11

Design by Contract: building reliable software

*E*quipped with the basic concepts of class, object and genericity, you can by now write software modules that implement possibly parameterized types of data structures. Congratulations. This is a significant step in the quest for better software architectures.

But the techniques seen so far are not sufficient to implement the comprehensive view of quality introduced at the beginning of this book. The quality factors on which we have concentrated — reusability, extendibility, compatibility — must not be attained at the expense of reliability (*correctness* and *robustness*). Although, as recalled next, the reliability concern was visible in many aspects of the discussion, we need more.

> The need to pay more attention to the semantic properties of our classes will be particularly clear if you remember how classes were defined: as implementations of abstract data types. The classes seen so far consist of attributes and routines, which indeed represent the functions of an ADT specification. But an ADT is more than just a list of available operations: remember the role played by the semantic properties, as expressed by the axioms and preconditions. They are essential to capture the true nature of the type's instances. In studying classes, we have — temporarily — lost sight of this semantic aspect of the ADT concept. We will need to bring it back into the method if we want our software to be not just flexible and reusable, but also correct and robust.

Assertions and the associated concepts, explained in this chapter, provide some of the answer. Although not foolproof, the mechanisms presented below provide the programmer with essential tools for expressing and validating correctness arguments. The key concept will be **Design by Contract**: viewing the relationship between a class and its clients as a formal agreement, expressing each party's rights and obligations. Only through such a precise definition of every module's claims and responsibilities can we hope to attain a significant degree of trust in large software systems.

In reviewing these concepts, we shall also encounter a key problem of software engineering: how to deal with run-time errors — with contract violations. This leads to the subject of *exception handling*, covered in the next chapter. The distribution of roles between the two chapters roughly reflects the distinction between the two components of reliability; as you will recall, correctness was defined as the software's ability to perform according to its specification, and robustness as its ability to react to cases not included in the specification. Assertions (this chapter) generally cover correctness, and exceptions (next chapter) generally cover robustness.

Some important extensions to the basic ideas of Design by Contract will have to wait until the presentation of inheritance, polymorphism and dynamic binding, enabling us to go from contracts to *subcontracting*.

11.1 BASIC RELIABILITY MECHANISMS

The preceding chapters already introduced a set of techniques that directly address the goal of producing reliable software. Let us review them briefly; it would be useless to consider more advanced concepts until we have put in place all the basic reliability mechanisms.

First, the defining property of object technology is an almost obsessive concern with the *structure* of software systems. By defining simple, modular, extendible architectures, we make it easier to ensure reliability than with contorted structures as often result from earlier methods. In particular the effort to limit inter-module communication to the strict minimum was central to the discussion of modularity that got us started; it resulted in the prohibition of such common reliability risks as global variables, and in the definition of restricted communication mechanisms, the client and inheritance relations. The general observation is that the single biggest enemy of reliability (and perhaps of software quality in general) is complexity. Keeping our structures as simple as possible is not enough to ensure reliability, but it is a necessary condition. So the discussion of the previous chapters provides the right starting point for the systematic effort of the present one.

Also necessary if not sufficient is the constant emphasis on making our software *elegant* and *readable*. Software texts are not just written, they are read and rewritten many times; clarity and simplicity of notation, such as have been attempted in the language constructs introduced so far, are a required basis for any more sophisticated approach to reliability.

Another indispensable weapon is automatic memory management, specifically *garbage collection*. The chapter on memory management explained in detail why, for any system that creates and manipulates dynamic data structures, it would be dangerous to rely on manual reclamation (or no reclamation). Garbage collection is not a luxury; it is a crucial reliability-enhancing component of any O-O environment.

The same can be said of another technique presented (in connection with genericity) in the last chapter: static typing. Without statically enforced type rules, we would be at the mercy of run-time typing errors.

All these techniques provide the necessary basis, from which we can now take a closer look at what it will take for a software system to be correct and robust.

11.2 ABOUT SOFTWARE CORRECTNESS

We should first ask ourselves what it *means* for a software element to be correct. The observations and deductions that will help answer this question will seem rather trivial at first; but let us not forget the comment (made once by a very famous scientist) that scientific reasoning is nothing but the result of starting from ordinary observations and continuing with simple deductions — only very patiently and stubbornly.

Assume someone comes to you with a 300,000-line C program and asks you "Is this program correct?". There is not much you can answer. (If you are a consultant, though, try answering "no" and charging a high fee. You might just be right.)

To consider the question meaningful, you would need to get not only the program but also a precise description of what it is supposed to do — a *specification*.

The same comment is applicable, of course, regardless of the size of a program. The instruction $x := y + 1$ is neither correct nor incorrect; these notions only make sense with respect to a statement of what one expects from the instruction — what effect it is intended to have on the state of the program variables. The instruction is correct for the specification

"Make sure that x and y have different values"

but it is incorrect vis-à-vis the specification

"Make sure that x has a negative value"

(since, assuming that the entities involved are integers, x may end up being non-negative after the assignment, depending on the value of y).

These examples illustrate the property that must serve as the starting point of any discussion of correctness:

Software Correctness property

Correctness is a relative notion.

A software system or software element is neither correct nor incorrect per se; it is correct or incorrect with respect to a certain specification. Strictly speaking, we should not discuss whether software elements are *correct*, but whether they are *consistent* with their specifications. This discussion will continue to use the well-accepted term "correctness", but we should always remember that the question of correctness does not apply to software elements; it applies to pairs made of a software element and a specification.

In this chapter we will learn how to express such specifications through **assertions**, to help us assess the correctness of our software. But we will go further. It turns out (and only someone who has not practiced the approach will think of this as a paradox) that just writing the specification is a precious first step towards *ensuring* that the software actually meets it. So we will derive tremendous benefits from writing the assertions at the same time as we write the software — or indeed before we write the software. Among the consequences we will find the following:

- Producing software that is correct from the start because it is designed to be correct. [Mills 1975]. The title of an article written by Harlan D. Mills (one of the originators of "Structured Programming") in the nineteen-seventies provides the right mood: *How to write correct programs and know it*. To "know it" means to equip the software, at the time you write it, with the arguments showing its correctness.
- Getting a much better understanding of the problem and its eventual solutions.
- Facilitating the task of software documentation. As we will see later in this chapter, assertions will play a central part in the object-oriented approach to documentation.
- Providing a basis for systematic testing and debugging.

The rest of this chapter explores these applications.

A word of warning: C, C++ and some other languages (following the lead of Algol W) have an "assert" instruction that tests whether a certain condition holds at a certain stage of the software's execution, and stops execution if it does not. Although relevant to the present discussion, this concept represents only a small part of the use of assertions in the object-oriented method. So if like many other software developers you are familiar with such instructions but have not been exposed to the more general picture, almost all the concepts of this chapter will be new.

11.3 EXPRESSING A SPECIFICATION

We can turn the preceding informal observations into a simple mathematical notation, borrowed from the theory of formal program validation, and precious for reasoning about the correctness of software elements.

Correctness formulae

Let A be some operation (for example an instruction or a routine body). A **correctness formula** is an expression of the form

$$\{P\}\ A\ \{Q\}$$

denoting the following property, which may or may not hold:

Meaning of a correctness formula $\{P\}\ A\ \{Q\}$

"*Any execution of* A*, starting in a state where* P *holds, will terminate in a state where* Q *holds.*"

Correctness formulae (also called *Hoare triples*) are a mathematical notation, not a programming construct; they are not part of our software language, but only designed to guide us through this discussion by helping to express properties of software elements.

In $\{P\}\ A\ \{Q\}$ we have seen that A denotes an operation; P and Q are properties of the various entities involved, also called assertions (the word will be defined more precisely later). Of the two assertions, P is called the precondition and Q the postcondition. Here is a trivial correctness formula (which, assuming that x is an integer entity, holds):

$$\{x >= 9\}\ \ x := x + 5\ \ \{x >= 13\}$$

The use of correctness formulae is a direct application of the Software Correctness Property. What the Property stated informally — that correctness is only meaningful relative to a particular specification — correctness formulae turn into a form that is directly usable for working on the software: from now on the discourse about software correctness will not be about individual software elements A, but about triples containing a software element A, a precondition P and a postcondition Q. The sole aim of the game is to establish that the resulting $\{P\}\ A\ \{Q\}$ correctness formulae hold.

The number 13 appearing in the postcondition is not a typo! Assuming a correct implementation of integer arithmetic, the above formula holds: if $x >= 9$ is true before the instruction, $x >= 13$ will be true after the instruction. Of course we can assert more interesting things: with the given precondition, the most interesting postcondition is the strongest possible one, here $x >= 14$; with the given postcondition, the most interesting precondition is the *weakest* possible one, here $x >= 8$. From a formula that holds, you can always get another one by strengthening the precondition or weakening the postcondition. We will now examine more carefully these notions of "stronger" and "weaker".

Weak and strong conditions

One way to look at a specification of the form $\{P\}\ A\ \{Q\}$ is to view it as a job description for A — an ad in the paper, which states "We are looking for someone whose work will be to start from initial situations as characterized by P, and deliver results as defined by Q".

Here is a small quiz to help you sharpen your understanding of the concepts.

Assume one of your friends is looking for a job and comes across several such ads, all with similar salary and benefits, but differing by their Ps and Qs. (Tough times have encouraged the companies that publish the ads to resort to this notation, which they like for its mathematical compactness since the newspaper charges by the word.) Like everyone else, your friend is lazy, that is to say, wants to have the easiest possible job. He is asking for your advice, always a dangerous situation. What should you recommend for P: choose a job with a *weak* precondition, or a *strong* one? Same question for the postcondition Q. (The answers appear right after this, but do take the time to decide the issue for yourself before turning the page.)

The precondition first. From the viewpoint of the prospective employee — the person who has to perform what has been called A — the precondition P defines the conditions under which the required job will start or, to put it differently, the set of cases that have to be handled. So a strong P is good news: it means that you only have to deal with a limited set of situations. The stronger the P, the easier for the employee. In fact, the perfect sinecure is the job defined by

Sinecure 1

$\{False\}\ A\ \{...\}$

The postcondition has been left unspecified because it does not matter what it is. Indeed if you ever see such an ad, do not even bother reading the postcondition; *take the job right away*. The precondition $False$ is the strongest possible assertion, since it is never satisfied in any state. Any request to execute A will be incorrect, and the fault lies not with the agent responsible for A but with the requester — the client — since it did not observe the required precondition, for the good reason that it is impossible to observe it. Whatever A does or does not do may be *useless*, but is always *correct* — in the sense, defined earlier, of being consistent with the specification.

The above job specification is probably what a famous police chief of a Southern US city had in mind, a long time ago, when, asked by an interviewer why he had chosen his career, he replied: "Obvious — it is the only job where the customer is always wrong".

For the postcondition Q, the situation is reversed. A strong postcondition is bad news: it indicates that you have to deliver more results. The weaker the Q, the better for the employee. In fact, the second best sinecure in the world is the job defined, regardless of the precondition, by

Sinecure 2

$\{...\}\ A\ \{True\}$

The postcondition $True$ is the weakest possible assertion, satisfied by all states.

The notions of "stronger" and "weaker" are formally defined from logic: $P1$ is said to be stronger than $P2$, and $P2$ weaker than $P1$, if $P1$ implies $P2$ and they are not equal. As every proposition implies $True$, and $False$ implies every proposition, it is indeed legitimate to speak of $True$ as the weakest and $False$ as the strongest of all possible assertions.

Why, by the way, is Sinecure 2 only the "second best" job in the world? The reason has to do with a fine point that you may have noticed in the definition of the meaning of $\{P\}\ A\ \{Q\}$ on the preceding page: termination. The definition stated that the execution must *terminate* in a state satisfying Q whenever it is started in a state satisfying P. With Sinecure 1 there are no states satisfying P, so it does not matter what A does, even if it is a program text whose execution would go into an infinite loop or crash the computer. Any A will be "correct" with respect to the given specification. With Sinecure 2, however, there

must be a final state; that state does not need to satisfy any specific properties, but it must exist. From the viewpoint of whoever has to perform *A*: you need to do nothing, *but you must do it in finite time*.

> Readers familiar with theoretical computing science or program proving techniques will have noted that the *{P} A {Q}* notation as used here denotes **total correctness**, which includes termination as well as conformance to specification. (The property that a program will satisfy its specification if it terminates is known as partial correctness.) See [M 1990] for a detailed presentation of these concepts.

The discussion of whether a stronger or weaker assertion is "bad news" or "good news" has taken the viewpoint of the prospective employee. If, changing sides, we start looking at the situation as if we were the employer, everything is reversed: a weaker precondition will be good news, as it means a job that handles a broader set of input cases; so will be a stronger postcondition, as it means more significant results. This reversal of criteria is typical of discussions of software correctness, and will reappear as the central notion of this chapter: *contracts* between client and supplier modules, in which a benefit for one is an obligation for the other. To produce effective and reliable software is to draw up the contract representing the best possible compromise in all applicable client-supplier communications.

11.4 INTRODUCING ASSERTIONS INTO SOFTWARE TEXTS

Once we have defined the correctness of a software element as the consistency of its implementation with its specification, we should take steps to include the specification, together with the implementation, in the software itself. For most of the software community this is still a novel idea: we are accustomed to programs as defining the operations that we command our hardware-software machines to execute for us (the *how*); it is less common to treat the description of the software's purposes (the *what*) as being part of the software itself.

To express the specification, we will rely on assertions. An assertion is an expression involving some entities of the software, and stating a property that these entities may satisfy at certain stages of software execution. A typical assertion might express that a certain integer has a positive value or that a certain reference is not void.

Mathematically, the closest notion is that of predicate, although the assertion language that we shall use has only part of the power of full predicate calculus.

Syntactically, the assertions of our notation will simply be boolean expressions, with a few extensions. One of these extensions, the **old** notation, is introduced later in this chapter. Another is the use of the semicolon, as in

```
n > 0 ; x /= Void
```

The meaning of the semicolon is equivalent to that of an **and**. As between declarations and instructions, the semicolon is actually optional, and we will omit it when assertion clauses appear on separate lines; just consider that there is an implicit **and** between successive assertion lines. These conventions facilitate identification of the individual components of an assertion. It is indeed possible, and usually desirable, to label these components individually, as in

Positive: *n* > 0
Not_void: *x* /= *Void*

If present, the labels (such as *Positive* and *Not_void* in this example) will play a role in the run-time effect of assertions — to be discussed later in this chapter — but for the moment they are mainly there for clarity and documentation.

The next few sections will review this principal application of assertions: as a conceptual tool enabling software developers to construct correct systems and to document *why* they are correct.

11.5 PRECONDITIONS AND POSTCONDITIONS

The first use of assertions is the semantic specification of routines. A routine is not just a piece of code; as the implementation of some function from an abstract data type specification, it should perform a useful task. It is necessary to express this task precisely, both as an aid in designing it (you cannot hope to ensure that a routine is correct unless you have specified what it is supposed to do) and, later, as an aid to understanding its text.

You may specify the task performed by a routine by two assertions associated with the routine: a *precondition* and a *postcondition*. The precondition states the properties that must hold whenever the routine is called; the postcondition states the properties that the routine guarantees when it returns.

A stack class

An example will enable us to become familiar with the practical use of assertions. In the previous chapter, we saw the outline of a generic stack class, under the form

class *STACK* [*G*] **feature**
… Declaration of the features:
count, empty, full, put, remove, item
end

An implementation will appear below. Before considering implementation issues, however, it is important to note that the routines are characterized by strong semantic properties, independent of any specific representation. For example:

- Routines *remove* and *item* are only applicable if the number of elements is not zero.
- *put* increases the number of elements by one; *remove* decreases it by one.

Such properties are part of the abstract data type specification, and even people who do not use any approach remotely as formal as ADTs understand them implicitly. But in common approaches to software construction software texts reveal no trace of them. Through routine preconditions and postconditions you can turn them into explicit elements of the software.

We will express preconditions and postconditions as clauses of routine declarations introduced by the keywords **require** and **ensure** respectively. For the stack class, leaving the routine implementations blank for the time being, this gives:

```
indexing
        description: "Stacks: Dispenser structures with a Last-In, First-Out %
                     %access policy"
class STACK1 [G] feature -- Access
        count: INTEGER
                -- Number of stack elements

        item: G is
                -- Top element
            require
                not empty
            do
                …
            end
feature -- Status report
        empty: BOOLEAN is
                -- Is stack empty?
            do … end

        full: BOOLEAN is
                -- Is stack representation full?
            do
                …
            end
feature -- Element change
        put (x: G) is
                -- Add x on top.
            require
                not full
            do
                …
            ensure
                not empty
                item = x
                count = old count + 1
            end

        remove is
                -- Remove top element.
            require
                not empty
            do
                …
            ensure
                not full
                count = old count – 1
            end
end
```

Both the **require** and the **ensure** clauses are optional; when present, they appear at the places shown. The **require** appears before the **local** clause, if present. The next sections explain in more detail the meaning of preconditions and postconditions.

Note the division into several **feature** clauses, useful to group the features into categories indicated by the clauses' header comments. Access, Status report and Element change are some of a dozen or so standard categories used throughout the libraries and, whenever applicable, subsequent examples in this book.

More on feature categories in "A stack class", page 348.

Preconditions

A precondition expresses the constraints under which a routine will function properly. Here:

- *put* may not be called if the stack representation is full.
- *remove* and *item* may not be applied to an empty stack.

A precondition applies to all calls of the routine, both from within the class and from clients. A correct system will never execute a call in a state that does not satisfy the precondition of the called routine.

Postconditions

A postcondition expresses properties of the state resulting from a routine's execution. Here:

- After a *put*, the stack may not be empty, its top is the element just pushed, and its number of elements has been increased by one.
- After a *remove*, the stack may not be full, and its number of elements has been decreased by one.

The presence of a postcondition clause in a routine expresses a guarantee on the part of the routine's implementor that the routine will yield a state satisfying certain properties, assuming it has been called with the precondition satisfied.

A special notation, **old**, is available in postconditions; *put* and *remove* use it to express the changes to *count*. The notation **old** *e*, where *e* is an expression (in most practical cases an attribute), denotes the value that *e* had on routine entry. Any occurrence of *e* not preceded by **old** in the postcondition denotes the value of the expression on exit. The postcondition of *put* includes the clause

count = **old** *count* + *1*

to state that *put*, when applied to any object, must increase by one the value of the *count* field of that object.

A pedagogical note

If you are like most software professionals who get exposed to these ideas for the first time, you may be itching to know what effect, if any, the assertions have on the execution of the software, and in particular what happens if one of them gets violated at run time — if *full* is true when someone calls *put*, or *empty* is true when *put* terminates one of its executions. It is too early to give the full answer but as a preview we can use the lawyer's favorite: *it depends*.

More precisely, it depends on what you want. You may decide to treat assertions purely as comments, with no effect on the software's execution; then a run-time assertion violation will remain undetected. But it is also possible to use assertions to check that everything goes according to plan; then during execution the environment will automatically monitor that all assertions hold when they should, and if one does not it will trigger an exception, usually terminating execution and printing a message indicating clearly what happened. (It is also possible to include an exception handling clause that will try to recover from the exception and continue execution; exception handling is discussed in detail in the next chapter.) To specify the policy that you want — no assertion checking, or assertion monitoring at one of various possible levels — you will use a compilation option, which you can set separately for each class.

See "Monitoring assertions at run time", page 393.

The full details of run-time assertion monitoring do appear later in this chapter. But it would be a mistake to attach too much importance to this aspect at this stage (one of the reasons why you were warned earlier not to think too much about the C notion of assertion if that has been your only exposure to the concept). Other aspects of assertions demand our attention first. We have only started to see assertions as a technique to help us get our software right in the first place; we still have much to discover of their *methodological* role as built-in guardians of reliability. The question of what happens if we do fail (in particular if an assertion, in spite of all our efforts, is not satisfied at some execution instant) is important too, but only after we have done all we could to prevent it from arising.

So (although it is never bad to think ahead) you do not need at this point to be too preoccupied by such questions as the possible performance penalty implied by the **old** construct. Must the run-time system preserve values before we start a routine, just to be able to evaluate an **old** expression appearing in the postcondition? *It depends*: in some circumstances (for example testing and debugging) it will indeed be useful to evaluate assertions; in others (for example production runs of fully validated systems) you can treat them as mere annotations to the software text.

All that counts for the next few sections is the methodological contribution of assertions, and of the associated method of Design by Contract: as a conceptual tool for analysis, design, implementation and documentation, helping us to build software in which **reliability is built-in**, rather than achieved or attempted after the fact through debugging; in Mills's terms, enabling us to build correct programs and know it.

11.6 CONTRACTING FOR SOFTWARE RELIABILITY

Defining a precondition and a postcondition for a routine is a way to define a *contract* that binds the routine and its callers.

Rights and obligations

By associating clauses **require** *pre* and **ensure** *post* with a routine *r*, the class tells its clients:

> "If you promise to call *r* with *pre* satisfied then I, in return, promise to deliver a final state in which *post* is satisfied."

In relations between people or companies, a contract is a written document that serves to clarify the terms of a relationship. It is really surprising that in software, where precision is so important and ambiguity so risky, this idea has taken so long to impose itself. A precondition-postcondition pair for a routine will describe the contract that the routine (the *supplier* of a certain service) defines for its callers (the *clients* of that service).

Perhaps the most distinctive feature of contracts as they occur in human affairs is that any good contract entails obligations as well as benefits for both parties — with an obligation for one usually turning into a benefit for the other. This is true of contracts between classes, too:

- The precondition binds the client: it defines the conditions under which a call to the routine is legitimate. It is an *obligation* for the client and a *benefit* for the supplier.
- The postcondition binds the class: it defines the conditions that must be ensured by the routine on return. It is a benefit for the client and an obligation for the supplier.

The benefits are, for the client, the guarantee that certain properties will hold after the call; for the supplier, the guarantee that certain assumptions will be satisfied whenever the routine is called. The obligations are, for the client, to satisfy the requirements as stated by the precondition; for the supplier, to do the job as stated by the postcondition.

Here is the contract for one of the routines in our example:

put	**OBLIGATIONS**	**BENEFITS**
Client	(***Satisfy precondition***:) Only call *put* (*x*) on a non-full stack.	(***From postcondition***:) Get stack updated: not empty, *x* on top (*item* yields *x*, *count* increased by 1).
Supplier	(***Satisfy postcondition***:) Update stack representation to have *x* on top (*item* yields *x*), *count* increased by 1, not empty.	(***From precondition***:) Simpler processing thanks to the assumption that stack is not full.

A routine contract: routine put for a stack class

Zen and the art of software reliability: guaranteeing more by checking less

Although you may not have noticed it yet, one of the contract rules given goes against the generally accepted wisdom in software engineering; shocking at first to many, it is among the method's main contributions to software reliability and deserves emphasis.

The rule reflects the above observation that the precondition is a *benefit* for the supplier and is expressed in the bottom-right box of the table: if the client's part of the

contract is not fulfilled, that is to say if the call does not satisfy the precondition, then the class is not bound by the postcondition. In this case the routine may do what it pleases: return any value; loop indefinitely without returning a value; or even crash the execution in some wild way. This is the case in which (in reference to the discussion at the beginning of this chapter) "the customer is wrong".

The first advantage of this convention is that it considerably simplifies the programming style. Having specified as a precondition the constraints which calls to a routine must observe, you, the class developer, may assume when writing the routine body that the constraints are satisfied; you do not need to test for them in the body. So if a square root function, meant to produce a real number as a result, is of the form

```
sqrt (x: REAL): REAL is
        -- Square root of x
    require
        x >= 0
    do … end
```

you may write the algorithm for computing the square root without any concern for the case in which *x* is negative; this is taken care of by the precondition and becomes the responsibility of your clients. (At first sight this may appear dangerous; but read on.)

Actually the method of Design by Contract goes further. Writing the **do** clause of the routine under the form

```
if x < 0 then
    "Handle the error, somehow"
else
    "Proceed with normal square root computation"
end
```

is not just unnecessary but unacceptable. This may be expressed as a methodological rule:

Non-Redundancy principle

Under no circumstances shall the body of a routine ever test for the routine's precondition.

This rule is the reverse of what many software engineering or programming methodology textbooks advocate, often under the name *defensive programming* — the idea that to obtain reliable software you should design every component of a system so that it protects itself as much as possible. Better check too much, this approach holds, than not enough; one is never too careful when dealing with strangers. A redundant check might not help, but at least it will not hurt.

Design by Contract follows from the opposite observation: redundant checks can and indeed will hurt. Of course this will at first seem strange; the natural reaction is to think that an extra check — for example routine *sqrt* containing the above conditional instruction testing for $x < 0$ even though callers have been instructed to ensure $x >= 0$ —

may at worst be useless, but cannot possibly cause any damage. Such a comment, however, comes from a microscopic understanding of reliability, focused on individual software elements such as the *sqrt* routine. If we restrict our view to the narrow world of *sqrt*, then the routine seems more robust with the extra test than without it. But the world of a system is not restricted to a routine; it contains a multitude of routines in a multitude of classes. To obtain reliable systems we must go from the microscopic view to a macroscopic view encompassing the entire architecture.

If we take this global view, *simplicity* becomes a crucial criterion. As was noted at the beginning of this chapter, complexity is the major enemy of quality. When we bring in this concern, possibly redundant checks do not appear so harmless any more! Extrapolated to the thousands of routines of a medium-size system (or the tens or hundreds of thousands of routines of a larger one), the **if** *x < 0* **then** … of *sqrt*, innocuous at first sight, begins to look like a monster of useless complexity. By adding possibly redundant checks, you add more software; more software means more complexity, and in particular more sources of conditions that could go wrong; hence the need for more checks, meaning more software; and so on ad infinitum. If we start on this road only one thing is certain: we will *never* obtain reliability. The more we write, the more we will have to write.

To avoid this infinite chase we should never start it. With Design by Contract you are invited to identify the consistency conditions that are necessary to the proper functioning of each client-supplier cooperation (each contract); and to specify, for each one of these conditions, **whose responsibility it is** to enforce it: the client's, or the supplier's. The answer may vary, and is partly a matter of design style; advice will be given below on how best to choose it. But once you have made the decision, you should stick to it: if a correctness requirement appears in the precondition, indicating that the requirement is part of the client's responsibility, there must not be a corresponding test in the routine; and if it is not in the precondition, then the routine must check for the requirement.

Defensive programming appears in contrast to cover up for the lack of a systematic approach by blindly putting in as many checks as possible, furthering the problem of reliability rather than addressing it seriously.

> Redundant checking, it should be noted, is a standard technique in hardware. The difference is that in a hardware system some object that was found to be in a correct state at some point may later have its integrity destroyed because of reasons beyond the control of the system itself, such as interference from another system, harmful external event, or simply wear and tear. For that reason it is normal practice, for example, to have both the sender and the receiver of an electronic signal check its integrity.
>
> But no such phenomenon occurs in software: if I can prove or check in some way that *a* is non-negative whenever *sqrt* (*a*) is called, I do not need to insert a check for *x* Š≥ *0*, where *x* is the corresponding formal argument, in the body of *sqrt*. Nothing will happen to *a* between the time it is "sent" by the caller and the time it is "received" (under the name *x*) by the routine. Software does not wear out when used for too long; it is not subject to line loss, to interference or to noise.
>
> Also note that in most cases what is called redundant checking in hardware is not really redundant: one actually applies *different* and complementary verifications, such as a parity check and some other test. Even when the checks are the same they are often

applied by different devices, as in the just mentioned case of a sender and receiver that both check a signal, or in a redundant computer system where several computers perform the same computation, with a voting mechanism to resolve discrepancies.

Another drawback of defensive programming is its costs. Redundant checks imply a performance penalty — often enough in practice to make developers wary of defensive programming regardless of what the textbooks say. If they do make the effort to include these checks, removing some of them later to improve performance will be tedious. The techniques of this chapter will also leave room for extra checks, but if you choose to enable them you will rely on the development environment to carry them out for you. To remove them, once the software has been debugged, it suffices to change a compilation option (details soon). The software itself does not contain any redundant elements.

Aside from performance considerations, however, the principal reason to distrust defensive programming is simply our goal of getting the best possible reliability. For a system of any significant size the individual quality of the various elements involved is not enough; what will count most is the guarantee that for every interaction between two elements there is an explicit roster of mutual obligations and benefits — the contract. Hence the Zen-style paradox of our conclusion: that to get *more* reliability the best policy is often to check *less*.

Assertions are not an input checking mechanism

It is useful here to emphasize a few properties of the approach which, although implicit in the preceding discussion, have been shown by experience to require further explanations. The following comments should help address some of the questions that may have been forming in your mind as you were reading about the basic ideas of Design by Contract.

To avoid a common misunderstanding, make sure to note that each of the contracts discussed holds between a routine (the supplier) and another routine (its caller): we are concerned about software-to-software communication, not software-to-human or software-to-outside-world. A precondition will not take care of correcting user input, for example in a *read_positive_integer* routine that expects the interactive user to enter a positive number. Including in the routine a precondition of the form

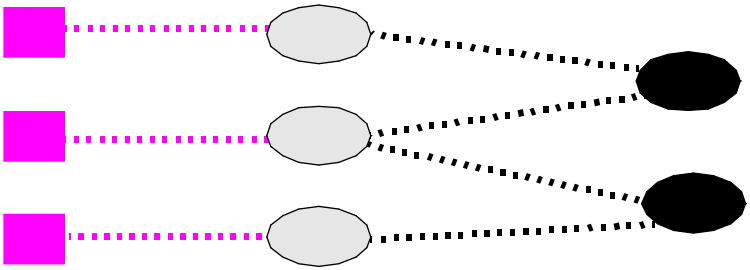
```
require
    input > 0
```

would be wishful thinking, not a reliability technique. Here there is no substitute for the usual condition-checking constructs, including the venerable **if** … **then** …; the exception handling mechanism studied in the next chapter may also be helpful.

"Modular protection", page 45.

Assertions do have a role to play in a solution to this problem of input validation. In line with the criterion of Modular Protection, the method encourages validating any objects obtained from the outside world — from sensors, from user input, from a network… — as close to the source of the objects as possible, using "filter" modules if necessary:

Using filter modules



External objects Input and validation modules Processing modules

In obtaining information from the outside (communication paths shown in color) you cannot rely on preconditions. But part of the task of the input modules shown in grey in the middle of the figure is to guarantee that no information is passed further to the right — to the modules responsible for the system's actual computations — unless it satisfies the conditions required for correct processing. In this approach there will be ample use of assertions in the software-to-software communication paths represented by the black dotted lines on the right. The postconditions achieved by the routines of the input modules will have to match (or exceed, in the sense of "stronger" defined earlier) the preconditions imposed by the processing routines.

> The routines of the filter classes may be compared to security officers in, say, a large government laboratory. To meet experts from the laboratory and ask them technical questions, you must submit to screening procedures. But it is not the same person who checks your authorization level and answers the questions. The physicists, once you have been officially brought into their offices, assume you satisfy the preconditions; and you will not get much help from the guards on theoretical physics.

Assertions are not control structures

Another common misunderstanding, related to the one just discussed, is to think of assertions as control structures — as techniques to handle special cases. It should be clear by now that this is not their role. If you want to write a routine *sqrt* that will handle negative arguments a certain way, and non-negative arguments another way, a **require** clause is not what you need. Conditional instructions (**if** … **then** … **else** …) and related constructs to deal with various cases (such as Pascal's **case** … **of** … or the **inspect** instruction of this book's notation) are perfectly appropriate for such purposes.

Assertions are something else. They express correctness conditions. If *sqrt* has its precondition, a call for which $x < 0$ is not a special case: it is a bug, plain and simple.

Assertion Violation rule (1)

A run-time assertion violation is the manifestation of a bug in the software.

"Bug" is not a very scientific word but is clear enough to anyone in software; we will look for more precise terminology in the next section. For the moment we can pursue the assertion violation rule further by noting a consequence of the contract view:

Assertion violation rule (2)

A precondition violation is the manifestation of a bug in the client.

A postcondition violation is the manifestation of a bug in the supplier.

A precondition violation means that the routine's caller, although obligated by the contract to satisfy a certain requirement, did not. This is a bug in the client itself; the routine is not involved. ("The customer is wrong".) An outside observer might of course criticize the contract as too demanding, as with the unsatisfiable **require** *False* precondition or our fictitious *Sinecure 1* example ("the customer is *always* wrong"), but this is too late to argue over the contract: it is the contract, and the client did not observe its part of the deal. So if there is a mechanism for monitoring assertions during execution — as will be introduced shortly — and it detects such a precondition violation, the routine should not be executed at all. It has stated the conditions under which it can operate, and these conditions do not hold; trying to execute it would make no sense.

A postcondition violation means that the routine, presumably called under correct conditions, was not able to fulfill its contract. Here too the distribution of guilt and innocence is clear, although it is the reverse of the previous one: the bug is in the routine; the caller is innocent.

Errors, defects and other creeping creatures

The appearance of the word "bug" in the preceding analysis of assertion violation causes is a good opportunity to clarify the terminology. In Edsger W. Dijkstra's view, using the word "bug" is a lame attempt by software people to blame someone else by implying that mistakes somehow creep into the software from the outside while the developers are looking elsewhere — as if were not the developers who made the mistakes in the first place.

Yet the term enjoys enduring success, if only because it is colorful and readily understood. Like the rest of the software literature, this book uses it freely. But it is appropriate to complement it by more specific (if more stodgy) terms for cases in which we need precise distinctions.

Terms to denote software woes

An *error* is a wrong decision made during the development of a software system.

A *defect* is a property of a software system that may cause the system to depart from its intended behavior.

A *fault* is the event of a software system departing from its intended behavior during one of its executions.

The causal relation is clear: faults are due to defects, which result from errors.

"Bug" usually has the meaning of defect ("are you sure there remains no other bug in this routine?"). This is the interpretation in this book. But in informal discussions it is also used in the sense of fault ("We have had bug-free operation for the last three weeks") or error ("the bug was that I used an unsorted list").

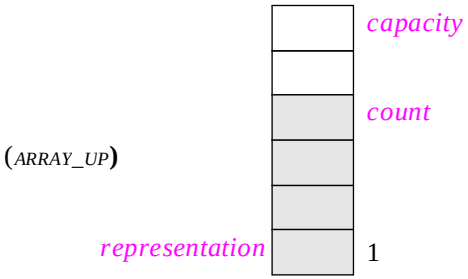
11.7 WORKING WITH ASSERTIONS

Let us now probe further the use of preconditions and postconditions, continuing with fairly elementary examples. Assertions, some simple, some elaborate, will be pervasive in the examples of the following chapters.

A stack class

The assertion-equipped *STACK* class was left in a sketchy form (*STACK1*). We can now come up with a full version including a spelled out implementation.

For an effective (directly usable) class we must choose an implementation. Let us use the array implementation illustrated at the beginning of the discussion of abstract data types:



"Push" operation:
count := *count* + *1*
representation [*count*] := *x*

Stack implemented with an array (see page 123 for other representations)

The array will be called *representation* and will have bounds 1 and *capacity*; the implementation also uses an integer, the attribute *count*, to mark the top of the stack.

Note that as we discover inheritance we will see how to write deferred classes that cover several possible implementations rather than just one. Even for a class that uses a particular implementation, for example by arrays as here, we will be able to *inherit* from the implementation class *ARRAY* rather than use it as a client (although some object-oriented developers will still prefer the client approach). For the moment, however, we can do without any inheritance-related technique.

For an array-based stack implementation using inheritance, see "IMPLEMENTATION INHERITANCE", 24.8, page 844.

Here is the class. Recall that if *a* is an array then the operation to assign value *x* to its *i*-th element is *a*.*put* (*x*, *i*), and the value of its *i*-th element is given by *a*.*item* (*i*) or, equivalently, *a* @ *i*. If, as here, the bounds of the array are 1 and *capacity*, then *i* must in all cases lie between these bounds.

```
indexing
    description: "Stacks: Dispenser structures with a Last-In, First-Out %
                 %access policy, and a fixed maximum capacity"
class STACK2 [G] creation
    make
feature -- Initialization
    make (n: INTEGER) is
            -- Allocate stack for a maximum of n elements
        require
            positive_capacity: n >= 0
        do
            capacity := n
            !! representation.make (1, capacity)
        ensure
            capacity_set: capacity = n
            array_allocated: representation /= Void
            stack_empty: empty
        end
feature -- Access
    capacity: INTEGER
            -- Maximum number of stack elements
    count: INTEGER
            -- Number of stack elements
    item: G is
            -- Top element
        require
            not_empty: not empty -- i.e. count > 0
        do
            Result := representation @ count
        end
feature -- Status report
    empty: BOOLEAN is
            -- Is stack empty?
        do
            Result := (count = 0)
        ensure
            empty_definition: Result = (count = 0)
        end
```

On the export status of capacity see exercise E11.4, page 410.

```
        full: BOOLEAN is
                -- Is stack full?
            do
                Result := (count = capacity)
            ensure
                full_definition: Result = (count = capacity)
            end
    feature -- Element change
        put (x: G) is
                -- Add x on top
            require
                not_full: not full        -- i.e. count < capacity in this representation
            do
                count := count + 1
                representation.put (count, x)
            ensure
                not_empty: not empty
                added_to_top: item = x
                one_more_item: count = old count + 1
                in_top_array_entry: representation @ count = x
            end
        remove is
                -- Remove top element
            require
                not_empty: not empty -- i.e. count > 0
            do
                count := count – 1
            ensure
                not_full: not full
                one_fewer: count = old count – 1
            end
    feature {NONE} -- Implementation
        representation: ARRAY [G]
                -- The array used to hold the stack elements
    invariant
        … To be filled in later (see page 365) …
    end -- class STACK2
```

This class text illustrates the simplicity of working with assertions. It is complete except for the **invariant** clause, which will be added later in this chapter. Let us explore its various properties.

Invariants are introduced in "CLASS INVARIANTS", 11.8, page 364

This is the first full-fledged class of this chapter, not too far from what you will find in professional libraries of reusable object-oriented components such as the Base libraries. (Apart from the use of inheritance and a few extra features, what still distinguishes this class from its real-life counterparts is the absence of the **invariant** clause.)

On multiple feature clauses and exporting to NONE see "SELECTIVE EXPORTS AND INFORMATION HIDING", 7.8, page 191.

Before studying the assertions, a general note about the structure of the class. As soon as a class has more than two or three features, it becomes essential to organize its features in a coherent way. The notation helps by providing the possibility of including multiple **feature** clauses. An earlier chapter introduced this facility as a way to specify a different export status for certain features, as done here for the last part of the class, labeled -- Implementation to specify that feature *representation* is secret. But as already previewed in *STACK1* you can take advantage of multiple feature clauses even when the export status is the same. The purpose is to make the class easier to read, and easier to manage, by grouping features into general categories. After each **feature** keyword appears a comment (known as the Feature Clause Comment) defining the general role of the features that follow. The categories used in the example are those of *STACK1*, plus *Initialization* for the creation procedure.

"Feature clause header comments", page 889.

The standard feature categories and associated Feature Clause Comments are part of the general rules for consistency and organization of reusable library classes. A more complete list appears in the chapter on style rules.

The imperative and the applicative

"Introducing a more imperative view", page 145.

The assertions of *STACK2* illustrate a fundamental concept of which we got a first glimpse when we studied the transition from abstract data types to classes: the difference between imperative and applicative views.

The assertions in *empty* and *full* may have caused you to raise an eyebrow. Here again is the text of *full*:

```
full: BOOLEAN is
            -- Is stack full?
        do
            Result := (count = capacity)
        ensure
            full_definition: Result = (count = capacity)
        end
```

The postcondition expresses that *Result* has the same value as *count = capacity*. (Since both sides of the equality, the entity *Result* and the expression *count = capacity*, are boolean, this means that the function returns **true** if and only if *count* is equal to *capacity*.) But what is the point of writing this postcondition, one may ask, since the body of the routine (the **do** clause) says exactly the same thing through the instruction *Result* := (*count = capacity*), whose only difference with the postcondition clause is its use of := rather than =? Is the postcondition not redundant?

Actually, there is a big difference between the two constructs, and no redundancy at all. The instruction *Result* := (*count* = *capacity*) is a command that we give to our virtual computer (the hardware-software machine) to change its state in a certain way; it performs an action. The assertion *Result* = (*count* = *capacity*) does not do anything: it specifies a property of the expected end state, as visible to the routine's caller.

The instruction is *prescriptive*; the assertion is *descriptive*. The instruction describes the "how"; the assertion describes the "what". The instruction is part of the implementation; the assertion is an element of specification.

The instruction is *imperative*; the assertion is *applicative*. These two terms emphasize the fundamental difference between the worlds of computing and mathematics:

- Computer operations may change the state of the hardware-software machine. Instructions of common programming languages are commands (imperative constructs) directing the machine to execute such operations.
- Mathematical reasoning never changes anything; as noted in the presentation of abstract data types, taking the square root of the number 2 does not change that number. Mathematics instead describes how to use properties of known objects, such as the number 2, to infer properties of others, such as $\sqrt{2}$, obtained from the former by *applying* (hence the name) certain mathematical derivations such as square root.

That the two notations are so close in our example — assignment := and equality = — should not obscure this fundamental difference. The assertion describes an intended result, and the instruction (the loop body) prescribes a particular way to achieve that result. Someone using the class to write a client module will typically be interested in the assertion but not in the implementation.

The reason for the closeness of notations for assignment and equality is that assignment is indeed in many cases the straightforward way to achieve equality; in our example the chosen implementation, *Result* := (*count* = *capacity*), is indeed the obvious one. But as soon as we move on to more advanced examples the conceptual difference between the specification and the implementation will be much larger; even in the simple case of a function to compute the square root of a real number *x*, where the postcondition is just something like *abs* (*Result* ^ 2 – *x*) <= *tolerance* with *abs* denoting absolute value and *tolerance* a tolerance value, the instructions in the function's body will be far less trivial since they have to implement a general algorithm for the computation of square roots.

Even for *put* in class *STACK2*, the same specification could have led to different implementations, although the differences are minor; for example the body could be

if *count* = *capacity* **then** *Result* := *True* **else** *Result* := *False* **end**

perhaps simplified (thanks to the rules of default initialization) into

if *count* = *capacity* **then** *Result* := *True* **end**

So the presence of related elements in the body and the postcondition is not evidence of redundancy; it is evidence of consistency between the implementation and the specification — that is to say, of correctness as defined at the beginning of this chapter.

In passing, we have encountered a property of assertions that will merit further development: their relevance for authors of client classes, whom we should not ask to read routine implementations, but who need a more abstract description of the routine's role. This idea will lead to the notion of **short form** discussed later in this chapter as the basic class documentation mechanism.

See "Including functions in assertions", page 401.

A caveat: for practical reasons we will allow assertions to include some seemingly imperative elements (functions). This issue will be explored at the end of this chapter.

As a summary of this discussion it is useful to list the words that have been used to contrast the two categories of software elements:

The imperative-applicative opposition

Implementation	Specification
Instruction	Expression
How	What
Imperative	Applicative
Prescription	Description

A note on empty structures

The precondition of the creation procedure *make* in class *STACK1* requires a comment. It states *n >= 0*, hence allowing empty stacks. If *n* is zero, *make* will call the creation procedure for arrays, also named *make*, with arguments *1* and *0* for the lower and upper bounds respectively. This is not an error, but follows from a convention regarding *ARRAY*'s creation procedure: using a first argument greater than the second by one creates an empty array.

A zero *n* for a stack, or a first creation argument greater than the second for an array, is not wrong but simply means that this particular stack or array should be empty. An error would only occur out of a call attempting to access an element from the structure, for example a *put* for the stack or an *item* for the array, both of whose preconditions will always be false for an empty structure ("my customer is always wrong").

When you define a general data structure such as a stack or array, you should determine whether the case of an empty structure is conceptually meaningful. In some cases it is not: for example most definitions of the notion of *tree* start from the assumption that there is at least one node, the root. But if the empty case raises no logical impossibility, as with arrays and stacks, you should plan for it in the design of your data structure, acknowledging that clients will, every once in a while, create empty instances, and should not suffer for it. An application system may for example need a stack for *n* elements, where *n* is an upper bound on the number of elements to be stacked, computed by the application just before it creates the stack; in some runs that number may be zero. This is not an error, simply an extreme case.

> The array mechanism of Algol W provides a counter-example. When a dynamically allocated array has an empty range, the program terminates in error — even if it was a perfectly valid array which simply happened to be empty on that particular run. This is too restrictive: an array with zero size is valid, it simply does not allow access to any element.

Precondition design: tolerant or demanding?

Central to Design by Contract is the idea, expressed as the Non-Redundancy principle, that for any consistency condition that could jeopardize a routine's proper functioning you should assign enforcement of this condition to only one of the two partners in the contract.

The principle was on page 344.

Which one? In each case you have two possibilities:

- Either you assign the responsibility to clients, in which case the condition will appear as part of the routine's precondition.
- Or you appoint the supplier, in which case the condition will appear in a conditional instruction of the form **if** *condition* **then** …, or an equivalent control structure, in the routine's body.

We can call the first attitude *demanding* and the second one *tolerant*. The *STACK2* class illustrates the demanding style; a tolerant version of the class would have routines with no preconditions, such as

Warning: not the recommended style.

```
remove is
            -- Remove top element
        do
            if empty then
                print ("Error: attempt to pop an empty stack")
            else
                count := count – 1
            end
        end
```

In the analogy with human contracts we can think of the demanding style as characterizing an experienced contractor who expects his clients to "do their homework" before calling on him; he has no trouble finding business, and will reject requests that appear too broad or unreasonable. The tolerant style evokes the image of a freshly established consulting practice, whose owner is so desperate for business that he will take anything, having put in his driveway a big sign:

Which is the better style? To a certain extent this is a matter of personal choice (as opposed to the Non-Redundancy principle, which was absolute in stating that it is *never* acceptable to deal with a correctness condition on both the client and supplier sides). A strong case can be made, however, for the demanding style illustrated by *STACK2*, especially in the case of software meant to be reusable — and in O-O development we should always write our software with the goal of ultimately making it reusable.

At first the tolerant style might appear better for both reusability and reliability; after all the demanding approach appears to put more responsibility on the clients, and there are typically many clients for a single supplier — even more so for a reusable class. Is it not preferable, then, to let the supplier take care of the correctness conditions once and for all, rather than require every client to do it for itself?

If we look more closely at the issue this reasoning does not hold. The correctness conditions describe what the routine requires to be able to do its job properly. The tolerant *remove* on the facing page is a good counter-example: what can a poor stack-popping routine do for an empty stack? It makes a brave attempt by outputting an error message, but this is clearly inadequate: a specialized utility module such as a stack handler has no business messing up the system's user output. We could try something more sophisticated, but *remove* simply does not have the proper context; the focus of class *STACK2* is too narrow to determine what to do in the case of an empty stack. **Only the client** — a module using stacks in some application, for example the parsing module in a compiler — has enough information to decide what an attempt to pop an empty stack really means: is it a normal although useless request that we should simply ignore, executing a null operation? Or is it an error, and if so, how should we handle it: raise an exception, correct the situation before trying again, or (the least likely answer) output a user-visible error message?

In the square root example, you may remember the fictitious routine text quoted in the discussion preceding the Non-Redundancy principle:

Page 344.

```
if x < 0 then
        "Handle the error, somehow"
else
        "Proceed with normal square root computation"
end
```

The operative word is "somehow". The **then** clause is incantation more than software: there is really no good general-purpose technique for handling the $x < 0$ case. Here again a general-purpose routine has no clue. Only the client author can know what the call means in this case — an error in the software, a case in which the expected result is 0, a reason to trigger an exception…

In this case as in the attempt at a tolerant *remove*, the position of the routine is not unlike that of a postman asked to deliver a postcard with no delivery address and no return address: the case falls outside of the contract, and there is no good way to decide what to do.

In the spirit of Design by Contract, the demanding approach to precondition design does not attempt to produce routines that are all things to all clients. Instead, it insists that each routine do a well-defined job and do it well (correctly, efficiently, generally enough to be reusable by many clients…), and specify clearly what cases it cannot handle. In fact you cannot hope that the routine will do its job well *unless* you have carefully circumscribed that job. A factotum routine, which wants to do a computation and check for abnormal cases and take corrective actions and notify the client and produce a result anyway, will most likely fail to fulfill any of these goals properly.

The routine author does not try to outsmart his clients; if he is not sure of what the routine is supposed to do in a certain abnormal situation, he excludes it explicitly through the precondition. This attitude is more generally a consequence of the overall theme in this book: building software systems as sets of modules that mind their own business.

If you read the supplementary mathematical section in the chapter on abstract data types, you may have noted the similarity between the present discussion and the arguments for using partial functions in the mathematical model, rather than special error values such as $\omega_{INTEGER}$. The two ideas are indeed very close, and Design by Contract is in part the application to software construction of the concept of partial function, so remarkably flexible and powerful in formal specification.

"Alternatives to partial functions", page 151.

A word of caution: the demanding approach is only applicable if the preconditions remain reasonable. Otherwise the job of writing a module would become easy: start every routine with **require** *False* so that, as we have seen, any routine body will be correct. What does "reasonable" concretely mean for the precondition of a routine? Here is a more precise characterization:

Reasonable Precondition principle

Every routine precondition (in a "demanding" design approach) must satisfy the following requirements:

- The precondition appears in the official documentation distributed to authors of client modules.
- It is possible to justify the need for the precondition in terms of the specification only.

The first requirement will be supported by the notion of short form studied later in this chapter. The second requirement excludes restrictions meant only for the supplier's convenience in implementing the routine. For example when you want to pop a stack the precondition **not** *empty* is a logical requirement that can be justified "in terms of the specification only", through the simple observation that in an empty stack there is nothing to pop; and when you want to compute the real square root of a number, the precondition *x >= 0* is a direct result of the mathematical property that negative real numbers do not have real square roots.

The general stack ADT was studied in chapter 6; the bounded stack ADT was the subject of exercise E6.9, page 162.

Some restrictions may arise from the general kind of implementation selected. For example the presence of **require not** *full* as precondition to the push operation *put* in *STACK2* is due to the decision of using an array for the implementation of stacks. But such a case does not violate the principle, as the bounded nature of *STACK2* stacks has been made part of the specification: the class does not claim to represent arbitrary stacks, but only stacks of finite maximum capacity (as expressed for example in the **indexing** clause of the class). The abstract data type serving as specification of this class is not the most general notion of stack, but the notion of bounded stack.

> In general, it is desirable to avoid bounded structures; even a stack implemented by arrays can use array resizing. This is the case with the most commonly used stack class in the Base libraries, which follows the *STACK2* style but without a notion of *capacity*; a stack that overflows its current capacity resizes itself silently to accommodate the new elements.

Preconditions and export status

You may have noted the need for a supplementary requirement on preconditions, which does not figure in the Reasonable Precondition principle: to be satisfiable by the clients, the precondition must not use features that are hidden from the clients as a result of export restrictions.

Assume for example the following situation:

```
        -- Warning: this is an invalid class, for purposes of illustration only.
class SNEAKY feature

    tricky is
        require
            accredited
        do
            …
        end

feature {NONE}

    accredited: BOOLEAN is do … end

end -- class SNEAKY
```

The specification for *tricky* states that any call to that procedure must satisfy the condition expressed by the boolean function *accredited*. But whereas the class exports *tricky* to all clients, it keeps *accredited* secret, so that clients have no way of finding out,

before a call, whether the call is indeed correct. This clearly unacceptable situation is akin, in human contracts, to a deal in which the supplier would impose some conditions not stated explicitly in the contract, and hence could reject a client's request as incorrect without giving the client any way to determine in advance whether it is correct.

The reason why the Reasonable Precondition principle does not cover such cases is that here a methodological principle does not suffice: we need a language rule to be enforced by compilers, not left to the decision of developers.

The rule must take into account all possible export situations, not just those illustrated above in which a feature is available to all clients (*tricky*) or to no client (*accredited*). As you will recall from the discussion of information hiding, it is also possible to make a feature available to some clients only, by declaring it in a feature clause appearing as **feature** {*A, B, …*}, which makes it available only to *A*, *B*, … and their descendants. Hence the language rule:

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

Precondition Availability rule

Every feature appearing in the precondition of a routine must be available to every client to which the routine is available.

With this rule every client that is in a position to call the feature will also be in a position to check for its precondition. The rule makes class *SNEAKY* invalid, since *tricky* is generally exported (available to all clients); you can turn it into a valid class by making *accredited* also generally exported. If *tricky* had appeared in a feature clause starting with **feature** {*A, B, C*}, then *accredited* would have to be exported at least to *A*, *B* and *C* (by appearing in the same feature clause as *tricky*, or by appearing in a clause of the form **feature** {*A, B, C*}, or **feature** {*A, B, C, D, …*}, or just **feature**). Any violation of this rule is a compile-time error. Class *SNEAKY*, for example, will be rejected by the compiler.

There is no such rule for postconditions. It is not an error for some clauses of a postcondition clause to refer to secret features, or features that are not as broadly exported as the enclosing routine; this simply means that you are expressing properties of the routine's effect that are not directly usable by clients. This was the case with the *put* procedure in *STACK2*, which had the form

```
put (x: G) is
            -- Add x on top
        require
            not full
        do
            …
        ensure
            … Other clauses …
            in_top_array_entry: representation @ count = x
        end
```

For the Other clauses *see page 351.*

The last postcondition clause indicates that the array entry at index *count* contains the element just pushed. This is an implementation property; even though *put* is generally available (exported to all clients), array *representation* is secret. But there is nothing wrong with the postcondition; it simply includes, along with properties that are directly useful to clients (the "Other clauses"), one that is only meaningful for someone who reads the entire class text. Such secret clauses will not appear in the "short" form of the class — the documentation for client authors.

A tolerant module

(On first reading you may skip this section or just look through it quickly.)

For filters of the first kind see "Assertions are not an input checking mechanism", page 346.

The simple but unprotected basic modules may not be robust enough for use by arbitrary clients. In some cases there will be a need for new classes to serve as filters, interposed not between the software and the external world (as with filters of the kind discussed earlier in this chapter) but between software and other software: possibly careless clients on one side, unprotected classes on the other.

Although we have seen that this is generally not the right approach, it is useful to examine how classes will look if we do decide to use the tolerant style in a specific case. Class *STACK3*, appearing next, illustrates the idea. Because the class needs to set integer error codes, it is convenient to rely on a property of the notation that has not been introduced yet: "unique" integer constants. If you declare a set of attributes as

a, b, c, …: *INTEGER* **is unique**

"UNIQUE VALUES", 18.6, page 654.

the effect is to define *a*, *b*, *c* … as integer constants with consecutive positive values. These values will be assigned by the compiler, and are guaranteed to be different for all constants thus declared, relieving you of having to invent separate codes. By convention, constant attributes such as these have names beginning with an upper-case letter, with the rest in lower case, as in *Underflow*.

Here, using this technique, is a tolerant version of our earlier stack class. Make sure to note that this class text (which you may just skim through on first reading) is included here only to make sure you understand the tolerant style; it is **not** an example of the generally recommended design — for reasons that will be discussed below, but will probably be clear enough as you browse through the text.

```
indexing
    description: "Stacks: Dispenser structures with a Last-In, First-Out %
                 %access policy, and a fixed maximum capacity; %
                 %tolerant version, setting an error code in case %
                 %of impossible operations."
class STACK3 [G] creation
    make
```

```
feature -- Initialization
    make (n: INTEGER) is
            -- Allocate stack for a maximum of n elements if n > 0;
            -- otherwise set error to Negative_size.
            -- No precondition!
        do
            if capacity >= 0 then
                capacity := n
                !! representation .make (capacity)
            else
                error := Negative_size
            end
        ensure
            error_code_if_impossible: (n < 0) = (error = Negative_size)
            no_error_if_possible: (n >= 0) = (error = 0)
            capacity_set_if_no_error: (error = 0) implies (capacity = n)
            allocated_if_no_error: (error = 0) implies (representation /= Void)
        end
feature -- Access
    item: G is
            -- Top element if present; otherwise the type's default value.
            -- with error set to Underflow.
            -- No precondition!
        do
            if not empty then
                                check representation /= Void end
                Result := representation.item
                error := 0
            else
                error := Underflow
                    -- In this case the result is the default value
            end
        ensure
            error_code_if_impossible: (old empty) = (error = Underflow)
            no_error_if_possible: (not (old empty)) = (error = 0)
        end
feature -- Status report
    empty: BOOLEAN is
            -- Number of stack elements
        do
            Result := (capacity = 0) or else representation.empty
        end
```

```
        error: INTEGER
            -- Error indicator, set by various features to a non-zero value
            -- if they cannot do their job
        full: BOOLEAN is
                -- Number of stack elements
            do
                Result := (capacity = 0) or else representation.full
            end
        Overflow, Underflow, Negative_size: INTEGER is unique
            -- Possible error codes
    feature -- Element change
        put (x: G) is
                -- Add x on top if possible; otherwise set error code.
                -- No precondition!
            do
                if full then
                    error := Overflow
                else
                                        check representation /= Void end
                    representation.put (x); error := 0
                end
            ensure
                error_code_if_impossible: (old full) = (error = Overflow)
                no_error_if_possible: (not old full) = (error = 0)
                not_empty_if_no_error: (error = 0) implies not empty
                added_to_top_if_no_error: (error = 0) implies item = x
                one_more_item_if_no_error: (error = 0) implies count = old count + 1
            end
        remove is
                -- Remove top element if possible; otherwise set error.
                -- No precondition!
            do
                if empty then
                    error := Underflow
                else
                                        check representation /= Void end
                    representation.remove
                    error := 0
                end
            ensure
                error_code_if_impossible: (old empty) = (error = Underflow)
                no_error_if_possible: (not old empty) = (error = 0)
                not_full_if_no_error: (error = 0) implies not full
                one_fewer_item_if_no_error: (error = 0) implies count = old count – 1
            end
```

```
feature {NONE} -- Implementation
        representation: STACK2 [G]
                -- The unprotected stack used as implementation
        capacity: INTEGER
                -- The maximum number of stack elements
end -- class STACK3
```

The operations of this class have no preconditions (or, more accurately, have *True* as their preconditions). For those that may result in abnormal situations, the postcondition has been refined to distinguish between correct and erroneous processing. An operation such as *s.remove*, where *s* is a *STACK3*, will set *s.error* to 0 or to *Underflow* (which, from the rules on unique values, is known to be strictly positive) and, in the latter case, do nothing else. It is still the caller's responsibility to check for *s.error* after the call. As noted, a general-purpose module such as *STACK3* has no way to decide what to do in the case of an erroneous popping attempt: produce an error message, take corrective action…

Such filter modules serve to separate algorithmic techniques to deal with normal cases and techniques for handling errors. This is the distinction between correctness and robustness explained at the beginning of this book: writing a module that performs correctly in legal cases is one task; making sure that other cases are also processed decently is another. Both are necessary, but they should be handled separately. Failure to do so is one of the principal reasons why so many software systems are hopelessly complex: any algorithm that does anything useful also takes care of checking that it is applicable, and for good measure tries to handle the cases in which it is not. Such software soon mushrooms into a total mess.

See "A REVIEW OF EXTERNAL FACTORS", 1.2, page 4.

A few technical comments apply to this example:

- An instance of *STACK3* is not an array but a structure containing a reference (*representation*) to an instance of *STACK2*, itself containing a reference to an array. These two indirections, detrimental to efficiency, can be avoided through inheritance as studied in later chapters.
- The boolean operator **or else** is similar to **or** but ignores the second operand if it does not affect the result and trying to evaluate it could cause trouble.
- The **check** instruction used in *put* and *remove* serves to state that a certain assertion is satisfied. It will be studied later in this chapter.

On **or else** *see "Non-strict boolean operators", page 454.*

Finally, you will have noted the heaviness of *STACK3*, especially if you compare it to the simplicity that *STACK2* achieves with its precondition. *STACK3* is good evidence that a tolerant style may lead to uselessly complex software. The demanding style, in contrast, follows from the general spirit of Design by Contract. Trying to handle all possible (and impossible) cases is not necessarily the best way to help your clients. If instead you build classes that impose possibly strict but reasonable usage conditions, and describe these conditions precisely as part of the official documentation for the class, you actually make life easier for the clients. This has been called the **tough love** approach: you can often serve your clients better by being more restrictive.

Better an efficient supplier that states its functionally justified limitations than a overzealous one that tries to second-guess its clients, making possibly inappropriate decisions for abnormal cases, and sacrificing simplicity and efficiency.

For modules whose clients are other software modules, the demanding approach is usually the right one. A possible exception is the case of modules intended for clients whose authors use a non-O-O language and may not have understood the basic concepts of Design by Contract.

"Assertions are not an input checking mechanism", page 346.

The tolerant approach remains useful for software elements that deal not with other software elements but with data coming from the outside world, such as user input, or sensor data. Then, as noted earlier, filter modules are often necessary to separate the actual processing modules (the physicists in our metaphor) from those which simply qualify data and reject anything that is not appropriate (the guards). This separation of concerns is essential for maintaining the simplicity of software elements on both sides. *STACK3* provides an idea of what such modules may look like.

11.8 CLASS INVARIANTS

Preconditions and postconditions describe the properties of individual routines. There is also a need for expressing global properties of the instances of a class, which must be preserved by all routines. Such properties will make up the class invariant, capturing the deeper semantic properties and integrity constraints characterizing a class.

Definition and example

Consider again the earlier implementation of stacks by arrays, the one without the protections (*STACK2*):

Page 350.

```
class STACK2 [G] creation
        make
feature
        … make, empty, full, item, put, remove …
        capacity: INTEGER
        count: INTEGER
feature {NONE} -- Implementation
        representation: ARRAY [G]
end
```

The attributes of the class — array *representation* and integers *capacity* and *count* — constitute the stack representation. Although routine preconditions and postconditions, given earlier, express some of the semantic properties of stacks, they fail to express other important consistency properties linking the attributes. For example, *count* should always remain between 0 and *capacity*:

```
0 <= count; count <= capacity
```

(implying also that *capacity >= 0*), and *capacity* should be the array size:

capacity = representation.capacity

A class invariant is such an assertion, expressing general consistency constraints that apply to every class instance as a whole; this is different from preconditions and postconditions, which characterize individual routines.

The above assertions involve only attributes. Invariants may also express the semantic relations between functions, or between functions and attributes. For example the invariant for *STACK2* may include the following property describing the connection between *empty* and *count*:

empty = (count = 0)

In this example, the invariant assertion links an attribute and a function; it is not particularly interesting as it merely repeats an assertion that appears in the postcondition of the function (here *empty*). More useful assertions are those which involve either only attributes, as above, or more than one function.

Here is another typical example. Assume — in line with previous examples dealing with the notion of bank account — that we have a class *BANK_ACCOUNT* with features *deposits_list*, *withdrawals_list* and *balance*. Then the invariant for such a class could include a clause of the form:

This example was first discussed in "Uniform Access", page 55. It will serve again to illustrate persistence issues: "Correction", page 1045.

consistent_balance: *deposits_list.total – withdrawals_list.total = balance*

where the function *total* gives the cumulated value of a list of operations (deposits or withdrawals). This states the basic consistency condition between the values accessible through features *deposits_list*, *withdrawals_list* and *balance*.

Form and properties of class invariants

Syntactically, a class invariant is an assertion, appearing in the **invariant** clause of the class, after the features and just before the **end**, as in

For the features of STACK2 see page 350.

```
class STACK4 [G] creation
        … As in STACK2 …
feature
        … As in STACK2 …
invariant
        count_non_negative: 0 <= count
        count_bounded: count <= capacity
        consistent_with_array_size: capacity = representation.capacity
        empty_if_no_elements: empty = (count = 0)
        item_at_top: (count > 0) implies (representation.item (count) = item)
end
```

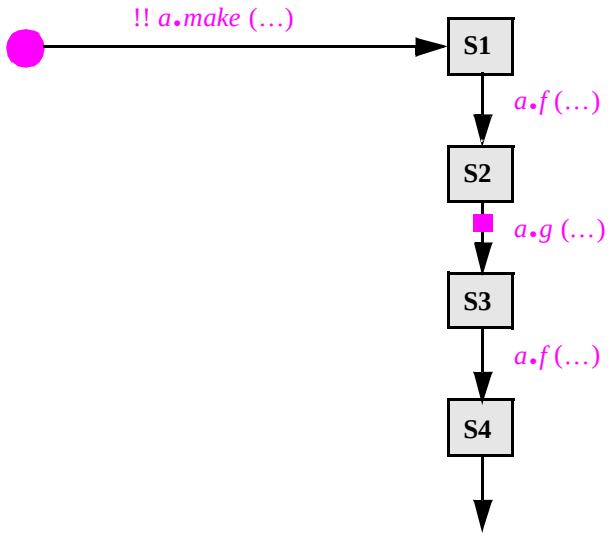
An invariant for a class *C* is a set of assertions that every instance of *C* will satisfy at all "stable" times. Stable times are those in which the instance is in an observable state:

- On instance creation, that is to say after execution of !! *a* or !! *a*.*make* (…), where *a* is of type *C*.

- Before and after every remote call *a*.*r* (…) to a routine *r* of the class.

The following figure, showing the life of an object, helps put the notions of invariant and stable time in place.

The life of an object



Life as an object, to tell the truth, is not that thrilling (in case you ever wondered). At the beginning — left of the figure — you do not exist. You are begot by a creation instruction !! *a* or !! *a*.*make* (…), or a *clone*, and reach your first station in life. Then things get quite boring: through some reference *a*, clients use you, one after the other, by applying operations of the form *a*.*f* (…) where *f* is a feature of your generating class. And so on forever, or at least until execution terminates.

The invariant is the characteristic property of the states represented by gray squares in the figure — S1 etc. These are the "stable times" mentioned above: those at which the object is observable from the outside, in the sense that a client can apply a feature to it. They include:

- The state that results from the creation of an object (S1 in the figure).

- The states immediately before and after a call of the form *a*.*some_routine* (…) executed by a client.

  Here the context is sequential computation, but the ideas will transpose to concurrent systems in a later chapter.

Chapter 30 covers concurrency; see in particular "Concurrent accesses to an object", page 982.

An invariant that varies

In spite of its name, the invariant does not need to be satisfied at all times, although in the *STACK4* example it does remain true after the initial creation. In the more general case, it is perfectly acceptable for a procedure *g* to begin by trying to work towards its goal — its postcondition — and in the process to destroy the invariant (as in human affairs, trying to do something useful may disrupt the established order of things); then it spends the second part of its execution scurrying to restore the invariant without losing too much of whatever ground has been gained. At some intermediate stages, such as the instant marked ■ in the figure, the invariant will not hold; this is fine as long as the procedure reestablishes the invariant before terminating its execution.

Who must preserve the invariant?

Qualified calls, of the form *a*.*f* (…), executed on behalf of a client, are the only ones that must always start from a state satisfying the invariant and leave a state satisfying the invariant; there is no such rule for unqualified calls of the form *f* (…), which are not directly executed by clients but only serve as auxiliary tools for carrying out the needs of qualified calls. As a consequence, the obligation to maintain the invariant applies only to the body of features that are exported either generally or selectively; a secret feature — one that is available to no client — is not affected by the invariant.

From this discussion follows the rule that precisely defines when an assertion is a correct invariant for a class:

Invariant rule

An assertion *I* is a correct class invariant for a class *C* if and only if it meets the following two conditions:

E1 • Every creation procedure of *C*, when applied to arguments satisfying its precondition in a state where the attributes have their default values, yields a state satisfying *I*.

E2 • Every exported routine of the class, when applied to arguments and a state satisfying both *I* and the routine's precondition, yields a state satisfying *I*.

Note that in this rule:

- Every class is considered to have a creation procedure, defined as a null operation if not explicitly specified.
- The state of an object is defined by all its fields (the values of the class attributes for this particular instance).
- The precondition of a routine may involve the initial state and the arguments.

- The postcondition may only involve the final state, the initial state (through the **old** notation) and, in the case of a function, the returned value, given by the predefined entity *Result*.
- The invariant may only involve the state.

  Assertions may use functions, but such functions are an indirect way of referring to the attributes — to the state.

A mathematical statement of the Invariant rule appears later in this chapter.

This is the topic of exercise E11.8, page 410. The reasoning for preconditions and postconditions was in "Assertions are not control structures", page 347

You can use the Invariant rule as a basis for answering a question that comes up in light of earlier discussions: what would it mean if an invariant clause turned out to be violated during system execution? We saw before that a precondition violation signals an error (a "bug") in the client, a postcondition violation an error in the supplier. The answer will be for invariants as for postconditions; you have all the elements for deriving this property by yourself.

The role of class invariants in software engineering

Property E2 indicates that we may consider the invariant as being implicitly added (**and**ed) to both the precondition and postcondition of every exported routine. So in principle the notion of invariant is superfluous: we could do without it by enriching the preconditions and postconditions of all routines in the class.

Such a transformation is of course not desirable. It would complicate the routine texts; but more importantly, we would lose the deeper meaning of the invariant, which transcends individual routines and applies to the class as a whole. One should in fact consider that the invariant applies not only to the routines actually written in the class, but also to any ones that might be added later, thus serving as control over future evolution of the class. This will be reflected in the inheritance rules.

In the view of software development introduced at the beginning of this book, we accept that change is inevitable, and we try to control it. Some aspects of a software system, and of its individual components — classes — may be expected to change faster than others. Adding, removing or changing features, in particular, is a frequent and normal event. In this volatile process one will want to cling to properties that, although they may change too — for we can hardly guarantee that any aspect of a system will remain set for eternity — will change far less often. Invariants, because they capture the fundamental semantic constraints applying to a class, play this role.

The *STACK2* example illustrates the basic ideas, but to appreciate the full power of the concept of invariant you should be on the lookout for further examples of invariants in the rest of this book. To me the notion of the invariant is one of the most illuminating concepts that can be learned from the object-oriented method. Only when I have derived the invariant (for a class that I write) or read and understood it (for someone else's class) do I feel that I know what the class is about.

Invariants and contracting

Invariants have a clear interpretation in the contract metaphor. Human contracts often contain references to general clauses or regulations that apply to all contracts within a certain category; think of a city's zoning regulations, which apply to all house-building contracts. Invariants play a similar role for software contracts: the invariant of a class affects all the contracts between a routine of the class and a client.

Let us probe further. It was noted above that we may consider the invariant as being added to both the precondition and postcondition of every exported routine. Let *body* be the body of a routine (the set of instructions in its **do** clause), *pre* its precondition, *post* its postcondition and *INV* the class invariant. The correctness requirement on the routine may be expressed, using the notation introduced earlier in this chapter, as:

{*INV* **and** *pre*} *body* {*INV* **and** *post*}

The notation was defined on page 335.

(As you will remember this means: any execution of *body*, started in any state in which *INV* and *pre* both hold, will terminate in a state in which both *INV* and *post* hold.)

For the supplier author — the person who writes *body* — is the invariant good news or bad news, that is to say, does it make the job easier or harder?

The answer, as you will have figured out from the earlier discussion, is: both. Remember our lazy job applicant, who wanted a *strong* precondition and a *weak* postcondition. Here adding *INV* makes stronger or equal both the precondition and the postcondition. (From the rules of logic, *a* **and** *b* always implies *a*, that is to say, is stronger than or equal to *a*.) So, if you are in charge of implementing the *body*, the invariant:

- Makes your job easier: in addition to the official precondition *pre*, you may assume that the initial state satisfies *INV*, further restricting the set of cases that you must handle.
- Makes your job harder: in addition to your official postcondition *post*, you must ensure that the final state satisfies *INV*.

These observations are consistent with the view of the invariant as a general consistency condition that applies to the class as a whole, and hence to all of its routines. As the author of such a routine, you have the benefit of being permitted to take this condition for granted at the start of the routine; but you have the obligation to ensure that the routine will satisfy it again on termination — so that the next routine to be executed on the same object can in turn take it for granted.

The class *BANK_ACCOUNT* mentioned above, with the invariant clause

deposits_list.*total* – *withdrawals_list*.*total* = *balance*

provides a good example. If you have to add a routine to the class, this clause gives you the guarantee that the features *deposits_list*, *withdrawals_list* and *balance* have consistent values, so you do not need to check this property (and then, as we have seen, you **must not** check it). But it also means that you must write the routine so that, whatever else it does, it will leave the object in a state that again satisfies the property. So a procedure *withdraw*,

used to record a withdrawal operation, should not just update *withdrawals_list*: it must also, if *balance* is an attribute, update the value of *balance* to take the withdrawal into account and restore the invariant, enabling any other routine called later on the same object to benefit from the same original assumption that facilitated the work of *withdraw*.

See "Uniform Access", page 55.

Rather than an attribute, *balance* could be a function, whose body computes and returns the value of *deposits_list.total – withdrawals_list.total*; in this case procedure *withdraw* does not need to do anything special to maintain the invariant. The ability to switch at will between the two representations without affecting the client is an illustration of the principle of Uniform Access.

This example shows the idea of class invariant as a transposition to software of one of the rules of polite behavior: that if you use a shared facility — say an office kitchen — you should leave it for others, after each use, in the state in which you would like to find it when you start.

11.9 WHEN IS A CLASS CORRECT?

If you prefer to skip the theory you should turn to "AN ASSERTION INSTRUCTION", 11.11, page 379.

Although we still have to see a few more constructs involving assertions, it is useful to take a brief pause and examine some of the implications of what we have learned about preconditions, postconditions and invariants. This section does not introduce any new constructs, but describes some of the theoretical background. Even on your first reading I think you should get familiar with these ideas as they are central to a proper understanding of the method, and will be precious when we try to figure out how to use inheritance well.

The correctness of a class

With preconditions, postconditions and invariants, we can now define precisely what it means for a class to be correct.

The basis for the answer appeared at the beginning of this chapter: a class, like any other software element, is correct or incorrect not by itself but with respect to a specification. By introducing preconditions, postconditions and invariants we have given ourselves a way to include some of the specification in the class text itself. This provides a basis against which to assess correctness: the class is correct if and only if its implementation, as given by the routine bodies, is consistent with the preconditions, postconditions and invariant.

The notation $\{P\}\ A\ \{Q\}$ introduced at the beginning of this chapter helps express this precisely. Remember that the meaning of such a correctness formula is: whenever A is executed in a state satisfying P, the execution will terminate in a state satisfying Q.

Let C be a class, *INV* its class invariant. For any routine r of the class, call $pre_r\ (x_r)$ and $post_r\ (x_r)$ its precondition and postcondition; x_r denotes the possible arguments of r, to which both the precondition and the postcondition may refer. (If the precondition or postcondition is missing from the routine text, then pre_r or $post_r$ is just *True*.) Call $Body_r$ the body of routine r.

Finally, let $Default_C$ be the assertion expressing that the attributes of C have the default values of their types. For example $Default_{STACK2}$, referring to the earlier stack class, is the assertion

$representation = Void$
$capacity = 0$
$count = 0$

These notations permit a general definition of class correctness:

Definition: class correctness

A class is correct with respect to its assertions if and only if:

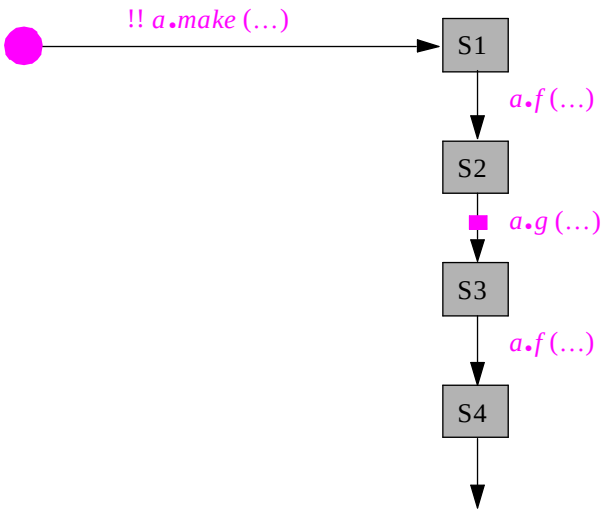
C1 • For any valid set of arguments x_p to a creation procedure p:

$\{Default_C \textbf{ and } pre_p\ (x_p)\}\ \ Body_p\ \ \{post_p\ (x_p) \textbf{ and } INV\}$

C2 • For every exported routine r and any set of valid arguments x_r:

$\{pre_r\ (x_r) \textbf{ and } INV\}\ \ Body_r\ \ \{post_r\ (x_r) \textbf{ and } INV\}$

This rule — previewed informally in the *BANK_ACCOUNT* example — is a mathematical statement of the earlier informal diagram showing the lifecycle of a typical object, which is worth looking at again:



The life of an object

(*This figure first appeared on page 366.*)

Condition C1 means that any creation procedure (such as *make* in the figure), when called with its precondition satisfied, must yield an initial state (S1 in the figure) that satisfies the invariant and the procedure's postcondition. Condition C2 expresses that any exported routine r (such as f or g in the figure), if called in a state (S1, S2 or S3) satisfying both its precondition and the invariant, must terminate in a state that satisfies both its postcondition and the invariant.

If we focus on invariants, we may look at the preceding definition of class correctness as working by induction on the set of instances of a class. Rule C1 is the base step of the induction, stating that the invariant holds for all newborn objects — those which directly result from a creation instruction. Rule C2 is the induction step, through which we determine that if a certain generation of instances satisfies the invariant, then the next generation — the set of instances obtained by applying exported features to the members of the current generation — will also satisfy it. Since by starting from newborn objects and going from generation to generation through exported features we obtain all possible instances of the class, the mechanism enables us to determine that all instances satisfy the invariant.

Two practical observations:

- If the class has no **creation** clause, we may consider that it has a single implicit creation procedure *nothing* with an empty body. Applying rule C1 to $B_{nothing}$ then means that $Default_C$ must imply *INV*: the default values must satisfy the invariant.
- A requirement of the form $\{P\}\ A\ \{Q\}$ does not commit A in any way for cases in which P is not initially satisfied. So the notation is in line with the property discussed in detail earlier in this chapter: the contract is not binding on the routine if the client fails to observe its part of the deal. Accordingly, the definition of class correctness leaves the routines of the class free to do as they please for any call that violates the precondition or the invariant.

What has just been described is how to *define* the correctness of a class. In practice, we may also want to *check* whether a given class is indeed correct. This issue will be discussed later in this chapter.

The role of creation procedures

See "CREATION PROCEDURES", 8.4, page 236, in particular "Rules on creation procedures", page 238

The discussion of invariants yields a better understanding of the notion of creation procedure.

A class invariant expresses the set of properties that objects (instances of the class) must satisfy in what has been called the stable moments of their lifetime. In particular, these properties must hold upon instance creation.

The standard object allocation mechanism initializes fields to the default values of the corresponding attribute types; these values may or may not satisfy the invariant. If not, a specific creation procedure is required; it should set the values of the attributes so as to satisfy the invariant. So creation may be seen as the operation that ensures that all instances of a class start their lives in a correct mode — one in which the invariant is satisfied.

The first presentation of creation procedures introduced them as a way to answer a more mundane (and obvious) question: how do I override the default initialization rules if they do not suit me for a particular class, or if I want to provide my clients with more than one initialization mechanism? But with the introduction of invariants and the theoretical discussion summarized by rule C1, we also see the more profound role of creation procedures: they are here to make sure that any instance of the class, when it starts its life, already satisfies the fundamental rules of its caste — the class invariant.

Arrays revisited

See "ARRAYS", 10.4, page 325.

The library class *ARRAY* was sketched in the previous chapter. Only now, however, are we in a position to give its definition properly. The notion of array fundamentally requires preconditions, postconditions and an invariant.

Here is a better sketch with assertions. Preconditions express the basic requirement on array access and modification: indices should be in the permitted range. The invariant shows the relation between *count*, *lower* and *upper*; it would allow *count* to be implemented as a function rather than an attribute.

```
indexing
        description: "Sequences of values, all of the same type or of a conforming one, %
                        %accessible through integer indices in a contiguous interval"
class ARRAY [G] creation
        make
feature -- Initialization
        make (minindex, maxindex: INTEGER) is
                        -- Allocate array with bounds minindex and maxindex
                        -- (empty if minindex > maxindex).
                require
                        meaningful_bounds: maxindex >= minindex – 1
                do
                        …
                ensure
                        exact_bounds_if_non_empty: (maxindex >= minindex) implies
                                ((lower = minindex) and (upper = maxindex))
                        conventions_if_empty: (maxindex < minindex) implies
                                ((lower = 1) and (upper = 0))
                end
feature -- Access
        lower, upper, count: INTEGER
                        -- Minimum and maximum legal indices; array size.
        infix "@", item (i: INTEGER): G is
                        -- Entry of index i
                require
                        index_not_too_small: lower <= i
                        index_not_too_large: i <= upper
                do … end
```

```
    feature -- Element change
        put (v: G; i: INTEGER) is
                -- Assign v to the entry of index i
            require
                index_not_too_small: lower <= i
                index_not_too_large: i <= upper
            do
                …
            ensure
                element_replaced: item (i) = v
            end
    invariant
        consistent_count: count = upper – lower + 1
        non_negative_count: count >= 0
    end -- class ARRAY
```

The only part left blank is the implementation of routines *item* and *put*. Because efficient array manipulation will require low-level system access, the routines will actually be implemented using **external** clauses, introduced in a later chapter.

This section explores the implications of previous concepts. Some readers may prefer to skip to "AN ASSERTION INSTRUCTION", 11.11, page 379

11.10 THE ADT CONNECTION

A class — you have heard this quite a few times by now — is an implementation of an abstract data type, whether formally specified or (as in many cases) just implicitly understood. As noted at the beginning of this chapter, we may view assertions as a way to re-introduce into the class the semantic properties of the underlying ADT. Let us perfect our understanding of assertion concepts by clarifying the connection of assertions to the components of an abstract data type specification.

Not just a collection of functions

As studied in the ADT chapter, an abstract data type is made of four elements:

- The name of the type, possibly with generic parameters (TYPES paragraph).
- The list of functions with their signatures (FUNCTIONS paragraph).
- The axioms (AXIOMS paragraph) expressing properties of the functions' results.
- The restrictions on the functions' applicability (PRECONDITIONS paragraph)

Simple-minded applications of abstract data types often overlook the last two parts. This removes much of the appeal of the approach, since preconditions and axioms express the semantic properties of the functions. If you omit them and simply view "stack" as encapsulating the (not specified further) operations *put*, *remove* etc., you retain the benefits of information hiding, but that is all. The notion of stack becomes an empty shell,

with no semantics other than suggested by the operation names. (And in the approach of this book that is of little comfort, since for reasons of structure, consistency and reusability we deliberately choose general names — *put*, *remove*, *item* … — rather than concrete, type-specific names such as *push*, *pop* and *top*.)

This risk transposes to programming in an O-O language: the routines which are supposed to implement the operations of the corresponding abstract data types could in principle perform just about any operations. Assertions avert that risk by bringing the semantics back in.

Class features *vs.* ADT functions

To understand the relation between assertions and ADTs we need first to establish the relation between class features and their ADT counterparts — the ADT's functions. An earlier discussion introduced three categories of function: creators, queries and commands. As you will recall, the category of a function

See "Function categories", page 134.

$$f : A \times B \times \ldots \rightarrow X$$

depended on where the ADT, say T, appeared among the types A, B, … X involved in this signature:

- If T appears on the right only, f is a creator; in the class it yields a creation procedure.
- If T appears only on the left of the arrow, f is a query, providing access to properties of instances of the class. The corresponding features are either attributes or functions (collectively called queries, for classes as well as ADTs).
- If T appears on both the left and the right, f is a command function, which yields a new object from one or more existing objects. Often f will be expressed, at the implementation stage, by a procedure (also called a command) which modifies an object, rather than creating a new object as a function would do.

See "Attributes and routines", page 173.

Expressing the axioms

From the correspondence between ADT functions and class features we can deduce the correspondence between semantic ADT properties and class assertions:

- A precondition for one of the specification's functions reappears as precondition clauses for the corresponding routine.
- An axiom involving a command function, possibly with one or more query functions, reappears as postcondition clauses of the corresponding procedure.
- Axioms involving only query functions reappear as postconditions of the corresponding functions or (especially if more than one function is involved, or if at least one of the queries is implemented as an attribute) as clauses of the invariant.
- Axioms involving constructor functions reappear in the postcondition of the corresponding creation procedure.

At this point you should go back to the preconditions and axioms of the ADT *STACK* and compare them with the assertions of class *STACK4* (including those of *STACK2*).

Exercise E11.2, page 409. The ADT specification is on page 139.

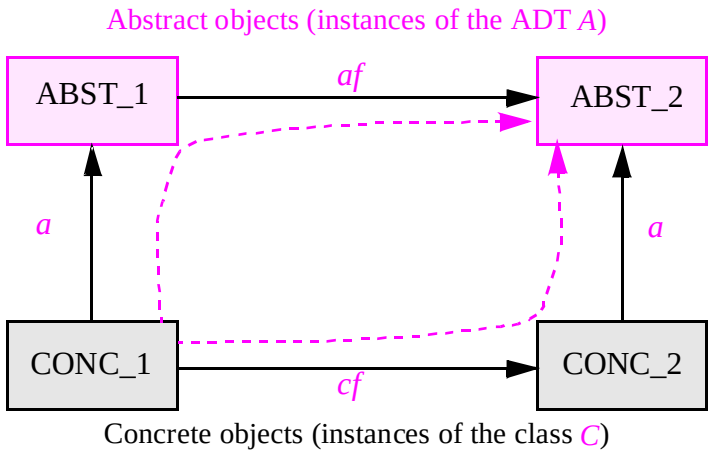
Non-mathematical readers may skip this section

The abstraction function

It is instructive to think of the preceding observations in terms of the following figure, inspired by the discussion in [Hoare 1972a], which pictures the notion "*C* is a correct implementation of *A*".

Transformations on abstract and concrete objects

(*See also the figure on page 229.*)



A is an abstract data type, and *C* as a class implementing it. For an abstract function *af* of the ADT specification — of which we assume for simplicity that it yields a result also of type *A* — there will be a concrete feature *cf* in the class.

The arrows labeled *a* represent the **abstraction function** which, for any instance of the class, or "concrete object", yields the abstract object (instance of the ADT) that it represents. As will be seen, this function is usually partial, and the inverse relation is usually not a function.

The implementation is correct if (for all functions *af* applicable to abstract data types, and their implementations *cf*) the diagram is commutative, that is to say:

Class-ADT Consistency property

$$(cf \,;\, a) = (a \,;\, af)$$

where **;** is the composition operator between functions; in other words, for any two functions *f* and *g*, *f* **;** *g* is the function *h* such that *h* (*x*) = *g* (*f* (*x*)) for every applicable *x*. (The composition *f* ; *g* is also written *g* ∘ *f* with the order of the operands reversed.)

The property states that for every concrete object CONC_1, it does not matter in which order you apply the transformation (abstract *af* or concrete *cf*) and the abstraction; the two paths, represented by dotted lines, lead to the same abstract object ABST_2. The result is the same whether you:

- Apply the concrete transformation *cf*, then abstract the result, yielding *a* (*cf* (CONC_1)).
- Abstract first, then apply the abstract transformation *af*, yielding *af* (*a* (CONC_1)).

Implementation invariants

Certain assertions appear in invariants although they have no direct counterparts in the abstract data type specifications. These assertions involve attributes, including some secret attributes which, by definition, would be meaningless in the abstract data type. A simple example is the following properties appearing in the invariant of *STACK4*:

STACK4 and its invariant appeared on page 365.

```
count_non_negative: 0 <= count
count_bounded: count <= capacity
```

Such assertions constitute the part of the class invariant known as the **implementation invariant**. They serve to express the consistency of the representation chosen in the class (here by attributes *count*, *capacity* and *representation*) vis-à-vis the corresponding abstract data type.

The figure on the previous page helps understand the concept of implementation invariant. It illustrates the characteristic properties of the abstraction function *a* (represented by the vertical arrows), which we should explore a little further.

First, is it correct to talk about *a* as being the abstraction *function*, as suggested by the upwards arrows representing *a* in the preceding figure? Recall that a function (partial or total) maps every source element to at most one target element, as opposed to the more general case of a relation which has no such restriction. If we go downwards rather than upwards in the figure and examine the inverse of *a*, which we may call the **representation relation**, we will usually find it not to be a function, since there are in general many possible representations of a given abstract object. In the array implementation that represents every stack as a pair *<representation, count>*, an abstract stack has many different representations, as illustrated by the figure on the facing page; they all have the same value for *count* and for the entries of array *representation* between indices *1* and *count*, but the size *capacity* of the array can be any value greater than or equal to *count*, and the array positions beyond index *count* may contain arbitrary values.

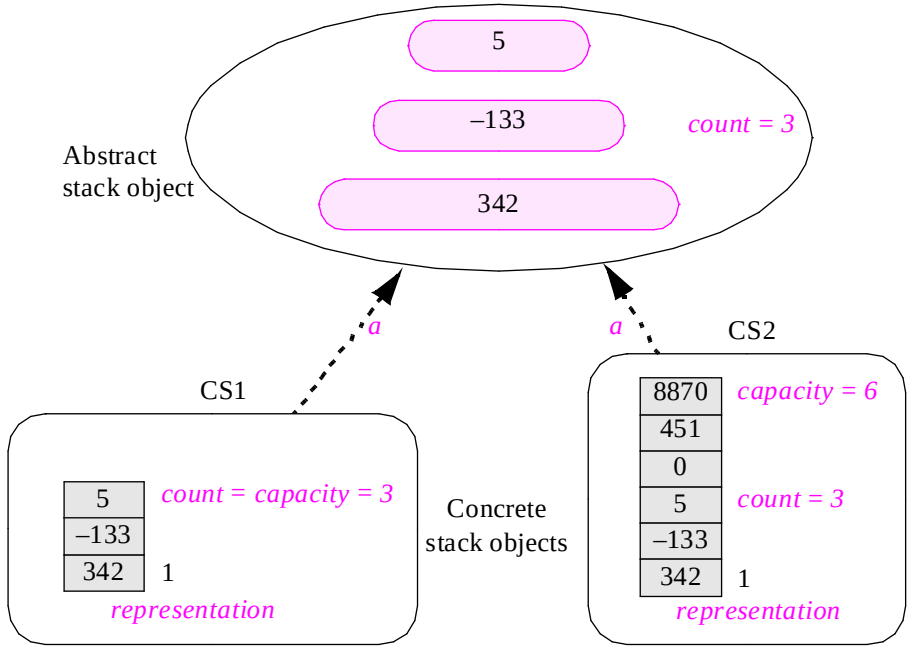
Since the class interface is restricted to the features directly deduced from the ADT's functions, clients have no way of distinguishing between the behaviors of several concrete objects that all represent the same abstract object (that is to say, all have the same *a* value). Note in particular that procedure *remove* in *STACK4* does its job simply by executing

```
count := count – 1
```

without bothering to clear the previous top entry, now at index *count + 1*; changing an entry of index higher than *count* modifies a concrete stack object CS, but has no effect on the associated abstract stack *a* (CS).

So the implementation relation is usually not a function. But its inverse the abstraction function *a* (the upwards arrows in both figures) is indeed a function since every concrete object represents at most one abstract object. In the stack example, every valid *<representation, count>* pair represents just one abstract stack (the stack with *count* elements, given, from the bottom up, by the entries of *representation* at indices *1* to *count*).

Same abstract object, two representations



Both of the concrete stacks in this figure are implementations of the abstract stack consisting of three elements of values 342, –133 and 5 from the bottom up. That *a* is a function is a universal requirement: if the same concrete object could be interpreted as implementing more than one abstract object, the chosen representation would be ambiguous and hence inadequate. So it is proper that the arrow associated with *a* points up in all the figures depicting connections between abstract and concrete types. (The discussion for inheritance will suggest a similar convention.)

The abstraction function *a* is usually a *partial* function: not every possible concrete object is a valid representation of an abstract object. In the example, not every *<representation, count>* pair is a valid representation of an abstract stack; if *representation* is an array of capacity three and *count* has value 4, they do not together represent a stack. Valid representations (members of the domain of the abstraction function) are those pairs for which *count* has a value between zero and the size of the array. This property is the implementation invariant.

In mathematical terms, the implementation invariant is the characteristic function of the **domain** of the abstraction function, that is to say, the property that defines when that function is applicable. (The characteristic function of a subset *A* is the boolean property that is true on *A* and false elsewhere.)

The implementation invariant is the one part of the class's assertions that has no counterpart in the abstract data type specification. It relates not to the abstract data type, but to its representation. It defines when a candidate concrete object is indeed the implementation of one (and then only one) abstract object.

11.11 AN ASSERTION INSTRUCTION

The uses of assertions seen so far — preconditions, postconditions and class invariants — are central components of the method. They establish the connection between object-oriented software construction and the underlying theory (abstract data types). Class invariants, in particular, cannot be understood, or even discussed, in a non-O-O approach.

Some other uses of assertions, although less specific to the method, are also precious in a systematic software development process and should be part of our notation. They include the **check** instruction, as well as loop correctness constructs (loop invariant and variant) which will be reviewed in the next section.

The **check** instruction serves to express the software writer's conviction that a certain property will be satisfied at certain stages of the computation. Syntactically, the construct is an instruction, written under the form

```
check
        assertion_clause1
        assertion_clause2
        …
        assertion_clausen
end
```

Including this instruction in the text of a routine is a way to state that:

"Whenever control reaches this instruction at execution time, the assertion shown (as given by its assertion clauses) will hold."

This is a way to reassure yourself that certain properties are satisfied, and (even more importantly) to make explicit for future readers of your software the hypotheses on which you have been relying. Writing software requires making frequent assumptions about properties of the objects of your system; as a trivial but typical example, any function call of the form *sqrt* (*x*), where *sqrt* is a routine requiring a non-negative argument, relies on the assumption that *x* is positive or zero. This assumption may be immediately obvious from the context, for example if the call is part of a conditional instruction of the form

```
if x >= 0 then y := sqrt (x) end
```

but the justification may also be more indirect, based for example on an earlier instruction that computed *x* as the sum of two squares:

```
x := a ^2 + b^2
```

The **check** instruction makes it possible to express such an assumption if it is not immediately obvious from the context, as in

```
x := a ^2 + b^2
… Other instructions …
        check
                x >= 0
                        -- Because x was computed above as a sum of squares.
        end
y := sqrt (x)
```

No **if** … **then** … protects the call to *sqrt* in this example; the **check** indicates that the call is correct. It is good practice to include, as here, a comment stating the reason invoked to support the assumption ("-- Because *x*…"). The extra two steps of indentation for the instruction are also part of the recommended style; they suggest that the instruction is not meant, in normal circumstances, to affect the algorithmic progression of the routine.

This example is typical of what is probably the most useful application of the **check** instruction: adding such an instruction just before a call to a routine that has a certain precondition (here we may assume that *sqrt* has a precondition requiring its argument to be non-negative), when you are convinced that the call satisfies the precondition but this is not immediately obvious from the context. As another example assume *s* is a stack and you include in your code a call

s.*remove*

at a position where you are certain that *s* is not empty, for example because the call has been preceded by *n* "*put*" and *m* "*remove*" instructions with *n* > *m*. Then there is no need to protect the call by an **if not** *s*.*empty* **then**…; but if the reason for the correctness of the call is not immediately obvious from the context, you may want to remind the reader that the omission of any protection was a conscious decision, not an oversight. You can achieve this by adding before the call the instruction

check not *s*.*empty* **end**

A variant of this case occurs when you write a call of the form *x*.*f* with the certainty that *x* is not void, so that you do not need to enclose this call in a conditional instruction **if** *x* /= *Void* **then** …, but the non-vacuity argument is not obvious from the context. We encountered this in the procedures *put* and *remove* of our "protected stack" class *STACK3*. The body of *put* used a call to the corresponding procedure in *STACK2*, as follows:

This is from the body of put on page 362.

```
if full then
        error := Overflow
else
                                check representation /= Void end
        representation.put (x); error := 0
end
```

Exercise E11.5, page 410, asks you for the implementation invariant of

Here a reader might think the call *representation*.*put* (*x*) in the **else** potentially unsafe since it is not preceded by a test for *representation* /= *Void*. But if you examine the class text you will realize that if *full* is false then *capacity* must be positive and hence *representation* cannot be void. This is an important and not quite trivial property, which should be part of the implementation invariant of the class. In fact, with a fully stated implementation invariant, we should rewrite the **check** instruction as:

```
check
    representation_exists: representation /= Void
        -- Because of clause representation_exists_if_not_full of the
        -- implementation invariant.
end
```

In ordinary approaches to software construction, although calls and other operations often (as in the various preceding examples) rely for their correctness on various assumptions, these assumptions remain largely implicit. The developer will convince himself that a certain property always holds at a certain point, and will put this analysis to good use in writing the software text; but after a while all that survives is the text; the rationale is gone. Someone — even the original author, a few months later — who needs to understand the software, perhaps to modify it, will not have access to the assumption and will have to figure out from scratch what in the world the author may have had in mind. The **check** instruction helps avoid this problem by encouraging you to document your non-trivial assumptions.

As with the other assertion mechanisms of this chapter, the benefit goes beyond helping you get things *right* in the first place, to helping you find that you got them *wrong*. You can, using a compilation option, turn the **check** into a true executable instruction, which will do nothing if all its assertion clauses are true, but will produce an exception and stop execution if any of them is false. So if one of your assumptions was actually not justified you should find out quickly. The mechanisms for enabling **check**-checking will be reviewed shortly.

11.12 LOOP INVARIANTS AND VARIANTS

Our last assertion constructs help us get loops right. They nicely complement the mechanisms seen so far, but are not really specific to the object-oriented method, so it is all right to skip this section on first reading.

If skipping, go to "USING ASSERTIONS", 11.13, page 390.

Loop trouble

The ability to repeat a certain computation an arbitrary number of times without succumbing to exhaustion, indeed without experiencing any degradation whatsoever, is the principal difference between the computational abilities of computers and those of humans. This is why loops are so important; just imagine what you could do in a language that only has the other two principal control structures, sequencing and conditional instructions, but no loops (and no support for recursive routine calls, the other basic mechanism permitting iterative computations).

But with power comes risk. Loops are notoriously hard to get right. Typical trouble includes:

- "Off-by-one" errors (performing one iteration too many or too few).
- Improper handling of borderline cases such as empty structures: for example a loop may work properly on a large array, but fail when the array has zero or one element.
- Failure to terminate ("infinite looping") in some cases.

Binary search — a staple of Computing Science 101 courses — is a good illustration of how tricky loops can be even when they appear trivial. Consider an array *t* of integers assumed to be in increasing order and indexed from 1 to *n*; binary search is a way to decide whether a certain integer value *x* appears in the array: if the array has no elements, the

See exercise E11.7, page 410.

answer is no; if the array has one element, the answer is yes if and only if that element has value *x*; otherwise compare *x* to the element at the array's middle position, and repeat on the lower or higher half depending on whether that element is greater or lesser than *x*. The four loop algorithms below all attempt to implement this simple idea; unfortunately all are wrong, as you are invited to check by yourself by finding, for each of them, a case in which it will not work properly.

Recall that *t* @ *m* denotes the element at index *i* in array *t*. The // operator denotes integer division, for example *7* // *2* and *6* //*2* have value *3*. The loop syntax is explained next but should be self-explanatory; the **from** clause introduces the loop initialization.

***Four (wrong) attempts at binary search. From* [M 1990]**

BS1

```
from
        i := 1; j := n
until i = j loop
        m := (i + j) // 2
        if t @ m <= x then
                i := m
        else
                j := m
        end
end
Result := (x = t @ i)
```

BS2

```
from
        i := 1; j := n; found := false
until i = j and not found loop
        m := (i + j) // 2
        if t @ m < x then
                i := m + 1
        elseif t @ m = x then
                found := true
        else
                j := m – 1
        end
end
Result := found
```

BS3

```
from
i := 0; j := n
until i = j loop
        m := (i + j + 1) // 2
        if t @ m <= x then
                i := m + 1
        else
                j := m
        end
end
if i >= 1 and i <= n then
        Result := (x = t @ i)
else
        Result := false
end
```

BS4

```
from
        i := 0; j := n + 1
until i = j loop
        m := (i + j) // 2
        if t @ m <= x then
                i := m + 1
        else
                j := m
        end
end
if i >= 1 and i <= n then
        Result := (x = t @ i)
else
        Result := false
end
```

Getting loops right

The judicious use of assertions can help avoid such problems. A loop may have an associated assertion, the *loop invariant* (not to be confused with the class invariant for the enclosing class); it may also have a *loop variant*, not an assertion but an integer expression. The invariant and variant will help us guarantee that a loop is correct.

To understand these notions it is necessary to realize that a loop is always a way to compute a certain result by **successive approximations**.

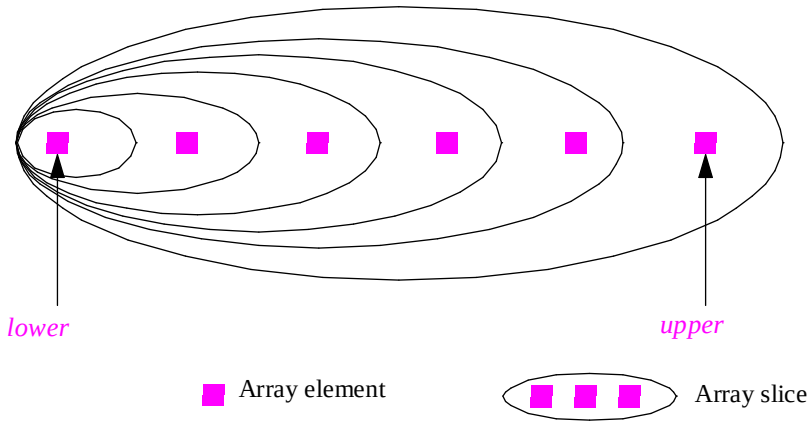
Take the trivial example of computing the maximum value of an array of integers using the obvious algorithm:

```
maxarray (t: ARRAY [INTEGER]): INTEGER is
        -- The highest of the values in the entries of t
    require
        t.capacity >= 1
    local
        i: INTEGER
    do
        from
            i := t.lower
            Result := t @ lower
        until i = t.upper loop
            i := i + 1
            Result := Result.max (t @ i)
        end
    end
```

We initialize *i* to the array's lower bound *i* := *t.lower* and the entity *Result* representing the future result to the value of the associated entry *t @ lower*. (We know that this entry exists thanks to the routine's precondition, which states that the array has at least one element.) Then we iterate until *i* has reached the upper bound, at each stage increasing *i* by one and replacing *Result* by the value of *t @ i*, the element at index *i*, if higher than the previous value of *Result*. (We rely on a *max* function for integers: *a.max (b)*, for two integers *a* and *b*, is the maximum of their values.)

This computation works by successive approximations. We approach the array by its successive slices: [*lower*,*lower*], [*lower*,*lower+1*], [*lower*,*lower+2*] and so on up to the full approximation [*lower*, *upper*].

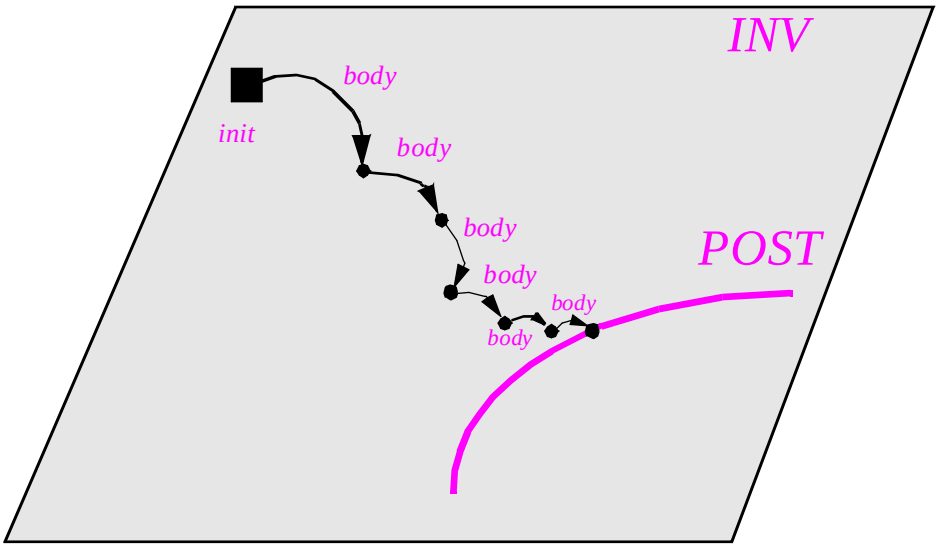
Approximating an array by successive slices



The invariant property is that at each stage through the loop *Result* is the maximum of the current approximation of the array. This is true after the initialization, since the instructions in the **from** clause ensure that *Result* is the maximum of the first approximation, the trivial slice [*lower*, *lower*] consisting of just one element. Then on each iteration we extend the slice by one element — improving our approximation of the array — and make sure to maintain the invariant by updating *Result* if the new value is higher than the previous maximum. At the end, the approximation covers the entire array, and since we have maintained invariant the property that *Result* is the maximum of the current approximation we know that it now is the maximum of the array as a whole.

Ingredients for a provably correct loop

The simple example of computing an array's maximum illustrates the general scheme of loop computation, which applies to the following standard situation. You have determined that the solution to a certain problem is an element belonging to an *n*-dimensional surface *POST*: to solve the problem is to find an element of *POST*. In some cases *POST* has just one element — *the* solution — but in general there may be more than one acceptable solution. Loops are useful when you have no way of shooting straight at *POST* but you see an indirect strategy: aiming first into an *m*-dimensional surface *INV* that includes *POST* (for $m > n$); then approaching *POST*, iteration by iteration, without ever leaving *INV*. The following figure illustrates this process.



A loop computation (from [M 1990])

A loop computation has the following ingredients:

- A goal *post*, the postcondition, defined as a property that any satisfactory end state of the computation must satisfy. Example: "*Result* is the maximum value in the array". The goal is represented in the figure by the set of states *POST* satisfying *post*.
- An invariant property *inv*, which is a generalization of the goal, that is to say includes the goal as a special case. Example: "*Result* is the maximum value in a non-empty array slice beginning at the lower bound". The invariant is represented in the figure by the set of states *INV* satisfying *inv*.
- An initial point *init* which is known to be in *INV*, that is to say to satisfy the invariant. Example: the state in which the value of *i* is the array's lower bound and the value of *Result* is that of the array element at that index, satisfying the invariant since the maximum of a one-element slice is the value of the element.
- A transformation *body* which, starting from a point in *INV* but not in *POST*, yields a point closer to *POST* and still in *INV*. In the example this transformation extends the array slice by one element, and replaces *Result* by the value of that element if higher than the previous *Result*. The loop body in function *maxarray* is an implementation of that transformation.
- An upper bound on the number of applications of *body* necessary to bring a point in *INV* to *POST*. This will be the variant, as explained next.

Computations by successive approximations are a mainstay of numerical analysis, but the idea applies more broadly. An important difference is that in pure mathematics we accept that a series of approximations may have a limit even though it cannot reach it through a finite number of approximations: the sequence *1*, *1/2*, *1/3*, *1/4*, …, *1/n*, … has

limit 0 but no element of the sequence has value zero. In computing, we want to see the results on our screen during our lifetime, so we insist that all approximation sequences reach their goal after a finite number of iterations.

> Computer implementations of numerical algorithms also require finite convergence: even when the mathematical algorithm would only converge at infinity, we cut off the approximation process when we feel that we are close enough.

The practical way to guarantee termination of a loop process is to associate with the loop an integer quantity, the loop variant, which enjoys the following properties:

- The variant is always non-negative.
- Any execution of the loop body (the transformation called *body* in the figure) decreases the variant.

Since a non-negative integer quantity cannot decrease forever, your ability to exhibit such a variant for one of your loops guarantees that the loop will always terminate.The variant is an upper bound, for each point in the sequence, of the maximum number of applications of *body* that will land the point in *POST*. In the array maximum computation, a variant is easy to find: *t.upper – i*. This satisfies both conditions:

The invariant of class ARRAY appeared on page 374.

- Because the routine precondition requires *t.capacity* to be positive (that is to say, the routine is only applicable to non-empty arrays) and the invariant of class *ARRAY* indicates that *capacity = upper – lower + 1*, the property *i <= t.upper* (part of the loop's invariant) will always be satisfied when *i* is initialized to *t.lower*.
- Any execution of the loop body performs the instruction *i := i + 1*, reducing the variant by one.

In this example the loop is simply an iteration over a sequence of integer values in a finite interval, known in common programming languages as a "for loop" or a "DO loop"; termination is not difficult to prove, although one must always check the details (here, for example, that *i* always starts no greater than *t.upper* because of the routine's precondition). For more sophisticated loops, the number of iterations is not that easy to determine in advance, so ascertaining termination is more of a challenge; the only universal technique is to find a variant.

One more notion is needed to transform the scheme just outlined into a software text describing a loop: we need a simple way of determining whether a certain iteration has reached the goal (the postcondition) *post*. Because the iteration is constrained to remain within *INV*, and *POST* is part of *INV*, it is usually possible to find a condition *exit* such that an element of *INV* belongs to *POST* if and only if it satisfies *exit*. In other words, the postcondition *post* and the invariant *inv* are related by

post = inv **and** *exit*

so that we can stop the loop — whose intermediate states, by construction, always satisfy *inv* — as soon as *exit* is satisfied. In the *maxarray* example, the obvious *exit* condition is *i = t.upper*: if this property is true together with the invariant, which states that *Result* is the maximum value in the array slice [*t.lower*, *i*], then *Result* is the maximum value in the array slice [*t.lower*, *t.upper*], hence in the entire array — the desired postcondition.

Loop syntax

The syntax for loops follows directly from the preceding rationale. It will include the elements listed as necessary:

- A loop invariant *inv* — an assertion.
- An exit condition *exit*, whose conjunction with *inv* achieves the desired goal.
- A variant *var* — an integer expression.
- A set of initialization instructions *init*, which always produces a state that satisfies *inv* and makes *var* non-negative.
- A set of body instructions *body* which, when started in a state where *inv* holds and *var* is non-negative, preserves the invariant and decreases the variant while keeping it non-negative (so that the resulting state still satisfies *inv* and has for *var* a value that is less than before but has not gone below zero).

The loop syntax combining these ingredients is straightforward:

from
    init
invariant
    inv
variant
    var
until
    exit
loop
    body
end

The **invariant** and **variant** clauses are optional. The **from** clause is required (but the *init* instructions may be empty). The effect of this instruction is to execute the *init* instructions and then, zero or more times, the *body* instructions; the latter are executed only as long as *exit* is false.

In Pascal, C etc. the loop would be a "while" loop, since the loop body is executed zero or more times, unlike the "**repeat** … **until**" loop for which the body is always executed at least once. Here the test is an exit condition, not a continuation condition, and the loop syntax includes room for initialization. So the equivalent in Pascal of **from** *init* **until** *exit* **loop** *body* **end** is

init;
while not *exit* **do** *body*

Warning: this is Pascal, not the O-O notation.

With a variant and an invariant the loop for *maxarray* appears as

```
from
        i := t.lower; Result := t @ lower
invariant
        -- Result is the maximum of the elements of t at indices t.lower to i.
variant
        t.lower – i
until
        i = t.upper
loop
        i := i + 1
        Result := Result.max (t @ i)
end
```

"The expressive power of assertions", page 400.

Note that the invariant is expressed informally as a comment; the discussion section of this chapter will explain this limitation of the assertion language.

Here is another example, first shown without variant or invariant. The purpose of the following function is to compute the greatest common divisor (gcd) of two positive integers *a* and *b* with Euclid's algorithm:

```
gcd (a, b: INTEGER): INTEGER is
        -- Greatest common divisor of a and b
require
        a > 0; b > 0
local
        x, y: INTEGER
do
        from
                x := a; y := b
        until
                x = y
        loop
                if x > y then x := x – y else y := y – x end
        end
        Result := x
ensure
        -- Result is the greatest common divisor of a and b
end
```

How do we know that function *gcd* ensures its postcondition — that it indeed computes the greatest common divisor of *a* and *b*? One way to check this is to note that the following property is true after loop initialization and preserved by every iteration:

x > *0*; *y* > *0*

-- The pair <*x*, *y*> has the same greatest common divisor as the pair <*a*, *b*>

This will serve as our loop invariant *inv*. Clearly, *INV* is satisfied after execution of the **from** clause. Also, if *inv* is satisfied before an execution of the loop body

if *x* > *y* **then** *x* := *x* – *y* **else** *y* := *y* – *x* **end**

under the loop continuation condition *x* /= *y*, then *inv* will still be satisfied after execution of this instruction; this is because replacing the greater of two positive non-equal numbers by their difference leaves them positive and does not change their gcd.

We have shown *inv* to be satisfied before the first iteration and preserved by every iteration. It follows that on loop exit, when *x* = *y* becomes true, *inv* still holds; that is to say:

x = *y* **and** "The pair <*x*, *y*> has the same greatest common divisor as the pair <*a*, *b*>"

which implies that the gcd is *x* because of the mathematical property that the gcd of any integer *x* and itself is *x*.

How do we know that the loop will always terminate? We need a variant. If *x* is greater than *y*, the loop body replaces *x* by *x* – *y*; if *y* is greater than *x*, it replaces *y* by *y* – *x*. We cannot choose *x* as a variant, because we cannot be sure that an arbitrary loop iteration will decrease *x*; nor can we be sure that it will decrease *y*, so *y* is also not an appropriate variant. But we can be sure that it will decrease *either* *x* or *y*, and hence their maximum *x*.*max* (*y*); this maximum will never become negative, so it provides the sought variant. We may now write the loop with all its clauses:

```
from
        x := a; y := b
invariant
    x > 0; y > 0
    -- The pair <x, y> has the same greatest common divisor as the pair <a, b>
variant
    x.max (y)
until
    x = y
loop
    if x > y then x := x – y else y := y – x end
end
```

As noted, the **invariant** and **variant** clauses in loops are optional. When present, they help clarify the purpose of a loop and check its correctness. Any non-trivial loop may be characterized by an interesting invariant and variant; many of the examples in subsequent chapters include variants and invariants, providing insights into the underlying algorithms and their correctness.

11.13 USING ASSERTIONS

We have now seen all the constructs involving assertions and should review all the benefits that we can derive from them. There are four main applications:

- Help in writing correct software.
- Documentation aid.
- Support for testing, debugging and quality assurance.
- Support for software fault tolerance.

Only the last two assume the ability to monitor assertions at run time.

Assertions as a tool for writing correct software

The first use is purely methodological and perhaps the most important. It has been explored in detail in the preceding sections: spelling out the exact requirements on each routine, and the global properties of classes and loops, helps developers produce software that is correct the first time around, as opposed to the more common approach of trying to debug software into correctness. The benefits of precise specifications and a systematic approach to program construction cannot be overemphasized. Throughout this book, whenever we encounter a program element, we shall seek to express as precisely as possible the formal properties of that element.

The key idea runs through this chapter: the principle of **Design by Contract**. To use features from a certain module is to contract out for services. Good contracts are those which exactly specify the rights and obligations of each party, and the *limits* to these rights and obligations. In software design, where correctness and robustness are so important, we need to spell out the terms of the contracts as a prerequisite to enforcing them. Assertions provide the means to state precisely what is expected from and guaranteed to each side in these arrangements.

Using assertions for documentation: the short form of a class

The second use is essential in the production of reusable software elements and, more generally, in organizing the interfaces of modules in large software systems. Preconditions, postconditions and class invariants provide potential clients of a module with basic information about the services offered by the module, expressed in a concise and precise form. No amount of verbose documentation can replace a set of carefully expressed assertions, appearing *in the software itself*.

"POSTSCRIPT: THE ARIANE 5 CRASH", page 411.

To learn how a particular project ignored this rule and lost an entire space mission at a cost of $500 million, see the very last section of this chapter.

The automatic documentation tool **short** uses assertions as an important component in extracting from a class the information that is relevant to potential clients. The short form of a class is a high-level view of the class. It only includes the information that is useful to authors of client classes; so it does not show anything about secret features and, for public features, it does not show the implementation (the **do** clauses). But the short form does retain the assertions, which provide essential documentation by stating the contracts that the class offers to its clients.

Chapter 23 discusses **short** *in detail.*

Here is the short form of class *STACK4*:

STACK4 appeared on page 365, based on STACK2 from page 350.

```
indexing
        description: "Stacks: Dispenser structures with a Last-In, First-Out %
                     %access policy, and a fixed maximum capacity"
class interface STACK4 [G] creation
        make
feature -- Initialization
        make (n: INTEGER) is
                        -- Allocate stack for a maximum of n elements
                require
                        non_negative_capacity: n >= 0
                ensure
                        capacity_set: capacity = n
                end
feature -- Access
        capacity: INTEGER
                        -- Maximum number of stack elements
        count: INTEGER
                        -- Number of stack elements
        item: G is
                        -- Top element
                require
                        not_empty: not empty -- i.e. count > 0
                end
feature -- Status report
        empty: BOOLEAN is
                        -- Is stack empty?
                ensure
                        empty_definition: Result = (count = 0)
                end
        full: BOOLEAN is
                        -- Is stack full?
                ensure
                        full_definition: Result = (count = capacity)
                end
```

```
feature -- Element change
    put (x: G) is
            -- Add x on top
        require
            not_full: not full
        ensure
            not_empty: not empty
            added_to_top: item = x
            one_more_item: count = old count + 1
        end
    remove is
            -- Remove top element
        require
            not_empty: not empty -- i.e. count > 0
        ensure
            not_full: not full
            one_fewer: count = old count – 1
        end
invariant
    count_non_negative: 0 <= count
    count_bounded: count <= capacity
    empty_if_no_elements: empty = (count = 0)
end -- class interface STACK4
```

This short form is not a syntactically valid class text (hence the use of **class interface** rather than the usual **class** to avoid any confusion), although it is easy to turn it into a valid *deferred* class, a notion to be seen in detail in our study of inheritance.

See chapter 36 about the environment.

In the ISE environment, you obtain the short form of a class by clicking on the corresponding button in a Class Tool displaying a class; you can generate plain text, as well as versions formated for a whole host of formats such as HTML (for Web browsing), RTF (Microsoft's Rich Text Format), FrameMaker's MML, TEX, troff and others. You can also define your own format, for example if you are using some text processing tool with its specific conventions for specifying fonts and layout.

If you compare the short form's assertions to those of the class, you will notice that all the clauses involving *representation* have disappeared, since that attribute is not exported.

The **short** form of documentation is particularly interesting for several reasons:

- The documentation is at a higher level of abstraction than what it describes, an essential requirement for quality documentation. The actual implementation, describing the *how*, has been removed, but the assertions, explaining the *what* (or in some cases the *why*) are still there. Note that the header comments of routines, which complement assertions by giving a less formal explanation of each routine's purpose, are retained, as well as the *description* entry of the **indexing** clause.

- A direct consequence of the Self-Documentation principle studied in our review of modularity concepts, the short form treats documentation not as a separate product but as information contained in the software itself. This means that there is only one product to maintain, a requirement that runs through this book. There is also, as a result, a much better chance that the documentation will be correct, since by having everything at the same place you decrease the risk of forgetting to update the documentation after a change to the software, or conversely.

"Self-Documentation", page 54.

- The short form can be extracted from the class by automatic tools. So the documentation is not something that you have to write; instead it is something that you ask "the computer" to produce, at the click of a mouse button, when you need it.

It is interesting to compare this approach with the notion of package interface present in Ada ("specification part"), where you write a module (package) in two parts: the interface and the implementation. Java uses a similar mechanism. The interface of a package has some similarities to the short form of a class, but also significant differences:

- There are no assertions, so all the "specification" that you can give is in the form of type declarations and comments.
- The interface is not produced by a tool but written separately. So the developer has to state many things twice: the headers of routines, their signatures, any header comments, declarations of public variables. This forced redundancy is tedious (it would be even more so with assertions) and, as always, raises the risk of inconsistency, as you may change one of the two parts and forget to update the other.

The short form (complemented by its variant the flat-short form, which deals with inheritance and is studied in a later chapter) is a principal contribution of the object-oriented method. In the daily practice of O-O development it appears all the time not just as a tool for documenting software, particularly reusable libraries, but also as the standard format in which developers and managers study existing designs, prepare new designs, and discuss proposed designs.

"The flat-short form", page 543.

The reason for the central role of the short form in O-O development is that it finally fulfills the goal defined by the analysis of reusability requirements at the beginning of this book. There we arrived at the requirement for *abstracted modules* as the basic unit of reuse. A class in its short (or flat-short) form is the abstracted module that we have been seeking.

"Reuse of abstracted modules", page 73.

Monitoring assertions at run time

It is time now to deal in full with the question "what is the effect of assertions at run time?". As previewed at the beginning of this chapter, the answer is up to the developer, based on a compilation option. To set that option, you should not, of course, have to change the actual class texts; you will rely instead on the Ace file. Recall that an Ace file, written in Lace, allows you to describe how to assemble and compile a system.

Lace and Ace files were introduced in "Assembling a system", page 198.

> Recall too that Lace is just one possible control language for assembling O-O systems, not an immutable component of the method. You will need something like Lace, even if it is not exactly Lace, to go from individual software components to complete compilable systems.

Here is how to adapt a simple Ace (the one used as example in the original presentation of Lace) to set some assertion-monitoring options:

Warning: this text is in Lace, not in the O-O notation.

```
system painting root
        GRAPHICS
default
        assertion (require)
cluster
        base_library: "\library\base"
        graphical_library: "\library\graphics"
                option
                        assertion (all): BUTTON, COLOR_BITMAP
                end
        painting_application: "\user\application"
                option
                        assertion (no)
                end
end -- system painting
```

The **default** clause indicates that for most classes of the system only preconditions will be checked (**require**). Two clusters override this default: *graphical_library*, which will monitor all assertions (**all**), but only for classes *BUTTON* and *COLOR_BITMAP*; and *painting_application*, which has disabled any assertion checking for all its classes. This illustrates how to define an assertion monitoring level for the system as a whole, for all the classes of a cluster, or for some classes only.

The following assertion checking levels may appear between parentheses in *assertion* (…):

- **no**: do not execute anything for assertions. In this mode assertions have no more effect on execution than comments.
- **require**: check that preconditions hold on routine entry.
- **ensure**: check that postconditions hold on routine exit.
- **invariant**: check that class invariants hold on routine entry and exit for qualified calls.

A qualified call is a call from the outside, as in $x.f$*, as opposed to a plain internal call* f*. See "Qualified and unqualified calls", page 186.*

- **loop**: check that loops invariants hold before and after every loop iteration, and that variants decrease while remaining non-negative.
- **check**: execute **check** instructions by checking that the corresponding assertions hold. **all** is a synonym for **check**.

Excluding **no**, each of these levels implies the previous ones; in particular it does not make sense to monitor postconditions unless you also monitor preconditions, since the principles of Design by Contract indicate that a routine is required to ensure its postcondition only if it was called with its precondition satisfied (otherwise "the customer is wrong"). This explains why **check** and **all** are synonyms.

If turned on, assertion monitoring will have no visible effect, except for the CPU cycles that it takes away from your computation, as long as the assertions that it monitors all evaluate to true. But having any assertion evaluate to false is a rather serious event which will usually lead to termination. Actually it will trigger an exception, but unless you have taken special measures to catch the exception (see next) everything will stop. An *exception history table* will be produced, of the general form

```
Failure: object: O2 class: YOUR_CLASS routine: your_routine
    Cause: precondition violation, clause: not_too_small
Called by: object: O2 class: YOUR_CLASS routine: his_routine
Called by: object: O1 class: HER_CLASS routine: her_routine
…
```

This gives the call chain, starting from the routine that caused the exception, the object to which is was applied and its generating class. Objects are identified by internal codes. The form shown here is only a sketch; the discussion of exceptions will give a more complete example of the exception history table.

See page 421 for the detailed form.

The optional labels that you can add to the individual clauses of an assertion, such as *not_too_small* in

```
your_routine (x: INTEGER) is
        require
                not_too_small: x >= Minimum_value
        …
```

prove convenient here, since they will be listed in the exception trace, helping you identify what exactly went wrong.

How much assertion monitoring?

What level of assertion tracing should you enable? The answer is a tradeoff between the following considerations: how much you trust the correctness of your software; how crucial it is to get the utmost efficiency; how serious the consequences of an undetected run-time error can be.

In extreme cases, the situation is clear:

- When you are debugging a system, or more generally testing it prior to release, you should enable assertion monitoring at the highest level for the classes of the system (although not necessarily for the libraries that it uses, as explained next). This ability is one of the principal contributions to software development of the method presented in this book. Until they have actually had the experience of testing a large, assertion-loaded system using the assertion monitoring mechanisms described in this section, few people realize the power of these ideas and how profoundly they affect the practice of software development.
- If you have a fully trusted system in an efficiency-critical application area — the kind where every microsecond counts — you may consider removing all monitoring.

The last advice is somewhat paradoxical since in the absence of formal proving techniques (see the discussion section of this chapter) it is seldom possible to "trust a system fully" — except by monitoring its assertions. This is a special case of a general observation made with his customary eloquence by C.A.R. Hoare:

From [Hoare 1973].

> *It is absurd to make elaborate security checks on debugging runs, when no trust is put in the results, and then remove them in production runs, when an erroneous result could be expensive or disastrous. What would we think of a sailing enthusiast who wears his life-jacket when training on dry land but takes it off as soon as he goes to sea*?

An interesting possibility is the option that only checks preconditions: *assertion* (**require**). In production runs — that is to say, past debugging and quality assurance — it has the advantage of avoiding catastrophes that would result from undetected calls to routines outside of their requirements, while costing significantly less in run-time overhead than options that also check postconditions and invariants. (Invariants, in particular, can be quite expensive to monitor since the method suggests writing rich invariants that include all relevant consistency conditions on a class, and the invariant is checked on entry and exit for every qualified call.)

Precondition checking is indeed the default compilation option if you do not include a specific *assertion* option in your Ace, so that the clause **default** *assertion* (**require**) appearing in the example Ace for system *painting* was not necessary.

Second Assertion Violation rule, page 347.

This option is particularly interesting for libraries. Remember the basic rule on assertion violations: a violated precondition indicates an error in the client; a violated postcondition or invariant indicates an error in the supplier. So if you are relying on reusable libraries that you assume to be of high quality, it is generally not desirable to monitor their postconditions and invariants: this would mean that you suspect the libraries themselves, and although the possibility of a library error is of course always open it should only be investigated (for a widely used library coming from a reputable source) once you have ruled out the presence, *a priori* much more likely, of an error in your own client software. But even for a perfect library it is useful to check **preconditions**: the goal is to find errors in client software.

See the class text starting on page 373.

Perhaps the most obvious example is array bound checking. In the *ARRAY* class we saw that *put*, *item* and the latter's synonym **infix** "@" all had the precondition clauses

```
index_not_too_small: lower <= i
index_not_too_large: i <= upper
```

Enabling precondition checking for the class solves a well-known problem of any software that uses arrays: the possibility of an out-of-bounds array access, which will usually scoop some memory area containing other data or code, causing ravages. Many compilers for conventional programming languages offer special compilation options to monitor array access at run time. But in object technology, just as we treat arrays through general notions of class and object rather than special constructs, we can handle array bound monitoring through the general mechanism for precondition checking. Just use a version of *ARRAY* compiled with *assertion* (**require**).

Should bounds always be checked? Hoare thinks so:

From [Hoare 1981]; *slightly abridged.*

> *In our Algol compiler every occurrence of every subscript of every array element was on every occasion checked at run time against the declared bounds. Many years later we asked our customers whether they wished us to provide an option to switch off these checks in the interest of efficiency in production runs. Unanimously they urged us not to — they already knew how frequently index errors occur on production runs where failure could be disastrous. I note with fear and horror that even today, language designers and users have not learned this lesson. In any respectable branch of engineering, failure to observe such elementary precautions would have long been against the law.*

These comments should be discussed not just for arrays but for preconditions in general. If indeed "index errors frequently occur on production runs" this must be true of other precondition violations too.

One may defend a less extreme position. (Some might of course see here an attempt at self-preservation, coming from a "language designer" who has provided a way to turn off assertion checking, through Lace options such as *assertion* (**no**), and presumably does not like being branded as acting "against the law".) First, a company which delivers software in which precondition errors "frequently occur on production runs" probably has a problem with its software quality practices, which run-time assertion monitoring will not solve. Monitoring addresses the symptoms (*faults* in the terminology introduced earlier in this chapter), not the cause (defects and errors). True, assertion monitoring is in such a case beneficial to the software's end-users: however unpleasant it is to have a system end its interruption with some message spouting insults about preconditions and other venomous beasts unknown to a layman, this is better than continuing operation and producing bad results. But in the long term a practice of always delivering systems with some level of assertion monitoring also has negative effects: it can encourage among developers, even unconsciously, a happy-go-lucky attitude towards correctness, justified by the knowledge that if an error remains it will be caught by the users through an assertion violation, reported to the supplier, and fixed for the following release. So can't we stop testing right now and start shipping?

It is hard to give an absolute answer to the question "should we leave some assertion monitoring on?" without some knowledge of the performance overhead of assertion monitoring. If adding some monitoring multiplied the execution time by ten, few people outside of the mission-critical-computing community would support Hoare's view; if the overhead were two percent, few people would disagree with it. In practice, of course, the penalty will be somewhere in-between.

How much is it, by the way? This clearly depends on what the software does and how many assertions it has, but it is possible to give empirical observations. In ISE's experience the cost for monitoring preconditions (the default option, including of course array bounds checking) is on the order of 50%. What is frustrating is that more than 75% of that cost is due not to precondition checking per se but to the supporting machinery of monitoring calls — recording every routine entry and every routine exit — so that if a precondition fails the environment can say which one and where. (A message of the form *Execution*

stopped because some assertion was violated somewhere would not be very useful.) This may be called the Precondition Checking Paradox: precondition checking is by itself cheap enough, but to get it you have to pay for something else. As to postcondition and invariant checking, they can bring the penalty to 100% to 200%. (Although circumstances vary, preconditions are often relatively simple consistency conditions such as *x > 0* or *a /= Void*, whereas many postconditions and invariants express more advanced semantic properties.)

One might fear that bringing performance into this discussion may lead to compromising on correctness, against the principle expressed at the beginning of this book:

Page 15.

> *Necessary as tradeoffs between quality factors may be, one factor stands out from the rest*: *correctness*. *There is never any justification for compromising on correctness for the sake of other concerns, such as efficiency. If the software does not perform its function, the rest is useless*.

Considering performance when we decide whether to leave assertion monitoring on is not, however, a violation of this principle. The point is not to sacrifice correctness for efficiency, but to determine what we should do for systems that are *not* correct — obviously because we have not worked hard enough at making them correct.

In fact, efficiency may be part of correctness. Consider a meteorological system that takes twelve hours to predict the next-day's weather (two hours would be more useful, of course). The system has been thoroughly optimized; in particular it does not have run-time checking for array bound violations or other such faults. It has also undergone careful development and extensive testing. Now assume that adding the run-time checks multiplies the execution time by two, giving a forecasting system that takes 24 hours to predict tomorrow's weather. Would you enable these checks? No.

> Although the examples that first come to mind when discussing such performance *vs.* safety issues tend to be of the Patriot-against-Scud variety, I prefer the weather forecasting example because here one cannot dismiss the efficiency issue offhand by saying "just buy a faster microprocessor". In meteorological computing, the hardware tends *already* to be the fastest parallel computer available on the market.

Let us not stop here but ask the really hard questions. Assume the original running time of twelve hours was with checking *enabled*. Would you disable it to get a six-hour forecast? Now assume that you also have the option of applying the improved efficiency to keep the same running time but use a more accurate forecasting model (since you can afford more grid points); would you do it? I think that in either case, if offered "*an option to switch off the checks in the interest of efficiency in production runs*", almost everyone will say yes.

So in the end the choice of assertion monitoring level at production time is not as simple as Hoare's rule suggests. But a few precise and strict principles do hold:

- Remember that a software system should be made reliable *before* it begins operation. The key is to apply the reliability techniques described in the software engineering literature, including those which appear in this chapter and throughout this book.
- If you are a project manager, *never let the developers assume* that the production versions will have checks turned on. Make everyone accept that — especially for the

biggest production runs, those which by nature make the consequences of potential errors most frightening — all checks may be off.

- Make sure that during development assertion checking is always turned on at least at the precondition level.
- Perform extensive testing with *all* the checks enabled. Also turn all checks on as soon as any bug is encountered during development.
- For the standard production versions, decide whether to choose a no-check version or a protected version (usually at the precondition level) based on your assessment, from an engineering perspective, of the relative weight of the three factors cited at the beginning of this discussion: how much you trust the correctness of your software (meaning in part how hard you have *worked* at making it correct and convincing yourself and others that it is); how crucial it is to get the utmost efficiency; and how serious the consequences of an undetected run-time error can be.
- If you decide to go for a no-check version, also include in your delivery a version that checks at least for preconditions. That way, if the system starts exhibiting abnormal behavior against all your expectations, you can ask the users — those at least who have not been killed by the first erroneous production runs — to switch to the checking version, helping you find out quickly what is wrong.

Used in this way, run-time assertion monitoring provides a remarkable aid for quickly weeding out any errors that may have survived a systematic software construction process.

11.14 DISCUSSION

The assertion mechanism presented in this chapter raises some delicate issues, which we must now examine.

Why run-time monitoring?

Should we really have to check assertions at run time? After all we were able, using assertions, to give a theoretical definition of what it means for a class to be correct: every creation procedure should ensure the invariant, and every routine body, when started in a state satisfying the precondition and the invariant, should maintain the invariant and ensure the postcondition. This means that we should simply *prove* the $m + n$ corresponding properties mathematically (for m creation procedures and n exported routines), and then do away with run-time assertion monitoring.

See "WHEN IS A CLASS CORRECT?", 11.9, page 370.

We should, but we cannot. Although mathematical program proving has been an active area of research for many years, and has enjoyed some successes, it is not possible today to prove the correctness of realistic software systems written in full-fledged programming languages.

> We would also need a more extensive assertion language. The IFL sublanguage, discussed below, could be used as part of a multi-tier proof strategy.

Even if proof techniques and tools eventually become available, one may suspect that run-time checks will *not* go away, if only to cope with hard-to-predict events such as hardware faults, and to make up for possible bugs in the proof software itself — in other words to apply the well-known engineering technique of multiple independent checking.

The expressive power of assertions

As you may have noted, the assertion language that we have used is essentially the language of boolean expressions, extended with a few concepts such as **old**. As a result, we may find it too restrictive when we would like to include in our classes some of the properties that were easy to express in the mathematical notation for abstract data types.

The assertions for stack classes provide a good example of what we can and cannot say. We found that many of the preconditions and axioms from the original ADT specification of chapter 6 gave assertion clauses; for example the axiom

A4 • **not** *empty* (*put* (*s*, *x*))

gives the postcondition **not** *empty* in procedure *put*. But in some cases we do not have the immediate counterpart in the class. None of the postconditions for *remove* in the stack classes given so far includes anything to represent the axiom

A2 • *remove* (*put* (*s*, *x*)) = *s*

We can of course add an informal property to the postcondition by resorting to a comment:

```
remove is
        -- Remove top element
    require
        not_empty: not empty -- i.e. count > 0
    do
        count := count – 1
    ensure
        not_full: not full
        one_fewer: count = old count – 1
        LIFO_policy: -- item is the last element pushed (by put)
                     -- and not yet removed, if any.
    end
```

Both on page 388.

Similar informal assertions, syntactically expressed as comments, appeared in the loop invariants for *maxarray* and *gcd*.

In such a case, two of the principal uses of assertions discussed earlier remain applicable at least in part: help in composing correct software, and help in documentation (an assertion clause that is syntactically a comment will appear in the short form). The other uses, in particular debugging and testing, assume the ability to evaluate assertions and do not apply any more.

It would be preferable to express all assertions formally. The best way to reach this goal is to extend the assertion language so that it can describe arbitrary properties; this requires the ability to describe complex mathematical objects such as sets, sequences, functions and relations, and including the full power of first-order predicate calculus, which allows quantified expressions ("for all" and "there exists"). Formal specification languages exist which provide at least part of this expressive power. The best known are Z, VDM, Larch and OBJ-2; both Z and VDM have had object-oriented extensions, such as Object-Z, in recent years, and the last two were close to O-O concepts already. The bibliographic notes to chapter 6 provide references.

Including a full specification language into the language of this book would have completely changed its nature. The language is meant to be simple, easy to learn, applicable to all areas of software construction, and implementable efficiently (with a final run-time performance similar to that of Fortran and C, and a fast compilation process).

Instead, the assertion mechanism is an engineering tradeoff: it includes enough formal elements to have a substantial effect on software quality; but stops at the point of diminishing return — the threshold beyond which the benefits of more formality might start being offset by the decrease of learnability, simplicity and efficiency.

> Determining that threshold is clearly a matter of personal judgment. I have been surprised that, for the software community at large, the threshold has not moved since the first edition of this book. Our field needs more formality, but the profession has not realized it yet.

So for the time being, and probably for quite a while, assertions will remain boolean expressions extended with a few mechanisms such as the **old** expression in postconditions. The limitation is not as stringent as it seems at first, because boolean expressions can use *function calls*.

Including functions in assertions

A boolean expression is not restricted to using attributes or local entities. We have already used the possibility of calling *functions* in assertions: the precondition for *put* in our stack classes was **not** *full*, were *full* is the function

```
full: BOOLEAN is
        -- Is stack full?
    do
        Result := (count = capacity)
    ensure
        full_definition: Result = (count = capacity)
    end
```

This is our little assertion secret: we get out of the stranglehold of propositional calculus — basic boolean expressions involving attributes, local entities and boolean operators such as **and**, **or**, **not** — thanks to function routines, which give us the power to compute a boolean value in any way we like. (You should not be troubled by the presence

of a postcondition in *full* itself, as it does not create any harmful circularity. Details shortly.)

Using function routines is a way to obtain more abstract assertions. For example, some people may prefer replacing the precondition of the array operations, expressed earlier as

```
index_not_too_small: lower <= i
index_not_too_large: i <= upper
```

by a single clause of the form

```
index_in_bounds: correct_index (i)
```

with the function definition

```
correct_index (i: INTEGER): BOOLEAN is
        -- Is i within the array bounds?
    do
        Result := (i >= lower) and (i <= upper)
    ensure
        definition: Result = ((i >= lower) and (i <= upper))
    end
```

Another advantage of the use of functions in assertions is that it may provide a way to circumvent the limitations on expressive power arising from the absence of first-order predicate calculus mechanisms. The informal invariant of our *maxarray* loop

```
-- Result is the maximum of the elements of t at indices t.lower to i
```

may be expressed formally as

```
Result = (t.slice (lower, i)).max
```

assuming a function *slice* which yields the set of elements between two indices of an array, and a function *max* which yields the maximum element in a set.

This approach has been explored in [M 1995a] as a way to extend the power of the assertion mechanism, possibly leading to a fully formal development method (that is to say, to software that may be *proven* correct mathematically). Two central ideas in this investigation are the use of libraries in any large-scale proof process, so that one could prove real, large-scale systems in a multi-tier proof structure using conditional proofs, and the definition of a restricted language of a purely applicative nature — IFL, for Intermediate Functional Language — in which to express the functions used in assertions. IFL is a subset of the notation of this book, which excludes some imperative constructs such as arbitrary assignments.

"The imperative and the applicative", page 352.

The risk that such efforts try to address is clear: as soon as we permit functions in assertions, we introduce potentially imperative elements (routines) into the heretofore purely applicative world of assertions. Without functions, we had the clear and clean separation of roles emphasized in the earlier discussion: instructions prescribe, assertions describe. Now we open the gates of the applicative city to the imperative hordes.

Yet it is hard to resist the power of using functions, as the alternatives are not without their drawbacks either:

- Including a full specification sublanguage could, as noted, cause problems of ease of learning and efficiency.

- Perhaps worse, it is not even clear that commonly accepted assertion languages would suffice. Take what most people versed in these topics would suggest as the natural candidate: first-order predicate calculus. This formalism will not enable us to express some properties of immediate interest to developers and common use in assertions, such as "the graph has no cycles" (a typical invariant clause). Mathematically this would be stated as $r^+ \cap r = \varnothing$ where r is the graph's relation and $^+$ is transitive closure. Although it is possible to conceive of a specification language that supports these notions, most do not.

The transitive closure of a relation is obtained by iterating it any number of times. For example "ancestor" is the transitive closure of "parent".

This is all the more troubling because, for a programmer, writing a boolean-valued function routine *cyclic* that explores the graph and returns true if and only if there is a cycle, is not particularly hard. Such examples provide a strong argument for contenting ourselves with a basic assertion language and using functions for anything beyond its expressive power.

But the need to separate applicative and imperative elements remains. Any function routine used in an assertion to specify the properties of a software element should be "beyond reproach", more precisely beyond imperative reproach; it should not cause any permanent change of the abstract state.

This informal requirement is clear enough in practice; the IFL sublanguage formalizes it by excluding all the imperative elements which either change the global state of the system or do not have trivial applicative equivalents, in particular:

- Assignments to attributes.
- Assignments in loops.
- Calls to routines not themselves in IFL.

If you exert the proper care by sticking to functions that are simple and self-evidently correct, the use of function routines in assertions can provide you with a powerful means of abstraction.

A technical point may have caught your attention. A function *f* used by an assertion for a routine *r* (or the invariant of the class containing *r*) may itself have assertions, as illustrated by both the *full* and *correct_index* examples. This raises a potential problem for run-time assertion monitoring: if as part of a call to *r* we evaluate an assertion and this causes a call to *f*, we do not want the call to evaluate any assertion that *f* itself may have. For one thing, it is easy to construct examples that would cause infinite recursion. But even without that risk it would be just wrong to evaluate the assertions of *f*. This would mean that we treat as peers the routines of our computation, such as *r*, and their assertions's functions, such as *f* — contradicting the rule that assertions should be on a

higher plane than the software they protect, and their correctness crystal-clear. The rule is simple:

Assertion Evaluation rule

During the process of evaluating an assertion at run-time, routine calls shall be executed without any evaluation of the associated assertions.

If a call to *f* occurs as part of assertion checking for *r*, that is too late to ask whether *f* satisfies its assertions. The proper time for such a question is when you decide to use *f* in the assertions applicable to *r*.

We can use an analogy introduced earlier. Think of *f* as a security guard at the entrance of a nuclear plant, in charge of inspecting the credentials of visitors. There are requirements on guards too. But you will run the background check on a guard in advance; not while he is screening the day's visitors.

Class invariants and reference semantics

The object-oriented model developed so far includes two apparently unrelated aspects, both useful:

- The notion of class invariant, as developed in this chapter.
- A flexible run-time model which, for various reasons detailed in an earlier chapter (in particular the modeling needs of realistic systems), makes considerable use of references.

See chapter 8, in particular "DEALING WITH REFERENCES: BENEFITS AND DANGERS", 8.9, page 265.

Unfortunately these individually desirable properties cause trouble when put together.

The problem is, once again, dynamic aliasing, which prevents us from checking the correctness of a class on the basis of that class alone. We have seen that the correctness of a class means $m + n$ properties expressing that (if we concentrate on the invariant *INV*, ignoring preconditions and postconditions which play no direct role here):

P1 • Every one of the m creation procedures produces an object that satisfies *INV*.

P2 • Every one of the n exported routines preserves *INV*.

These two conditions seem sufficient to guarantee that *INV* is indeed invariant. The proof is apparently trivial: since *INV* will be satisfied initially, and preserved by every routine call, it should by induction be satisfied at all stable times.

This informal proof, however, is not valid in the presence of reference semantics and dynamic aliasing. The problem is that attributes of an object may be modified by an operation on another object. So even if all $a.r$ operations preserve *INV* on the object OA attached to a, some operation $b.s$ (for b attached to another object) may destroy *INV* for OA. So even with conditions P1 and P2 satisfied, *INV* may not be an invariant.

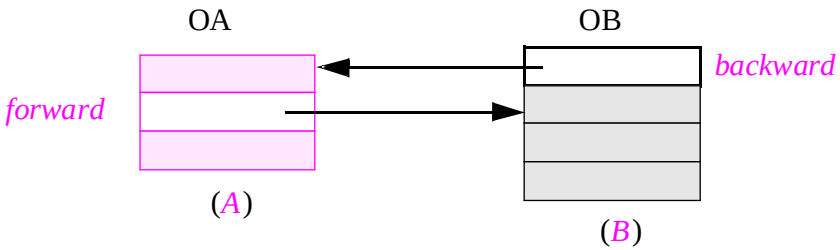
Here is a simple example. Assume classes *A* and *B*, each with an attribute whose type is the other's class:

class *A* … **feature** *forward*: *B* … **end**
class *B* … **feature** *backward*: *A* … **end**

We require that following the *forward* reference (if defined) from an instance of *A* and then the *backward* reference from the corresponding *B* will yield the original *A*. This may be expressed as an invariant property of *A*:

round_trip: (*forward* /= *Void*) **implies** (*forward*.*backward* = *Current*)

Here is a situation involving instances of both classes and satisfying the invariant:



Consistency of forward and backward references

Invariant clauses of the *round_trip* form are not uncommon; think of *forward* in class *PERSON* denoting a person's residence, and *backward* in class *HOUSE* denoting a house's resident. Then *round_trip* states that the resident of any person's residence is that person, a simple consistency requirement. Another example is the linked implementation of trees, where the attributes of a tree node include references to its first child and to its parent, introducing the following *round_trip*-style property in the invariant:

(*first_child* /= *Void*) **implies** (*first_child*.*parent* = *Current*)

Assume, however, that the invariant clause of *B*, if any, says nothing about the attribute *backward*. The following version of *A* appears consistent with the invariant:

```
class A feature
        forward: B
        attach (b1: B) is
                        -- Link b1 to current object.
                do
                        forward := b1
                                -- Update b1's backward reference for consistency:
                        if b1 /= Void then
                                b1.attach (Current)
                        end
                end
invariant
        round_trip: (forward /= Void) implies (forward.backward = Current)
end
```

The call *b1*.*attach* is meant to restore the invariant after an update of *forward*. Class *B* must provide its own *attach* procedure:

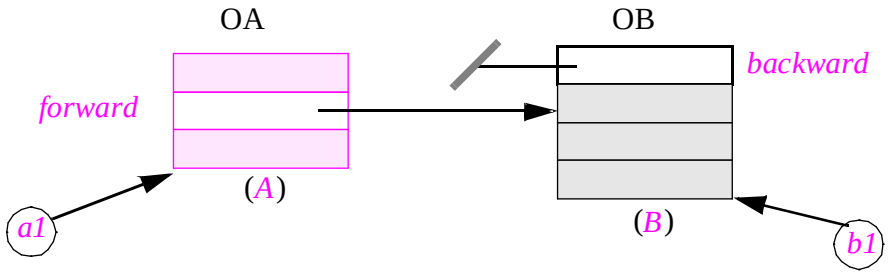
```
class B feature
        backward: B
        attach (a1: A) is
                -- Link a1 to current object.
            do
                backward := a1
            end
end
```

Class *A* appears to be correct: a procedure-less creation instruction ensures the invariant *round_trip* (since it initializes *forward* to a void reference), and its sole procedure will always preserve *round_trip*. But consider execution of the following:

```
a1: A; b1: B
…
!! a1; !! b1
a1.attach (b1)
b1.attach (Void)
```

Here is the situation after the last instruction:

Violating the invariant



The invariant is violated on OA! This object is now linked to OB, but OB is not linked to OA since its *backward* field is void. (A call to *b1*.*attach* (…) could also have linked OB to an instance of *A* other than OA, which would be equally incorrect.)

What happened? Dynamic aliasing has struck again. The proof of correctness of class *A* outlined above is valid: every operation of the form *a1*.*r*, where *a1* is a reference to object OA, will preserve *round_trip* since the corresponding features of *A* (here there is only one, *attach*) have been designed accordingly. But this is not sufficient to preserve the consistency of OA, since properties of OA may involve instances of other classes, such as *B* in the example, and the proof says nothing about the effect of these other classes' features on the invariant of *A*.

This problem is important enough to deserve a name: **Indirect Invariant Effect**. It may arise as soon as we allow dynamic aliasing, through which an operation may modify

an object even without involving any entity attached to it. But we have seen how much we need dynamic aliasing; and the *forward-backward* scheme, far from being just an academic example, is as noted a useful pattern for practical applications and libraries.

What can we do? The immediate answer involves the conventions for run-time monitoring of assertions. You may have wondered why the effect of enabling assertion monitoring at the *assertion* (**invariant**) level was described as

"*Check that class invariants hold on routine entry and exit for qualified calls.*" *Page 394.*

Why both entry and exit? Without the Indirect Invariant Effect, it would suffice to check the invariant when exiting qualified calls. (It is also checked at the end of creation calls.) But now we have to be more careful, since between the termination of a call and the beginning of the next one on the same object, some call may have affected that object even though its target was another object.

A more satisfactory solution would be to obtain a statically enforceable validity rule, which would guarantee that whenever the invariant of a class *A* involves references to instances of a class *B*, the invariant of *B* includes a mirror clause. In our example we can avoid trouble by including in *B* an invariant clause *trip_round* mirroring *round_trip*:

trip_round: (*backward* /= *Void*) **implies** (*backward*.*forward* = *Current*)

It may be possible to generalize this observation to a universal mirroring rule. Whether such a rule indeed exists, solving the Indirect Invariant Effect and removing the need for double run-time monitoring, requires further investigation.

More to come

We are not done with Design by Contract. Two important consequences of the principles remain to be studied:

- How they lead to a disciplined exception handling mechanism; this is the topic of the next chapter.
- How they combine with inheritance, allowing us to specify that any semantic constraints that apply to a class also apply to its descendants, and that semantic constraints on a feature apply to its eventual redeclarations; this will be part of our study of inheritance. *"INHERITANCE AND ASSERTIONS", page 569.*

More generally, assertions and Design by Contract will accompany us throughout the rest of this book, enabling us to check, whenever we write software elements, that we know what we are doing.

11.15 KEY CONCEPTS INTRODUCED IN THIS CHAPTER

- Assertions are boolean expressions expressing the semantic properties of classes and reintroducing the axioms and preconditions of the corresponding abstract data types.
- Assertions are used in preconditions (requirements under which routines are applicable), postconditions (properties guaranteed on routine exit) and class

invariants (properties that characterize class instances over their lifetime). Other constructs that involve assertions are loop invariants and the **check** instruction.

- A precondition and a postcondition associated with a routine describe a contract between the class and its clients. The contract is only binding on the routine inasmuch as calls observe the precondition; the routine then guarantees the postcondition on return. The notion of contracting provides a powerful metaphor for the construction of correct software.
- The invariant of a class expresses the semantic constraints on instances of the class. The invariant is implicitly added to the precondition and the postcondition of every exported routine of the class.
- A class describes one possible representation of an abstract data type; the correspondence between the two is expressed by the abstraction function, which is usually partial. The inverse relation is in general not a function.
- An implementation invariant, part of the class invariant, expresses the correctness of the representation vis-à-vis the corresponding abstract data type.
- A loop may have a loop invariant, used to deduce properties of the result, and a variant, used to ascertain termination.
- If a class is equipped with assertions, it is possible to define formally what it means for the class to be correct.
- Assertions serve four purposes: aid in constructing correct programs; documentation aid; debugging aid; basis for an exception mechanism.
- The assertion language of our notation does not include first-order predicate calculus, but can express many higher-level properties through function calls, although the functions involved must be simple and of unimpeachable correctness.
- The combination of invariants and dynamic aliasing raises the Indirect Invariant Effect, which may cause an object to violate its invariant through no fault of its own.

11.16 BIBLIOGRAPHICAL NOTES

According to Tony Hoare:

From [Hoare 1981].

> *An early advocate of using assertions in programming was none other than Alan Turing himself. On 24 June 1950 at a conference in Cambridge, he gave a short talk entitled "Checking a Large Routine" which explains the idea with great clarity. "How can one check a large routine in the sense that it's right? In order that the man who checks may not have too difficult a task, the programmer should make a number of definite* assertions *which can be checked individually, and from which the correctness of the whole program easily follows."*

The notion of assertion as presented in this chapter comes from the work on program correctness pioneered by Bob Floyd [Floyd 1967], Tony Hoare [Hoare 1969] and Edsger

Dijkstra [Dijkstra 1976], and further described in [Gries 1981]. The book *Introduction to the Theory of Programming Languages* [M 1990] presents a survey of the field.

The notion of class invariant comes from Hoare's work on data type invariants [Hoare 1972a]. See also applications to program design in [Jones 1980] [Jones 1986]. A formal theory of morphisms between abstract data types may be found in [Goguen 1978].

Formal specification languages include Z, VDM, OBJ-2 and Larch; see the bibliographical references to chapter 6. Object-oriented formal specification languages include Object Z, Z++, MooZ, OOZE, SmallVDM and VDM++, all of which are described in [Lano 1994] which gives many more references.

The IEEE Computer Society publishes standards for the terminology of software errors, defects, failures [IEEE 1990] [IEEE 1993]. Its Web page is at *http://www.computer.org*.

Surprisingly, few programming languages have included syntactical provision for assertions; an early example (the first to my knowledge) was Hoare's and Wirth's Algol W [Hoare 1966], the immediate precursor of Pascal. Others include Alphard [Shaw 1981] and Euclid [Lampson 1977], which were specifically designed to allow the construction of provably correct programs. The connection with object-oriented development introduced by the notation developed in this book was foreshadowed by the assertions of CLU [Liskov 1981] which, however, are not executable. Another CLU-based book by Liskov and Guttag [Liskov 1986], one of the few programming methodology texts to discuss in depth how to build reliable software, promotes the "defensive programming" approach of which the present chapter has developed a critique.

The notion of Design by Contract presented in this chapter and developed in the rest of this book comes from [M 1987a] and was further developed in [M 1988], [M 1989c], [M 1992b] and [M 1992c]. [M 1994a] discusses the tolerant and demanding approaches to precondition design, with particular emphasis on their application to the design of reusable libraries, and introduces the "tough love" policy. Further developments of the ideas have been contributed by James McKim in [McKim 1992a] (which led to some of the initial ideas for IFL), [McKim 1995], [McKim 1996], [McKim 1996a]; see also [Henderson-Sellers 1994a] which examines the viewpoint of the supplier.

EXERCISES

E11.1 Complex numbers

Write the abstract data type specification for a class *COMPLEX* describing the notion of complex number with arithmetic operations. Assume perfect arithmetic.

E11.2 A class and its ADT

The ADT specification appears on page 139. STACK4 is on page 365 and includes STACK2, page 350

Examine all the preconditions and axioms of the *STACK* abstract data type introduced in an earlier chapter and study whether and how each is reflected in class *STACK4*.

E11.3 Complete assertions for stacks

Show that by introducing a secret function *body* which returns the body of a stack, it is possible to make the assertions in a *STACK* class reflect the full corresponding abstract data type specification. Discuss the theoretical and practical value of this technique.

E11.4 Exporting the size

Page 350.

Why is *capacity* exported for the bounded implementation of stacks, class *STACK2*?

E11.5 An implementation invariant

"A tolerant module", page 360.

Write the implementation invariant for class *STACK3*.

E11.6 Assertions and exports

Page 402.

The discussion of using functions in assertions introduced a function *correct_index* for the precondition of *item* and *put*. If you add this function to class *ARRAY*, what export status must you give it?

E11.7 Finding the bugs

Page 382.

Show that each of the four attempts at binary search algorithms advertized as "wrong" is indeed incorrect. (**Hint**: unlike proving an algorithm correct, which requires showing that it will work for all possible cases, proving it incorrect only requires that you find *one* case in which the algorithm will produce a wrong result, fail to terminate, or execute an illegal operation such as an out-of-bounds array access or other precondition violation.)

E11.8 Invariant violations

The discussion in this chapter has shown that a precondition violation indicates an error in the client, and a postcondition violation indicates an error in the supplier. Explain why an invariant violation also reflects a supplier error.

E11.9 Random number generators

Write a class implementing pseudo-random number generation, based on a sequence $n_i = f(n_{i-1})$ where f is a given function and the seed n_0 will be provided by clients of the class. Functions should have no side effects. (Assume f is known; you can find such functions in textbooks such as [Knuth 1981], and in numerical libraries.)

E11.10 A queue module

Write a class implementing queues (first-in, first-out policy), with appropriate assertions, in the style of the *STACK* classes of this chapter.

E11.11 A set module

Write a class implementing sets of elements of an arbitrary types, with the standard set operations (membership test, addition of a new element, union, intersection etc.). Be sure to include the proper assertions. Any correct implementation, such as linked lists or arrays, is acceptable.

POSTSCRIPT: THE ARIANE 5 CRASH

As this book was being printed, the European Space Agency released the report of the international investigation into the test flight of the Ariane 5 launcher, which crashed on June 4, 1996, 40 seconds after lift-off, at a reported cost of 500 million dollars (uninsured).

For a more detailed discussion see [M 1997a].

The cause of the crash: a failure of the on-board computer systems. The cause of that failure: a conversion from a 64-bit floating-point number (the mission's "horizontal bias") to a 16-bit signed integer produced an exception because the number was not representable with 16 bits. Although some other possible exceptions were monitored (using the Ada mechanisms described in the next chapter) prior analysis had shown that this particular one could not occur; so it was decided not to encumber the code with an extra exception handler.

The real cause: insufficient specification. The analysis that the value would always fit in 16 bits was in fact correct — but for the Ariane 4 flight trajectory! The code was reused for Ariane 5, and the assumption, although stated in an obscure part of some technical document, was simply forgotten. It did not apply any more to Ariane 5.

With the Design by Contract approach, it would have been stated in a precondition:

```
require
    horizontal_bias <= Maximum_horizontal_bias
```

naturally prompting the quality assurance team to check all uses of the routine and to detect that some could violate the assertion. Although we will never know, it seems almost certain that the mistake would have been caught, probably through static analysis, and at worst during testing thanks to the assertion monitoring mechanisms described in this chapter.

The lesson is clear: **reuse without contracts is folly**. The "abstracted modules" that we have defined as our units of reuse must be equipped with clear specifications of their operating conditions — preconditions, postconditions, invariants; and these specifications must be *in the modules themselves*, not in external documents. The principles that we have learned, particularly Design by Contract and Self-Documentation, are a required condition of any successful reusability policy. Even if your mistakes would cost less than half a billion dollars, remember this rule as you go after the great potential benefits of reuse: to be reusable, a module must be specified; and the programming language must support assertion mechanisms that will put the specification in the software itself.