12

When the contract is broken: exception handling

*L*ike it or not, it is no use pretending: in spite of all static precautions, some unexpected and undesired event will sooner or later occur while one of your systems is executing. This is known as an exception and you must be prepared to deal with it.

12.1 BASIC CONCEPTS OF EXCEPTION HANDLING

The literature on exception handling is often not very precise about what really constitutes an exception. One of the consequences is that the exception mechanisms present in such programming languages as PL/I and Ada are often misused: instead of being reserved for truly abnormal cases, they end up serving as inter-routine **goto** instructions, violating the principle of Modular Protection.

Fortunately, the Design by Contract theory introduced in the preceding chapter provides a good framework for defining precisely the concepts involved.

Failures

Informally, an exception is an abnormal event that disrupts the execution of a system. To obtain a more rigorous definition, it is useful to concentrate first on a more elementary concept, failure, which follows directly from the contract idea.

A routine is not just some arbitrary sequence of instructions but the implementation of a certain specification — the routine's contract. Any call must terminate in a state that satisfies the precondition and the class invariant. There is also an implicit clause in the contract: that the routine must not have caused an abnormal operating system signal, resulting for example from memory exhaustion or arithmetic overflow and interrupting the normal flow of control in the system's execution.

It *must* refrain from causing such events, but of course not everything in life is what it must be, and we may expect that once in a while a routine call will be unable to satisfy its contract — triggering an abnormal signal, producing a final state that violates the postcondition or the invariant, or calling another routine in a state that does not satisfy that routine's precondition (assuming run-time assertion monitoring in the last two cases).

Such a case will be called a failure.

Definitions: success, failure

A routine call succeeds if it terminates its execution in a state satisfying the routine's contract. It fails if it does not succeed.

The discussion will use the phrase "routine failure", or just "failure", as an abbreviation for "failure of a routine call". Of course what succeeds or fails is not a routine (an element of the software text) but one particular call to that routine at run time.

Exceptions

From the notion of failure we can derive a precise definition of exceptions. A routine fails because of some specific event (arithmetic overflow, assertion violation…) that interrupts its execution. Such an event is an exception.

Definition: exception

An exception is a run-time event that may cause a routine call to fail.

Often an exception *will* cause failure of the routine. But you can prevent this from occurring by writing the routine so that it will catch the exception and try to restore a state from which the computation will proceed. This is the reason why failure and exception are different concepts: every failure results from an exception, but not every exception results in failure.

The study of software anomalies in the previous chapter introduced the terms *fault* (for a harmful execution event), *defect* (for an inadequacy of system, which may cause faults) and *error* (for a mistake in the thinking process, which may lead to defects). A failure is a fault; an exception is often a fault too, but not if its possible occurrence has been anticipated so that the software can recover from the exception.

See "Errors, defects and other creeping creatures", page 348.

Sources of exceptions

The software development framework introduced so far opens the possibility of specific categories of exception, listed at the top of the facing page.

Case E1 reflects one of the basic requirements of using references: a call $a.f$ is only meaningful if a is attached to an object, that is to say non-void. This was discussed in the presentation of the dynamic model.

"Void references and calls", page 240.

Case E2 also has to do with void values. Remember that "attachment" covers assignment and argument passing, which have the same semantics. We saw in the discussion of attachment that it is possible to attach a reference to an expanded target, the result being to copy the corresponding object. This assumes that the object exists; if the source is void, the attachment will trigger an exception.

"Hybrid attachments", page 263.

Definition: exception cases

An exception may occur during the execution of a routine *r* as a result of any of the following situations:

E1 • Attempting a qualified feature call *a.f* and finding that *a* is void.

E2 • Attempting to attach a void value to an expanded target.

E3 • Executing an operation that produces an abnormal condition detected by the hardware or the operating system.

E4 • Calling a routine that fails.

E5 • Finding that the precondition of *r* does not hold on entry.

E6 • Finding that the postcondition of *r* does not hold on exit.

E7 • Finding that the class invariant does not hold on entry or exit.

E8 • Finding that the invariant of a loop does not hold after the **from** clause or after an iteration of the loop body.

E9 • Finding that an iteration of a loop's body does not decrease the variant.

E10 • Executing a **check** instruction and finding that its assertion does not hold.

E11 • Executing an instruction meant explicitly to trigger an exception.

Case E3 follows from signals that the operating system sends to an application when it detects an abnormal event, such as a fault in an arithmetic operation (underflow, overflow) or an attempt to allocate memory when none is available.

Case E4 arises when a routine fails, as a result of an exception that happened during its own execution and from which it was not able to recover. This will be seen in more detail below, but be sure to note the rule that results from case E4:

Failures and exceptions

A *failure* of a routine causes an *exception* in its caller.

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

Cases E5 to E10 can only occur if run-time assertion monitoring has been enabled at the proper level: at least *assertion* (**require**) for E5, *assertion* (**loop**) for E8 and E9 etc.

Case E11 assumes that the software may include calls to a procedure *raise* whose sole goal is to raise an exception. Such a procedure will be introduced later.

Causes of failure

Along with the list of possible exception cases, it is useful for the record to define when a *failure* (itself the source of an exception in the caller, as per case E4) can occur:

Definition: failure cases

A routine call will fail if and only if an exception occurs during its execution and the routine does not recover from the exception.

We have yet to see what it means for a routine to "recover" from an exception.

The definitions of failure and exception are mutually recursive: a failure arises from an exception, and one of the principal sources of exceptions in a calling routine (E4) is the failure of a called routine.

12.2 HANDLING EXCEPTIONS

We now have a definition of what may happen — exceptions — and of what we would prefer not to happen as a result — failure. Let us equip ourselves with ways to deal with exceptions so as to avoid failure. What can a routine do when its execution is suddenly interrupted by an unwelcome diversion?

As so often in this presentation, we can get help towards an answer by looking at examples of how *not* to do things. Here the C mechanism (coming from Unix) and an Ada textbook will oblige.

How not to do it — a C-Unix example

The first counter-example mechanism (most notably present on Unix, although it has been made available on other platforms running C) is a procedure called *signal* which you can call under the form

signal (*signal_code*, *your_routine*)

with the effect of planting a reference to *your_routine* into the software, as the routine that should be called whenever a signal of code *signal_code* occurs. A signal code is one of a number of possible integers such as *SIGILL* (illegal instruction) and *SIGFPE* (floating-point exception). You may include as many calls to *signal* as you like, so as to associate different routines with different signals.

Then assume some instruction executed after the call to *signal* triggers a signal of code *signal_code*. Were it not for the *signal* call, this event would immediately terminate the execution in an abnormal state. Instead it will cause a call to *your_routine*, which presumably performs some corrective action, and then will … resume the execution exactly at the point where the exception occurred. This is dangerous, as you have no guarantee that the cause of the trouble has been addressed at all; if the computation was interrupted by a signal it was probably impossible to complete it starting from its initial state.

What you will need in most cases is a way to correct the situation and then **restart** the routine in a new, improved initial state. We will see a simple mechanism that implements this scheme. Note that one can achieve it in C too, on most platforms, by combining the *signal* facility with two other library routines: *setjmp* to insert a marker into the execution record for possible later return, and *longjmp* to return to such a marker, even if several calls have been started since the *setjmp*. The *setjmp*-*longjmp* mechanism is,

however, delicate to use; it can be useful in the target code generated by a compiler — and can indeed serve, together with *signal*, to implement the high-level O-O exception mechanism introduced later in this chapter — but is not fit for direct consumption by human programmers.

How not to do it — an Ada example

Here is a routine taken from an Ada textbook:

From Sommerville and Morrison, "Software Development with Ada", Addison-Wesley, 1987. Letter case, indentation, semicolon usage and the name of the floating-point type have been adapted to the conventions of the present book; Non_positive has been changed to Negative.

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

This example was probably meant just as a syntactical illustration of the Ada mechanism, and was obviously written quickly (for example it fails to return a value in the exceptional case); so it would be unfair to criticize it as if it were an earnest example of good programming. But it provides a useful point of reference by clearly showing an undesirable way of handling exceptions. Given the intended uses of Ada — military and space systems — one can only hope that not too many actual Ada programs have taken this model verbatim.

The goal is to compute the real square root of a real number. But what if the number is negative? Ada has no assertions, so the routine performs a test and, if it finds *n* to be negative, raises an exception.

The Ada instruction **raise** *Exc* interrupts execution of the current routine, triggering an exception of code *Exc*. Once raised, an exception can be caught, through a routine's (or block's) **exception** clause. Such a clause, of the form

```
exception
    when code_a1, code_a2, …=> Instructions_a;
    when code_b1, … => Instructions_b;
    …
```

is able to handle any exception whose code is one of those listed in the **when** subclauses; it will execute *Instructions_a* for codes *code_a1*, *code_a2*, … and so on for the others. One of the subclauses may, as in the example, start with **when others**, and will then handle any exception not explicitly named in the other subclauses. If an exception occurs but its code

is not listed (explicitly or through **when others**), the routine will pass it to its caller; if there is no caller, meaning that the failed routine is the main program, execution terminates abnormally.

In the example there is no need to go to the caller since the exception, just after being raised, is caught by the **exception** clause of the routine itself, which contains a subclause **when** *Negative => …*

But what then do the corresponding instructions do? Here they are again:

```
put ("Negative argument")
return
```

In other words: print out a message — a delicate thought, considering was happens next; and then return to the caller. The caller will not be notified of the event, and will continue its execution as if nothing had happened. Thinking again of typical applications of Ada, we may just wish that artillery computations, which can indeed require square root computations, do not follow this scheme, as it might direct a few missiles to the wrong soldiers (some of whom may, however, have the consolation of seeing the error message shortly before the encounter).

This technique is probably worse than the C-Unix *signal* mechanism, which at least picks up the computation where it left. A **when** subclause that ends with **return** does not even continue the current routine (assuming there are more instructions to execute); it gives up and returns to the caller as if everything were fine, although everything is *not* fine. Managers — and, to continue with the military theme, officers — know this situation well: you have assigned a task to someone, and are told the task has been completed — but it has not. This leads to some of the worst disasters in human affairs, and in software affairs too.

This counter-example holds a lesson for Ada programmers: under almost no circumstances should a **when** subclause terminate its execution with a **return**. The qualification “almost” is here for completeness, to account for a special case, the *false alarm*, discussed below; but that case is very rare. Ending exception handling with a **return** means pretending to the caller that everything is right when it is not. This is dangerous and unacceptable. If you are unable to correct the problem and satisfy the Ada routine’s contract, you should make the routine fail. Ada provides a simple mechanism to do this: in an **exception** clause you may execute a **raise** instruction written as just

```
raise
```

whose effect is to re-raise the original exception to the caller. This is the proper way of terminating an execution that is not able to fulfill its contract.

Ada Exception rule

The execution of any Ada exception handler should end by either executing a **raise** instruction or retrying the enclosing program unit.

Exception handling principles

These counter-examples help show the way to a disciplined use of exceptions. The following principle will serve as a basis for the discussion.

Disciplined Exception Handling principle

There are only two legitimate responses to an exception that occurs during the execution of a routine:

R1 •**Retrying**: attempt to change the conditions that led to the exception and to execute the routine again from the start.

R2 •**Failure** (also known as **organized panic**): clean up the environment, terminate the call and report failure to the caller.

In addition, exceptions resulting from some operating system signals (case E3 of the classification of exceptions) may in rare cases justify a **false alarm** response: determine that the exception is harmless and pick up the routine's execution where it started.

The classification of exception cases, including E3, is on page 413.

Let us do away first with the false alarm case, which corresponds to the basic C-Unix mechanism as we have seen it. Here is an example. Some window systems will cause an exception if the user of an interactive system resizes a window while some process is executing in it. Assume that such a process does not perform any window output; then the exception was harmless. But even in such case there are usually better ways, such as disabling the signals altogether, so that no exception will occur. This is how we will deal with false alarms in the mechanism of the next sections.

False alarms are only possible for operating system signals — in fact, only for signals of the more benign kind, since you cannot ignore an arithmetic overflow or an inability to allocate requested memory. Exceptions of all the other categories indicate trouble that cannot be ignored. It would be absurd, for example, to proceed with a routine after finding that its precondition does not hold.

So much for false alarms (unfortunately, since they are the easiest case to handle). For the rest of this discussion we concentrate on true exceptions, those which we cannot just turn off like an oversensitive car alarm.

Retrying is the most hopeful strategy: we have lost a battle, but we have not lost the war. Even though our initial plan for meeting our contract has been disrupted, we still think that we can satisfy our client by trying another tack. If we succeed, the client will be entirely unaffected by the exception: after one or more new attempts following the initial failed one, we will return normally, having fulfilled the contract. ("Mission accomplished, Sir. The usual little troubles along the way, Sir. All fine by now, Sir.")

What is the "other tack" to be tried on the second attempt? It might be a different algorithm; or it might be the same algorithm, executed again after some changes have been brought to the state of the execution (attributes, local entities) in the hope of preventing the exception from occurring again. In some cases, it may even be the original routine tried again without any change whatsoever; this is applicable if the exception was due to some

external event — transient hardware malfunction, temporarily busy device or communication line — which we do not control although we expect it will go away.

With the other response, *failure*, we accept that we not only have lost the battle (the current attempt at executing the routine body) but cannot win the war (the attempt to terminate the call so as to satisfy the contract). So we give up, but we must first ensure two conditions, explaining the use of "organized panic" as a more vivid synonym for "failure":

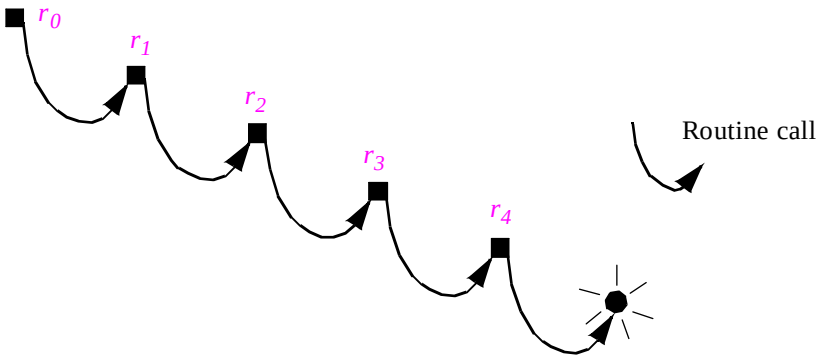
- Making sure (unlike what happened in the *sqrt* counter-example) that the caller gets an exception. This is the *panic* aspect: the routine has failed to live up to its contract.
- Restoring a consistent execution state — the *organized* aspect.

What is a "consistent" state? From our study of class correctness in the previous chapter we know the answer: a state that satisfies the invariant. We saw that in the course of its work a routine execution may temporarily violate the invariant, with the intention of restoring it before termination. But if an exception occurs in an intermediate state the invariant may be violated. The routine must restore it before returning control to its caller.

The call chain

To discuss the exception handling mechanism it will be useful to have a clear picture of the sequence of calls that may lead to an exception. This is the notion of call chain, already present in the explanation of the Ada mechanism.

The call chain



Let r_0 be the root creation procedure of a certain system (in Ada r_0 would be the main program). At any time during the execution, there is a *current routine*, the routine whose execution was started last; it was started by the execution of a certain routine; that routine was itself called by a routine; and so on. If we follow this called-to-caller chain all the way through we will end up at r_0. The reverse chain (r_0, the last routine r_1 that it called, the last routine r_2 that r_1 called, and so on down to the current routine) is the call chain.

If a routine produces an exception (as pictured at the bottom-right of the figure), it may be necessary to go up the chain until finding a routine that is equipped to handle the exception — or stop execution if we reach r_0, not having found any applicable exception handler. This was the case in Ada when no routine in the call chain has an **exception** clause with a **when** clause that names the exception type or **others**.

12.3 AN EXCEPTION MECHANISM

From the preceding analysis follows the exception mechanism that fits best with the object-oriented approach and the ideas of Design by Contract.

The basic properties will follow from a simple language addition — two keywords — to the framework of the preceding chapters. A library class, *EXCEPTIONS*, will also be available for cases in which you need to fine-tune the mechanism.

Rescue and Retry

First, it must be possible to specify, in the text of a routine, how to deal with an exception that occurs during one of its calls. We need a new clause for that purpose; the most appropriate keyword is **rescue**, indicating that the clause describes how to try to recover from an undesirable run-time event. Because the **rescue** clause describes operations to be executed when the routine's behavior is outside of the standard case described by the precondition (**require**), body (**do**) and postcondition (**ensure**), it will appear, when present, after all these other clauses:

```
routine is
        require
                precondition
        local
                … Local entity declarations …
        do
                body
        ensure
                postcondition
        rescue
                rescue_clause
        end
```

The *rescue_clause* is a sequence of instructions. Whenever an exception occurs during the execution of the normal *body*, this execution will stop and the *rescue_clause* will be executed instead. There is at most one **rescue** clause in a routine, but it can find out what the exception was (using techniques introduced later), so that you will be able to treat different kinds of exception differently if you wish to.

The other new construct is the retry instruction, written just **retry**. This instruction may only appear in a **rescue** clause. Its execution consists in re-starting the routine body from the beginning. The initializations are of course not repeated.

These constructs are the direct implementation of the Disciplined Exception Handling principle. The **retry** instruction provides the mechanism for retrying; a **rescue** clause that does not execute a **retry** leads to failure.

How to fail without really trying

The last observation is worth emphasizing:

> **Failure principle**
>
> Execution of a **rescue** clause to its end, not leading to a **retry** instruction, causes the current routine call to fail.

So if you have wondered how routines can fail in practice — causing case E4 of the exception classification — this is it.

See page 413.

As a special case, consider a routine which does *not* have a **rescue** clause. In practice this will be the case with the vast majority of routines since the approach to exception handling developed here suggests equipping only a select few routines with such a clause. Ignoring possible local entity declarations, arguments, precondition and postcondition, the routine appears as

```
routine is
        do
                body
        end
```

Then if we consider — as a temporary convention — that the absence of a **rescue** clause is the same thing as an empty rescue clause, that is to say

```
routine is
        do
                body
        rescue
                -- Nothing here (empty instruction list)
        end
```

the Failure principle has an immediate consequence: if an exception occurs in a routine without **rescue** clause it will cause the routine to fail, triggering an exception in its caller.

Treating an absent **rescue** clause as if it were present but empty is a good enough approximation at this stage of the discussion; but we will need to refine this rule slightly when we start looking at the effect of exceptions on the class invariant.

For the exact convention see "When there is no rescue clause", page 430.

An exception history table

If a routine fails, either because it has no **rescue** clause at all or because its **rescue** clause executes to the end without a **retry**, it will interrupt the execution of its caller with a "Routine failed" (E4) exception. The caller is then faced with the same two possibilities: either it has a **rescue** clause that can execute a successful **retry** and get rid of the exception, or it will fail too, passing the exception one level up the call chain.

If in the end no routine in the call chain is able to recover from the exception, the execution as a whole will fail. In such a case the environment should print out a clear description of what happened, the exception history table. Here is an example:

An exception history table

Object	Class	Routine	Nature of exception	Effect
O4	*Z_FUNCTION*	*split* (from *E_FUNCTION*)	Feature *interpolate*: Called on void reference.	Retry
O3	*INTERVAL*	*integrate*	*interval_big_enough*: Precondition violated.	Fail
O2	*EQUATION*	*solve* (from *GENERAL_EQUATION*)	Routine failure	Fail
O2	*EQUATION*	*filter*	Routine failure	Retry
O2	*MATH*	*new_matrix* (from *BASIC_MATH*)	*enough_memory*: Check violated.	Fail
O1 (root)	*INTERFACE*	*make*	Routine failure	Fail

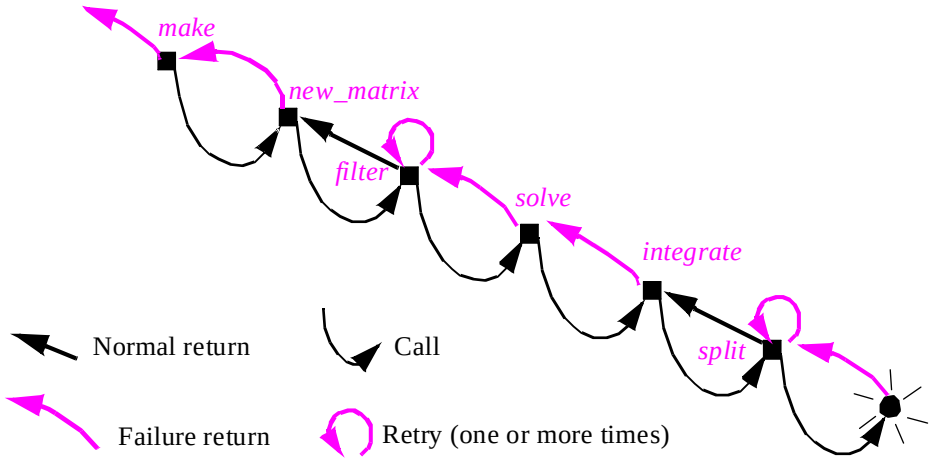
This is a record not only of the exceptions that directly led to the execution's failure but of all recent exceptions, up to a limit of 100 by default, including those from which the execution was able to recover through a **retry**. From top to bottom the order is the reverse of the order in which calls were started; the creation procedure is on the last line.

The **Routine** column identifies, for each exception, the routine whose call was interrupted by the exception. The **Object** column identifies the target of that call; here the objects have names such as O1, but in a real trace they will have internal identifiers, useful to determine whether two objects are the same. The **Class** column gives the object's generating class.

The **Nature of exception** column indicates what happened. This is where, for a assertion violation as in the second entry from the top, the environment can take advantage of assertion labels, *interval_big_enough* in the example, to identify the precise clause that was violated.

The last column indicates how the exception was handled: Retry or Fail. The table consists of a sequence of sections separated by thick lines; each section except the last led to a Retry. Since a Retry enables the execution to restart normally, an arbitrary number of calls may have occurred between two calls separated by a thick line.

Ignoring any such intermediate calls — successful and as such uninteresting for the purposes of this discussion — here is the call and return chain corresponding to the above exception history table. To reconstruct the action you should follow the arrows counter-clockwise from the call to *make* at the top left.



A failed execution

12.4 EXCEPTION HANDLING EXAMPLES

We now have the basic mechanism. Let us see how to apply it to common situations.

Fragile input

Assume that in an interactive system you need to prompt your system's user to enter an integer. Assume further that the only procedure at your disposal to read the integer, *read_one_integer*, leaving its result in the attribute *last_integer_read*, is not robust: if provided with something else than integer input, it may fail, producing an exception. Of course you do not want your own system to fail in that case, but since you have no control over *read_one_integer* you must use it as it is and try to recover from the exception if it occurs. Here is a possible scheme:

```
get_integer is
        -- Get integer from user and make it available in last_integer_read.
        -- If input initially incorrect, ask again as many times as necessary.
    do
        print ("Please enter an integer: ")
        read_one_integer
    rescue
        retry
    end
```

This version of the routine illustrates the retry strategy: we just keep retrying.

An obvious criticism is that if a user keeps on entering incorrect input, the routine will forever keep asking for a value. This is not a very good solution. We might put an upper bound, say five, on the number of attempts. Here is the revised version:

```
Maximum_attempts: INTEGER is 5
        -- Number of attempts before giving up getting an integer.
get_integer is
        -- Attempt to read integer in at most Maximum_attempts attempts.
        -- Set value of integer_was_read to record whether successful.
        -- If successful, make integer available in last_integer_read.
    local
        attempts: INTEGER
    do
        if attempts < Maximum_attempts then
            print ("Please enter an integer: ")
            read_one_integer
            integer_was_read := True
        else
            integer_was_read := False
            attempts := attempts + 1
        end
    rescue
        retry
    end
```

This assumes that the enclosing class has a boolean attribute *integer_was_read* which will record how the operation went. Callers should use the routine as follows to try to read an integer and assign it to an integer entity *n*:

```
get_integer
if integer_was_read then
    n := last_integer_read
else
    "Deal with case in which it was impossible to obtain an integer"
end
```

Recovering from hardware or operating system exceptions

Among the events that trigger exceptions are signals sent by the operating system, some of which may have originated with the hardware. Examples include: arithmetic overflow and underflow; impossible I/O operations; "illegal instruction" attempts (which, with a good object-oriented language, will come not from the O-O software but from companion routines, written in lower-level languages, which may overwrite certain areas of memory); creation or clone operations that fail because no memory is available; user interrupts (a user hitting the "break" key or equivalent during execution).

Theoretically you may view such conditions as assertion violations. If *a* + *b* provokes overflow, it means that the call has not observed the implicit precondition on the + function for integer or real numbers, stating that the mathematical sum of the two

arguments should be representable on the computer. A similar implicit precondition on the allocation of a new object (creation or clone) is that enough memory is available; if a write fails, it is because the environment — files, devices, users — did not meet the applicability conditions. But in such cases it is impractical or impossible to express the assertions, let alone check them: the only solution is to attempt the operation and, if the hardware or operating system signals an abnormal condition, to treat it as an exception.

Consider the problem of writing a function *quasi_inverse* which for any real number x must return either its inverse $\frac{1}{x}$ or, if that is impossible to compute because x is too small, the value 0. This type of problem is essentially impossible to solve without an exception mechanism: the only practical way to know whether x has a representable inverse is to attempt the division $\frac{1}{x}$; but if this provokes overflow and you cannot handle exceptions, the program will crash and it will be too late to return 0 as a result.

> On some platforms it may be possible to write a function *invertible* such that *invertible* (*x*) is true if and only if the inverse of *x* can be computed. You can then use *invertible* to write *quasi_inverse*. But this is usually not a practical solution since such a function will not be portable across platforms, and in time-sensitive numerical computations will cause a serious performance overhead, a call to *invertible* being at least as expensive as the inversion itself.

With the **rescue-retry** mechanism you can easily solve the problem, at least on hardware that triggers a signal for arithmetic underflow:

```
quasi_inverse (x: REAL): REAL is
            -- 1/x if possible, otherwise 0
        local
            division_tried: BOOLEAN
        do
            if not division_tried then
                Result := 1/x
            end
        rescue
            division_tried := True
            retry
        end
```

The initialization rules set *division_tried* to false at the start of each call. The body does not need any **else** clause because these rules also initialize *Result* to 0.

Retrying for software fault tolerance

Assume you have written a text editor and (shame on you) you are not quite sure it is entirely bug-free, but you already want to get some initial user feedback. Your guinea pigs are willing to tolerate a system with some remaining errors; they might accept for example that once in a while it will be unable to carry out a command that they have requested; but

they will not use it to enter serious texts (which is what you want them to do, to test your editor under realistic conditions) if they fear that a failure may result in a catastrophe, such as brutal exit and loss of the last half-hour's work. With the Retrying mechanism you can provide a defense against such behavior.

Assume that the editor, as will usually be the case, contains a basic command execution loop of the form

```
from … until exit loop
        execute_one_command
end
```

where the body of routine *execute_one_command* is of the form

```
"Decode user request"
"Execute appropriate command in response to request"
```

Chapter 21.

The "Execute…" instruction chooses among a set of available routines (for example delete a line, change a word etc.) We will see in a later chapter how the techniques of inheritance and dynamic binding yield simple, elegant structures for such multi-way decisions.

The assumption is that the different routines are not entirely safe; some of them may fail at unpredictable times. You can provide a primitive but effective protection against such an event by writing the routine as

```
execute_one_command is
            -- Get a request from the user and, if possible,
            -- execute the corresponding command.
        do
            "Decode user request"
            "Execute appropriate command in response to request"
        rescue
            message ("Sorry, this command failed")
            message ("Please try another command")
            message ("Please report this failure to the author")
            "Instructions to patch up the state of the editor"
            retry
        end
```

This scheme assumes in practice that the types of supported user request include "save current state of my work" and "quit", both of which had better work correctly. A user who sees the message *Sorry, this command failed* will most likely want to save the results of the current session and get out as quickly as possible.

Some of the routines implementing individual operations may have their own **rescue** clauses, leading to failure (so that the above **rescue** clause of *execute_one_command* takes over) but only after printing a more informative, command-specific message.

N-version programming

Another example of retrying for software fault tolerance is an implementation of the "N-version programming" approach to improving software reliability.

N-version programming was inspired by redundancy techniques that have proved their usefulness in hardware. In mission-critical setups it is frequent to encounter redundant hardware configurations, where several devices — for example computers — perform an identical function, and an arbitrating mechanism compares the results, deciding for the majority in case of discrepancy. This approach guards against single-component failures and is common in aerospace applications. (In a famous incident, an early space shuttle launch had to be delayed because of a bug in the software for the *arbitrating* computer itself.) N-version programming transposes this approach to software by suggesting that for a mission-critical development two or more teams, working in environments as distinct as possible, should produce alternative systems, in the hope that errors, if any, will be different.

See A. Avizienis, "The N-Version Approach to Fault-Tolerant Software", IEEE Trans. on Soft. Eng., SE-11, 12, Dec. 1985, pp. 1491-1501.

This is a controversial idea; one may argue that the money would be better spent in improving the correctness and robustness of a single version than in financing two or more imperfect implementations. Let us, however, ignore these objections and refrain from any judgment on the idea itself, but see how the **retry** mechanism would support the idea of using several implementations where one takes over if the others fail:

```
do_task is
        -- Solve a problem by applying one of several possible implementations.
    require
        …
    local
        attempts: INTEGER
    do
        if attempts = 0 then
            implementation_1
        elseif attempts = 1 then
            implementation_2
        end
    ensure
        …
    rescue
        attempts := attempts + 1
        if attempts < 2 then
            "Perhaps some instructions to reset to stable state"
            retry
        end
    end
```

The generalization to more than two alternative implementations is immediate.

This example is typical of the use of **retry**. The rescue clause *never* attempts to reach the original goal using a substitute implementation; reaching this goal, as expressed by the postcondition if there is one, is the privilege of the normal body. Note that after two attempts (or *n* in the general case) the routine simply executes its **rescue** clause to the end and so fails.

Let us look more closely at what happens when an exception is triggered during the execution of *r*. The normal execution (the body) stops; the rescue clause is executed instead. Then two cases may occur:

- The rescue clause may execute a **retry**, usually after some other instructions. In this case, execution of the routine will start anew. This new attempt may succeed; then the routine will terminate normally and return to its client. The call is a success; the contract has been fulfilled. Execution of the client is not affected, except of course that the call may have taken longer than normal. If, however, the retry attempt again causes an exception, the process of executing the rescue clause will start anew.

- If the rescue clause does not execute a **retry**, it will continue to its end. (This happens in the last example when *attempts >= 2*.) In this case the routine fails: it returns control to its caller, signaling an exception. Because the caller gets an exception, the same rule determines how its own execution continues.

This mechanism strictly adheres to the Disciplined Exception Handling principle: either a routine succeeds, that is to say its body executes to the end and satisfies the postcondition, or it fails. When interrupted by an exception, you may either report failure or try your normal body again; in no way can you exit through the rescue clause and pretend to your caller that you succeeded.

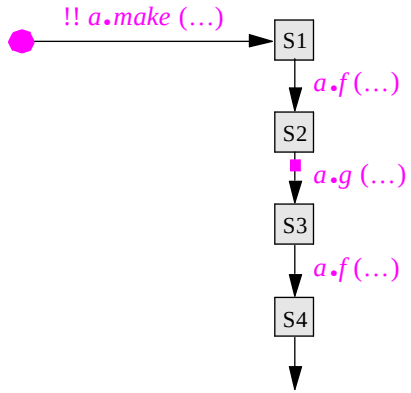
12.5 THE TASK OF A RESCUE CLAUSE

The last comments get us started towards a better understanding of the exception mechanism by suggesting the theoretical role of rescue clauses. Some formal reasoning will help us obtain the complete picture.

The correctness of a rescue clause

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

The formal definition of class correctness stated two requirements on the features of a class. One (C1) requires creation procedures to start things off properly. The other (C2), more directly relevant for the present discussion, states that to satisfy its contract, every routine, started with its precondition and the class invariant both satisfied, must preserve the invariant and ensure its postcondition. This was illustrated by the diagram depicting the typical object lifecycle:



The life of an object

(*Original page 366.*)

The formal rule read:

> C2 • For every exported routine r and any set of valid arguments x_r:
>
> $\{pre_r\ (x_r)$ **and** $INV\}\ \ Body_r\ \ \{post_r\ (x_r)$ **and** $INV\}$

This rule appeared on page 371.

where pre_r is the precondition, INV the class invariant, $Body_r$ the body of the routine, and $post_r$ the postcondition. To keep things simple let us ignore the arguments x_r.

Let $Rescue_r$ be the rescue clause of a routine, ignoring any branch that leads to a **retry** — that is to say keeping only those branches that will result in failure if executed. Rule C2 is a specification of the body $Body_r$ of the routine, in terms of what initial states it assumes and what final states it can guarantee. Can we obtain a similar specification for $RESCUE_r$? It should be of the form

$$\{\ ?\ \}\ \ Rescue_r\ \ \{\ ?\ \}$$

with the question marks replaced by actual assertions. (Here it is useful to try answering the question for yourself before reading on: how would you fill in the question marks?)

Consider first the input assertion — the question mark on the left of $Rescue_r$. **Anything** non-trivial that we write there would be wrong! Remember the discussion of attractive job offers: for whoever implements the task A in $\{P\}\ A\ \{Q\}$, the stronger the precondition P, the easier the job, since a precondition restricts the set of input cases that you must handle. Any precondition for $Rescue_r$ would make the job easier by restricting the set of states in which $Rescue_r$ may be called to action. But we may not assume any such restriction since exceptions, by their very nature, may happen at any time. If we knew when an exception will happen, it probably would not be an exception any more. Think of hardware failures: we have no clue as to when a computer can start to malfunction. Nor do we know, in an interactive system, when a user will feel like hitting the “break” key.

See “Weak and strong conditions”, page 335.

So the only P assertion that we can afford here (to replace the question mark on the left) is the one that asserts nothing at all: $True$, the assertion that all states satisfy.

For a lazy $Rescue_r$ implementor — again in reference to the discussion of job offers in the previous chapter — this is bad news; in fact the precondition $True$ is always the worst possible news for a supplier, the case in which “the customer is always right”!

What about the output assertion (the Q)? As discussed earlier, a rescue clause that leads to a failure must, before returning control to its caller with an exception, restore a stable state. This means reestablishing the invariant.

Hence the rule that we seek, with no more question marks:

Correctness rule for failure-inducing rescue clauses

C3 • $\{True\}\ \ Rescue_r\ \ \{INV\}$

Similar reasoning yields the corresponding rule for any branch $Retry_r$ of the rescue clause leading to a **retry** instruction:

Correctness rule for retry-inducing rescue clauses

C4 • $\{True\}\ \ Retry_r\ \ \{INV\ \textbf{and}\ pre_r\}$

A clear separation of roles

It is interesting to contrast the formal roles of the body and the rescue clause:

C2 • $\{pre_r\ \textbf{and}\ INV\}\ \ Body_r\ \ \{post_r\ (x_r)\ INV\}$

C3 • $\{True\}\ \ Rescue_r\ \ \{INV\}$

The input assertion is stronger for $Body_r$: whereas the rescue clause is not permitted to assume anything at all, the routine's body may assume the precondition and the invariant. This makes its job easier.

The output assertion, however, is also stronger for $Body_r$: whereas the rescue clause is only required to restore the invariant, the normal execution must also ensure the postcondition — the official job of the routine. This makes its job harder.

These rules reflect the separation of roles between the body (the **do** clause) and the rescue clause. The task of the body is to ensure the routine's contract; not directly to handle exceptions. The task of the rescue clause is to handle exceptions, returning control to the body or (in the failure case) to the caller; not to ensure the contract.

As an analogy — part of this book's constant effort to provide readers not just with theoretically attractive concepts but also with practical skills that they can apply to the pursuit of their careers — consider the difficulty of choosing between two noble professions: *cook* and *firefighter*. Each has its grandeur, but each has its servitudes. A gratifying quality of the cook's job is that he may assume, when he shows up at work in the morning, that the restaurant is not burning (satisfies the invariant); presumably his contract does not specify any cooking obligation under burning circumstances. But with a non-burning initial state the cook must prepare meals (ensure the postcondition); it is also a component of his contract, although perhaps an implicit one, that throughout this endeavor he should maintain the invariant, if he can, by not setting the restaurant on fire.

The firefighter, for his part, may assume nothing as to the state in which he finds the restaurant when he is called for help at any time of day or night. There is not even any guarantee that the restaurant is indeed burning — no precondition of the form *is_burning*, or of any other form save for *True* — since any call may be a false alarm. In some cases, of course, the restaurant will be burning. But then a firefighter's only duty is to return it to a non-burning state; his job description does not require that he also serve a meal to the assembly of patiently waiting customers.

When there is no rescue clause

Having formalized the role of rescue clauses we can take a second look at what happens when an exception occurs in a routine that has no such clause. The rule introduced earlier — with a warning that it would have to be revised — stated that an absent rescue clause was equivalent to a present but empty one (**rescue end**). In light of our formal rules, however, this is not always appropriate. C3 requires that

$$\{True\}\ \ Rescue_r\ \ \{INV\}$$

If $Rescue_r$ is an empty instruction and the invariant *INV* is anything other than *True*, this will not hold.

Hence the exact rule. The class *ANY* — mother of all classes — includes a procedure

We will study ANY, whose features are present in all classes, in "THE GLOBAL INHERITANCE STRUCTURE", 16.2, page 580.

```
default_rescue is
            -- Handle exception if no Rescue clause.
            -- (Default: do nothing)
    do
    end
```

A routine that does not have a Rescue clause is considered to have one that, rather than being empty as first suggested, has the form

```
rescue
    default_rescue
```

Every class can redefine *default_rescue* (using the feature redefinition mechanism studied as part of inheritance in a later chapter) to perform some specific action, instead of the default empty effect defined in *GENERAL*.

Rule C3 indicates the constraint on any such action: starting in any state, it must restore the class invariant *INV*. Now you will certainly remember that producing a state that satisfies the invariant was also the role of the **creation procedures** of a class, as expressed by the rule labeled C1. In many cases, you will be able to write the redefinition of *default_rescue* so that it relies on a creation procedure.

See "The role of creation procedures", page 372. Rule C1 was on page 371.

12.6 ADVANCED EXCEPTION HANDLING

The extremely simple mechanism developed so far handles most of the needs for exception handling. But certain applications may require a bit of fine-tuning:

- You may need to find out the nature of the latest exception, so as to handle different exceptions differently.
- You may want to specify that certain signals should not trigger an exception.
- You may decide to trigger an exception yourself.

We could extend the language mechanism accordingly, but this does not seem the right approach, for at least three reasons: the facilities are needed only occasionally, so that we would be needlessly burdening the language; some of them (in particular anything that has to do with signals) may be platform-dependent, whereas a language definition should be portable; and when you select a set of these facilities it is hard to be sure that you will not at some later time think of other useful ones, which would then force a new language modification — not a pleasant prospect.

For such a situation we should turn not to the language but to the supporting library. We introduce a library class *EXCEPTIONS*, which provides the necessary fine-tuning capabilities. Classes that need these capabilities will inherit *EXCEPTIONS*, using the inheritance mechanism detailed in later chapters. (Some developers may prefer to use the client relation rather than inheritance.)

Exception queries

Class *EXCEPTIONS* provides a number of queries for obtaining some information about the last exception. You can find out the integer code of that exception:

```
exception: INTEGER
        -- Code of last exception that occurred
original_exception: INTEGER
        -- Original code of last exception that triggered current exception
```

The difference between *exception* and *original_exception* is significant in the case of an “organized panic” response: if a routine gets an exception of code *oc* (indicating for example an arithmetic overflow) but has no rescue clause, its caller will get an exception whose own code, given by the value of *exception*, indicates “failure of a called routine”. It may be useful at that stage, or higher up in the call chain, to know what the original cause was. This is the role of *original_exception*.

The exception codes are integers. Values for the predefined exceptions are given by integer constants provided by *EXCEPTIONS* (which inherits them from another class *EXCEPTION_CONSTANTS*). Here are some examples:

```
Check_instruction: INTEGER is 7
        -- Exception code for violated check
Class_invariant: INTEGER is …
        -- Exception code for violated class invariant
Incorrect_inspect_value: INTEGER is …
        -- Exception code for inspect value which is not one
        -- of the inspect constants, if there is no Else_part
Loop_invariant: INTEGER is …
        -- Exception code for violated loop invariant
Loop_variant: INTEGER is …
        -- Exception code for non-decreased loop variant
No_more_memory: INTEGER is …
        -- Exception code for failed memory allocation
Postcondition: INTEGER is …
        -- Exception code for violated postcondition
Precondition: INTEGER is …
        -- Exception code for violated precondition
Routine_failure: INTEGER is …
        -- Exception code for failed routine
Void_assigned_to_expanded: INTEGER is …
```

Since the integer values themselves are irrelevant, only the first one has been shown.

A few other self-explanatory queries provide further information if needed:

```
meaning (except: INTEGER)
        -- A message in English describing the nature of exceptions
        -- of code except
is_assertion_violation: BOOLEAN
        -- Is last exception originally due to a violated assertion
        -- or non-decreasing variant?
    ensure
        Result = (exception = Precondition) or (exception = Postcondition) or
                 (exception = Class_invariant) or
                 (exception = Loop_invariant) or (exception = Loop_variant)
is_system_exception: BOOLEAN
        -- Is last exception originally due to external event (operating system error)?
is_signal: BOOLEAN
        -- Is last exception originally due to an operating system signal?
tag_name: STRING
        -- Tag of last violated assertion clause
original_tag_name: STRING
        -- Assertion tag for original form of last assertion violation.
```

```
recipient_name: STRING
        -- Name of routine whose execution was interrupted by last exception
class_name: STRING
        -- Name of class including recipient of last exception
original_recipient_name: STRING
        -- Name of routine whose execution was interrupted by
        -- original form of last exception
original_class_name: STRING
        -- Name of class including recipient of original form of last exception
```

With these features a rescue clause can handle different kinds of exception in different ways. For example you can write it, in a class inheriting from *EXCEPTIONS*, as

```
rescue
    if is_assertion_violation then
        "Process assertion violation case"
    else if is_signal then
        "Process signal case"
    else
        ...
    end
```

or, with an even finer grain of control, as

```
rescue
    if exception = Incorrect_inspect_value then
        "Process assertion violation case"
    else if exception = Routine_Failure then
        "Process signal case"
    else
        ...
    end
```

quasi_inverse was on page 424.

Using class *EXCEPTIONS*, we can modify the *quasi_inverse* example so that it will only attempt the **retry** if the exception was overflow. Other exceptions, such as one generated when the interactive user presses the Break key, will not cause the retry. The instruction in the rescue clause becomes:

```
if exception = Numerical_error then
    division_tried := True; retry
end
```

Since there is no **else** clause, exceptions other than *Numerical_error* will result in failure, the correct consequence since the routine has no provision for recovery in such cases. When writing a rescue clause specifically to process a certain kind of possible exception, you may use this style to avoid lumping other, unexpected kinds with it.

How fine a degree of control?

One may express reservations about going to the level of specific exception handling illustrated by the last two extracts. This chapter has developed a view of exceptions as undesired events; when one happens, the reaction of the software and its developer is "I don't want to be here! Get me out as soon as possible!". This seems incompatible with exerting fine control, depending on the exception's source, over what happens in a rescue clause.

For that reason, I tend in my own work to avoid using detailed case analysis on exception sources, and stick to exception clauses that try to fix things if they can, and then fail or retry.

This style is perhaps too austere, and some developers prefer a less restricted exception handling scheme that makes full use of the query mechanisms of class *EXCEPTIONS* while remaining disciplined. If you want to use such a scheme you will find in *EXCEPTIONS* all that you need. But do not lose sight of the following principle, a consequence of the discussion in the rest of this chapter:

Exception Simplicity principle

All processing done in a rescue clause should remain simple, and focused on the sole goal of bringing the recipient object back to a stable state, permitting a retry if possible.

Developer exceptions

All the exceptions studied so far resulted from events caused by agents external to the software (such as operating system signals) or from involuntary consequences of the software's action (as with assertion violations). It may be useful in some applications to cause an exception to happen on purpose.

Such an exception is called a developer exception and is characterized by an integer code (separate from the general exception code, which is the same for all developer exceptions) and an associated string name, which may be used for example in error messages. You can use the following features to raise a developer exception, and to analyze its properties in a rescue clause:

```
trigger (code: INTEGER; message: STRING)
                -- Interrupt execution of current routine with exception
                -- of code code and associated text message.
developer_exception_code: INTEGER
                -- Code of last developer exception
developer_exception_name: STRING
                -- Name associated with last developer exception
```

```
        is_developer_exception: BOOLEAN
                -- Was last exception originally due to a developer exception?
        is_developer_exception_of_name (name: STRING): BOOLEAN
                -- Is the last exception originally due to a developer
                -- exception of name name?
            ensure
                Result := is_developer_exception and then
                                equal (name, developer_exception_name)
```

It is sometimes useful to associate with a developer exception a *context* — any object structure that may be useful to the software element handling the exception:

```
        set_developer_exception_context (c: ANY)
                -- Define c as the context associated with subsequent developer
                -- exceptions (as caused by calls to trigger).
            require
                context_exists: c /= Void
        developer_exception_context: ANY
                -- Context set by last call to set_developer_exception_context
                -- void if no such call.
```

These facilities enable a style of development that heavily relies on some software elements triggering exceptions that others will process. In one compiler that I have seen, the developers took advantage of this mechanism, in the parsing algorithm, to stick to a relatively linear control structure, recognizing the various elements of the input text one after the other. Such sequential treatment is only possible if the elements parsed are the expected ones; any syntactical error disrupts the process. Rather than complicating the control structure by adding possibly nested **if** … **then** … **else** constructs, the developers chose to raise a developer exception whenever encountering an error, then dealt with it separately in the calling routine. As hinted earlier, this is not my favorite style, but there is nothing inherently wrong with it, so the developer exception mechanisms are there for those who want them.

12.7 DISCUSSION

We have now completed the design of an exception mechanism for object-oriented software construction, compatible with the rest of the approach and resulting directly from the ideas of Design by Contract developed in the preceding chapter. Thanks in particular to the **retry** instructions the mechanism is more powerful than what you will find in many languages; at the same time it may appear stricter because of its emphasis on retaining the ability to reason precisely about the effect of each routine.

Let us explore a few alternative design ideas that could have been followed, and why they were not retained.

Disciplined exceptions

Exceptions, as they have been presented, are a technique to deal with erroneous conditions that may be arise at run time: assertion violations, hardware signals, attempts to access void references.

The approach we have explored is based on the contracting metaphor: under no circumstances should a routine pretend it has succeeded when in fact it has failed to achieve its purpose. A routine may only succeed (perhaps after experiencing some exceptions but recovering from them through one or more **retry**, unbeknownst to the client) or fail.

Exceptions in Ada, CLU or PL/I do not follow this model. Using the Ada model, and instruction

raise *exc*

cancels the routine that executed it and returns control to its caller, which may handle the exception *exc* in a special handler clause or, if it has no such handler, will itself return control to its caller. But there is no rule as to what a handler may do. Hence it is perfectly possible to ignore an exception, or to return an alternate result. This explains why some developers use this exception mechanism simply to deal with cases other than the easiest one for an algorithm. Such applications of exceptions really use **raise** as a **goto** instruction, and a fairly dangerous one since it crosses routine boundaries. In my opinion, they are abuses of the mechanism.

There have traditionally been two viewpoints on exceptions: many practicing programmers, knowing how essential it is to retain control at run time whenever an abnormal condition is detected (whether due to a programming error or to an unforeseeable hardware event, say numerical overflow or hardware failure), consider them an indispensable facility. Computing scientists preoccupied with correctness and systematic software construction have often for their part viewed exceptions with suspicion, as an unclean facility used to circumvent the standard rules on control structures. The mechanism developed above will, it is hoped, appeal to both sides.

Should exceptions be objects?

An object-oriented zealot (and who, having discovered and mastered the beauty of the approach, does not at times risk succumbing to zeal?) may criticize the mechanism presented in this chapter for not treating exceptions as first-class citizens of our software society. Why is an exception not an object?

One recent language, the object-oriented extension of Pascal for Borland's Delphi environment has indeed taken the attitude that exceptions should be treated as objects.

It is not clear that such a solution would bring any benefit. The reasoning is in part a preview of the more general discussion that will help us, in a later chapter, tackle the question "how do we find the objects and classes?" An object is an instance of an abstractly defined data type, characterized by features. An exception has some features, of

"No-command classes", page 729.
See exercise E12.2, page 438.

course, which we saw in class *EXCEPTIONS*: its type, which was given by an integer code; whether it is a signal, an assertion violation, a developer exception; its associated message if it is a developer exception. But these features are *queries*; in most classes describing true "objects" there should also be *commands* changing the objects' state. Although one might conceive of commands applicable to exception objects, for example to disarm an exception after it has occurred, this seems incompatible with reliability requirements. Exceptions are not under the control of the software system; they are triggered by events beyond its reach.

Making their properties accessible through the simple queries and commands of the class *EXCEPTIONS* seems enough to satisfy the needs of developers who want fine-grain access to the exception handling mechanism.

The methodological perspective

"DEALING WITH ABNORMAL CASES", 23.6, page 797.

A final note and preview. Exception handling is not the only answer to the general problem of *robustness* — how to deal with special or undesired cases. We have gained a few methodological insights, but a more complete answer will appear in the chapter discussing the design of module interfaces, allowing us to understand the place of exception handling in the broader arsenal of robustness-enhancing techniques.

12.8 KEY CONCEPTS INTRODUCED IN THIS CHAPTER

- Exception handling is a mechanism for dealing with unexpected run-time conditions.
- A failure is the impossibility, for a routine execution, to fulfill the contract.
- A routine gets an exception as a result of the failure of a routine which it has called, of an assertion violation, of an abnormal condition signaled by the hardware or operating system.
- It is also possible for a software system to trigger a "developer exception" explicitly.
- A routine will deal with an exception by either Retry or Organized Panic. Retry reexecutes the body; Organized Panic causes a routine failure and sends an exception to the caller.
- The formal role of an exception handler not ending with a **retry** is to restore the invariant — not to ensure the routine's contract, as that is the task of the body (the **do** clause). The formal role of a branch ending with **retry** is to restore the invariant and the precondition so that the routine body can try again to achieve its contract.
- The basic language mechanism for handling exceptions should remain simple, if only to encourage straightforward exception handling — organized panic or retrying. For applications that need finer control over exceptions, their properties and their processing, a library class called *EXCEPTIONS* is available; it provides a number of mechanisms for distinguishing between exception types, as well as for triggering developer-defined exceptions.

12.9 BIBLIOGRAPHICAL NOTES

[Liskov 1979] and [Cristian 1985] offer other viewpoints on exceptions. Much of the work on software fault tolerance derives from the notion of "recovery block" [Randell 1975]; a recovery block for a task is used when the original algorithm for the task fails to succeed. This is different from rescue clauses which never by themselves attempt to achieve the original goal, although they may restart the execution after patching up the environment.

[Hoare 1981] contains a critique of the Ada exception mechanism.

The approach to exception handling developed in this chapter was first presented in [M 1988e] and [M 1988].

EXERCISES

E12.1 Largest integer

Assume a machine that generates an exception when an integer addition overflows. Using exception handling, write a reasonably efficient function that will return the largest positive integer representable on the machine.

E12.2 Exception objects

Notwithstanding the skeptical comments expressed in the discussion section as to the usefulness of treating exceptions as objects, press the idea further and discuss what a class *EXCEPTION* would look like, assuming an instance of that class denotes an exception that has occurred during execution. (Do not confuse this class with *EXCEPTIONS*, the class, meant to be used through inheritance, which provides general exception properties.) Try in particular to include commands as well as queries.