.

Linked lists are a useful representation of sequential structures because they facilitate operations of *insertion* and *deletion*. The successive elements will be housed in individual cells, or *linkables*, each containing a value and a reference to another linkable:

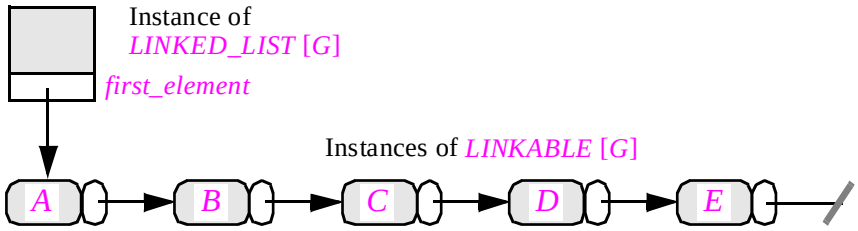
A linkable



The corresponding class, *LINKABLE*, should be generic, since we want the structure to be applicable to linked lists of any type. The cell value will be given by feature *item*, of type *G*, the generic parameter; this will be an in-place value if the actual generic parameter is expanded, for example for lists of integers or reals, and a reference otherwise. The other attribute, *right*, of type *LINKABLE* [*G*], always represents a reference.

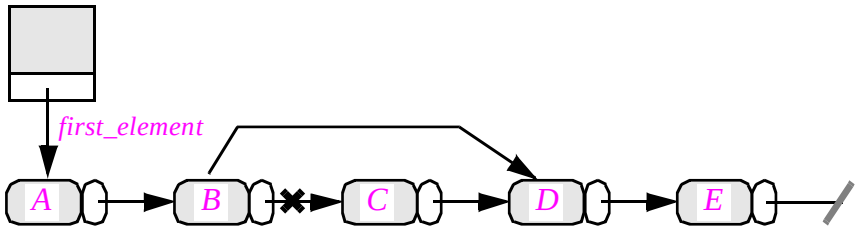
The list itself is represented by a separate cell, the header, containing a reference *first_element* to the first linkable, and possibly some bookkeeping information such as the number of items, *count*. The figure shows the representation of a list of characters.

A linked list



This representation makes insertion or deletion fast if you have a reference to the linkable immediately to the left of the operation's target: a few reference manipulations will do, as shown here for the deletion of the third element.

Deletion in a linked list



On the other hand, linked representation is not very good for finding an element known by its value or its position: these operations require sequential list traversal. Array representations, in contrast, are good for accessing by position, but poor for insertions and deletions. Many other representations exist, some of which manage to combine some of the best of both worlds. The basic linked list remains one of the most commonly used implementations, and is indeed an effective technique for applications that require many local insertions and deletions but few random accesses.

A technical point: the figure does not detail attributes of *LINKED_LIST* other than *first_element*, showing simply a shaded area. Although we could do with just *first_element*, the classes below will include an attribute *count* to record the number of elements of the list. This query could also be a function, but it would then be inefficient, requiring a traversal of the list to count its items each time a client asks us how many we have. Of course if you use an attribute you must make sure that every insertion or deletion updates it. The Uniform Access principle applies here: you can change the implementation without disturbing clients, which will in all cases use the same notation, *l*.*count*, to obtain the item count.

See "Uniform Access", page 55.

Passive classes

We clearly need two classes: *LINKED_LIST* for lists (more precisely, list headers), *LINKABLE* for list elements (linkables). Both are generic.

The notion of *LINKABLE* is essential for the implementation, but not relevant to most clients. We should strive for an interface that provides client modules with list primitives but does not bother them with such implementation details as the presence of linkable elements. The attributes, corresponding to the earlier figure, will appear as:

```
indexing
        description: "Linkable cells, for use in connection with linked lists"
        note: "Partial version, attributes only"
class
        LINKABLE1 [G]
feature {LINKED_LIST}
        item: G
                        -- The cell value
        right: LINKABLE [G]
                        -- The right neighbor
end -- class LINKABLE1
```

For the type of *right* we might consider **like** *Current*, but it is preferable at this stage to keep more redefinition freedom as we do not know yet what may need to be changed by the possible descendants of *LINKABLE*.

To have a true class we need to add routines. What should clients be allowed to do on a linkable? They will need the ability to change the *item* and *right* fields. Also, we may expect that most clients creating a linkable will specify its initial value, requiring a creation procedure. This yields a proper version of the class:

```
indexing
        description: "Linkable cells, for use in connection with linked lists"
class LINKABLE [G] creation
        make
feature {LINKED_LIST}
        item: G
                -- The cell value
        right: LINKABLE [G]
                -- The right neighbor
        make (initial: G) is
                -- Initialize with item value initial.
            do put (initial) end
        put (new: G) is
                -- Replace value with new.
            do item := new end
        put_right (other: LINKABLE [G]) is
                -- Put other to the right of current cell.
            do right := other end
end -- class LINKABLE
```

For brevity the class omits the obvious procedure postconditions (such as **ensure** *item = initial* for *make*). There are no preconditions.

So much for *LINKABLE*. Now consider the linked lists themselves, to be accessed internally through their headers. Among others we need exported features to: obtain the number of elements (*count*); find out whether the list is empty (*empty*); obtain the value of the *i*-th element, for any legal index *i* (*item*); insert a new element at a certain position (*put*); change the value of the *i*-th element (*replace*); search for an element having a certain value (*occurrence*). We will also need a query returning a reference to the first element (void if the list is empty); it does not need to be exported.

Here is a sketch of a first version. Some of the routine bodies have been omitted.

```
indexing
        description: "One-way linked lists"
        note: "First version, passive"
class
        LINKED_LIST1 [G]
feature -- Access
        count: G
        empty: BOOLEAN is
                -- Is list empty?
            do
                Result := (count = 0)
            ensure
                empty_if_no_element: Result = (count = 0)
            end
```

```
item (i: INTEGER): G is
        -- Value of i-th list element
    require
        1 <= i; i <= count
    local
        elem: LINKABLE [G]; j: INTEGER
    do
        from
            j := 1; elem := first_element
        invariant j <= i; elem /= Void variant i — j until
            j = i
        loop
            j := j + 1; elem := elem.right
        end
        Result := elem.item
    end
occurrence (v: G): INTEGER is
        -- Position of first element of value v in list (0 if none)
    do … end
```

```
feature -- Element change
    put (v: G; i: INTEGER) is
            -- Insert a new element of value v
            -- so that it becomes the i-th element
        require
            1 <= i; i <= count + 1
        local
            previous, new: LINKABLE [G]; j: INTEGER
        do
                -- Create new cell
            !! new.make (v)
            if i = 1 then
                    -- Insert at head of list
                new.put (first_element); first_element := new
            else
                from
                    j := 1; previous := first_element
                invariant
                    j >= 1; j <= i — 1; previous /= Void
                    -- previous is the j-th list element
                variant
                    i — j — 1
                until j = i — 1 loop
                    j := j + 1; previous := previous.right
                end
                    -- Insert after previous
                previous.put_right (new)
                new.put_right (previous.right)
            end
            count := count + 1
        ensure
            one_more: count = old count + 1
            not_empty: not empty
            inserted: item (i) = v
            -- For 1 <= j < i, the element of index j has not changed its value
            -- For i < j <= count,
                -- the element of index j has the value
                -- that the element of index j — 1 had before the call
        end
```

```
        replace (i: INTEGER; v: G) is
                -- Replace by v the value of i-th list element.
            require
                1 <= i; i <= count
            do
                …
            ensure
                replaced: item (i) = v
            end
    feature -- Removal
        prune (i: INTEGER) is
                -- Remove i-th list element
            require
                1 <= i; i <= count
            do
                …
            ensure
                one_less: count = old count — 1
            end
        … Other features …
    feature {LINKED_LIST} -- Implementation
        first_element: LINKABLE [G]
    invariant
        empty_definition: empty = (count = 0)
        empty_iff_no_first_element: empty = (first_element = Void)
    end -- class LINKED_LIST1
```

It is a good idea to try to complete *occurrence*, *replace* and *prune* for yourself in this first version. (Make sure to maintain the class invariant.)

Encapsulation and assertions

Before we consider better versions, a few comments are in order on this first attempt.

Class *LINKED_LIST1* shows that even on fairly simple structures reference manipulations are tricky, especially when combined with loops. The use of assertions helps get them right (see procedure *put* and the invariant); but the sheer difficulty of this type of operations is a strong argument for encapsulating them once and for all in reusable modules, as promoted by the object-oriented approach.

Also note the application of the Uniform Access principle: although *count* is an attribute and *empty* a function, clients do not need to know these details. They are protected against any later reversal of these implementation decisions.

The assertions for *put* are complete, but, because of the limitations of the assertion language, not completely formal. Similarly extensive preconditions should be added to the other routines.

A critique of the class interface

How usable is *LINKED_LIST1*? Let us evaluate its design.

A worrying aspect is the presence of significant redundancies: *item* and *put* contain almost identical loops, and similar ones will need to be included in the routines whose code has been left to the reader (*occurrence*, *replace*, *remove*). Yet it does not seem possible to factor out the common part. Not a promising start.

This is an implementation problem, internal to the class: lack of reusability of the internal code. But it points to a more serious flaw — a poorly designed class interface.

Consider routine *occurrence*. It returns the index at which a given element has been found in the list, or zero if the element is not present. One drawback is that this only gives the first occurrence; what if the client wants to obtain the successive occurrences of a value? But there is a more serious difficulty. A client that has performed a successful search may, among other typical needs, want to change the value of the element found, to delete that element, or to insert a new one next to it. But any one of these operations requires traversing the list again! For example, *put* (*v*, *i*) goes through the first *i* elements, even if *i* is the result of *occurrence* — obtained by a similar traversal.

In the design of a general-purpose library component that will get used over and over, one cannot treat such inefficiencies lightly. Any performance overhead due to the increased generality of a reusable solution must remain negligible; otherwise developers will not accept paying the price, dooming any reuse policy. Here the price is not acceptable.

Simple-minded solutions

How can we remove the inefficiency? Two possible solutions come to mind:

- We could make *occurrence* return, instead of an integer, the *LINKABLE* reference to the cell where the requested value appears, or void for an unsuccessful search. Then the client has a direct handle on the actual linkable cell and may perform the needed operations without retraversal; it can for example use *LINKABLE*'s *put* procedure to change the value, and its *put_right* procedure to insert a new element. (Deletion is more delicate since the client would need the previous element too.)
- We could try to provide enough primitives to deal with various combinations of operations: search and replace, search and insert, search and delete and so on.

The first solution, however, defeats the whole idea of encapsulating data structures in classes: clients would directly manipulate the representations, with all the dangers involved. The notion of linkable is internal; we want client programmers to think in terms of lists and list values, not of list cells and pointers. Otherwise we lose data abstraction.

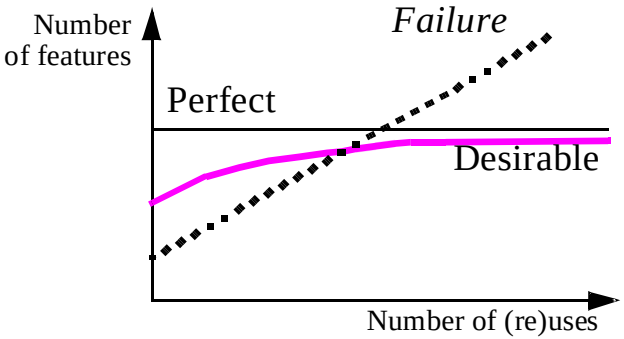
The second solution was attempted in an early version of ISE's libraries, which made an effort to provide routines covering common combinations of operations. To insert an element just before the occurrence of a known value, a client would use, rather than a call to *search* followed by a call to *put*, a single call to

```
insert_before_by_value (v: G; v1: G) is
        -- Insert a new element of value v in front of first occurrence
        -- of v1 in list, or at end of list if no such occurrence
    do
        …
    end
```

This solution keeps the internal representation hidden from clients, while avoiding the inefficiencies of the initial version.

But we soon realized we were in for a long journey. Consider all the potentially useful variants: *search_and_replace*, *insert_before_by_value*, *insert_after_by_value*, *insert_after_by_position*, *insert_after_by_position*, *delete_before_by_value*, *insert_at_end_if_absent*, and more.

This raises troubling questions about the viability of the approach, forcing a reflection on library design. Writing general-purpose reusable software is a difficult task, and there is no guarantee that you will get everything right the first time — with a design that would follow the horizontal line in the figure below. You should be prepared to extend classes with new features as the library's usage reaches new users and new application domains. As represented by the colored line of the picture, however, the process must converge: after an initial tune-up period, the design should reach a stable state.



Evolution of a library class

If not — that is to say, if almost every new use brings in the need for extension or modification, as represented by the dotted line in the figure — the approach to reusability is obviously flawed. This appeared to be the case with the list class we had at that point: it looked as if every time we put the list class to a new use the need would arise for yet another routine, representing a new combination of the basic operations.

To make matters worse, all such routines are rather complex, with loops similar to the one for *put*; they have much in common but all differ from each other by small details. The prospect of a robust, reusable linked list class seems to be receding.

Introducing a state

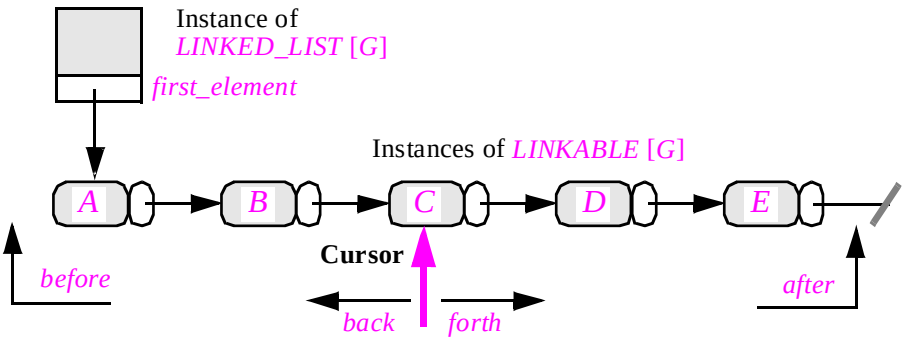
Fortunately, there is a way out. To find it requires taking a different view of the underlying abstract data type.

So far a list has been treated as a passive repository of information. To provide its clients with a better service, the list should become more active by "remembering" where the last operation was performed.

As noted earlier in this chapter, we should not hesitate to look at objects as machines with an internal state, and introduce both commands that change the state and queries on the state. In the first solution a list object already had a state, defined by its contents and modifiable by commands such as *put* and *remove*; but by adding more components to the state we will obtain a better interface, making the class both simpler and more efficient.

Besides the list contents, the state will include the notion of currently active position, or cursor; the interface will allow clients to move the cursor explicitly.

List with cursor



We permit the cursor to be on a list element (if any), or one position to the left of the first, in which case the boolean query *before* will return true, or one position to the right of the last, making *after* true.

An example of a command that may move the cursor is the procedure *search*, replacing the function *occurrence*. A call to *l.search* (*v*) will move the cursor to the first element of value *v* to the right of the current cursor position, or move it *after* if there is none. Note that in passing this solves the problem of finding multiple occurrences of *v*: just call *search* as many times as needed. (For symmetry we could also have *search_back*.)

The basic commands to manipulate the cursor are:

- *start* and *finish* to move the cursor to the first and last position if any.
- *forth* and *back* to move the cursor to the next and previous position.
- *go* (*i*) to move it to a stated position *i*.

Besides *before* and *after*, queries on the cursor position include *index*, its integer index (starting at 1 for the first element) as well as the booleans *is_first* and *is_last*.

The procedures to build and modify a list — insertion, deletion, replacement — become simpler because they do not have to worry about positions: they will simply act on elements at the current cursor position. All the loops disappear! For example, *remove* will not be called as *l*.*remove* (*i*) any more, but simply as *l*.*remove*, to delete the element at the current cursor position. We need to establish precise and consistent conventions about what happens to the cursor after each operation:

- *remove*, with no argument, deletes the element at cursor position and puts the cursor under its right neighbor (so that the value of *index* does not change in the end).
- *put_right* (*v*: *G*) inserts an element of value *v* to the right of the cursor and does not move the cursor (*index* is unchanged).
- *put_left* (*v*: *G*) inserts an element of value *v* to the left of the cursor and does not move the cursor (increasing the value of *index* by 1).
- *replace* (*v*: *G*) changes the value of the element at cursor position. The value of this element is given by the query function *item*, which now has no argument (and so could be implemented as an attribute).

Maintaining consistency: the implementation invariant

In building the class for such a fundamental data structure we must be careful to get everything right. Here assertions are indispensable. Without them we would be almost sure to miss some details. For example:

- Is a call to *start* permitted if the list is empty and, if so, what is its effect?
- What happens to the cursor after a *remove* if the cursor was on the last element? In other cases the cursor should go to the element immediately to the right of the deleted one, but here there is none. This is one of the reasons for the convention that was stated informally — allowing the cursor to move one position off to the right or to the left — but we need a more precise statement of this property, addressing all cases unambiguously.

Answers to questions of the first kind will be described by preconditions and postconditions.

See "Implementation invariants", page 377.

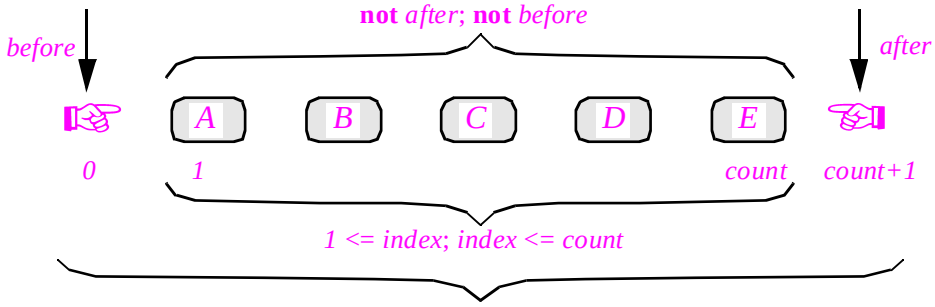
For such properties as the permitted cursor positions, we should use the invariant, more precisely the clauses constituting the implementation invariant. Remember that an implementation invariant expresses the consistency of a representation, given by a class, vis-à-vis the underlying abstract data type. Here it will include the property

0 <= index; *index <= count + 1*

What about an empty list? We need to respect the symmetry between left and right. One solution, adopted in an earlier version of the library, is to consider that an empty list is both *before* and *after*, and constitutes the only case in which both of these properties may be true together. This works but leads, in the routines' algorithms, to frequent tests of the form **if** *after* **and not** *empty*… to distinguish between true cases of *after* and accidental

ones resulting from *empty*. It turns out to be preferable to take the view that, conceptually, a list always has two extra *sentinel* elements, shown as ☞ and ☜ in the figure:

List with sentinels



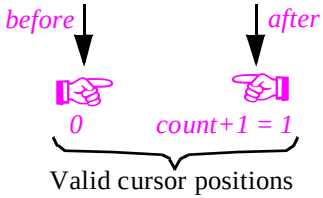
The sentinel elements help us reason about the structure, but we will not necessarily store them in the representation. The implementation discussed next stores the left sentinel but not the right one; it is also possible to use an implementation that stores neither but still conforms to the conceptual model represented by the above figure.

Since we often want to state, for example as the precondition for an operation on an element given by its index, that the index indeed marks a position where the list has an element, we need a query to express this condition:

```
on_item (i: INTEGER): BOOLEAN is
                -- Is there an element at position i?
        do
                Result := ((index >= 1) and (index <= count))
        ensure
                within_bounds: Result = ((index >= 1) and (index <= count))
                no_elements_if_empty: Result implies (not empty)
        end
```

To state that there is an element at the cursor position, we may define query *readable*, whose value is that of *on_item* (*index*). This is a good example of the Shopping List principle: because *readable* is conceptually redundant, a minimalist policy would get rid of it; by including it we provide our clients with a better abstraction, freeing them from having to remember what exactly constitutes a valid item index at the implementation level.

The invariant will state that **not** (*after* **and** *before*). In the boundary case of an empty list, the picture becomes:



Empty list with sentinels

So an empty list will have two possible states: *empty* **and** *before* and *empty* **and** *after*, corresponding to the two cursor positions in the figure. This seems strange at first but has no unpleasant consequence, and is in practice preferable to the earlier convention that *empty* = (*before* **and** *after*), now replaced by *empty* **implies** (*before* **or** *after*).

Note two general lessons here: the usefulness, as in many mathematics or physics problems, of checking boundary cases to verify that a general solution is sound; and the importance of relying on assertions to express the precise properties of a design. Here are some of the principal clauses of the invariant:

```
0 <= index; index <= count
before = (index = 0); after = (index = count + 1)
is_first = ((not empty) and (index = 1)); is_last = ((not empty) and (index = count+1))
empty = (count = 0)
        -- The next three clauses are theorems (deducible from the previous ones):
empty implies (before or after)
not (before and after)
empty implies ((not is_first) and (not is_last))
```

This example illustrates the general observation that writing the invariant is the best way to get a real understanding of what a class is about. The clauses seen so far apply equally to all implementations of sequential lists; they will shortly be complemented by a few others which are specific to the choice of a linked representation.

For more clauses see page 791.

The last three clauses, as noted, are deducible from the others (prove them!). Invariants are not required to be minimal; it is often useful to list additional clauses such as these if they state important, non-trivial properties of the class. As we saw in the study of abstract data types, an ADT, and hence its implementation as a class, is a theory — here the theory of linked lists. The basic invariant clauses express the axioms of the theory; but any useful theory has interesting theorems too.

Exercise E23.6, page 807.

Of course if you intend to monitor invariants at run time — meaning that you are not quite sure yet that the theory is sound! — you should also consider the effect of added clauses on execution time. But this only matters for development and debugging. In a usual production context there is no reason for monitoring the invariants.

The client's view

This design provides a simple and elegant interface to the implementation of linked lists. Operations such as "search and then insert" use two successive calls, although with no significant loss of efficiency:

```
l: LINKED_LIST [INTEGER]; m, n: INTEGER
…
l.search (m)
if not after then l.put_right (n) end
```

The call *search* (*m*) moves the cursor to the next occurrence of *m* after the current cursor position, or *after* if there is none. (The extract assumes that the cursor is initially known to be on the first element; if not, the client should execute *l.start* first.)

To delete the third occurrence of a certain value, a client will execute:

```
l.start; l.search (m); l.search (m); l.search (m)
if not after then l.remove end
```

To insert a value at position *i*:

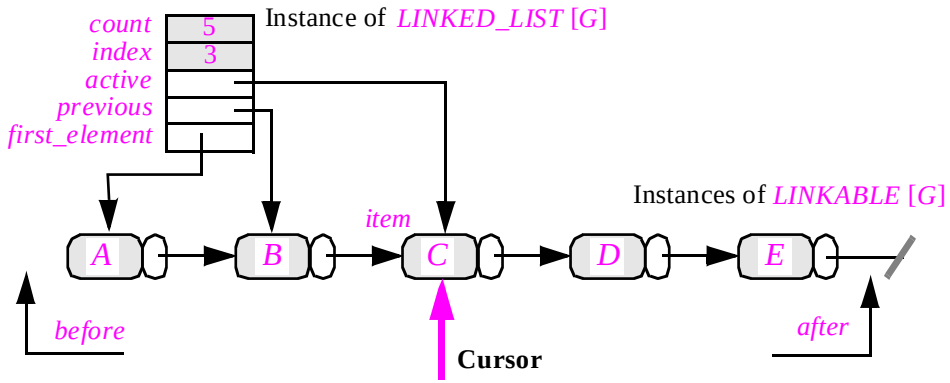
```
l.go (i); l.put_left (i)
```

and so on. We have obtained a clear and easy to use interface by making the internal state explicit, and providing clients with the appropriate commands and queries on this state.

The internal view

The new solution simplifies the implementation just as it improves the interface. Most importantly, by giving each routine a simpler specification, concentrated on just one task, it removes unjustified redundancies, in particular all the unneeded loops. Insertion and deletion procedures no longer have to traverse the list; they just carry out a local modification. The responsibility of positioning the cursor now lies with other routines (*back*, *forth*, *go*, *search*), only some of which (*go* and *search*) need loops.

Cursor list representation (first variant)



Along with *first_element* it will be useful to keep two references in the list header, enabling us to perform insertions and deletions efficiently: *active*, attached to the cursor item at cursor position, and *previous* attached to the previous one.

Clients may know the state of the list by accessing public integer attributes *count* and *index* and boolean queries *before*, *after*, *is_first*, *is_last*, *item*. Here are two typical functions:

You should complete before *and* is_last *based on this model.*

```
after: BOOLEAN is
        -- Is there no valid position to right of cursor?
    do Result := (index = count + 1) end

is_first: BOOLEAN is
        -- Is cursor on first item?
    do Result := (index = 1) end
```

Note the phrasing of the header comments. For *after*, "Is cursor to the right of last element?" would not be quite correct, since *after* may be true even if there is no element at all. Writing header comments so that they are clear, terse and accurate is an art form.

See "Routine header comments: an exercise in corporate downsizing", page 886.

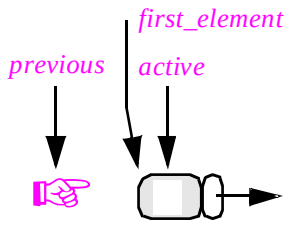
The query *item* returns the element at cursor position, if any:

```
item: G is
        -- Element at cursor position
    require
        readable: readable
    do
        Result := active.item
    end
```

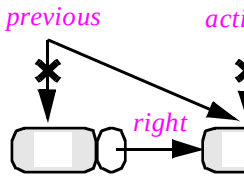
Remember that *readable* indicates whether the cursor is on an element (*index* between 1 and *count*). Also note that *item* in *active.item* refers to the attribute in *LINKABLE*, not to the function of *LINKED_LIST* itself.

Here now are the basic cursor manipulation commands; they are fairly delicate to get right but, as a consolation, you may note that only a handful of routines, such as *start*, *forth*, *put_right*, *put_left* and *remove*, must perform non-trivial operations on references. Let us try *start* and *forth*. Procedure *start* must work for an empty list as well as a non-empty one; for an empty list the convention is that *start* brings the cursor to the second sentinel.

```
start1 is
        -- Move cursor to first position.
        -- (Provisional version; see next.)
    do
        index := 1
        previous := Void
        active := first_element
    ensure
        moved_to_first: index = 1
        empty_convention: empty implies after
    end
```



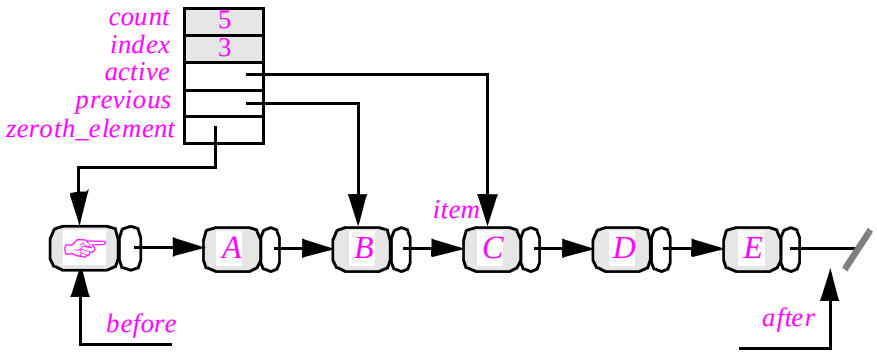
```
forth1 is
                -- Move cursor to next position.
                -- (Provisional version; see next.)
        require
                not_after: not after
        do
                index := index + 1
                if before then
                        active := first_element; previous := Void
                else
                                check active /= Void end
                        previous := active; active := active.right
                end
        ensure
                moved_by_one: index = old index + 1
        end
```



Here we stop! This is becoming too complicated and too inefficient. The performance of procedure *forth* is crucial, since a typical use of a list by a client is **from** *start* **until** *after* **loop** *…*; *forth* **end**. Can we get rid of the test?

We can, by taking the left sentinel seriously and always creating it when we create a list; the creation procedure *make* of *LINKED_LIST* is left as an exercise. We replace *first_element* by a reference *zeroth_element* to the sentinel:

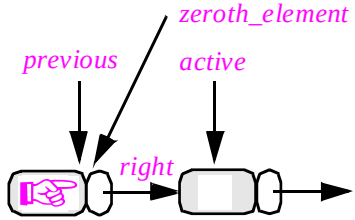
Cursor list representation (revised variant)



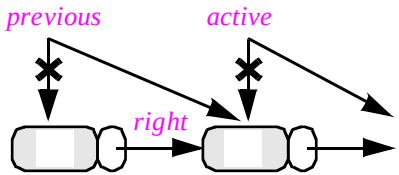
The properties *zeroth_element* /= *Void* and *previous* /= *Void* will be part of the invariant (you must of course make sure that the creation procedure ensures them). They are precious since they will save many repeated tests.

Procedure *forth*, given here after the new *start*, is simpler and faster (no test!):

```
start is
        -- Move cursor to first position.
    do
        index := 1
        previous := zeroth_element
        active := previous.right
    ensure
        moved_to_first: index = 1
        empty_convention: empty implies after
        previous_is_zeroth: previous = zeroth_element
    end
```

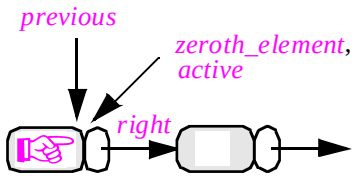



```
forth is
        -- Move cursor to next position.
        -- (Version revised for efficiency; no test!)
    require
        not_after: not after
    do
        index := index + 1
        previous := active
        active := active.right
    ensure
        moved_by_one: index = old index + 1
    end
```



It is convenient to define *go_before* which positions the cursor on the left sentinel:

```
go_before is
        -- Move cursor before.
    do
        index := 0
        previous := zeroth_element
        active := zeroth_element
    ensure
        before: before
        previous_is_zeroth: previous = zeroth_element
        previous_is_active: active = previous
    end
```



Procedure *go* is entirely defined in terms of *go_before* and *forth*:

```
go (i: INTEGER) is
                -- Move cursor to i-th position.
        require
                not_offleft: i >= 0
                not_offright: i <= count + 1
        do
                from
                        if i < index then go_before end
                invariant index <= i variant i – index until index = i loop
                        forth
                end
        ensure
                moved_there: index = i
        end
```

Note the care exercised in avoiding useless traversal steps in *go*, the only one of procedures seen so far that needs a loop. For symmetry we should add *finish*, which brings the cursor to the last position and can be implemented as just *go* (*count + 1*).

Although not really indispensable, it is convenient (Shopping List principle!) to export *go_before*. Then for symmetry we should also include and export *go_after*, which does *go* (*count + 1*), and export it.

Also for symmetry is *back*, using *go*'s loop:

```
back is
                -- Move cursor to previous position.
        require
                not_before: not before
        do
                        check index – 1 >= 0 end
                go (index – 1)
        ensure
                index = old index – 1
        end
```

Exercise E23.7, page 807.

However pleasing, the symmetry between *back* and *forth* is not without danger, since it may lead client authors to use both procedures freely even though *back*, which has to restart from the beginning of the list and perform *index – 1* iterations of *forth*, is much more expensive. If you perform anything more than a few occasional *back*, the one-way linked list is inappropriate; you can for example use two-way linked lists. The corresponding class may be built as an heir to *LINKED_LIST* (a valid use of inheritance, since a list linked both ways is also linked one way) and is left as an exercise. Make sure to do this exercise at some stage if you want to reach a full mastery of the concepts.

The earlier invariant clauses, as noted, were implementation-independent. Here are a few more clauses capturing some of what we have learned about our implementation:

```
empty = (zeroth_element.right = Void)

zeroth_element /= Void; previous /= Void

(active = Void) = after; (active = previous) = before
(not before) implies (previous.right = active)

(previous = zeroth_element) = (before or is_first)
is_last = ((active /= Void) and then (active.right = Void))
```

The first set of clauses was on page 785.

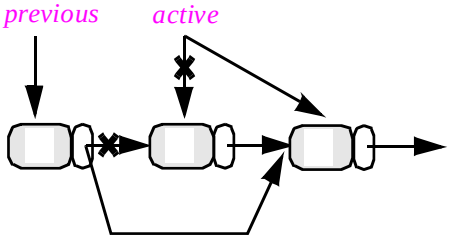
Most of the queries are straightforward. *before* should return the boolean value of (*index = 0*) and *after* that of (*index = count + 1*). The element at cursor position is given by

```
item: G is
        -- Value of element at cursor position
    require
        readable: readable
    do
        Result := active.item
    end
```

Procedure *search* is similar to *go* and left to the reader. You should also write the procedure *i_th* (*i*: *INTEGER*) which returns the value of the element at position *i*; although concrete side effects are acceptable, be sure not to introduce any abstract side effect.

The last category of features includes procedures for insertion and deletion. The basic deletion operation is:

```
remove is
        -- Delete element at cursor position and move cursor to its right neighbor.
        -- (If no right neighbor, list becomes after).
    require
        readable: readable
    do
        active := active.right
        previous.put_right (active)
        count := count — 1
    ensure
        same_index: index = old index
        one_less_element: count = old count – 1
        empty_implies_after: empty implies after
    end
```



The routine looks trivial; but this is only thanks to the technique of keeping the left sentinel around as a physical object, avoiding constant tests of the form *previous /= Void* and *first_element /= Void*. It is worth considering the more complicated and less efficient routine body that we would have obtained without this simplification:

Warning: rejected version!

```
active := active.right
if previous /= Void then previous.put_right (active) end
count := count — 1
if count = 0 then
        first_element := Void
elseif index = 1 then
        first_element := active
-- else first_element does not change
end
```

In either case, the more you can express in assertions, the better you will understand what is going on and avoid mistakes.

Exercise E23.9, page 808.

You should exercise your understanding of these techniques by writing the insertion procedures *put_left* and *put_right*.

Abstract data types and abstract machines

The notion of active data structure is widely applicable and in line with earlier principles of this chapter, Command-Query Separation in particular. Giving data structures an explicit state often yields simple, easy to document interfaces.

One might fear that the resulting structures would become less abstract, but this is not the case. Abstract does not mean passive. What the theory of abstract data types tells us is that our objects should be known through abstract descriptions of the applicable operations and their properties; but this does not imply treating them as mere repositories of data. By introducing a state and operations on that state, we actually make the abstract data type specification richer as it has more functions and more properties. The state itself is a pure abstraction, always accessed indirectly through commands and queries.

The view of objects as state machines reflects abstract data types which are more *imperative*, not less abstract.

Separating the state

It is possible to take the preceding techniques further. So far the cursor was just a concept, implemented indirectly through attributes *previous*, *active* and *index* rather than directly through one of the classes of the software. We can define a class *CURSOR* with descendants for various kinds of cursor structure. Then we can separate, for a structure such as a list, the attributes that describe the list contents (*zeroth_element*, *count*) from the traversal-related attributes, which will be stored in cursor objects.

Although we do not need to pursue this idea here, it is useful to note its possible application to a concurrent context. If a number of clients need to access a shared structure, they can each have their own cursors.

Skip to "SELECTIVE EXPORTS", 23.5, page 796.

Merging the list and the sentinels

(This section describes an advanced optimization and may be skipped on first reading.)

The example of linked lists with the sentinels can benefit from one more optimization, which has indeed been applied to the latest versions of the ISE libraries. We will only take a peek at it because it is of a specialized nature and not relevant to normal application development. Such delicate optimizations should only be considered for widely used reusable components. (In other words: do not try this at home.)

Can we get the benefit of sentinels without wasting the corresponding space? As noted upon the introduction of the sentinel concept, we could treat the sentinels as fictitious; but then we would lose the crucial optimization which has enabled us to write the body of *forth* as just

```
index := index + 1
previous := active
active := active.right
```

without the expensive tests of the earlier versions. We avoid these tests by making sure that, for a list in non-*after* state, *active* is never void (the corresponding invariant clause is (*active* = *Void*) = *after*); this is because we always have a real cell, the sentinel, available to serve as initialization for *active*, even for an empty list.

For a routine other than *forth*, the optimization would not be such a big deal. But *forth*, as noted, is the bread and butter of list processing by clients, resulting from the sequential nature of the lists; typical usage is of the form

```
from your_list.start until your_list.after loop …; your_list.forth end
```

and it is not uncommon, if you use a profiler tool to measure what happens during execution, to discover that the computation spends a good part of its time in *forth*. So it pays to optimize it, and the test-free form above indeed provides a dramatic improvement over the test-full one.

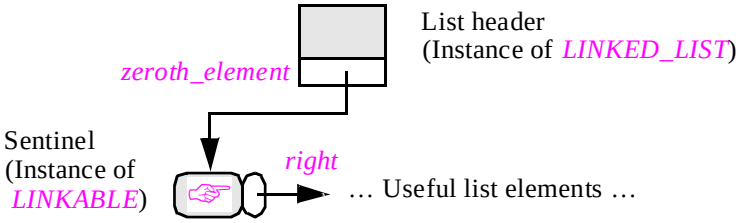
To get this time improvement, however, we pay a space penalty: each list now has an extra element, with no actual information. This would seem to cause a problem only if we have many short lists. But the problem can become more serious:

- In many cases, as hinted earlier, you will need two-way linked lists, fully symmetric, with *BI_LINKABLE* elements chained both ways. Class *TWO_WAY_LIST* (which, by the way, may be written as inheriting twice from *LINKED_LIST*, relying on repeated inheritance techniques) will need both a left and a right sentinel elements.
- Linked trees present an even more serious problem. An important practical class is *TWO_WAY_TREE*, providing a convenient doubly-linked representation of trees. Building on ideas developed in the presentation of multiple inheritance, this class merges the notion of node and tree; it inherits from both *TWO_WAY_LIST* and *BI_LINKABLE*. But then every node is a list, a two-way one at that, and may have to carry both sentinels.

"Trees are lists and list elements", page 525.

Although there are other ways to solve the second case — such as renouncing the inheritance structure — let us see if we can get the best of all worlds.

To find a solution let us ask an impertinent question. In the structure

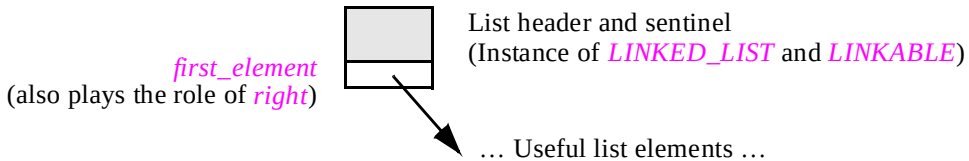


do we really need *two* bookkeeping-only objects? The truly useful information is in the part not shown on the figure, the actual list elements; to manage them we have added both a list header and a sentinel — two sentinels in the case of a two-way list. For long lists we are able to ignore this bloated bookkeeping structure, like a large company that has accumulated many layers of middle management in times of economic prosperity; but when the going gets tough it is time to take a closer look and see if we cannot merge a few of these management functions.

first_element appeared among others in the figures of page 786.

Can we indeed make the list header *itself* play the role of sentinel? It turns out we can. All that a *LINKABLE* needs is an *item* field and a *right* field. For a sentinel, in fact, only the *right* field. That field denotes the first of the list elements; so if we put it in the list header it will play the same role as what used to be called *first_element* in the first variant of the sentinel implementation. The problem, of course, was that *first_element* could be void, for an empty list, polluting all our algorithms with tests of the form **if** *before* **then**… We certainly do not want to go back to that situation. But we can keep the representation of the figure at the top of this page as the conceptual model, while getting rid of the sentinel object in the implementation. The concrete picture becomes

Header as sentinel (non-empty list)



The key to understanding this solution and getting things right is to remember that this solution is exactly the same conceptually as the last one, but replaces *zeroth_element* by a reference to the list header itself (*Current* in class *LINKED_LIST*), using *first_element* to represent what used to be *zeroth_element*.*right* (possibly void, but always defined since *zeroth_element* was never void). We still need a convention for the empty list, with no "Useful list elements"; in that case the last figure becomes

Header as sentinel (empty list)

with a simple convention: attaching *first_element* back to the list header itself. This way *first_element* will never be void — our crucial goal for keeping everything simple; we must just remember to replace, everywhere in class *LINKED_LIST*, any test of the form *zeroth_element*.*right* by *first_element = Current*.

We keep all the desirable invariant clauses of the previous sentinel versions:

previous /= *Void*
(*active* = *Void*) = *after*; (*active* = *previous*) = *before*
(**not** *before*) **implies** (*previous*.*right* = *active*)
is_last = ((*active* /= *Void*) **and then** (*active*.*right* = *Void*))

The clauses involving *zeroth_element*, which used to be

zeroth_element /= *Void*
empty = (*zeroth_element*.*right* = *Void*)
(*previous* = *zeroth_element*) = (*before* **or** *is_first*)

now yield:

first_element /= *Void*
empty = (*first_element* = *Current*)
(*previous* = *Current*) = (*before* **or** *is_first*)

All this is obtained simply (fasten your seat belts) by making *LINKED_LIST* inherit from *LINKABLE*:

class *LINKED_LIST* [*G*] **inherit**
LINKABLE [*G*]
rename *right* **as** *first_element*, *put_right* **as** *set_first_element* **end**
… Rest of class as before, with the removal of *zeroth_element* as shown above …

Is it a kludge to let *LINKED_LIST* inherit from *LINKABLE*? Not at all! The whole idea was to merge the notions of list header and sentinel, that is to say, to consider a list header (an instance of *LINKED_LIST*) as a linkable too; so we have a perfect example of the "is-a" relation of inheritance. We have decided to treat every *LINKED_LIST* as a *LINKABLE*, so inheritance is the proper way to go. Here the client relation is not even in the race: not only would it not yield what we want, the removal of extra fat from our structures; it would *add* even more fields to our objects!

Make sure your seat belts are still securely fastened as we start considering what happens lower in the inheritance structure. *BI_LINKABLE* inherits twice from *LINKABLE*. Class *TWO_WAY_LIST* inherits from *LINKED_LIST* (once, or possibly twice depending on the implementation technique that we choose) and, in line with the technique just seen, from *BI_LINKABLE*. With all the repeated inheritance involved one might think that things would get out of hand and that our structures would start getting all kinds of unnecessary fields; but no, the rules on sharing and replication in repeated inheritance enable us to get exactly what we want.

The last step is *TWO_WAY_TREE* which, for good measure, inherits from both *TWO_WAY_LIST* and *BI_LINKABLE*. Enough, one might think, for a few heart attacks, but no; everything falls nicely into place. We get all the features we want, none of the features we do not want; all the sentinels are in place — conceptually — so that *forth*, *back* and all the consequent loops can be as fast as they need to be; and the sentinels do not take up any space at all.

This is indeed the scheme now applied to the affected classes in the Base libraries. Before we recover from the flight, a few observations are in order:

- Under no circumstance should this kind of work, involving tricky data structure manipulation, be undertaken without the full benefit of assertions. It is simply impossible to get them right without stating the invariant precisely, and checking that everything remains compatible with it.
- The machinery of repeated inheritance is essential. Without the techniques introduced by the notation of this book to enable a repeated descendant to obtain sharing or replication on a feature-by-feature basis, based on the simple criterion of feature names, it is impossible to handle effectively any situation involving serious use of repeated inheritance.
- To repeat the most important comment: such delicate optimizations are only worth considering in heavily used libraries of general-purpose reusable components. In normal application development, they are just too hairy to be worthwhile. The discussion has been included here to give the reader a glimpse of what it takes to craft professional components all the way to the end; but most developments will, happily, never have to undertake such efforts.

23.5 SELECTIVE EXPORTS

The relationship between classes *LINKABLE* and *LINKED_LIST* illustrates the importance, for a satisfactory application of the rule of Information Hiding, of supporting more than just two export modes, secret and generally available, for a feature.

Class *LINKABLE* should not make its features — *item*, *right*, *make*, *put*, *put_right* — generally available, since most clients have no business peeking into linkables, and should only use linked lists. But it cannot make them secret, for that would hide them from *LINKED_LIST*, their intended beneficiary. Such calls as *active*.*right*, essential to the operation of *forth* and other *LINKED_LIST* routines, would not be possible.

"SELECTIVE EXPORTS AND INFORMATION HIDING", 7.8, page 191.

Selective exports provide the solution by enabling *LINKABLE* to select a set of classes to which, and to which only, it will export its features:

```
class
        LINKABLE [G]
feature {LINKED_LIST}
        item: G
        right: LINKABLE [G]
        etc.
end -- class LINKABLE
```

Remember that this makes the features available to all descendants of *LINKED_LIST*, as is indispensable if they need to redefine some inherited routines or add their own.

Sometimes, as we saw in an earlier chapter, a class must export a feature selectively to itself. For example the heir *BI_LINKABLE* of *LINKABLE*, describing two-way linked lists with a field *left*, includes an invariant clause of the form

"Exporting to yourself", page 193.

(*left* /= *Void*) **implies** (*left*.*right* = *Current*)

requiring *right* to be declared in a clause **feature** {… Other classes …, *BI_LINKABLE*}; otherwise the call *left*.*right* would be invalid.

Selective export clauses are essential when a group of related classes, as *LINKABLE* and *LINKED_LIST* here, need some of each other's features for their implementations, although these features remain private to the group and should not be made available to other classes.

A reminder: in a discussion of an earlier chapter we saw that selective exports are a key requirement for the decentralized architectures of object-oriented software construction.

"The architectural role of selective exports", page 209.

23.6 DEALING WITH ABNORMAL CASES

Our next interface design topic is a problem that affects every software development: how to handle cases that deviate from the normal, desired schemes.

Whether due to errors made by the system's users, to abnormal conditions in the operating environment, to irregular input data, to hardware malfunction, to operating system bugs or to incorrect behavior of other modules, special cases are the scourge of developers. The necessity to account for all possible situations, erroneous user input, failures of the hardware or of the operating system, and other modules' possibly improper processing, is a powerful impediment in the constant battle against software complexity.

This problem strongly affects the design of module interfaces What software developer has not wished that it would just go away? Then we could write clear, elegant algorithms for normal cases, and rely on external mechanisms to take care of all the others. Much of the hope placed in exception mechanisms results from this dream. In Ada, for example, you may deal with an abnormal case by writing something like

```
if some_abnormal_situation_detected then
        raise some_exception;
end;
"Go on with normal processing"
```

where execution of the **raise** instruction stops the execution of the current routine or block and transfers control to an "exception handler" written in one of the direct or indirect callers. But this is a control structure, not a method for dealing with abnormal cases. In the end you still have to decide what to do in these cases: is it possible to correct the situation? If so, how, and what should the system do next? If not, how quickly and gracefully can you terminate the execution?

Chapter 12.

We saw in an earlier chapter that a disciplined exception mechanism fits well with the rest of the object-oriented approach and in particular with the notion of Design by Contract. But not all special cases justify resorting to exceptions. The design techniques that we will now examine are perhaps less impressive at first — "low-tech" might be a good characterization — but they are remarkably powerful and address many of the possible practical situations. After studying them we will review the cases in which exceptions remain indispensable.

The a priori scheme

Perhaps the most important criterion in dealing with abnormal cases at the module interface level is specification. If you know exactly what inputs each software element is prepared to accept, and what guarantees it ensures in return, half the battle is won.

"Zen and the art of software reliability: guaranteeing more by checking less", page 242.

This idea was developed in depth as part of the study of Design by Contract. We saw in particular that, contrary to conventional wisdom, one does not obtain reliability by including many possible redundant checks, but by assigning every consistency constraint to the responsibility of just one class, either the client or the supplier.

Including the constraint in a routine precondition means assigning it to the clients. The precondition expresses what is required to make the routine's operation possible:

```
operation (x: …) is
        require
                precondition (x)
        do
                … Code that will only work if precondition is met …
        end
```

The precondition should, whenever possible, be complete, in the sense of guaranteeing that any call satisfying will succeed. If so, there are two ways to write the corresponding client extracts. One is to test explicitly:

```
if precondition (y) then
        operation (y)
else
        … Appropriate alternate action …
end
```

(For brevity this example uses an unqualified call, but of course most calls will be of the qualified form *z*.*operation* (*y*).) The other possibility avoids the **if**…**then**…**else** by ensuring that the context leading to the call ensures the precondition:

```
… Some instructions that, among other possible effects, ensure precondition (y) …
                check precondition (y) end
operation (y)
```

On **check** *see "AN ASSERTION INSTRUCTION", 11.11, page 379.*

As shown here and in many other examples throughout this book, it is desirable in this case to include a **check** instruction, with two benefits: making it immediately clear,

for the reader of the software text, that you did not forget the precondition but instead checked that it would hold; and, in case your deduction was wrong, facilitating debugging when the software is executed with assertion monitoring on. (If you do not remember the details of the **check** instruction, make sure to re-read the corresponding section now.)

Such use of a precondition, which the client has to ensure beforehand — either by testing for it as in **if** *precondition* (*y*) …, or by relying on other instructions —, may be called the a priori scheme: the client is asked to take advance measures to avoid any error.

With the a priori scheme, any remaining run-time failure signals a design error — a client not abiding by the rules. Then the only long-term solution is to correct the error, although we have seen that for mission-critical systems it is possible to devise software-fault-tolerant solutions which, on assertion violation, will attempt partial recovery through **retry**.

Obstacles to the a priori scheme

Because of its simplicity and clarity, the a priori scheme is ideal in principle. Three reasons, however, prevent it from being universally applicable:

A1 • Efficiency considerations make it impractical in some cases to test for the precondition before a call.

A2 • Limitations of practical assertion languages imply that some of the assertions of interest cannot be expressed formally.

A3 • Finally, some of the conditions required for the successful execution of a routine depend on external events and are not assertions at all.

An example of case A1, from numerical computation, is a linear equation solver. A function for solving an equation of the form $a\,x = b$, where a is a matrix, and x (the unknown) and b are vectors, might take the following form in an appropriately designed *MATRIX* class:

inverse (*b*: *VECTOR*): *VECTOR*

so that a particular equation will be solved by x := *a*.*inverse* (*b*). A unique solution only exists if the matrix is not “singular”. (Singularity mans that one of the rows is a linear combination of others or, equivalently, that the determinant is zero.) We could make non-singularity the precondition of *inverse*, requiring client calls to be of the form

```
if a.singular then
        … Appropriate error action …
else
        x := a.inverse (b)
end
```

This technique works but is very inefficient: determining whether a matrix is singular is essentially the same operation as solving the associated linear equation. Standard algorithms (Gaussian elimination) will at each step compute a divisor, called the pivot; if the pivot found at some step is zero or below a certain threshold, this shows that the matrix was singular. This result is obtained as a byproduct of the equation-solving

algorithm; to obtain it separately would take almost as much computation time as to execute the entire algorithm. So doing the job in two steps — first finding out whether the matrix is singular, and then, if it is not, computing the solution — is a waste of effort.

Examples of A2 include cases in which the precondition is a global property of a data structure and would need to be expressed with quantifiers, for example the requirement that a graph contain no cycles or that a list be sorted. Our notation does not support this. As noted, we can usually rely on such assertions using functions; but then we might be back in case A1, as the precondition can be too costly to check before every call.

Finally, limitation A3 arises when it is impossible to test the applicability of the operation without attempting to execute it, because interaction with the outside world — a human user, a communication line, a file system — is involved.

The a posteriori scheme

When the a priori scheme does not work, a simple *a posteriori* scheme is sometimes possible. The idea is to try the operation first and then find out how it went; this will work if a failed attempt has no irrecoverable consequences.

The matrix equation problem provides a good example. With an a posteriori scheme, client code will now be of the form

```
a.invert (b)
if a.inverted then
        x := a.inverse
else
        … Appropriate error action …
end
```

Function *inverse* has been replaced by a procedure *invert*, for which a more accurate name might be *attempt_to_invert*. A call to this procedure sets the attribute *inverted* to true or false to indicate whether a solution was found; if it was, the procedure makes the solution itself available through attribute *inverse*. (An invariant clause in the matrix class may state that *inverted* = (*inverse* /= *Void*).)

With this method, any function that may produce an error condition is transformed into a procedure, the result being accessible, if it exists, through an attribute set by the procedure. To save space you may use a once function rather than an attribute if at most one answer is needed at any time.

This also works for input operations. For example a “read” function that may fail is better expressed as a procedure that attempts to read, and two attributes, one boolean indicating whether the operation succeeded and the other yielding the value read if any.

This technique, as you will have noted, is in line with the Command-Query Separation principle. A function that may fail to compute its intended result is not side-effect-free, and so is better decomposed into a procedure that attempts to compute the value and two queries (functions or attributes), one to ascertain success and the other to

yield the value in case of success. The technique is also consistent with the idea of objects as machines, whose state can be changed by commands and accessed by queries.

The example of input functions is typical of cases that can benefit from this scheme. Most of the read functions provided by programming languages or the associated libraries are of the form "next integer", "next string" etc., requiring the client to state in advance the type of the element to be read. Inevitably, they will fail when the actual input does not match the expectation. A read procedure, on the other hand, can attempt to read the next input item without any preconception of what it will be, and then return information about its type through one of the queries available to clients.

This example highlights one of the constant rules for dealing with failure: whenever available, a method for engineering out failures is preferable to methods for recovering from failures.

The role of an exception mechanism

The preceding discussion has shown that in most cases methods based on standard control structures, principally essentially conditional instructions, are adequate for dealing with abnormal cases. Although the a priori scheme is not always practical, it is often possible to check success after attempting an operation.

There remain, however, cases in which both a priori and a posteriori techniques are inadequate. The above discussion leaves only three categories of such cases:

- Some abnormal events such as numerical failure or memory exhaustion can lead to preemptive action by the hardware or operating system, such as raising an exception and, unless the software catches the exception, terminating execution abruptly. This is often intolerable, especially in systems with continuous availability requirements (think of telephone switches and many medical systems).
- Some abnormal situations, although not detectable through a precondition, must be diagnosed at the earliest possible time; the operation must not be allowed to run to completion (for a posteriori checking) because it could lead to disastrous consequences, such as destroying the integrity of a database or even endanger human lives, as in a robot control system.
- Finally, the developer may wish to include some form of protection against the most catastrophic consequences of any remaining errors in the software; this is the use of exceptions for software fault tolerance.

"Why run-time monitoring?", page 399.

In such cases, exception-based techniques appear necessary. The orderly exception mechanism presented in an earlier chapter provides the appropriate tools.

Chapter 12.

23.7 CLASS EVOLUTION: THE OBSOLETE CLAUSE

We try to make our classes perfect. All the techniques accumulated in this discussion tend towards that goal — unreachable, of course, but useful as an ever present ideal.

Unfortunately (with no intention of offending the reader) we are not ourselves perfect. What happens if, after a few months or a few years, we realize that some of the interface of a class could have been designed better? The dilemma is not pleasant:

The real Hugo quote is about Liberty.

- Favor the current users: this will mean continuing to live with an obsolete design whose unpleasant effects will be felt more and more sorely as time passes. This is known in the computer industry as *upward compatibility*. Compatibility, how many crimes have been committed in thy name! (as Victor Hugo almost wrote).

  > According to Unix folklore, one of the less pleasant conventions of the Make tool, which has bothered quite a few novice users, was detected not too long after the first release. Since it implied a language change and the inconvenience was not a show-stopper, the decision was made to let things stand so as not to disturb the user community. The Make user community, at that time, must have included a dozen or two people at Bell Laboratories.

- Favor the future users: you cause trouble to the current ones, whose only sin was to trust you too early.

Sometimes — but sometimes only — there is a way out. We introduce into our notation the concept of **obsolete features** and **obsolete classes**. Here is an example of obsolete routine:

```
enter (i: INTEGER; v: G) is
        obsolete "Use put (value, index) instead"
    require
        correct_index (i)
    do
        put (v, i)
    ensure
        entry (i) = v
    end
```

This is a real example, although no longer current. Here is the context. Early in the evolution of the Base libraries, we realized that the names and conventions were not systematic enough; this is when the principles of style developed in chapter 26 of this book were codified. They entailed in particular using the name *put* rather than *enter* for the procedure that replaces an array element (and *item* rather than *entry* for the corresponding query) and, to make things worse, reversing the order of arguments, for compatibility with features of other classes in the library.

The above declaration smoothes out the evolution. Note how the old feature, *enter*, has a new implementation, relying on the new feature, *put*; you should normally use this scheme when making a feature obsolete, to avoid carrying along two competing implementations with the resulting reliability and extendibility risks.

What are the consequences of making a feature obsolete? Not much in practice. The tools of the environment must recognize this property, and output the corresponding

warnings when a client system uses the class. The compiler, in particular, will output a message, which includes the string that has been included after the keyword **obsolete**, such as Use *put* (*value, index*) instead in our example. That is all. The feature otherwise continues to be normally usable.

Similar syntax enables you to declare an entire class as obsolete.

What you are providing your client developers, then, is a migration path. By telling them that a feature will be removed, you encourage them to adapt their software; but you are not putting a knife to their throat. If the change is justified — as it should be — users of the class will not resent having to update their part; what is unacceptable is, when they receive a new version, to be forced to do all the changes immediately. Given a little time, they will readily comply.

In practice, the migration period should be bounded. At the next major release — a few months later, a year at most — you should remove the obsolete features and classes for good. Otherwise no one will take obsolescence warnings seriously. This is why the example was mentioned above as "no longer current": *enter* and *entry* were removed several years ago. But in their short lives they helped keep more than one developer happy.

Feature and class obsolescence only solve a specific problem. The comment made when we discussed the Open-Closed principle and how inheritance enables you to adapt a parent's design without disturbing the original is fully applicable here: when a design is *flawed*, the only reasonable approach is to correct it, while making your best efforts to help current users make the transition. Neither inheritance-cum-redefinition nor obsolescence should serve as cover-ups for bugs in existing software. But obsolescence is precious when the original design, while satisfactory in other respects, does not conform to your current views; it typically resulted from a narrower and less clear perspective than what you have gained now. Although there was nothing fundamentally wrong with the old design, you can do better: simpler interfaces, better consistency with the rest of the software, interoperability with other products, better naming conventions. In such cases, making a few features and classes obsolete is a remarkable way to protect the investment of your current users while moving ahead to an ever brighter future.

See "The module view", page 495.

23.8 DOCUMENTING A CLASS AND A SYSTEM

Having mastered the most advanced techniques of class interface design, you build a set of great classes. To achieve the success they deserve, they will need good interface documentation. We have seen the basic documentation tool: the short form and its variant the flat-short form. Let us summarize their use and examine a complementary mechanism that works on entire systems rather than just classes.

Mentions of the short form in this discussion will encompass the flat-short form as well. The difference between the two, as you will remember, is that the flat-short form takes inherited features into account, whereas the plain short form only relies on the immediate features introduced in the class itself. In most practical cases, the flat-short form is what client authors will need.

See "Using assertions for documentation: the short form of a class", page 390, and "The flat-short form", page 543.

Showing the interface

The short form directly applies the rule of Information Hiding by removing all secret information from client view. Secret information includes:

- Any non-exported feature and anything having to do with it (for example, a clause of an assertion which refers to the feature).
- Any routine implementation, as given by the **do** … clause.

What remains is abstract information about the class, providing authors of client classes, current or prospective, with the implementation-independent description that they need to use it effectively.

See the "common misunderstanding" cited on page 52.

> Remember that the purpose is abstraction, not protection. We do not necessarily wish to prevent client authors from accessing secret class elements; we wish to relieve them from having to do so. By separating function from implementation, information hiding decreases the amount of information to be mastered; client authors should view it as help rather than hindrance.

The short form avoids the technique (supported, without assertions, by Ada, Modula-2 and Java) of writing separate and partially redundant module interfaces, as this can mean trouble for evolution; as always in software engineering, repetition breeds inconsistency. Instead it puts everything into the class and relies on computer tools to extract abstract information.

"Self-Documentation", page 54.

The underlying principle was introduced at the beginning of this book: try to make the software as self-documenting as possible. In this effort, judiciously chosen assertions will play a fundamental part. Examining the examples of this chapter and constructing their short forms (at least mentally) should provide clear enough evidence.

To help the short form deliver the best possible results, you should keep it in mind when writing your classes, and apply the following principle:

Documentation principle

Try to write the software so that it includes all the elements needed for its documentation, recognizable by the tools that are available to extract documentation elements automatically at various levels of abstraction.

This simply translates the more general Self-Documentation principle into a practical rule to be applied day to day by developers. Particularly important will be:

- Well-designed preconditions, postconditions and invariants.
- Careful choice of names for both classes and features.
- Informative indexing clauses.

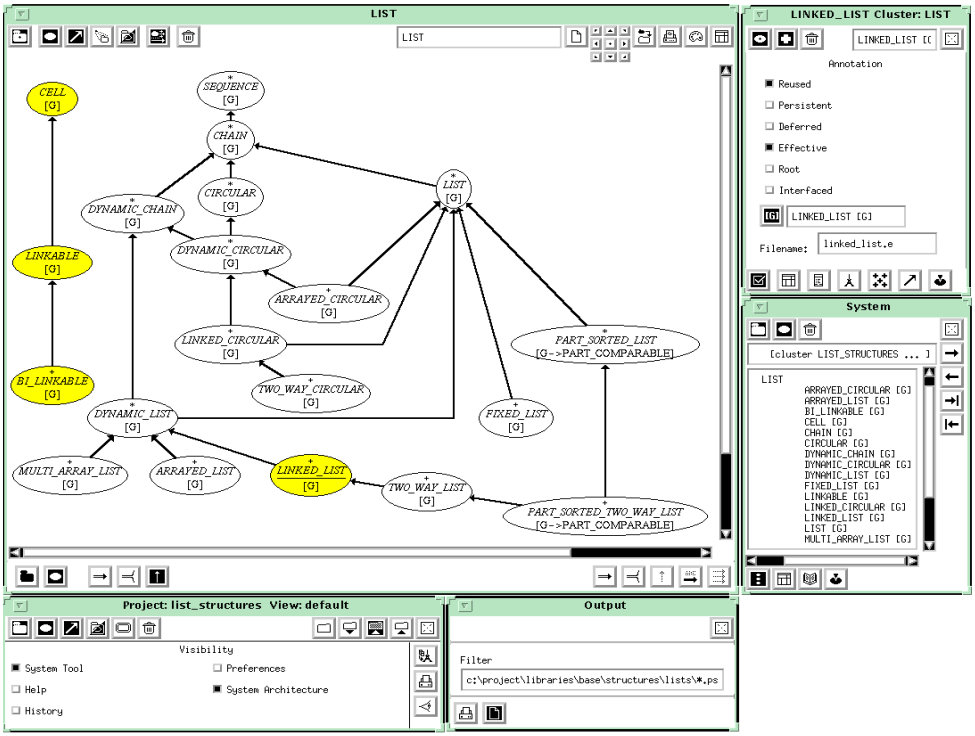
Chapter 26.

The chapter on style will give precise guidelines on the last two points.

System-level documentation

The **short** and **flat-short** tools, when applied to software developed according to the rules developed in this book (assertions, Design by Contract, information hiding, clear and systematic naming conventions, header comments etc.) apply the Documentation principle at the module level. There is also a need for higher-level documentation — documentation on an entire system, or one of its subsystems — applying the same principle. But here textual output, although necessary, is not sufficient. To grasp the organization of a possibly complex system, you will want graphical descriptions.

The Case tool of ISE's environment, based on Business Object Notation concepts, provides such system views, as illustrated below for a session devoted to reverse-engineering of the Base libraries.



A system architecture diagram

Although further details fall beyond the scope of this discussion, we may note that the tool supports the exploration of large systems through zooming, unzooming and other abstraction mechanisms such as the ability to focus on a cluster (subsystem) or one of its subclusters as well as the entire system; also, it combines graphical views, essential to provide a general glimpse of an architecture, with textual information about the components of a system, dictionaries of abstractions etc.

See [M 1995c].

All these tools are applications of the Documentation principle, tending towards the production of software which, thanks to carefully designed notations and with the help of advanced environments, should get us ever closer to the ideal of self-documentation.

23.9 KEY CONCEPTS INTRODUCED IN THIS CHAPTER

- A class should be known by its interface, which specifies the services offered independently of their implementation.
- Class designers should strive for simple, coherent interfaces.
- One of the key issues in designing modules is which features should be exported, and which should remain secret.
- The design of reusable modules is not necessarily right the first time, but the interface should stabilize after some use. If not, there is a flaw in the way the interface was designed. The mechanism of **obsolete** features and classes makes it possible to smooth over the transition to a better design.
- It is often fruitful to treat some data structures as active machines, with an internal state remembered from one feature call to the next.
- Proper use of assertions (preconditions, postconditions, invariants) is essential for documenting interfaces.
- Abnormal situations are best dealt with through standard control structures, either through the a priori scheme, which checks applicability before calling an operation, or through the a posteriori scheme, which attempts the operation and then examines whether it has succeeded. A disciplined exception mechanism remains necessary in cases when execution must immediately cancel a potential dangerous operation.

23.10 BIBLIOGRAPHICAL NOTES

The work of Parnas [Parnas 1972] [Parnas 1972a] introduced many seminal ideas on the design of interfaces.

The operand-option distinction, and the resulting principle, come from [M 1982a].

The notion of "active data structure" is supported in some programming languages by control abstractions called iterators. An iterator is a mechanism defined together with a data structure, which describes how to apply an arbitrary operation to every element of an instance of the data structure. For example, an iterator associated with a list describes a looping mechanism for traversing the list, applying a given operation to every list element; a tree iterator specifies a tree traversal strategy. Iterators are available in the programming language CLU [Liskov 1981]; [Liskov 1986] contains a detailed discussion of the concept. In object technology, we can implement iterators through classes rather than predefine them as language constructs; see [M 1994a], which applies to library design a number of ideas from the present chapter.

The example of the self-adaptive complex number implementation comes from [M 1979], where it was expressed in Simula.

Literate programming [Knuth 1984] emphasizes, like this chapter, that programs should contain their own documentation. Its concepts, however, are quite different from those of object technology; one of the exercises below invites you to compare the approaches.

Articles by James McKim and Richard Bielak [Bielak 1993], [McKim 1992a] [McKim 1995] present useful advice on class interface design based on the notion of Design by Contract.

EXERCISES

E23.1 A function with side effects

The example of component-level memory management for linked lists had a function *fresh* that calls a procedure, *remove* for stacks, and hence produces a side effect on the data structure. Discuss whether this is acceptable.

Function fresh appeared on page 299.

E23.2 Operands and options

Examine a class or routine library to which you have access and study its routines to determine, for each of them, which arguments are operands and which are options.

E23.3 Optional arguments

Some languages, such as Ada, offer the possibility for a routine of having optional arguments, each with an associated argument keyword; if the keyword is not included, the argument may be set to a default. Discuss which of the advantages of the Operand principle this technique retains, and which it fails to ensure.

E23.4 Number of elements as function

Adapt the definition of class *LINKED_LIST* [*G*] so that *count* is a function rather than an attribute, the interface of the class being unchanged.

E23.5 Searching in a linked list

Write the *LINKED_LIST* procedure *search* (*x*: *G*), searching for the next occurrence of *x*.

E23.6 Invariant theorems

Prove the three assertion clauses listed as theorems in the first part of the invariant for *LINKED_LIST*.

Page 785.

E23.7 Two-way lists

Write a class describing two-way linked lists, with the same interface as *LINKED_LIST*, but more efficient implementations of some operations such as *back*, *go* and *finish*.

E23.8 Alternative linked list class design

See [M 1988], *sections 9.1 and A.5.*

Devise a variant of the linked list class design using the convention that an empty list is considered both *after* and *before*. (This was the technique used in the first edition of this book.) Assess it against the approach developed in the present chapter.

E23.9 Insertion in a linked list

remove is on page 791.

Drawing inspiration from *remove*, write the procedures *put_left* and *put_right* to insert an element to the left and right of the cursor position.

E23.10 Circular lists

Explain why the *LINKED_LIST* class may not be used for circular lists. (Hint: show what assertions would be violated.) Write a class *CIRCULAR_LINKED* that implements circular lists.

E23.11 Side-effect-free input functions

Design a class describing input files, with input operations, without any side-effect-producing functions. Only the class interface (without the **do** clause describing the routine implementations, but with the routine headers and any appropriate assertions) is required.

E23.12 Documentation

Discuss, expand and refine the Self-Documentation principle and its various developments in this book, considering various kinds of documentation in software and examining what styles of documentation are appropriate in various circumstances and at various levels of abstraction.

E23.13 Self-documenting software

For references on literate programming see the bibliographic notes to this chapter.

The approach to self-documenting software advocated in this chapter emphasizes terseness and does not readily support long explanations of design decisions. Knuth's "Literate programming" style of design combines techniques from programming, writing and text processing to integrate a program, its complete design documentation and its design history within a single document. The method relies on a classical paradigm: top-down development of a single program. Starting from Knuth's work, discuss how his method could be transposed to the object-oriented development of reusable components.