Chapter 1
Design by Contract

Bertrand Meyer
Interactive Software Engineering Inc.

ABSTRACT: This chapter is a contribution to software reliability. It presents a view of software design based on a metaphor: software construction as a succession of documented contracting decisions.
This idea provides insights into a number of issues faced by programmers whenever they try to construct correct and robust software.
Two results of the approach developed in this paper are a coherent approach to the handling of failures and exceptions, believed to be both simpler and safer than earlier techniques, and a coherent interpretation of two important but potentially unsafe aspects of object-oriented programming: inheritance and dynamic binding.
The discussion relies on the Eiffel object-oriented language and environment.

1.1 SCOPE

How can we build reliable software – in other words, software that is both correct and robust?

Anyone who writes or uses programs knows how pressing this question is. Yet little of the theoretical work that has been devoted to techniques that could improve reliability (such as formal specification and verification) has found its way into the common practice of software development.

Worse yet, much of the literature on object-oriented programming – an approach that is increasingly recognized as the *sine qua non* of future advances in software technology – is all but silent on correctness and robustness, as if it were satisfactory enough to obtain the flexible and modular structures permitted by object-oriented methods. This silence is all the more surprising in light of the role played in object-oriented programming by reuse, perhaps the central concept of the whole approach: with a method based on general-purpose components meant to be used in scores of different applications, the correctness and robustness of these components becomes a critical issue, even more so than in traditional one-of-a-kind developments.

This chapter is a contribution to the search for software reliability, expanding on the discussions of an earlier book [8]. It does not, of course, offer any magical recipe, but describes engineering techniques which, if applied consistently, can considerably improve the trust that we place in our software.

A theory of software design runs through this presentation, based on a metaphor: software construction as a succession of documented contract decisions. This idea provides insights into many of the issues programmers face whenever they try to construct correct and robust software.

In particular, the discussion will analyze the dangers involved in some of the existing methods for handling failures and exceptions, and will introduce a simpler approach, which should also be safer. It will also throw some light on a key topic of object-oriented programming by showing how the contract theory clarifies inheritance and dynamic binding – two essential components of the object-oriented method, which are, however, the source of serious potential confusions and dangers unless developers understand their theoretical underpinnings.

The discussion relies on the Eiffel language [11] and method [8], whose mechanisms for assertions, exception handling and inheritance result largely from the reflections reported here. Some elements should be of interest to readers working in other contexts; there will even be some advice on how not to misuse exceptions in Ada.

1.2 THE NOTION OF CONTRACT

1.2.1 Decomposing into subtasks

Assume you are writing some program unit implementing a task to be performed at run-time. The unit describes the task as a combination of more elementary subtasks. To make things simple, this can be expressed just as a sequence of abstract instructions, each corresponding to a subtask:

```
my_task is
    do
        subtask_1 ;
        subtask_2 ;
        ...
        subtask_n ;
    end -- my_task
```

As the implementer of *my_task*, you are faced with a fundamental decision for each of the $subtask_i$: should you handle the subtask locally, or should you contract it out?

Concretely, the first solution means that you implement the task by writing one or more elementary instructions corresponding to $subtask_i$ within the body of *my_task*. The second solution means that you write a separate routine, or get access

to a routine written by someone else, and implement $subtask_i$ by a call to that routine.

The decision of contracting out a subtask to a routine may be made for either or both of two reasons:

- To keep the size of the current unit under control, by separating the details of a subtask from the unit's main stream.
- More pragmatically, to take advantage of commonality between subtasks, especially when an existing program element addresses the subtask at hand.

These two incentives respectively reflect the top-down and bottom-up components of software construction.

The need to choose constantly between direct implementation and contracting out is a key feature of software development. The design of a software system is the result of a myriad of minute decisions – "Do we contract out for this particular subtask, or do we do it here?" Good designers know how to find the appropriate balance between too much contracting, which produces overly fragmented architectures, and too little, which yields unmanageably large modules.

1.2.2 Writing down the contract

When you contract out, you must make sure that the contractor will perform the task as required. As in real life, this is only possible if the agreement is spelled out precisely in a contract document.

A contract document protects both sides:

- It protects the client by specifying how much should be done: the client is entitled to receive a certain result.
- It protects the contractor by specifying how *little* is acceptable: the contractor must not be liable for failing to carry out tasks outside of the specified scope.

A contract carries mutual obligations and benefits. If I (as the contractor) agree to build a building at least three stories high, for at most one million francs, on a parcel of land covering at least one hectare which you (the client) will have to provide, then you are not obliged to do anything if you do not find a parcel, or if your parcel is smaller than one hectare; and if I do comply with these obligations, you can reject my building if it is less than three stories high, or costs more than one million francs.

The obligations and benefits for both parties in this simplistic example are summarized in the table of Figure 1.1. Note how the obligation for each maps into a benefit for the other.

	Obligations	**Benefits**
Client	Provide land parcel of at least one hectare	Get building three-story high or higher, for at most one million francs
Contractor	Build building three-story high or higher for at most one million francs	No need to do anything if there is no land, or land is to small

Figure 1.1: A contract

Primitive as it is, this example serves as evidence of some the fundamental properties of contracts in human affairs:

- As noted, a contract implies obligations and benefits for both parties. Usually, an obligation for one maps into a benefit for the other.
- The obligations and benefits are explicit: the role of the contract document is precisely to spell them out in detail, avoiding ambiguity inasmuch as humanly feasible.
- Some general clauses may remain implicit because they automatically apply to all contracts. They reflect the law of the land and other prevailing regulations.
- One less immediate but equally essential property of contracts is that even an "obligations" box in a table such as the above is actually also a "benefit" for the corresponding party. The reason is that such a clause implicitly expresses that the obligations mentioned are the *only* ones that bind that party (apart from the just mentioned universal clauses, if any).

The last property, which may be called the **no hidden clauses** rule, is fundamental to the smooth functioning of contracts in a law-based society: when you see a list of your obligations, it does not just bring "bad news" by stating work that you must perform; it is also "good news" because it states the limits on the duties imposed on you.

1.3 ASSERTIONS: CONTRACTING FOR SOFTWARE

It is surprising that software contracts – routine calls – are not similarly documented in standard approaches to programming. Yet if we entertain any hope of producing correct and robust software, the very least we can do is to make explicit the obligations and guarantees on any call.

The mechanisms for expressing such conditions are called assertions. Some assertions, called preconditions and postconditions, apply to individual routines; others, called class invariants, constrain all the routines of a given class.

1.3.1 Assertions on individual routines

To specify the terms of a software contract, we may associate a **precondition** and a **postcondition** with each routine. In Eiffel, they appear in the syntax for routine declarations, as follows:

```
routine_name (argument declarations) is
            -- Header comment
        require
            precondition
        do
            routine body, i.e. instructions
        ensure
            postcondition
        end -- routine_name
```

The **require** and **ensure** clauses (as well as the header comment) are optional. The precondition and postcondition are "assertions", or lists of boolean expressions, separated by semicolons, which are equivalent to boolean "ands" but allow individual identification of the assertion clauses. The precondition expresses requirements that any call must satisfy if it is to be correct; the postcondition expresses properties that are ensured in return by the execution of the call.

A missing precondition clause is equivalent to **require true**, and a missing postcondition to **ensure true**. Assertion **true** is the least committing of all possible assertions, and is always satisfied.

Consider for example a routine *put* for adding an element of some type T to a table. A character key is associated with every table element. Assume the table is managed by a scheme such as hash-coding, where the insertion position is determined by the insertion algorithm on the basis of the key (rather than specified by the client). The routine may be written in the following form:

```
put (element: T, key: STRING) is
        -- Insert element with key key
    require
        count < capacity
    do
        ... "Insertion algorithm" ...
    ensure
        count <= capacity;
        item (key) = element;
        count = old count + 1
    end -- put
```

The following explanations will be useful for the reader not familiar with Eiffel. First we are in an object-oriented environment in which every operation is relative to a certain class of objects. The routine will thus be in some class, say *TABLE*, describing the behavior of tables through the operations and attributes available on them. (The class should be declared as *TABLE* [*T*], where the generic type parameter *T* allows use of the same class for tables of elements of various types, in a type-safe fashion.) The routine will be applied by a client to a table *ta* through a call of the form *ta•put* (*val*) where *val* has the appropriate type.

A class is a set of encapsulated services offered to clients on elements of a certain type, here *TABLE*; these services, known as **features**, are implemented either as routines (procedures or functions), performing some computation, or as attributes, which simply describe some field present at run-time in the representation of every object of the type. Here the class will contain the following features beyond *put*:

- Function *item*, such that *item* (*k*), called by clients in the form *ta•item* (*k*) for some table *ta* and some string *k*, gives the value associated with key *k* in the table.
- Attributes *capacity* and *count* (for clients, *ta•capacity* and *ta•count*), giving the table size and current number of used entries.

The precondition and postcondition of *put* express the terms of the contract imposed on any client that wishes to use this routine: *put* accepts a call if and only if the table is not full (in other words, has a number of inserted elements, *count*, less than its *capacity*) and yields a table with one more element, such that the value associated with *key* is the *element* inserted. The notation **old** *count*, in the postcondition, denotes the value of *count* on routine entry; the unary operator **old** is used only in postconditions.

1.3.2 A premature question

At this point many readers will already have mentally raised their hands to ask the question "What happens at run-time if the table **is** in fact full when *put* is called?".

A pessimist might view the importance that software developers seem to attach to this question as a sad comment on the state of software engineering: a sign

that developers are more interested in trying to limit the consequences of their errors than in learning how to avoid errors in the first place, or to correct them.

Errors and exceptional conditions do occur, of course; accordingly, we need to study in detail the effect of assertions on program execution, and the question of how a program can recover from a run-time assertion violation. This will be done below. But if we are responsible professionals and have set our priorities straight we must first look for ways to produce correct software, and *then* consider what happens if we have failed to do so. For the time being, assertions are a pure design and documentation aid: a conceptual tool for building better software, and explaining the essentials of a software component to its potential users.

1.3.3 Observations on software contracts

The preconditions and postconditions express the terms of the contract. The roster of benefits and obligations may be given for *put* in the same style as Figure 1.1:

	Obligations	**Benefits**
Client	Call *put* only on a non-full table	Get modified table in which *x* is associated with *key*
Contractor	Insert *x* so that it may be retrieved through *key*	No need to deal with the case in which table is full before insertion.

Figure 1.2: A software contract

The bottom-right entry of the table is particularly noteworthy. If the precondition is not satisfied, the routine is not bound to do anything, as a house builder who is not given any land on which to build. This means that the routine body should **not** be of the form

```
if count = capacity then
        ...
else
        ... "Deal with normal case" ...
end
```

which would defeat the whole purpose of having a precondition (**require** clause). This is an absolute rule: either you have the condition in the **require**, or you have it in an **if** instruction in the body of the routine, but never in both.

Because of this, preconditions are sometimes viewed with suspicion. Shouldn't a routine be prepared to handle all possible inputs?

It should not. Again, the contract metaphor provides the proper perspective to discuss this issue. The stronger the precondition, the higher the burden on the client, and the easier for the contractor. (The most comfortable job in the world is that of a routine implementor presented with the precondition **false** – any implementation will do, since no call will ever be correct.) The matter of who should deal with abnormal values is essentially a pragmatic decision about division of labor: the best solution is the one that achieves the simplest architecture.

If every routine checked for every possible error in its calls, no useful work would ever be performed. If both the client and the routine check for the same conditions, the resulting redundancy, when accumulated over a large system, will yield the complexity and unwieldiness that are so characteristic of today's software.

In many existing programs, one can hardly find the islands of useful processing in oceans of error-checking code. Much of the redundancy in error checking is understandable: better check twice than not at all, reason the designers. But with techniques for defining precisely each party's responsibility, as provided by assertions, such redundancy, so harmful to the overall program structure, is no longer necessary.

Not many software engineering textbooks talk in any detail about how to obtain reliable software. Most of those which do (see for example [5]) state that individual routines should be able to cope with as many cases as possible – that is to say, have the weakest possible preconditions. In spite of this conventional wisdom, however, a good case may be made for routines with strong preconditions. Such routines will concentrate on performing well a precisely defined task, rather than attempt to handle all possible abnormal cases.

Developers are traditionally reluctant to distribute modules which will only work under strict constraints; but this is due to the lack of a standard for documenting the constraints simply and clearly. Preconditions provide this standard.

1.3.4 Against defensive programming

The method outlined above may be viewed as the exact opposite of the traditional advice given to programmers preoccupied with reliability: "defensive programming" – the recommendation to protect every software module by as many checks as possible, even those which are redundant with checks made by the clients. Include them anyway, the standard advice goes, just in case: if they do not help, at least they will not harm.

But they do harm. The result of such blind checking can only be an increase in software complexity, which inevitably leads to a decrease in reliability.

The contract theory suggests a different approach. It prompts developers to specify precisely every consistency condition that could go wrong, and to assign explicitly the responsibility of its enforcement to one software element, supplier or client. With these responsibilities clearly defined through contracts, there is no further need for ad hoc redundant checks.

1.3.5 Who should check?

The rejection of defensive programming means that we never ask both client and supplier to be responsible for a consistency condition. Either the condition is part of the precondition, and must be guaranteed by the client; or it is not stated in the precondition, in which case the supplier must handle it.

Which of these two solutions should we choose? Here there is no absolute rule; several styles of writing routines are possible, ranging from "demanding" ones where the precondition is strong (putting the responsibility on clients) to "tolerant" ones where it is weak (increasing the routine's burden). Choosing between them is to a certain extent a matter of personal preference; again, the key criterion is to maximize the overall simplicity of the architecture.

The standard recommended style in Eiffel is on the demanding side: it encourages writing simple routines with a well-defined contract, rather than routines which will attempt to handle every imaginable case. Client programmers do not expect miracles: as long as the conditions on the use of a routine make sense and, above all, the routine's documentation states these conditions (the contract) precisely, they will be able to use the routine properly by observing their part of the deal.

This demanding style is consistently used in the Eiffel Libraries; to take an example among hundreds, function *first* in any of the list classes, returning the first element of a list, has a precondition stating that the list is non-empty. Fair enough.

An objection sometimes heard against this style is that it seems to force every client to make the same checks, corresponding to the precondition, and so to result in unnecessary and damaging repetitions.

On further examination, however, this objection does not hold.

First, the presence of a precondition p in a routine r does not necessarily mean that every call must test for p, as in

/A/

```
if x.p then
        x.r
else
        ... Special treatment ...
end
```

This is only one possible form. What the precondition means is that the client must *guarantee* property p, which is not the same as *testing for* this condition before each call. If the context of the call implies p, then there is no need for such a test. A typical form which avoids the test is

```
x.s;   x.r
```

where the postcondition of s is such that it implies p. For example, x might be a data structure such as a queue, priority list or stack, r the operation

```
remove is
            -- Remove an element
      require
            not empty
      ...
```

and s the operation *put*, which adds an element, and so has the condition **not** *empty* as part of its postcondition. If the call to *remove* follows a call to *put*, there is no need to check for the precondition.

Assume now that this case does not hold and that many clients will indeed need to check for the precondition, as in form /A/ above. What matters then is the "Special treatment" in the **else** clause (dealing with the case in which the precondition is not satisfied). There are two possibilities:

- The "Special treatment" may be the same for all calls. Then there is indeed unpleasant repetition in many clients. But this is almost certainly a sign of a poor contract for the routine r. If there is a well-defined standard action for the case **not** p, then the routine's precondition as given is too restrictive; its contract should be extended (renegotiated, if you like) to include the case **not** p, the "Special treatment" being moved from the individual clients to the routine itself.
- If, however, the "Special treatment" is different for various clients, then the individual test for p by every client is inevitable. Each has defined its own way of dealing with the case for which p is not satisfied.

The second possibility indeed occurs frequently, since in many cases a general-purpose supplier module simply lacks the proper context inability of many supplier modules to define the handling of abnormal cases, for lack of the proper context. How could a general-purpose *QUEUE* class know what to do when requested to remove an element from an empty queue, or a general-purpose graphics class know how to react when asked to display a circle on a screen without graphics

capabilities? Only the clients, in such situations, have enough context information to decide on the proper action.

1.3.6 Documenting a software contract

For the contract theory to work properly and lead to correct systems, we must provide client programmers with a proper description of the interface properties of a class and its routines.

Here assertions can play a key role since they help express the purpose of a software element such as a routine without reference to its implementation.

The **short** command of the Eiffel environment serves to document a class by extracting interface information. In this approach, software documentation is not treated as a product to be produced and maintained separately from the actual code; instead, it is the more abstract part of that code, and may be extracted by computer tools.

Command **short** will retain only the exported features of a class and, for an exported routine, will drop the routine body and any other implementation-related details. However pre- and postconditions are kept. (So is the header comment if present.) For example **short** yields the following for the *put* routine:

```
put (element: T, key: STRING)
        -- Insert element with key key
    require
        count < capacity
    ensure
        count <= capacity;
        item (key) = element;
        count = old count + 1
```

This expresses simply and concisely the purpose of the routine, without reference to a particular implementation.

All documentation on Eiffel classes (for example the class specifications in the book on the Eiffel Library) is produced automatically in this fashion; for classes that inherit from others, **short**, as will be seen below, should be combined with another tool, **flat**.

1.4 CLASS INVARIANTS AND CLASS CORRECTNESS

Routine preconditions and postconditions could be added to any programming language supporting routines. More specific to an object-oriented context is the notion of class invariant, which is also needed to define what it means for a class to be correct.

1.4.1 Class invariants

A class invariant is a property that must be satisfied by all instances of the class, transcending particular routines. For example, all tables (instances of class *TABLE*) must satisfy:

0 <= *count* <= *capacity*

This is a typical invariant property, which in Eiffel appears in the **invariant** clause of a class:[1]

class *TABLE* [*T*] **feature**

... Attribute and routine declarations for
put, item, delete, count, capacity, ...

invariant
0 <= *count* <= *capacity*
end -- class *TABLE*

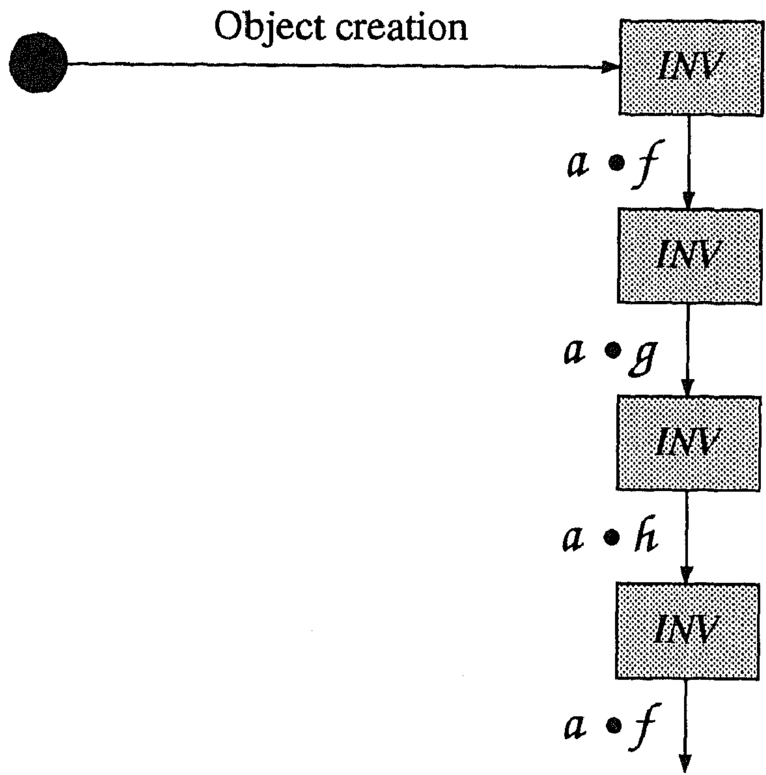

Figure 1.3: Object lifecycle

[1] Eiffel also supports another form of invariant, the loop invariant, which will not be studied any further in this chapter. See [8], and [9] for the theoretical background.

Two properties characterize a class invariant:

- It must be satisfied after the creation of every instance of the class (every table in this example). This means that every creation procedure of the class, called at object creation time, is required to ensure it.
- It must be preserved by every exported routine of the class (that is to say, every routine available to clients): any such routine must guarantee that the invariant is satisfied on exit if it was satisfied on entry.

In effect, then, the invariant is added to the precondition and postcondition of every exported routine of the class. But the invariant characterizes the class as a whole rather than its individual routines.

Figure 1.3 illustrates these requirements by picturing the lifecycle of any object as a sequence of transitions between "observable" states. The observable states, shown as rectangles, are the state immediately following object creation, and any state subsequently reached after the execution of an exported routine of the object's generating class. The invariant is the consistency constraint on observable states. Note that it is not necessarily satisfied in-between these states.

From the contract viewpoint, the invariant may be viewed as a general clause to be added to a group of contracts and constraining both the client (as it is added to the routine preconditions) and the contractor (as it must be preserved in the postconditions). Real-life contracts often contain such general clauses of the form "All provisions of the XX code shall apply to this contract".

1.4.2 Class correctness

With the above notions it is possible to define what it means for a class to be correct.

Correctness is always a relative notion, since no software element is correct or incorrect per se: the only practically useful notion is that of *consistency* with some specification. Preconditions, postconditions and invariants give that specification.

The following notation from program proving theory serves to make these ideas more precise. The formula written

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

where P and Q are assertions, and A a sequence of instructions, means: "If A is executed starting in a state in which P is satisfied, the resulting state will satisfy Q".[2]

Note that the person in charge of developing A, who may view the above formula as a job description, will prefer Q to be as weak as possible (limiting the ambition of the results to be achieved) and P to be as strong as possible (limiting

[2] This formulation does not address the question of whether A's execution indeed terminates. For a more rigorous discussion see [9].

the extent of cases to be covered). For client programmers, of course, the situation is reversed.

Also useful are the notations pre_r for the postcondition of a routine r, $post_r$ for its postcondition, do_r for its body (implementation), and *INV* for the class invariant. Then we can define the correctness of a class (that is to say, its consistency with its specification) as follows. Every creation procedure c must satisfy:

/1/

$$\{pre_c\}\ do_c\ \{INV\}$$

Furthermore, every exported routine r must satisfy:

/2/

$$\{pre_r\ \&\ INV\}\ do_r\ \{post_r\ \&\ INV\}$$

Rule /2/ shows well the dual role of the invariant which, for the implementer of the supplier class, is both "good news" and "bad news". The presence of *INV* on the left is good news since, as noted above, it limits the number of cases to be covered, restricting the routine's scope to consistent states (those satisfying the invariant). The occurrence on the right, however, is bad news since it requires the routine to restore the invariant on exit, in addition to ensuring the contract (the postcondition).

1.5 MORE ON ASSERTIONS

We now have a good view of the theoretical role of assertions, and the part they play in the design process for obtaining correct software on the basis of well-defined contracts.

Before we move on to further applications of the contract theory, in particular the handling of abnormal cases and the understanding of inheritance, we should answer the question that was set aside earlier: what happens if a system's execution violates an assertion at run time? This will also lead us to an important (although surprising at first) property: the "paradox of assertion semantics", and to a clarification of the assertion sublanguage and its limitations.

1.5.1 Monitoring assertions

What happens if, during execution, a system violates one of its own assertions?

In the Eiffel environment, the answer depends on a compilation option. For each class, you may choose between various levels of assertion monitoring, such as:

1 • No assertion checking at all.

2 • Preconditions only.

3 • Preconditions and postconditions.

4 • Preconditions, postconditions, invariants.

For a class compiled under option 1, assertions have no effect on system execution. With option 4, every routine call or return triggers a check: precondition and invariant (call), postcondition and invariant (return). Option 2 causes checks for preconditions only. Option 2 is the default.

The effect of an assertion violation (under the last three options) is to raise an exception. Section 1.8 below explains what behavior will result from an exception raised during the execution of a system.

1.5.2 Why monitor?

It should be clear from the preceding discussion that an assertion violation is **not** a special but expected situation that is meant to be handled in a particular way (such as out-of-range user input). For such cases, habitual control structures such as the if-then-else are entirely appropriate. Rather, an assertion violation is always the consequence of an error of specification, design or implementation – in ordinary computer parlance, a bug.

Assertion monitoring, then, has only two applications:

- The most common use is simply debugging. Turning assertion checking on (at either the PRECONDITIONS or ALL_ASSERTIONS level) makes it possible to detect mistakes.
- A less frequent application (that some readers may find objectionable) is software fault tolerance. If a large system is released with the expectation that it may still contain errors, then assertion monitoring will serve to trigger an exception in such a case; the exception handling facilities described below may in certain cases be used to recover from the exception.

Let us concentrate on the first application, by far the most frequently and unquestionably useful. Assertions are ways to express assumptions about the properties that will hold at various stages of the software's execution (especially routine entry and return). In classical approaches to software construction, although programmers typically use many such assumptions in the design process, they remain informal and implicit. Here the assertion mechanism enables us to express them fully and explicitly, providing many productive checks in case of a buggy system – one that does not meet the assumptions.

This use of assertions is one of the important applications of the contract theory; it provides for a debugging, testing and quality assurance mechanism that is considerably superior to more traditional techniques, since (as opposed to "blind" testing) it is based on high-level consistency information provided by the developers.

1.5.3 Assertion monitoring and the software lifecycle

An assertion violation, it was said above, always reflects a bug. The contract theory indicates a different situation for preconditions and postconditions:

- A precondition violation shows a bug in the client: the calling routine did not observe its part of the deal.
- A postcondition violation shows a bug in the supplier: the called routine did not perform its task. (An invariant violation is also a supplier bug.)

The first case is particularly interesting in connection with the general software engineering strategy for building systems in the object-oriented method. As explained in more detail in [7], the recommended approach is to build successive "clusters" of classes in a bottom-up order, from more general (reusable) to more specific (application-dependent). When a developer or (more commonly) a project manager decides to release a cluster C for general use, this normally implies a high degree of confidence in its quality; in other words, a belief that no bugs remain in C. If this is the case and performance constraints suggest economizing on run-time checks, C may be distributed in a mode that does not monitor postconditions or invariants. Monitoring preconditions is still useful, however, since the next clusters to be produced in the cluster development cycle will initially be in a more tentative state and may contain errors, resulting in violations of preconditions in the trusted cluster C.

This is one of the reasons why PRECONDITIONS is the default compilation option. Another, more obvious one, is that it is a reasonable tradeoff between performance and safety. Checking just preconditions is usually much less of an overhead than checking everything. Yet it will suffice to avoid the bad consequences that could result from a client's bug in light of the policy defined in 1.3.4 ("Against defensive programming"): since suppliers do not protect themselves against violated preconditions, they may exhibit arbitrary behavior in such a case (the bottom-right box of contracts such as the one of Figure 1.2).

Pursuing again the contract metaphor, we may view the assertion monitoring mechanism as the authority, not bound to either clients or suppliers, that checks the proper observance of contracts: the Chamber of Commerce, perhaps, or the Better Business Bureau.

1.5.4 The paradox of assertion semantics

It may be shocking at first to see a compilation option (the option that governs the level assertion monitoring) producing a widely different result at execution time. It is generally accepted that compilation options may change various aspects of run-time behavior, for example the execution speed in the case of an optimizing option, but not the essential semantics of a system execution.

Closer analysis reveals, however, that this convention is indeed appropriate, although it does lead to an interesting paradox, which may be called the paradox of assertion semantics.

The reason for the paradox is that all reasonable definitions of the semantics of a programming language (an informal one written in English, or a formal specification using a method such as denotational or axiomatic semantics as e.g. in [9]) are written with the assumption that the programs being specified are correct. It would be very strange indeed to write the semantic specification of incorrect programs. In the case of assertions, however, it has been emphasized that a violation can only occur for an incorrect system. As a result, the semantics of a language such as Eiffel which includes an assertion mechanism need not take assertions into account: the language's semantic definition need not provide for evaluation of assertions at run-time, and even if it does, it does not matter what actions it specifies for when an assertion is violated, since in a correct system all assertions will always be satisfied!

The paradox comes from the observation that there is often no better way of finding out whether the system is correct than... to monitor its assertions.

We should not forget, however, that this is only a debugging facility, which should not affect the behavior of correct systems. In an ideal world, we should be able to use the assertions to **prove** the correctness of a class, using definitions /1/ and /2/ of ; then (assuming that the hardware and operating system are correct ...) we would not need any run-time monitoring of assertions. Until such a situation is reached, however, run-time checks provide a highly useful, if theoretically imperfect, alternative mechanism.[3]

1.5.5 The assertion language

Although the above examples gave a good idea of typical assertions, no formal definition has yet been given of what is permitted in an assertion clause.

Eiffel assertions are boolean expressions, with a few extensions such as the **old** notation. Since the whole power of boolean expressions is available, they may include function calls.

In some cases, one might want to use quantified expressions, of the form "For all x of type T, $p\ (x)$ holds" or "There exists x of type T, such that $p\ (x)$", where p is a certain boolean property. To include such properties in assertions, you will need to simulate them using function calls, which would provide loops to represent the quantifiers.

[3] Although to my knowledge no Eiffel proof system has been built at the time of this writing, a partial but useful multi-tiered proof system seems feasible thanks to the presence of the assertion mechanism and to the peculiar nature of Eiffel software development, with its heavy reliance on libraries. This appears to be a promising area for research.

Eiffel could be extended to include a full-fledged formal specification language, with first-order predicate calculus. As it stands, however, Eiffel is a programming language meant for practical software development, and the embedded assertion language is the result of an engineering tradeoff between different design goals: on the one hand, support for reliable software development; on the other hand, ability to generate efficient code and overall language simplicity.

In fact, first-order predicate calculus would not necessarily be sufficient: many properties, such as "the graph is acyclic" (a typical invariant clause) are not expressible in this framework, at least not in any simple way. In such a case a simple and clear boolean function that looks for cycles is just as convincing.

The use of functions – that is to say, computations – is not, of course, without its dangers. As opposed to routine bodies, which are software elements, by nature "prescriptive" and imperative, assertions should belong to the descriptive, mathematical world of specifications. By re-introducing software (functions) into the assertions, we let the imperative fox into the chicken coop.

In practice, this means that any function used in assertions must be of unimpeachable quality, avoiding any change to the current state, and any operation that could result in abnormal situations. In particular, for reasons that should be obvious, the assertion monitoring mechanism always disables itself temporarily when, as part of evaluating an assertion, it must call a function.

1.6 DEALING WITH ABNORMAL SITUATIONS

The preceding discussion provides a convenient basis for discussing a problem that plagues software developers: how to handle "abnormal" or "exceptional" cases. This notion is often defined only vaguely. Here we can provide a more precise definition: an abnormal case arises whenever one of the parties in a contract is unable to fulfil its obligations.

Exploring the implications of the contract theory will lead us to a better understanding of the notion of exception, and to a disciplined set of mechanisms for dealing with abnormal cases.

1.6.1 Honest contracting

Once the terms of every contract have been properly laid out, the role of each party is clear. In particular, the responsibility for ensuring the precondition rests with the client. (Shifting some of the responsibility to the routine would mean loosening the precondition; the extreme case is that of a routine with precondition **true**, for which all calls are correct.) Thus for any call in a correct software system:

- Either it should be demonstrable from the context of the call that the precondition will be always be satisfied.
- Or the call should be protected by a conditional construct or equivalent.

For example, a call to *put* should be of the form

```
if t.count < t.capacity then
        t.put (x)
else
        ... Deal with case of full table ...
end
```

unless it can be inferred from the context that the table may never be full at the point of the call.[4]

This straightforward "a priori" scheme is not always applicable. Even if it is possible to express the precondition formally, it may be impractical in some situations to require that clients test it before every call.

Consider for example a class *MATRIX*, with a function

solution (*b*: *VECTOR*): *VECTOR*

for solving linear equations. Here *a*.*solution* (*b*) is the solution of $ax = b$. The precondition for the solution to exist is that the matrix be regular, which may be expressed by a boolean function *regular* in class *MATRIX*. The a priori scheme would mean that any call of the form *a*.*solution* (*b*) in a client would have to be protected by a test for *a*.*regular*.

In practice, however, testing whether a matrix is regular is essentially the same problem as solving the equation. More precisely, a standard elimination algorithm used to solve the equation will detect at some step that the matrix is not regular (by finding a "pivot" that is zero or, in practice, too small). In other words, non-regularity is detected as a byproduct of attempting to solve the equation. Few programmers preoccupied with efficiency would accept to write code of the form

```
if a.regular then
        c := a.solution (b)
else
        ...
end
```

where *solution* repeats the job done by *regular*.

In such cases, standard control structures are still appropriate if an **a posteriori** scheme is used: attempt the operation, and then see whether it has succeeded. For

[4] In this example it may be preferable to add to the class a boolean-valued function *full* which indicates whether the table is full. Then the precondition becomes **not** *full*, and the property to be tested by the client becomes **not** *t*.*full*, which is more abstract and avoids the need to refer explicitly to *capacity* and *count*.

example, we may replace function *solution* by a procedure *solve*, such that *a.solve* (*b*) will try to solve the equation (*attempt_to_solve* might be a better name). Feature *regular* is now a boolean attribute, whose value is set by *solve*: it will be **true** if and only if the operation has succeeded. If so, *solve* will also have set the value of attribute *solution* to the value of the solution. The client scheme becomes:

```
a.solve;
if a.regular then ...
        The solution is available here as a.solution ...
else
        ... Non-regular case ...
end
```

1.6.2 When standard control structures do not suffice

Combined, a priori and a posteriori techniques cover most of the problems of dealing with abnormal cases. There remain three categories of situations, however, in which they are not sufficient:

1 • Operations whose applicability can only be ascertained by attempting execution.

2 • Frequent operations with small likelihood of failure.

3 • Software fault tolerance.

The first category covers operations for which, as in the a posteriori case, the only way to determine the operation's applicability is to try to carry it out; but if it is not applicable, such an attempt may result in disastrous events. Here are two typical cases:

- Arithmetic overflow: it is hardly possible to determine whether $a + b$ will be representable on the machine at hand without attempting to compute this sum, but in case of overflow this may trigger a fatal hardware or operating system event unless you have taken special precautions.
- Input and output: for example, to determine whether a disk write operation is possible, there is often no other way than to attempt the operation and see what happens.

As a further example, although it is possible to require clients of a file operation to test first whether the file exists, such a test is not fully trustworthy: between the time a client tests for (say) *f.readable* and the time it reads from f, some other client may have destroyed the file. This raises the more general question of adapting the contract theory to the context of concurrent programming, which is discussed in an article [10] presenting the Eiffel model for concurrent computation and describing the work currently being carried out in this area.

The second category covers frequent operations with infrequent failure. These are often basic operations, which we expect to succeed most of the time. Arithmetic operations, mentioned as part of the first category, are also representative of this

one; another example is object creation. We may consider such operations to have preconditions: for arithmetic operations, the mathematical result must fit in the machine's number system; for object creation, there must be enough free space available. In principle, then, we could require clients to perform the corresponding test before every operation. For example, every object creation would be written

```
if not_enough_space then
        special_treatment
else
        ... Actual creation ...
end
```

In such cases, however, the operations are so common that such explicit a priori checking, or some a posteriori variant, would make the software extremely complex.

The third category reflects the problem of software errors and fault-tolerant computing. You may have a system that you believe is correct, every call being executed under the proper precondition. Yet you know that you and the other developers are only human and may have left an error. If it leads to abnormal behavior at run-time, you still want to be able to detect it and, at the least, terminate the execution in an orderly fashion.

Standard control structures cannot fully handle situations in these three categories.

1.6.3 Traditional exception mechanisms

To deal with abnormal cases, language designers have introduced the notion of exception. Well-known languages offering such a facility include CLU and Ada. But the use of exceptions in such languages is much broader than implied by the above discussion.

One of the main applications of these exception mechanisms is to separate textually the treatment of normal and abnormal cases. The design of such language support for exceptions stemmed in part from a desire to avoid the pollution of program structure implied by the mixture of "useful" processing and handling of abnormal cases.

If you have exceptions are your disposal, you will treat an abnormal case not by the standard control structure

```
if something_wrong then
        handle abnormal case
else
        further_processing
end
```

or its "a posteriori" counterpart, but (using the Ada scheme as example) by

```
if something_wrong then
    raise an_exception
end;
further_processing
```

The effect of the **raise** instruction is to interrupt processing and pass control to another segment of the program. Because **raise** is a control structure, affecting the control flow, control is guaranteed never to reach *further_processing* if *something_wrong* was true.

With this technique normal cases (*further_processing*) are separated from abnormal ones, handled by exception handlers. An exception handler is a clause that may be attached to a block or routine, and has the form (again using Ada syntax)

```
exception
    when excep1 => action1;
    when excep2 => action2;
    ...
```

There may also be a branch **when others** => ... which will catch all exceptions not explicitly named.

When a **raise** *an_exception* instruction is executed, the closest appropriate handler will be invoked. This is the first handler in the dynamic chain (that is to say the sequence including the current block or routine, its caller, the caller's caller etc.) whose **exception** clause has a **when** branch listing *an_exception* or **others** as its left-hand side. The corresponding right-hand side will be executed, and control will return to the caller of the unit to which the selected handler belongs. If no unit in the dynamic chain is has an appropriate handler, the program as a whole fails, returning control to the operating system.

Apart from exceptions explicitly triggered by programs through **raise** instructions, the underlying hardware and operating system may raise predefined exceptions such as *NUMERIC_ERROR* and *STORAGE_ERROR*. As will be seen below, predefined exceptions are the most useful because they reflect low-level failures that programmers may not easily avoid by a priori checks.

But let us concentrate for the moment on programmer-raised exceptions. How useful are they? To find examples, I surveyed a number of commonly available Ada textbooks, as well as the Ada reference manual and literature on the CLU language, which has a different exception mechanism.[5] These references yielded several categories of exception usage.

1.6.4 Ada exceptions

A typical example of when *not* to use exceptions [13] presents a routine for computing a square root:[6]

```
sqrt (n: REAL) return REAL is begin
        if x < 0.0 then
                raise Negative;
        else
                normal_square_root_computation;
        end if;
exception
        when Negative =>
                put ("Negative argument");
                return;
        when others => ...
end sqrt
```

Here when a square root routine is erroneously applied to a negative argument, the routine prints an error message and ... returns to its caller! The caller has no way of knowing that anything out of the ordinary has happened.

This is a rather surprising treatment of an abnormal situation: continuing the computation as if nothing had happened, using meaningless values. Sure, an error message will printed somewhere, in an attempt to notify the poor user. Thinking of a realistic use of this routine, like trajectory computation in a missile control system, we can only wonder whether the general will see the message on the console before or after he is hit by the missile sent to the wrong side of the battlefield.

It may be unfair to attach too much significance to this example which, in its original context, was just meant to introduce the Ada language mechanisms for exception handling. But its very status of elementary programming example in a book intended to teach "software development" shows, better than any critic of the language could ever hope to do, the dangers of an ad hoc exception mechanism; if the elementary pedagogical examples are that scary, what then must uses of the mechanism look like in "real-world" Ada programs?

---

[5] The following texts were surveyed. ANSI and AJPO: *Military Standard: Ada Programming Language* (American National Standards Institute and US Government Department of Defense, Ada Joint Program Office), February 17, 1983, ANSI/MIL-STD-1815A- 1983. Grady Booch: *Software Engineering with Ada*, Benjamin/Cummings Publishing Co., Menlo Park (Calif.), 1983. A. Nico Habermann and Dewayne E. Perry: *Ada for Experienced Programmers*, Addison-Wesley, Reading (Mass.), 1983. Barbara Liskov and John Guttag: *Abstraction and Specification in Program Development*, MIT Press, Cambridge (Mass.), 1986. Sabina Saib: *Ada: An Introduction*, Holt, Rinehart and Winston, New York, 1985. Ian Sommerville and Ron Morrison: *Software Development with Ada*, Addison-Wesley, Wokingham (England), 1987. Putnam P. Texel: *Introductory Ada: Packages for Programming*, Wadsworth Publishing Company, Belmont (Calif.), 1986.

[6] In this and subsequent examples, minor changes have been made for consistency; they only affect letter case, identifier names and indentation. Also, in this particular example, the word *Non_positive*, used in the original, has been replaced by *Negative*.

The missing element, whose absence leads to the dangers so apparent in this example, is a sound notion of what is "normal" and what is "exceptional", as provided by the contract theory. Implementing this notion in a programming language requires an assertion mechanism. This is what Ada lacks.

1.6.5 More uses of exceptions

Other examples use exceptions in a way that appears less harmful but simply unnecessary.

A CLU-based discussion by Liskov and Guttag [5] considers the example of a function *search* which returns an index at which an element x appears in a list l. When you start from such a specification, you are faced with a definition problem: what should *search* return when x does not appear in l?

The solution retained by Liskov and Guttag is to write *search* as a function that returns an index if x occurs in l, and otherwise triggers an exception – which the caller must then handle.

The use of exceptions for such a simple example appears rather overblown. Exceptions should be reserved for truly exceptional run-time conditions that cannot be handled by standard techniques. This view is reinforced by the basic paper on the the CLU exception mechanism, written by some of the same authors [6], which states that it is acceptable for an implementation to sacrifice some performance in the handling of exceptional cases, provided that non-exceptional ones are handled efficiently. But why should unsuccessful search, hardly an uncommon case, not be subject to the same efficiency requirements as successful search?

Here, of course, exceptions are not needed. A standard technique is to return a special value in the abnormal case, say 0 if the range of valid indices for the list is 1..*count*. Other solutions rely on the notion of "active data structure" and "cursor" (see [8], chapter 9).

1.6.6 Handling abnormal cases

In other cases, exceptions are simply there because there is no notion of precondition or postcondition. In an extract from the Ada Reference Manual, which has served as inspiration for examples found in many Ada textbooks, a module implementing stacks (using an array called *space*) has a *pop* routine of the form

```
procedure pop (top: out ELEMENT) is
    begin
        if count = 0 then
            raise Stack_underflow
        end if;
        top := space (count);
        count := count – 1;
    end pop;
```

Clearly, the exception contained in this example (as in the square root example) corresponds to an unformulated precondition: *pop* should never be called on an empty stack.

Although the references surveyed contain a number of similar examples of *raising* such an exception, I have found no realistic examples showing how to *handle* it. Yet this is the really interesting problem!

Let us try to see how the exception could be handled.

It is improper to add an **exception** clause to *pop* and handle the exception locally, as the routine does not know what to do when it is called erroneously on an empty stack – in the same way that the above square root routine could not know how to deal with a negative argument. The responsibility lies with the clients.

In any significant system using stacks in several ways, such as a compiler, any useful treatment of the exception must be specific to each call to *pop*. This means that every routine calling *pop* must include an **exception** clause with a branch

[EXC]

```
    when Stack_underflow => ... Instructions to deal with empty stack ...
```

But does this really make any sense? There are only two possibilities: either the calling routine is indeed prepared to deal with empty stacks; or it includes no provision for empty stacks.

In the first situation the exception structure is inadequate: the "Instructions to deal with empty stack" are in a handler, away from the actual call, and lack the proper context to know what to do with an empty stack. It would have been considerably simpler and clearer to write the call using the standard a priori protection scheme:

[TEST]

```
    ...
    if "Stack empty" then
        ... Instructions to deal with empty stack ...
                    -- Note that here the proper context is available
    else
        pop (...)
    end
```

where the test for "Stack empty" is a function that any stack module will readily provide. With this formulation the exception will never occur for this call.

The other situation arises when there is indeed a possibility that the exception will occur; this means that the call has not been properly protected. But then, using [EXC] is inappropriate: if the programmer has not overlooked the possibility of an empty stack, then he could just as well have written [TEST]. When you discover that you have made an error (forgetting about possible empty stacks) and want to update the software so as to cancel the effect of that mistake, it would take a rather convoluted mind to conclude that the needed change is the addition of a new clause to recover from the resulting run-time failures! The right action is the obvious one: just fix the bug.

Too often the Ada mechanism lures programmers into believing that by just raising exceptions they can forget about awkward cases. But in the end this only makes the system either unsafe or more complicated.

How pleasant indeed our life would become if through some incantations we could make all special cases vanish, and free ourselves of any need for the if-then-elses of this world. Alas, the programmer is no Aladdin, and **raise** is no good genie.

1.6.7 Fault tolerance

There remains only one type of meaningful handling for an exception such as *Stack_underflow*: using the exception mechanism for fault-tolerant programming. This is the situation, mentioned above, in which you believe that your program is correct and the exception may never occur; but you are a cautious person, having perhaps seen too many examples of supposedly correct systems that were not so correct after all, and want to make sure that if a bug remains the system will end its operation in a clean state and produce meaningful error messages.

In this case the handler should re-raise the exception, so as to notify the caller. The exception will be propagated along the dynamic chain; the handler in the main program (the last unit in every dynamic chain) should print an error message and terminate execution.

This type of exception handling is supported by the parameterless form of the Ada **raise** instruction, which does not name an exception. A parameterless **raise** occurring in a handler simply re-raises the exception being handled. The exception clause of the above square root routine should of course have ended with such a **raise** rather than the unacceptable **return**.

The use of exceptions that we have just seen is a technique for coping with software errors. The handler is used as a last resort when all normal branches of the program are unable to cope with the situation. But the exception is not supposed to arise in normal operation of the program. This protective technique – handlers included for situations that should not occur if the program contains no error – seems legitimate in view of human imperfection.

1.6.8 Resumption

The above response to exceptions may be described as **organized panic**: cut your losses and terminate the current program unit.

Another legitimate form appears in some examples. Sometimes an exception is triggered because some operation was attempted and failed, but this failure is not necessarily fatal as in the previous case. It may be possible to fix the conditions that caused the failure and try again. This form of exception handling is known as **resumption**. It is not supported by the CLU mechanism; it may be programmed in Ada, but often (if one is to judge from the examples surveyed) through rather complicated control structures.

An example of resumption is provided by a routine that reads integer input from an interactive user. If the input is incorrect, the routine cannot obtain an integer, but it can prompt the user for a new value. Whenever possible, such examples should be implemented by standard control structures, of the form:

```
[LOOP]
    "Get user input";
    while "Input not correct" loop
        print ("Input must be an integer. Please enter again.");
        "Get user input"
    end
```

A difficulty arises, however, if "Get user input" is performed by an existing low-level routine that will fail if the user's input is incorrect. In Ada, the input routine will trigger an exception; the client may catch this exception and retry the operation.

Several of the Ada texts surveyed treat such an example, but they choose the case in which the input must be one among a small number of character strings (such as Y and N), for which the above structure, [LOOP], is adequate in any language.

Transposing the example to integers rather than characters yields the form shown on the next page. Here the program prompts the user at most five times; this is achieved through a **for** loop. Note the necessity to exit from the middle of the loop by an **exit** instruction. The final **raise** instruction is appropriate since the routine has been unable to correct the failure, and signals it to its caller in the hope that the caller can deal with it better.

```
for i in 1..5 loop
    print ("Enter an integer");
    begin
                -- An internal block is needed here
                -- to introduce a local exception handler
            get (answer);
                -- answer is an integer variable
            exit;       -- Leave the loop
    exception
            when DATA_ERROR =>
                if i < 5 then
                    print ("Input must be an integer. Please enter again.");
                else
                    raise;
                end if;
    end;
end for;
```

1.7 PRINCIPLES OF EXCEPTION HANDLING

The first requisite of a well-drawn contract is that a violation by either party should be easy for the other party to detect and prove.[7]

1.7.1 The first law of exception handling

The preceding discussion of Ada exceptions, which applies in part to the CLU model except for resumption, seems to point to a number of cases where exceptions do fill a need. But exceptions as they exist in such programming languages generation are subject to serious criticism.

The first criticism is not meant at the mechanism but at its use, or misuse. We have seen cases in which exceptions tend to be used although they are unneeded; standard control structures are much preferable.

The main criticism, however, is the danger of the mechanism, which stems from the absence of a precise methodological approach of software reliability, serving as a basis for the exception mechanism. In other words, the notion of contract is missing.

The exception mechanism of Ada and CLU is not a technique for handling errors; it is simply a control structure, allowing jumps of a rather bold nature since

---

[7] From: Nathan Rosenberg and L.E. Birdzell, Jr., *How the West Grew Rich: The Economic Transformation of the Industrial World*, Basic Books, New York, 1986.

their scope is only determined at run-time by the dynamic chain (whereas a goto, at least, is statically bound). This mechanism is not defined with respect to a precise view of what a routine is about. Without such a view, exceptions are too easily misused. Specifically, the mechanism violates the following principle:

> **First law of exception handling:** There are only two ways a routine call may terminate: either the routine fulfils its contract, or its fails to fulfil it.

This law may seem trivial at first sight; but it is violated by the Ada exception mechanism, as evidenced by the square root routine: when that routine is incapable of fulfilling its contract, it "returns" as if nothing had happened, not even notifying the caller that an abnormal case was encountered.

This possibility of an Ada routine to fail but "pretend" to its caller that everything just went fine is probably the most dangerous aspect of the mechanism. It defeats the whole purpose of routine calling, which is to get some specific job done. A client may be prepared to deal with a contractor that fails to do its job, but cannot accept a execution which appears to return when in fact it has not achieved its stated purpose.

1.7.2 The notion of exception and the second law of exception handling

The discussion so far yields a clear definition of one of the two basic concepts involved in understanding exception handling – failure:

> **Definition** (failure): A routine's failure is its inability to satisfy its contract.

The other notion is that of exception itself. Informally, an exception is an abnormal event occurring at run-time. We can now be more precise, however. At the beginning of this chapter, we considered a routine performing a task divided into subtasks:

my_task **is**
    do
        $subtask_1$;
        $subtask_2$;
        ...
        $subtask_n$;
    end -- *my_task*

Every one of these subtasks should have its own contract, and any one of them may fail to achieve that contract. Such an event is what causes an exception for *my_task*. In other words:

> **Definition** (exception): An exception in a routine's execution is the failure of any one of the actions performed by that execution.

This definition uses the word "action" although the above definition of failures applies to "routines". The difference is only a pragmatic one: in principle, we could consider that in the body of a routine all the actions (the $subtask_i$) are routine calls. In practice, some of these calls are to predefined actions such as arithmetic operations or object creations, and do not use the syntax of routine calls. From a theoretical perspective, however, these actions are equivalent to routines, with a well-defined contract; just as programmer-defined routines, they may fail to fulfil the contract.

The definition of exception reflects a corollary of the First Law, which may be phrased as follows:

> **Second law of exception handling**: A routine's failure must always cause an exception in the execution of the routine's caller.

As indicated above, this excludes the "dishonest" case in which the routine hides the failure from its caller. To tell that caller, it must trigger an exception.

1.7.3 Strategies for handling exceptions

It may seem at first that exceptions are not that different from failures. If a routine uses a certain strategy to achieve its contract, and one of the components of the strategy (one of the $subtask_i$) fails, doesn't this imply that the routine itself has failed?

Often – but not always. The difference comes from the possibility of resumption: the calling routine may have an alternative strategy for achieving its contract, to which it will resort if the first attempted one fails.

This yields the third and last law:

> **Third law of exception handling:** There are only two ways a routine may react as a result of an exception (that is to say, after a first strategy to fulfil its contract has not worked):
> - Put back the objects in a stable state, and make a new attempt, using the same or another strategy (**resumption.**)
> - Put back the objects in a stable state, give up on the contract, and report failure to the caller by triggering an exception (**organized panic.**)

In both cases the routine must first "clean up its act" by putting back any objects involved into a stable state. We know of course what this means in practice: restoring the invariant. This obligation, studied in more detail below, accounts for the "organized" part of the panic in the second case.

In the resumption case, the alternative strategy may in fact be the same as the original one. A typical example is the handling of an exception caused by some malfunction that may be temporary. For example, one of the Eiffel routines shown below attempts to transfer a messager over an unreliable communication line. If the transmission fails, the routine just tries again.

Translated into Ada terms, the above two laws imply a strict rule (violated by the square root example):

> **Ada exception rule:** The execution of every exception handler should end by either retrying the unit or executing a **raise** instruction.

Even if this rule is observed, however, the Ada exception mechanism is still too general. In particular, it yields a style that seems to require **exit** instructions and, in at least some resumption cases, gotos (as seen in the example of 1.8.7 below). This is all the more worrying that exceptions are already jumps themselves, and, as noted, fairly wild ones at that.

1.8 A DISCIPLINED EXCEPTION HANDLING MECHANISM

The preceding discussion shows the need for a more disciplined exception mechanism. It forms the rationale for the mechanism present in Eiffel.

Before presenting the mechanism, it is useful to repeat the methodological limitations on its use. Whenever possible, special cases should be handled by standard control structures – *not* exceptions. The exception handling facilities are meant for cases that elude these normal structures. The preceding discussion left only three such cases: operations whose applicability can only be determined by attempting them, hence risking failure; very frequent operations with infrequent failures; and fault-tolerant programming.

1.8.1 Causes for exceptions

In the execution of an Eiffel system, an exception may occur as a result of any of the following events:

In practice there are four types of exception:

1 • An explicit assertion is found to be violated: a precondition on routine entry, a postcondition on routine exit, an invariant at either time.

2 • A called routine fails.

3 • The hardware or operating system sends a signal as a result of some abnormal event such as numerical error, input-output error, user interrupt or memory exhaustion.

4 • An attempt is made to apply a routine to a non-existing object: in *x*.*f* (...), the fundamental operation in the Eiffel model of object-oriented programming, *x* is a a void reference, not attached to any object.

The exception handling mechanism, which follows directly from the contract theory, is very simple: two language keywords, **rescue** and **retry**, plus a class in the basic library, *EXCEPTIONS*, which is not part of the language proper and not indispensable for simple uses.

1.8.2 Rescue and retry

What happens when one of the above events causes an exception in the corresponding routine? The answer is a direct application of the above laws of exception handling. Only two responses make sense: resumption and organized panic.

To specify how a routine should behave after an exception, the routine's author may include a rescue clause, which expresses the alternate behavior of the routine. The rescue clause is triggered whenever an exception occurs during the execution of the routine. Execution is interrupted and the rescue clause is executed. The rescue clause contains one or more instructions; **retry** may be among them. Execution of the rescue clause terminates in one of the following two ways:

- If the rescue clause terminates without executing a **retry**, then the routine fails; it will report failure to its caller by triggering a new exception.
- If the rescue clause executes **retry**, then the body of the routine (**do** clause) is executed again.

This mechanism satisfies the three laws. When a routine detects that it is unable to fulfil its contract because an exception has arisen, it is physically prevented from hiding this fact from its client: it may only return either after one or more **retry** that lead to success, or by exiting from the rescue clause and signaling failure.

The rescue clause is similar to clauses that occur in human contracts, to allow for exceptional, unplanned circumstances.

In general, only a few routines in a system will have an explicit rescue clause. By default, any other routine is considered to have a rescue clause with a null effect, so that any exception occurring during an execution of the routine will cause failure. We will see in 1.8.5 that it is possible to override this default rescue behavior by a class-specific behavior..

This is all there is to the language mechanism. It is complemented by a class *EXCEPTIONS*, available in the basic Eiffel Class library, which provides some facilities for dealing with exceptions.

1.8.3 Examples

The integer reading routine seen above in Ada may be written in Eiffel as follows:

```
get_integer_from_user: INTEGER is
            -- Read an integer (allow user up to five attempts)
    local
            failures: INTEGER
    do
            Result := getint
    rescue
            failures := failures + 1;
            if failures < 5 then
                    message ("Input must be an integer. Please enter again.");
                    retry
            end;
    end -- get_integer_from_user
```

Result, in a function, is the predefined entity whose final value will be returned by the function; *failures* is declared as a local variable, initialized to zero at the beginning of any execution of the routine. (The initialization rules are part of the language definition.) After five attempts, the function fails, as is always the case when a rescue clause terminates other than by a **retry**.

Another example is adapted from one by Booch [2]. We want to compute the inverse of a real number x, or 0 if the inverse cannot be computed because x is too small. We assume that in this case an attempt to divide 1 by x would trigger a predefined (hardware or operating system) exception. Even though the specification seems simple, it is typical of problems that are almost impossible to solve without some form of exception handling mechanism. Here we may use a simple scheme:

```
quasi_inverse (x: REAL): REAL is
        -- 1/x if representable, 0 otherwise
    local
        division_attempted: BOOLEAN
    do
        if not division_attempted then
            Result := 1/x
        else
            Result := 0
        end
    rescue
        division_attempted := true;
        retry
    end -- quasi_inverse
```

Boolean local variables such as *division_attempted* are initialized to **false** on routine entry.

1.8.4 Discriminating between exceptions

The above rescue clauses do not attempt to discriminate between possible exceptions. For an exception other than arithmetic overflow (in the last example, if the interactive user types BREAK during the execution of the routine) you will probably want the routine to fail.

The *EXCEPTIONS* class from the Basic Eiffel Library provides a mechanism for such discrimination: it contains an attribute *exception* which yields the code of the last exception, and predefined constants such as *Numerical_error* and *Violated_assertion* which yield the codes of predefined exceptions. To guarantee that the **retry** will only be invoked in the proper case, *quasi_inverse* should be in a class inheriting from *EXCEPTIONS* and have its rescue clause rewritten as:

```
rescue
    if exception = Numerical_error then
        division_attempted := true;
        retry
    end
```

This way, any exception whose code is not *Numerical_error* will cause the routine to fail rather than return 0. The other examples may be similarly adapted.

Class *EXCEPTIONS* provides a number of other facilities for fine-tuning the exception mechanism. For example, in addition to the integer code *exception*, string attributes yield a character code for the last exception, a plain English explanation (which may be used to display a message), the names of the class and routine in which the exception occurred, the object identification etc. The class also introduces a procedure *raise* allowing programmers to trigger exceptions explicitly.

These facilities should be used with care – especially those which make it possible to ascertain the nature of an exception. As pointed out by Hoare in his Turing lecture [4]:

> *The danger of exception handling is that an "exception" is too often a symptom of some entirely unrelated problem. For example, a floating-point overflow may be the result of an incorrect pointer used some 43 seconds before; and that was due perhaps to programmer oversight, transient hardware fault, or even a subtle compiler bug.*

In most cases, the rescue clause should treat all exceptions alike; if it does test for individual types of exceptions, this should be because it is specifically meant for one of them, as with *Numerical_error* in the above example. It should not try to discriminate between many different cases; more generally, a rescue clause should be extremely simple and short. Otherwise the danger exists for the exception mechanism to follow the Ada path and be increasingly used as a substitute for standard control structures.

1.8.5 Rules on rescue clauses

The rescue clause of a routine describes a standby algorithm that is to be used when the primary algorithm, given in the body, fails to achieve the contract. The rescue clause does **not**, however, attempt to perform the original contract, as expressed by the postcondition; for if there was a way to achieve this contract in the presence of an exception, it should be included in the body.

All the rescue clause can do is to "patch things up" (for example, in a data base transaction, to undo any harmful effect of the aborted operation) and either fail or retry. In the former case, the rescue clause is still subject to a contract, albeit a reduced one. This contract does not require the rescue clause to achieve the routine's postcondition: again, this is not its job. Even though the routine call has failed, however, it is essential that the failed rescue clause should leave the corresponding object in a clean state. In Eiffel we know exactly what a "clean state" means for an object: it is a state in which the class **invariant** is satisfied.

We may deduce from these observations the contract which is imposed on any branch of rescue clause that does not end with a **retry**:

- Because an exception may occur at any step during execution of the routine, the branch may not make any assumption on the state in which it will be triggered. In other words, it must admit the weakest possible condition, **true**, as precondition.
- Because the branch of the rescue clause must leave the object in a "clean" state, it must admit the class invariant as postcondition.

This yields the formal requirement on such rescue clause branches:

$$\{\mathbf{true}\}\ rescue_r\ \{INV\}$$

This rule should be contrasted with rule /2/ (pages 14) on routine bodies (**do** clauses). A routine body must ensure not only the invariant but also the routine's postcondition as defined by the **ensure** clause.

> You may think of the body as the cook in a restaurant, and of the rescue clause as the fire brigade. The cook must serve meals *and* make sure that the restaurant does not burn. The fire brigade must return the restaurant to a non-burning state, but is not additionally required to serve meals to customers. The input requirements, however, are harder on the fire brigade: whereas the cook may expect to find the restaurant initially non-burning (invariant) and open (precondition), there is no such guarantee for the fire brigade, which may be called at any time, as reflected by the use of **true** as its precondition.

It was noted above that by default an absent rescue clause is equivalent to one with a null effect. But developers need the ability to override this default rescue behavior, since it does not guarantee that the invariant will be restored after a failure. The exact rule follows from this observation: a routine without an explicit rescue clause is considered to have an implicit clause of the form

rescue
    default_rescue

where *default_rescue* is a routine of class *ANY*, the "universal" library class which, as guaranteed by the language rules, is an ancestor of every possible class. The version of *default_rescue* in *ANY* has a null body; but it is possible to redefine *default_rescue* in any class *C* to prescribe some non-null behavior. Then if a failure occurs in the execution of a routine *r* of *C*, and *r* has no explicit rescue clause, the mechanism will trigger the specific *default_rescue*.

Clearly, a class author who suspects that exceptions may occur in routines of the class, and who does not want to write individual rescue clauses for each of them, should redefine *default_rescue* so as to ensure the invariant. In simple cases one of the creation procedures of the class may provide a ready-made implementation for *default_rescue* since (as seen in /1/, page 14) the contract of a creation procedure is precisely to ensure the invariant.

A branch of the rescue clause that ends with **retry** is subject to the same requirements as a branch leading to failure, but, in addition to the invariant, must also re-establish the routine precondition before resumption.

1.8.6 Checking the checker

The requirements on rescue clauses are reflected in the policy implemented by the Eiffel environment at run-time: to avoid infinite loops, the checking of assertions is turned off during the execution of rescue code (as it is during the evaluation of an assertion, which may contain calls to boolean functions). This is yet another reason to make sure that any rescue code (as well as any non-purely-applicative component

of an assertion) must be of unimpeachable quality. If it fails, there is no guarantee as to what will happen.

This requirement is not unrealistic. First, any checking method must assume that the checking mechanism itself is safe; when you allow auditors into a bank, or inspectors into a nuclear plant, you have no choice but to hope that they will not introduce anomalies. Second, rescue clauses and assertions should in practice be kept clean and simple, enabling easy manual verification that they will indeed work in all cases.

1.8.7 N-version programming

Our last example of exception handling will be one of resumption. Taken from Saib [12], it is an elementary case of "*n*-version programming" [1] – a method which seeks to attain better software reliability by using methods adapted from hardware engineering, relying on fault-tolerance and redundancy. Two or more teams are asked to implement an identically specified module; each version serves as standby if the other fails.

Regardless of one's judgment about this approach to software reliability, the example provides a good programming exercise. For purposes of comparison let us keep Saib's model, which keeps alternating between the two versions as long as one fails, although in practice it would seem more reasonable to stop if both attempts fail. Here is the Ada version:

```
procedure try is begin
    <<Start>>                -- Start is a label
    loop
        begin
            algorithm_1;
            exit; -- Algorithm 1 was successful
        exception
            when others =>
                begin
                    algorithm_2;
                    exit; -- Algorithm 2 was successful
                exception
                    when others => goto Start;
                end
        end
    end
end main;
```

The control structure necessary to achieve the result looks rather contorted: two blocks, two exception handlers, two exits from within a loop, and one goto which traverses two exception handlers, two blocks and a loop! This would be enough to bring "structured programming" back into fashion. A much simpler structure does

not appear possible with the Ada exception mechanism. Compare the Eiffel version (which is easy to adapt so as to try each algorithm at most once):

```
try is
        local
                even: BOOLEAN
        do
                if even then algorithm_2 else algorithm_1 end
        rescue
                even := not even; retry
        end -- main
```

The choice between the two versions is left to the reader's taste.

1.9 INHERITANCE AND DYNAMIC BINDING

The contracting paradigm has led us to a new approach to exception handling – which appears to be safer than existing approaches, while leading to simpler solutions in many cases.

Another application of this paradigm, which is particularly important for object-oriented design and programming, is to shed a new light on the concept of inheritance. The notions of redefinition and dynamic binding, in particular, are much better understood if we are able to associate a contract with every routine.

The results of the following discussion have played a central role in the design of Eiffel's inheritance mechanism.

1.9.1 Redefinition

Inheritance is a key aspect of object-oriented programming, permitting the definition of new classes from previously defined ones. A class that inherits from another has all the features (routines and attributes) defined in that class, plus its own.

An important technique associated with inheritance is **redefinition**. Often, when inheriting from a class, it is necessary to provide new implementations of some features. For example, an heir to the *TABLE* class sketched at the beginning of this paper could include a new definition of *put*, as follows:

```
class OTHER_TABLE [T] inherit
        TABLE
                redefine put end
feature
        put (element: T) is
                do
                        ... New implementation of the insertion operation ...
                end; -- put

        ... Other features ...
end -- class OTHER_TABLE
```

Redefinition is fundamental for reusability because in practice we can seldom afford to reuse a software component exactly as it stands: most of the time, some local adjustments are needed. Inheritance with redefinition provides the appropriate degree of flexibility, which has no equivalent in other approaches.

Redefinition is complemented by two other extremely powerful techniques: polymorphism and dynamic binding.

Polymorphism allows assignments of the form

ta := *o_ta*

where *ta* is of type *TABLE* and *o_ta* of type *OTHER_TABLE*. In Eiffel, which is a strictly typed language, this is possible only because *OTHER_TABLE* is a descendant (direct or indirect heir) of *TABLE*: the reverse assignment would be prohibited.

When a call of the form *t*.*put* (...) is executed, dynamic binding means that the operation to be executed depends on the run-time form of *ta*: the *TABLE* version will be executed by default, but the *OTHER_TABLE* version will be executed after the above assignment.

Dynamic binding is a fundamental technique of object-oriented programming and has a number of far-reaching implications for software reusability and extendibility. But it also carries potential risks: what is to prevent a descendant class (direct or indirect heir) from redefining *put* into a procedure that actually performs a deletion or some other operation?

1.9.2 Honest subcontracting

Without assertions and the notion of contracting, inheritance and dynamic binding may indeed be misunderstood and misused. Contracting provides the appropriate view: inheritance with redefinition means **subcontracting**. When, as a contractor, you are charged with a certain task, you do not always carry it out yourself; sometimes it it more convenient to turn to somebody else who can do the job *better* or *cheaper* or both.

This is exactly what happens with redefinition and dynamic binding: a routine subcontracts its actual implementation to a version better adapted to the run-time form of its target. For example, the general table insertion routine will subcontract to a different algorithm for tables of the *OTHER_TABLE* form. Presumably, this algorithm will be more efficient than the default in this case; this accounts for the "cheaper".

But an honest subcontractor is not permitted to do just anything he likes. If the original contractor is to fulfil the client's request properly, the subcontractor must be bound by the same contract. The subcontractor may not place higher demands on the client – require a 2-hectare plot of land, for example, where the original requirement was just 1 hectare; and he may not return less than was originally pledged – a 2-story building rather than the promised 3 stories, or one costing more than *n* francs.

These rules are readily translated into rules on the assertions of redefined routines. The precondition and postcondition of a routine must apply to its redefined versions in descendants. This is the basic constraint needed to harness the power of redefinition and dynamic binding.

The exact rule is more subtle. The assertions on the redefined routine do not need to be exactly the same as those of the original. As noted, the subcontractor may do the job "better" as well as cheaper. Here there are two ways one may do the job better:

- By accepting cases which would have been rejected by the original contractor.
- By returning a better result than initially agreed on.

For example the above subcontractor is certainly permitted to use a technique that will work on a half-hectare parcel, or to produce a building 4 stories or higher. For assertions, the rule is expressed as follows:

> **Redefinition rule:** In the redefinition of a routine, the precondition must be weaker than the original, and the postcondition must be stronger than the original.

In this definition, an assertion is said to be *stronger* than another if it implies it; for example $x > 3$ is stronger than $x > 1$. "Weaker" is the reverse notion. (More correct phrases would be "Stronger [Weaker] than or equal to".)

The possibility of strengthening the postcondition of a redefined routine is essential in practice, as a redefinition will generally use more specific properties of the descendant class, adding new properties to the result. For example, a descendant *ARRAY_TABLE* of class *TABLE*, using an array implementation, might have a new integer attribute *insertion_index*, now set by *put* to the value of the index at which the last insertion was made. The new postcondition will be

/B/

```
count <= capacity;
item (key) = element;
count = old count + 1;
        -- Below is the new clause:
array_item (last_index) = element
```

Here *array_item* (i) is the value of the i-th element of the array.

Another example of postcondition strengthening is the redefinition of a routine computing a certain mathematical function of the argument, say its cosine, within a certain precision ε. A redefinition is certainly permitted to provide a better approximation, say within $\varepsilon/2$ of the exact result. What of course it may *not* do is to decrease the precision of the result: the client is entitled to a precision of ε.

For preconditions, the situation is symmetric. The precondition of *put* was *count < capacity*. In this initial implementation, once a table fills up, clients cannot insert any more. This will be the case if *TABLE* relies on a fixed-size implementation. A descendant may introduce a mechanism which automatically resizes the table when it fills up. If there is a limit on the number of secondary blocks, for example ten times the size of the primary table, the new precondition is

```
count <= 11 * capacity
```

If the descendant fully removes size limitations, the precondition disappears altogether, or, formally, becomes **true**. Both cases are correct since they weaken the original precondition.

Here again the subcontractor does "better" than required from the original, this time by being less demanding on its clients: it accepts cases that the prime contractor would have rejected. What would not be acceptable is a *more* demanding subcontractor.

1.9.3 Assertions in redefinitions: the language rule

Since it would place an undue burden on compilers to check that the precondition of a redefined routine is weaker than the original and the postcondition stronger, Eiffel directly enforces the above principles through language rules.

In the redefined version of a routine, it is not permitted to have plain **require** and **ensure** clauses. Instead, the precondition and postcondition clauses, if any, must be of the form

```
require else
        new_pre
```

and

ensure then
new_post

These notations yield the following as new precondition and postcondition for the redefined version of the routine:

new_pre **or else** *original_precondition*

new_post **and then** *original_postcondition*

where **or else** and **and then** are the non-commutative versions of the "or" and "and" operators, which evaluate their second argument only if necessary.

With this rule, the postcondition clause for the redefinition of *put* mentioned above (see /B/) becomes simply

ensure then
array_item (*last_index*) = *element*

which is automatically "anded" with the original to yield the semantics of /B/.

Similarly, the new precondition for an improved version of *put* may be

require else
count <= 11 * *capacity*

In this example, the resulting precondition is

count <= *capacity* **or** else *count* <= 11 * *capacity*

which of course is equivalent to its second term. It is not impossible that a compiler could simplify such assertions, at least in simple cases such as this one (which assumes a supplementary assertion stating that *capacity* is positive).

1.9.4 Documentation

If a class includes redefined routines with new assertions, the question arises of giving the proper information to a reader of the class text. Clearly, the **require else** and **ensure then** clauses do not suffice in this case, so that the short form of the class will be insufficient.

This is in fact a consequence of a general problem raised by inheritance: one cannot fully understand a class without its ancestry. In Eiffel, the problem has a simple solution: flattening.

The **flat** command of the Eiffel environment reconstructs an inheritance-free version of a class, with every inherited feature copied from the appropriate ancestor; renaming and redefinition are of course taken into account, and the class invariant is expanded so as to accumulate all ancestors' invariant clauses. For a class that has parents, interface documentation is obtained by applying **short** not directly to the class text, but to the flattened version produced by **flat**.

As an obvious consequence of the redefinition rule, then, **flat** must expand the assertions of redefined routines, so as to take into account the original assertions

through the **or else** and **and then** operators. (Again, an advanced version of **flat** might use the laws of boolean algebra to simplify some of the resulting assertions.) The "flat-short" interface documentation will then show the correct precondition and postcondition.

1.9.5 Taking advantage of improvements

Informally, the redefinition rule expresses that the interface specification provided to clients by a redefined version must be better than the original through a stronger postcondition and a weaker precondition.

But you will have noted that the clients of the original contractor are in fact unable to make use of the enhancement offered by the subcontractor: client classes can only be written in reference to the original preconditions and postconditions. So even if they end up using the better algorithm thanks to dynamic binding, they can only rely on the original interface specification.

This means that a client will be able to benefit from the improved performance that a redefinition may yield (the "cheaper" part), but not from improved functionality (the "better" part). How useful, then, is it to provide a better precondition or postcondition?

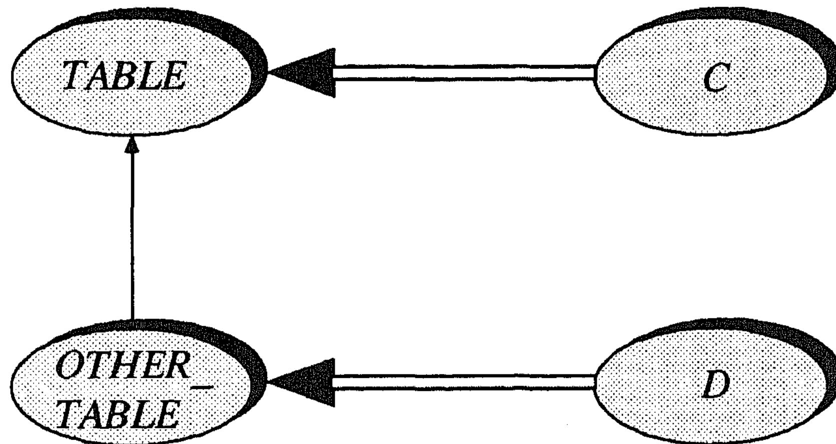

Figure 1.4: Clients and descendants

The answer is that the enhancement is indeed of no use to clients of the original contractors, but may be put to profit by direct clients of the subcontractor. Consider the situation illustrated by Figure 1.4 above (which uses the standard graphical conventions in Eiffel analysis and design, with single arrows for inheritance and double arrows for clients).

OTHER_TABLE is assumed to be a descendant of *TABLE* offering an enhanced interface for *put* (weaker precondition, stronger postcondition). *C* is a client of *TABLE*; it contains code of the form

```
ta: TABLE;
...
t.put (...)
```

Similarly, *D* is a client of *OTHER_TABLE*. The designer of *C* may only rely on the original assertions, even though at run-time dynamic binding will cause the *OTHER_TABLE* version of *put* to be executed if *ta* is attached to an object of type *OTHER_TABLE*. This will happen for example after an assignment

```
ta := o_ta
```

with *o_ta* of type *OTHER_TABLE*.

In other words, the contractors' and subcontractors' bureau, although honest – the result you get is always guaranteed to be at least as good as what you paid for, and can even be better – is also stingy: you are not guaranteed anything more than what you paid for.

There is, of course, a way for the client to benefit from the the subcontractor's improved services: bypass the original contractor and become a direct client of the subcontractor, without dynamic binding. In Figure 1.4, this means drawing a double arrow directly from *C* to *OTHER_TABLE*. In human contracts too, if you discover that your supplier uses the services of a subcontractor, you may sometimes decide to avoid the intermediary – a choice the original contractor will usually not like.

1.9.6 The horrors of static binding

To conclude this discussion of what the contracting theory brings to inheritance, it is appropriate to take a look at a technique which can only be characterized as a distortion of the principles of object-oriented programming, although, sadly enough, it is used in some languages that claim to be object-oriented (but, in an effort to protect the guilty, shall here remain unnamed).

As noted, redefinition and polymorphism lead to dynamic binding: when we apply an operation to an object through the notation *x.f*, we want to use the version of *f* that is directly adapted to the nature of the object. If the object is of type *D*, and *f*, coming from an ancestor *C* of *D*, has been redefined for *D*, then the *D* version should be applied.

That *x* may be declared of type *C* is irrelevant here: *x* is a polymorphic entity which may become attached to objects of various types (all descendants of *C*, such as *D*). This makes it possible for a client to write *x.f* without having to know what exact kind of object *x* will represent at execution-time. But this facility only makes sense precisely because the client has the guarantee that the **right** version of *f* will be applied in each case.

Static binding, implying that we apply the C variant, would be a gross mistake: a guarantee that we apply the **wrong** version!

The principles developed in this chapter provide a more theoretical perspective for the same arguments. Consider the requirements on an object's lifecycle, as illustrated by Figure 1.3 (page 12). A routine r, defined in a class C, must preserve the invariant INV_C of C (this is property /2/ on page 14):

$$\{pre_r \ \& \ INV_C\} \ do_r \ \{post_r \ \& \ INV_C\}$$

A version of r redefined in a descendant D of C must preserve the invariant INV_D of this new class, which is stronger than INV_C. Calling the redefined version s:

$$\{pre_s \ \& \ INV \ sub \ D\} \ do_s \ \{post_s \ \& \ INV_D\}$$

There is no reason, however, for the original do_r to preserve the stronger INV_D. In fact, class C need not know about any descendant that it may have.

Static binding, then, would mean the possibility of applying do_r to an object of type D. Since the C implementation is not required to preserve INV_D, this can produce an inconsistent object (one which does not satisfy its own class invariant), the worst possible situation in the execution of an object-oriented program, from which it is essentially impossible to recover (especially since no exception is triggered, the execution appearing to be normal).

The only argument that can be made in favor of static binding is one of performance: with static binding, there is no run-time overhead to look for the appropriate routine. But this argument does not make sense:

- Performance is never an excuse for executing a program incorrectly. If one drops the correctness requirement, it becomes *very* easy to write *very* fast programs.
- If properly implemented, dynamic binding can be quite cheap. A good implementation of Eiffel will find the needed routine in constant time (even in the presence of multiple inheritance), and with an overhead that remains small compared to the normal cost of routine call in any language.
- In some cases, it is appropriate to get rid of even this limited overhead. This occurs, for example, when a routine f is never redefined, or when an entity x is not polymorphic (that is to say, can become attached at run-time to objects of only one type). Then static binding or dynamic binding have the same semantics. But the detection of such situations, which requires a global system analysis, is the job of a computer, not of a human being! It is far too tedious and error-prone to be left to programmers. In ISE's Eiffel compiler, the optimizer performs this safe application of static binding to cases in which it is equivalent to dynamic binding.

1.9.7 Inheritance: assessment

The perspective provided by the contract theory seems necessary for a full understanding of the notion of inheritance. It may in fact contain the root for an axiomatic semantics of inheritance that would complement Cardelli's denotational specification [3].

Redefinition and dynamic binding are too often presented as clever techniques – almost as tricks – designed to make software more flexible. In the subcontracting metaphor developed here, these mechanisms take a precise and fruitful meaning. In particular, we have seen that routine redefinitions should not be arbitrary: they are constrained by the original assertions. It is the original designer's responsibility to choose assertions that are precise enough to attach a useful semantics to the routine throughout its avatars in descendants, yet leave enough room to future redefiners. **Redefinition is a semantics-preserving transformation.**

1.10 A PLEA FOR PARTIAL FUNCTIONS

The metaphor of programming as a contractual activity has led us to a number of important issues of software design: how to deal properly with abnormal cases; how to devise an exception mechanism that does not violate rules of systematic program construction; how to harness the power of inheritance.

One of the ideas guiding this discussion has been the inevitability of **possibly partial functions.**

In mathematics, a partial function is one which is not defined for some elements of its source set. Consider for example the inverse function *inv* on real numbers, viewed as a function in $\mathbf{R} \nrightarrow \mathbf{R}$ (where $\mathbf{R}$ is the set of real numbers, and $X \nrightarrow Y$ is the set of possibly partial functions with source set X and target set Y). Function *inv* is partial since it is not defined for the real number 0.[8]

In principle, we could always do without partial functions: if f is a function in $X \nrightarrow Y$ and the domain of f is A, a subset of X, we can consider f as a total function in $A \nrightarrow Y$. For example, *inv* is a total function in $\mathbf{R}^{\bullet} \nrightarrow \mathbf{R}$, where $\mathbf{R}^{\bullet}$ is the set of non-zero reals. This technique, however, complicates discussions of functions considerably since it leads to treating functions with different domains, such as *inv*, the square root function and the tangent function, as being of different "types".

In computing, routines are implementations of mathematical functions. Almost every specification of interest will include operations that are not always applicable;

---

[8] As in [9], we call "total" a function in $X \nrightarrow Y$ which is defined for all members of X, and "partial" a function which is not total, that is to say, such that for at least one member x of X x is not in the domain of f. "Function" without further qualifier means "possibly partial function" – that is to say, either partial or total.

even the most common "toy" example used in formal specification, stacks, has an operation *top* for which there is no reasonable default result when the operation is applied to an empty stack. This should not be surprising to anyone who has read this discussion so far: routine *top*, in a class representing stacks, will have a precondition other than **true**. Such a routine implements a mathematical function which is partial; it may itself be called a partial routine.

But partial routines are not popular. For example, one recent text on programming methodology [5] which, not surprisingly, promotes the CLU style of programming with its heavy reliance on exceptions, states that

Partial [*routines*] *lead to programs that are not robust.*

based on the obvious argument that such routines will not work for all calls.

But this argument neglects a fundamental aspect of software design: in the end, what makes a software system robust or not is not the greater or lesser tolerance of every individual routine. Once the system has been written, it contains only a fixed set of calls to each of its routines. So even if the routines are partial the problem of deciding whether all calls are correct is finite.

The robustness of the system is fundamentally affected, however, by the coherence of the structure, the consistency of module interfaces, and the simplicity of each individual module.

These goals are often met by accepting that the functions provided are partial, so that each program unit may do a well-defined job and do it well without having to check for a thousand different normality conditions, once it has been determined that the responsibility for establishing these conditions lies with the clients.

Liskov and Guttag, the authors of [5], rightly warn against the temptation

not to bother with the checks, or to use them only while debugging.

But in many cases there is a quite valid argument for omitting checks: simplicity of design. If the contracts are spelled out clearly, and a formal enough set of preconditions and postconditions is associated with the routines, I would venture the inverse warning: guard against the temptation to overcheck, which will lead to complex interfaces and over-ambitious techniques (such as unjustified uses of exceptions), and from there to decreased robustness – which in software is the almost inevitable consequence of undue complexity.

This view does leave a role for exceptions and recovery techniques, but only as a general mechanism that monitors the correct execution of contracts and, whenever possible, attempts to rescue clients and contractors from the failure of either party.

The approach developed in this chapter accepts partial functions as a fact of mathematical life and their counterparts, partial routines, as a fact of programming. Rejecting the elusive goal of building systems from components that would work under any possible circumstances, it prefers to aim at a more modest but perhaps more realistic principle: making sure that each component of a system, however humble and partial, states as clearly as possible what it will do, and what it will not do – which is what contracts are for.

REFERENCES

[1] Algirdas Avizienis, "The N-version aproach to Fault-Tolerant Software", *IEEE Transactions on Software Engineering*, vol. SE-11, no. 12, pp. 1491-1501, December 1985.

[2] Grady Booch, *Software Engineering with Ada*, Benjamin/Cummings Publishing Co., Menlo Park (Calif.), 1983 (new edition, 1986).

[3] Luca Cardelli, "A Semantics of Multiple Inheritance", in *Semantics of Data Types*, ed. Gilles Kahn, David B. McQueen and Gordon Plotkin, pp. 51-67, Springer-Verlag, Berlin-New York, 1984.

[4] C.A.R. Hoare, "The Emperor's Old Clothes", *Communications of the ACM*, vol. 21, no. 8, pp. 75-83, February 1981.

[5] Barbara Liskov and John Guttag, *Abstraction and Specification in Program Development*, MIT Press, Cambridge (Mass.), 1986.

[6] Barbara A. Liskov and Alan Snyder, "Exception Handling in CLU", *IEEE Transactions on Software Engineering*, vol. SE-5, no. 6, pp. 546-558, November 1979.

[7] Bertrand Meyer, "The New Culture of Software Development: Reflections on the Practice of Object-Oriented Design", in *D. Mandrioli and B. Meyer (eds.), Advances in Object-Oriented Software Engineering*, Prentice Hall, 1991. (This volume.), pp. 51-64.

[8] Bertrand Meyer, *Object-Oriented Software Construction*, Prentice Hall, 1988.

[9] Bertrand Meyer, *Introduction to the Theory of Programming Languages*, Prentice Hall, 1990.

[10] Bertrand Meyer, "Sequential and Concurrent Object-Oriented Programming", in *TOOLS 2 (Technology of Object-Oriented Languages and Systems)*, pp. 17-28, Angkor/SOL, Paris, June 1990.

[11] Bertrand Meyer, *Eiffel: The Language*, Prentice Hall, 1991.

[12] Sabina Saib, *Ada: An Introduction*, Holt, Rinehart and Winston, New York, 1985.

[13] Ian Sommerville and Ron Morrison, *Software Development with Ada*, Addison-Wesley, Wokingham (England), 1987.

APPENDIX: FURTHER SOURCES

The primary source and inspiration for this work is the research on program proving and systematic program construction pioneered by Floyd [C], Hoare [D] and Dijkstra [B].

The view of programs as computing partial functions plays an important part in the VDM method as presented in [E], which emphasizes the use of preconditions, postconditions and invariants.

The approach to inheritance presented here, and the use of assertions in an object-oriented language, appear specific to Eiffel; more details are given in [I].

Non-object-oriented languages that support assertions include Euclid [F] and Alphard [K]; see also the Ada-based specification language "Anna" [G]. CLU, cited in the text, includes non-formal assertions.

Another view of exceptions may be found in [A].

The notion of rescue clause bears some resemblance to Randell's recovery blocks [J], but the spirit and aims are different. Recovery blocks as defined by Randell are alternate implementations of the original goal of a routine, to be used when the initial implementation fails to achieve this goal. In contrast, a rescue clause does not attempt to carry on the routine's official business; it simply patches things up by bringing the object to a stable state. Any **retry** attempt uses the original implementation again. Also, recovery blocks require that the initial system state be restored before an alternate implementation is tried after a failure; this is hardly implementable in practice. No such provision is made with rescue clauses in Eiffel; the only requirement is that the rescue clause must restore the class invariant and, if resumption is attempted, the routine precondition.

As it exists in Eiffel, the notion of rescue clause actually derives from a corresponding formal notion of "surrogate function", also called "doppelgänger", in the specification method and language M [H]. M is a formal specification language, not an executable programming language like Eiffel. Functions in an M specification may be partial; a surrogate is associated with a partial function, and serves as a backup for arguments that do not belong to the domain of that function. It should be mentioned, however, that at the time of writing the design of M has not been fully ironed out.

[A] Flaviu Cristian, "On Exceptions, Failures and Errors", *Technology and Science of Informatics*, vol. 4, no. 1, January 1985.

[B] Edsger W. Dijkstra, *A Discipline of Programming*, Prentice Hall, Englewood Cliffs (N.J.), 1976.

[C] Robert W. Floyd, "Assigning Meanings to Programs", in *Proceedings American Mathematical Society Symposium in Applied Mathematics*, vol. 19, pp. 19-31, 1967.

[D] C.A.R. Hoare, "An Axiomatic Basis for Computer Programming", *Communications of the ACM*, vol. 12, no. 10, pp. 576-580, 583, October 1969.

[E] Cliff B. Jones, *Systematic Software Development Using VDM*, Prentice-Hall, Englewood Cliffs (N.J.), 1986.

[F] Butler W. Lampson, Jim J. Horning, Ralph L. London, J. G. Mitchell and Gerard L. Popek, "Report on the Programming Language Euclid", *SIGPLAN Notices*, vol. 12, no. 2, pp. 1-79, February 1977.

[G] David Luckham and Friedrich W. von Henke, "An Overview of Anna, a Specification Language for Ada", *IEEE Software*, vol. 2, no. 2, pp. 9-22, March 1985.

[H] Bertrand Meyer, "M: A System Description Method", Technical Report TRCS85-15, University of California, Santa Barbara, Computer Science Department, August 1986.

[I] Bertrand Meyer, *Object-Oriented Software Construction*, Prentice-Hall, 1988.

[J] Brian Randell, "System Structure for Software Fault Tolerance", *IEEE Transactions on Software Engineering*, vol. SE-1, no. 2, pp. 220-232, June 1975.

[K] Mary Shaw and others, *Alphard: Form and Content*, Springer-Verlag, Berlin-New York, 1981.