This draft: Version 4.82-00-00, 21 September 02 12:15 (Zürich time)
Extracted from ongoing work on future third edition of "Eiffel: The Language".

17 Exception handling *(in progress)*

17.1 OVERVIEW

During the execution of an Eiffel system, various abnormal events may occur. A hardware or operating system component may be unable to do its job; an arithmetic operation may result in overflow; an improperly written software element may produce an unacceptable outcome.

Such events will usually trigger a signal, or exception, which interrupts the normal flow of execution. If the system's text does not include any provision for the exception, execution will terminate. The system may, however, be programmed so as to *handle* exceptions, which means that it will respond by executing specified actions and, if possible, resuming execution after correcting the cause of the exception.

This chapter presents the exception mechanism by explaining what conditions lead to exceptions, and how systems can be written so as to handle exceptions.

It also introduces the *EXCEPTION* Kernel Library class and some of its descendants, which provides tools for fine-tuning the exception mechanism.

When using the exception facility, remember to take its name literally. The constructs discussed in this chapter – Rescue clause, Retry instruction – are not control structures on a par with those of the previous chapter; they should be reserved for those unexpected cases which cannot be detected a priori. Complex algorithmic structures, if any, should appear in Routine_body parts, not in exception handlers. If your system has many sophisticated exception handling clauses, it is probably misusing the mechanism.

17.2 WHAT IS AN EXCEPTION?

Failure, exception

Under certain circumstances, the execution of a construct (such as an instruction) may be unable to proceed as defined by the construct's semantics. It is then said to result in a **failure**.

If, during the execution of a routine, the execution of one of the components of the routine's Body fails, this prevents the routine's execution from continuing the Body's execution normally; such an event is called an **exception**.

"Failure" is in fact the more primitive notion; an exception is the consequence of a failure.

See chapter 9 about assertions.

The following categories of exception may occur (be **triggered**) during the execution of a routine:

See below about routine failure.

1 • Assertion violation (in an assertion monitoring mode).

2 • Failure of a called routine.

See 23.8, page 505, on Call with a void target, and the table on page ====on assigning void to an expanded. Chapter 18 covers Creation.

3 • Using an entity with a void value for an operation which requires an object; examples include feature call, for which the target must be non-void, or assignment to an entity of expanded type, for which the source must be non-void.

4 • Impossible operation, such as a Creation instruction attempted when not enough memory is available, or an arithmetic operation which would cause an overflow or underflow in the computer's number system.

"Machine" means hardware combined with operating system. See 2.11, page====.

5 • Interruption signal sent by the machine for example after a user has hit the "break" key or the window of the current process has been resized. Events which conceptually fall under category 1, such as an attempt to read from a non-existent process, are often caught first by the machine in this way.

The software may raise an exception through procedure 'raise' in class EXCEPTIONS. See 5.11, page ====below.

6 • An exception raised by the software itself, through the facilities offered in the Kernel Library class *EXCEPTIONS*, as explained below.

These categories distinguish the manifestation of the exception, not its real cause. Causes of exceptions essentially boil down to two possibilities: an error (a bug) in the software, or the inability of the underlying machine to carry out a certain operation. Assertion violations are a clear example of the first cause – a correct program always satisfies its assertions at run time – whereas running out of memory for a Creation is an example of the second.

In a way, the second of these types of cause is a variant of the first: if systems never executed an operation without checking first that it is feasible, then a correct system would never run into an exception. But it would be hardly practical to have every Creation instruction preceded by a check for available space, or every addition preceded by a check that the result will fit in the machine's number system – assuming such checks were possible.

In an environment supporting virtual memory and a garbage collector, unsuccessful Creation only occurs when the system has exhausted virtual memory and the collector is unable to reclaim any space. See 20.16, page ====.

In cases like these, a priori checking is expensive, and only a small percentage of executions are likely not to pass the checks. These are the cases requiring exceptions – ways to detect an abnormal situation, and possibly recover from it, *after* it has occurred.

17.3 EXCEPTION HANDLING POLICY

What can happen after an exception? In other words, what can we do when the unexpected occurs?

To answer this question properly, we must remember that a routine or other software component is not just the description of some computation. What transcends that particular computation is the goal that it is meant to achieve – what in the Eiffel theory is called the **contract**. The component provides just one way to achieve the contract; often, other implementations are possible. For simple components the contract is defined by the language: for example the contract of a Creation instruction is to create an object, initialize its fields and attach it to an entity. For more complex components you may express the contract through assertions: for example, a routine's contract may be defined by a precondition, a postcondition, and the class invariant. Even if there are no explicit assertions, the contract implicitly exists, perhaps expressed informally by the routine's Header_comment.

See "Object-Oriented Software Construction" for more in-depth discussions of exception handling principles. References in appendix C.

If we want to remain in control of what our software does, we must concentrate on the notion of contract to define possible responses to an exception. The contract of a software component defines the observable aspects of its behavior, those which its clients expect. Any exception handling policy must be compatible with that expectation.

An exception is the occurrence of an event which prevents a component from fulfilling the current execution of its contract. An unacceptable reaction would be to terminate the component's execution and to return silently to the client, which would then proceed on the wrong assumption that everything is normal. Since things are *not* normal – the client's expectations were not fulfilled – such a policy would almost inevitably lead to disaster in the client's execution.

What then is an acceptable reaction? Depending on the context, only three possibilities make sense for handling an exception:

- A favorable albeit unlikely case is one in which the exception was in fact not justified. This is called the **false alarm**.
- When writing the component, you may have anticipated the possibility of an exception, and provided for an alternative way to fulfil the contract. Then the execution will try that alternative. This case is called **resumption**.
- If you have no way of fulfilling the contract, then you should try to return the objects involved into an acceptable state, and signal your failure to the client. This is called **organized panic**.

The language mechanism described below – Rescue clauses and Retry instructions – directly supports resumption and organized panic. The rather infrequent case of false alarm is handled through features of the Kernel Library class *EXCEPTIONS*.

These mechanisms are defined at the routine level. For components at a lower level, such as an instruction or a call, you have no language mechanism to specify potential recovery. This means that for an unsuccessful attempt at executing such a component (for example an attempt at object creation when there is not enough memory, or at feature call on a void target) only policy E3 is possible: the component's execution will fail immediately, causing an exception. The exception interrupts the last started routine, called the *recipient* of the exception:

Recipient of an exception

The recipient of an exception is the current routine at the time of the exception, that is to say, the last routine whose execution was started before the component failure that caused the exception.

Depending on the recipient routine and its class, the exception will be handled through one of the three techniques listed above.

See page ====for a precise definition of the "current routine"

For the rest of this chapter, then, the unit of discourse is the routine. Any exception has a recipient, which is a routine. By writing an appropriate Rescue clause, you may specify the routine's response as resumption or organized panic; through the appropriate calls to library features, you may in some cases proceed with the routine's execution after a false alarm.

In the case of an assertion violation, the rule for determining the recipient appears on page ====.

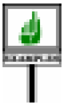
The next sections explain how to specify one of these three possibilities as your choice for exception handling.

17.4 RESCUE CLAUSES AND ORGANIZED PANIC

The construct which specifies a routine's response to exceptions that may occur during an execution of the routine is the Rescue clause.

This is an optional part of a Routine declaration, introduced by the keyword **rescue**.

Here is a sketch of a routine with a Rescue clause:



```
attempt_transaction (arg: CONTEXT) is
--Try transaction with arg; if impossible,
--reset current object
require
...
do
...
ensure
...
rescue
reset (arg)
end -- attemp_transaction
```

Any exception triggered during the execution of the Routine_body (**do**... clause) will cause execution of the Rescue clause. Here this clause calls procedure *reset*, meant to restore the current object to a stable state; such a state should satisfy the class invariant.

Termination of the Rescue clause also terminates the routine execution; in this case, however, as opposed to what would happen if the **do**.. clause was executed to the end with no exception, the call to *attempt_transaction* will fail. This is indeed the only way for a routine call to fail: being the recipient of an exception and executing its Rescue clause to the end, not ending with a Retry instruction (described below).

In other words, the routine illustrates the policy defined above as organized panic – put back the object in an acceptable state (satisfying the invariant) and terminate, notifying your caller, if any, of the failure. The technique used for this notification is to trigger a new exception, with the caller as recipient.

As noted, organized panic should restore the invariant. The formal version of this requirement, given below as part of the definition of exception correctness, is that any branch of a Rescue clause not terminating with a Retry should yield a state satisfying the invariant, independently of the state in which it is triggered.

As you may remember from the definition of class consistency, this requirement of ensuring the invariant also applies in another context: creation procedures of a class. This suggests that it is sometimes possible to write a Rescue clause as a call to a creation procedure, which will reset the object to a state which it could have reached just after creation. Of course, other situations may require more specific Rescue clauses, taking into account the routine that failed and the context of the failure.

Class consistency is defined on page ====.

17.5 THE DEFAULT RESCUE

In most systems, the vast majority of routines will not have an explicit Rescue clause. What happens if an exception is triggered during the execution of such a routine?

The convention a routine of a class *C* is considered, if it has no explicit Rescue clause, to have an implicit Rescue of the form ‘

```
rescue
    def_sec
```

where *def_resc* is the version of *default_rescue* in the enclosing class. Procedure *default_rescue* is introduced in the universal class *ANY*, where it is defined so as to have no effect:

On ANY, see 6.13, page 115, and chapter 32. The "version" of a routine in a descendant of its class of origin is the result of any redefinition and renaming that may have occurred along the inheritance path; see 11.12, page ====

```
default_rescue is
    do
    end -- default_rescue
```

Any developer-defined class, which is automatically a descendant of *ANY*, may redefine this routine to serve in case of organized panic. The redefined version will be called by any routine of the class which does not have a specific Rescue clause. Like any other routine, the redefined version is passed on to every heir, which will use it as default Rescue clause unless there is a new redefinition in the heir.

The reason for using the name *def_resc* rather than *default_rescue* in expressing the above equivalence is that in the process of inheriting from *ANY*, directly or indirectly, classes may rename features. For clarity, however, it is recommended to keep the original name *default_rescue*.

If, following the possibility suggested above, you use a creation procedure as default Rescue, you may rely on the following scheme, where *default_rescue* and the creation procedure are declared as synonym features:

Synonyms were discussed in 5.18, page 93. Recall that to redefine a feature from ANY you must explicitly list ANY It is also possible to undefine 'default_rescue' and rename it as 'make'.This, however, would lose the original name.

```
    class C creation
make, ... other creation procedures if any ...
    inherit
ANY
redefine default_rescue end
...
    feature
make, default_rescue is
-- No precondition
do
... Appropriate implementation;
... must ensure the invariant.
end; -- make, default_rescue
... Other features ...
    end -- class C
```

With this scheme, since *default_rescue* has no argument, there must also be no argument for the creation procedure chosen as synonym, here *make*.

The *default_rescue* convention explains what happens if a routine such as *attempt_transaction* above fails and its caller had no explicit Rescue. The caller will simply execute its version of *default_rescue* – which means doing nothing at all if it still has the original version inherited from *ANY* Then it will fail and trigger an exception in its client, which will itself be faced with the same situation. The effect of executing this Rescue chain all the way to the original root call will be described below.

'attempt_ transaction' was on page ====.

17.6 RETRY INSTRUCTIONS AND RESUMPTION

Sometimes you can do better than just conceding defeat and cutting your losses. This is where the Retry instruction is useful.

This instruction, which supports the resumption policy, may only appear in a Rescue clause. It has a very simple form, being just the keyword

```
retry
```

The effect of a Retry is to execute again the body of the routine. A Rescue clause which executes a Retry escapes failure – perhaps only temporarily, of course, since the body may again cause an exception.

Here is a general scheme that covers many uses of Retry. To solve a problem, you normally use method 1; if that method does not work, however, it may trigger an exception, and method 2 may yield the desired result.

```
try_once_or_twice is
-- Solve problem using method 1 or, if
unsuccessful, method 2
local
already_tried: BOOLEAN
do
if not already_tried then
method_1
else
method_2
end
rescue
if not already_tried then
already_tried := true;
retry
end
end -- try_once_or_twice
```

This example relies on the default initialization rules for local entities: *already_tried*, being of type *BOOLEAN*, is initialized to **false** on routine entry. This initialization is not repeated if the rescue block executes a Retry.

Local entity initialization is specified in the discussion of call semantics, 23.15, page 518.

If *method_2* triggers an exception, that is to say if both methods have failed, the Rescue clause will execute an empty Compound (since the Conditional has no Else_part). So the routine execution will fail, triggering an exception in the caller. This is because *try_once_or_twice* had two methods to reach a goal, and neither succeeded.

You may of course prefer a routine that behaves less dramatically when it cannot produce a result. Rather than sending an exception to the caller, it will just record the result in a boolean attribute *impossible* of the enclosing class: .

```
try_once_or_twice is
-- Solve problem using method 1 or, if
unsuccessful, method 2
    --if unsuccessful, method 2. Set
impossible to true
    --if neither method succeeded,
false otherwise.
local
already_tried: BOOLEAN
do
if not already_tried then
method_1
elseif not impossible then
method_2
end
rescue
if not already_tried then
impossible:= true;
    end;
    already_tried := true;
retry
end -- try_once_or_twice
```

This routine will never fail, since its Rescue clause always terminates with a Retry. This is not a paradox: the contract here is simply broader. As opposed to the contract for *try_once_or_twice*, it does not require the routine to solve the problem, but, more tolerantly, either to solve the problem (and set attribute *impossible* to true) or to set *impossible* to false if it is unable to solve the problem. Clearly, it is always possible to satisfy such a requirement; so there is no cause for failure.

You may easily generalize either version – *try_once_or_twice*, which may fail, and *try_and_record*, which never fails but sets a boolean success indicator – to try more than two alternative methods: just replace *already_tried* by a local entity *attempts* of type *INTEGER*, which will be initialized to zero.

As a special case, the resumption may in some situations simply amount to trying the same policy again. This applies when the exception was caused by an intermittent malfunction in external device, for example a busy communication line, or by an erroneous human input; by trying the line again, or outputting an error message asking the user to correct his input, you may hope to succeed. Here is the general scheme:

```
try_repeatedly_and_record is
		-- Attempt to solve problem in at most Maximum trials.
	local
		attempts: INTEGER
	do
		if attempts <= Maximum then
			attempt_to_solve
		else
			impossible := true
		end
	rescue
		attempts := attempts + 1
		... Other corrective actions, such as outputting
		an error message ...
		retry
	end
```

Maximum is a constant attribute with a positive value. The integer attribute *attempts* will be initialized to zero on routine entry.

The strategy used by *try_repeatedly_and_record* derives from *try_and_record* rather than *try_once_or_more*: if unable to perform its duty, it does not fail but simply sets attribute *impossible* to true. Adapting to the other style, which causes the routine to fail and trigger an exception in its caller, is easy and is left as an exercise.

17.7 SYSTEM FAILURE AND THE EXCEPTION HISTORY TABLE

In the organized panic case, a failed execution of a routine *r* triggers an exception in the caller. But what if there is no caller?

This can occur only if the execution that fails is the "original call": the execution of the root's creation procedure which started system execution. Remember that executing a system means creating an instance of its root class and applying a creation procedure to that instance. The creation procedure usually calls other routines, which themselves execute further calls. This means that any routine execution except the original call has a caller.

The semantics of system execution was defined on page ====.

A failure of the original call produces a **system failure**. Execution of the system terminates, producing an appropriate diagnostic about the system's inability to fulfil its task.

This rule does not just apply to exceptions triggered directly by the original call – an infrequent case since root creation procedures tend in practice to perform only simple actions before creating other objects or calling other routines. The more interesting case is the failure of a routine execution deep down in the call sequence, for which all direct and indirect callers eventually fail because they are not able to apply resumption. Then the failure bubbles up the call chain until it finally causes system failure.

This scenario in fact applies to the simplest case, in which no routine of the system has a Rescue clause, and no class redefines *default_rescue*: then any exception occurring during execution will propagate to the root's creation procedure, and result in a system failure.

What happens after a system failure? As noted, the tool that handles execution (the run-time system) should produce a diagnostic. The exact form of that diagnostic is not part of the language specification. Here is the format used in one particular implementation. After a system failure, that implementation prints an error message and an **exception history table** such as the following:

This is the format used by ISE's implementation. Others may use different conventions.

ObjectClassRoutine Nature of exceptionEffect
2FB44INTERFACEm_creation Feature "quasi_inverse": Called on void reference.Retry
2F188MATHquasi_inverse "positive_or_null": (from BASIC_MATH) Precondition violated.Fail

ObjectClassRoutine Nature of exceptionEffect
2F188MATHraise "*Negative_value": (from EXCEPTIONS)Developer exception.Fail*
2F188MATHfilter "*Negative_value*": *Developer exception.Retry*
2F32MATHnew_matrix "enough_memory": (*from BASIC_MATH*)*Check violated.Fail*
2FB44INTERFACEsetRoutine failureFail

For an exception whose recipient was a routine *r*, during a call on an object OBJ, the first column identifies OBJ (through an internal object identifier), the second column identifies the generating class of OBJ (the base class of its type), and the "Routine" column identifies *r*. The next column indicates the nature of the exception; for developer-defined exceptions and assertion violations this includes a tag (the Assertion_tag for assertions clauses). The last column indicates the effect of the exception: resumption (appearing as *Retry*) or organized panic (appearing as *Fail*).

The table contains not just a trace of the calls that led to the final failure but also the entire history of recent exceptions. Some exceptions may have been caught and handled through resumption, only to lead to further exceptions. This is why the exception history is divided into periods, each terminated by a Retry. The table shows these periods separated by a double line; exceptions appear in the order in which they occurred, which is the reverse of the order of the calls.

The case illustrated on the table, which resulted from a specially contrived system meant to illustrate the various possibilities – involving exceptions of many kinds, and resumptions that trigger new exceptions – is unusual. Exception handling in well-written systems should remain simple, and as much effort as possible should go into avoiding exceptions rather than handling them a posteriori. Exception handling does play a crucial role, however, for those hard to prevent cases which, in the absence of an appropriate exception mechanism, would leave defenseless the system, its users and its developers.

17.8 SYNTAX AND VALIDITY OF THE EXCEPTION CONSTRUCTS

It is time now to look at the precise properties of the two constructs associated with exceptions: rescue clauses of routines and retry instructions.

The grammar is straightforward: I

Rescue $\triangleq$ **rescue** Compound

Retry $\triangleq$ **retry**

A Rescue clause is part of a Routine. A Retry instruction is one of the choices for the Instruction construct.

See page ====for the syntax of Routine and page ====for the syntax of Instruction.

A constraint applies to Rescue clauses:

Rescue clause rule *CXRC*

It is valid for a Routine to include a Rescue clause if and only if its Routine_body is of the Internal form.

VXRC

An Internal body is one which begins with either **do** or **once**. The other possibilities are Deferred, for which it would be improper to define an exception handler since the body does not specify an algorithm, and an External body, where the algorithm is specified outside of the Eiffel system, which then lacks the information it would need to handle exceptions.

The various kinds of Routine_body are discussed in 8.6, page 148.

The constraint on Retry instructions has already been mentioned:

Retry rule *CXRT*

A Retry instruction is valid if and only if it appears in a Rescue clause.

VXRT

Because this constraint requires the Retry to appear physically within the Rescue clause, it is not possible for a Rescue to call a procedure containing a Retry. This means in particular that a redefined version of *default_rescue* may not contain a Retry. In other words, the default exception processing may not lead to resumption. The reason for this rule is one of simplicity and readability: outside of the Rescue clause to which it applies directly, a Retry would be little more informative than an arbitrary branch instruction.

17.9 SEMANTICS OF EXCEPTION HANDLING

To define the semantics of exception handling, it is convenient to consider that every routine has an implicit or explicit "rescue block":

Rescue block

Any Internal routine *r* of a class *C* has a **rescue block** *rb*, which is a Compound defined as follows:

1 • If *r* has a Rescue clause, *rb* is the Compound contained in that clause.

2 • If *r* has no Rescue clause, *rb* is a Compound made of a single instruction: a call to the version of *default_rescue* in *C*.

An exception triggered during an execution of a routine *r* leads, if it is neither ignored nor continued, to the following sequence of events.

1 • Some or all of the remaining instructions are not executed.

2 • The rescue block of the routine is executed.

3 • If the rescue block executes a Retry, the body of the routine is executed again. This terminates processing of the current exception. Any new triggering of an exception is a new occurrence, which will (recursively) be handled according to the present semantics.

4 • If the rescue block is executed to the end without executing a Retry, this terminates the processing of the current exception and the current execution of *r*, causing a **failure** of that execution. If there is a calling routine, this failure triggers an exception in the calling routine, which will be handled (recursively) according to the same semantics. If there is no calling routine, *r* is the root's creation procedure; its execution will terminate.

After failure and termination, the run-time should normally produce a diagnostic similar to the exception history table of page ====.

The definition mentions that it applies only to an exception which is neither ignored nor continued. This corresponds to two facilities provided through features of the Kernel Library class *EXCEPTIONS*, implementing the false alarm response:

False alarm was response E1 introduced on page ====as part of the exception handling policy discussed

- You may specify that a certain type of exception must be altogether ignored.
- You may specify that a certain type of exception must cause execution of a designated procedure and then continuation.

In step 3, the Retry will only re-execute the Routine_body of *r*; it does not repeat argument passing and local entity initialization. This may be used to take a different path on a new attempt.

17.10 EXCEPTION CORRECTNESS

The role of Rescue clauses is to cope with unexpected events. Although in a well-designed system Rescue clauses will only be executed in rare, special conditions, they still have an obligation to maintain the consistency of objects.

In particular, a routine failure should leave the current object (corresponding to the target of the latest call) in a consistent state, satisfying the invariant, so as not to hamper further attempts to use the object if another routine is able, through resumption, to recover from the failure. Also, a Retry instruction, which will restart the Routine_body, should re-establish the routine's precondition, if any, since the precondition is required for the Routine_body to operate properly.

These two requirements yield the notion of **exception correctness**, one of the conditions which make up class correctness. As you may recall, a class is correct if it is consistent (every Routine_body, started in a state satisfying the precondition and the invariant, terminates in a state satisfying the postcondition and the invariant), loop-correct (loops maintain their invariant and every iteration decreases the variant), check-correct (the conditions of Check instructions are satisfied) and exception correct, a notion which was sketched in the general discussion of correctness but may now be made more precise.

Chapter 9 addressed correctness, with the full definition on page ====, as part of 9.16.

Exception-correct

A routine *r* of a class *C* is **exception-correct** if and only if, for every branch *b* of its rescue block:

1 • If *b* ends with a Retry: $\{\mathbf{true}\}\ b\ \{INVC \wedge prer\}$

2 • If *b* does not end with a Retry: $\{\mathbf{true}\}\ b\ \{INVC\}$

In this rule, *INVC* is the invariant of class *C* and *prer* is the precondition of *r*.

Here .*INVC* is the class invariant and *prer* is the precondition of *r*.

The definition involves the rescue block of a routine. Remember that the rescue block always exists: if the routine has a Rescue clause, then its Compound is the rescue block; otherwise the rescue block is the local version of *ANY*'s procedure *default_rescue*.

As with other correctness conditions, exception correctness should ideally be provable automatically, but in practice you will most likely have to ascertain it through informal means.

17.11 FINE - TUNING THE MECHANISM

In some cases it is useful to have finer control over the handling of exceptions. Features from the Kernel Library class *EXCEPTIONS* address this need. These features will be available to any descendant of *EXCEPTIONS*: to use them in a class *C* which is not already a descendant, just add a Parent part for *EXCEPTIONS* to the Inheritance clause of *C*.

Let us take a look at the facilities offered. A later chapter gives the full short-flat form of *EXCEPTIONS*.

See chapter 34 about the details of class EXCEPTIONS.

First, *EXCEPTIONS* introduces an integer code for every possible type of exception. Examples include

Precondition (code for a violated precondition)

Routine_failure

Incorrect_inspect_value

Void_call_target

No_more_memory

The integer-valued feature *exception* is then guaranteed, after an exception occurs, to have the value of the code for that exception. This makes it possible to write Rescue clauses such as

```
rescue
    if exception = No_more_memory
then
    ... Specific treatment...
    else
    default_rescue
    end
```

The call to *default_rescue* in the Else_part is not required, of course, but as a general guideline if you do need to treat certain categories of exception in a special way then such treatment should remain simple and apply to a small number of categories. You should handle the remaining categories through *default_rescue* or another general-purpose mechanism.

Another integer-valued feature, *original_exception*, complements the information given by *exception*. It yields the "real" cause of an exception, disregarding any resulting failures of intermediate routines. Consider for an example a Postcondition violation which causes a routine *t* to fail, triggering an exception whose recipient is *t*'s caller, *s*; the Rescue clause of *s*, if any, does not execute a Retry, so *s* in turn fails, sending an exception to its own caller, *r*. If the Rescue clause of *r* looks at *exception* to determine what happened, it will find as exception code the value of *Routine_failure*. This gives the immediate cause of *r*'s exception (the failure of *s*) but not the real source of the problem – *t*'s Postcondition violation. Feature *original_exception* provides more precise information in such cases. Its value is the code of the oldest exception not handled by a Retry. In the example, this will be the value of *Precondition*.

Features which provide further information about the original exception include *routine_name* (name of the original recipient) and *tag_name* (tag of the violated Assertion_clause, for an assertion violation).

Class *EXCEPTION* also provides a way to raise an exception on purpose. This is called a **developer exception** and is triggered by the procedure call

raise (*code*, *name*)

whose arguments are an exception code *code*, which must be a negative integer (non-negative values are reserved for predefined exceptions), and a string *name* describing the nature of the exception. To obtain that string when handling the exception, use feature *developer_exception_name*.

To know the general category of an exception given of a given *code* (usually *exception* or *original_exception*), use one of the boolean functions

is_assertion_violation (code)
is_developer_exception (code)
is_signal (code)

To prescribe a **false alarm** response you may use one of the procedure calls

ignore (code)

continue (code)

A call to *ignore* simply prescribes that later occurrences of the event with the given *code* must not cause an exception.

After a call to *continue* with *code* as argument, any occurrence of the corresponding signal will cause execution of the appropriate version of procedure *continue_action*, followed by continuation of the Routine_body which was the signal's recipient. Procedure *continue_action* is introduced in class *EXCEPTIONS* with an empty body, but (like *default_rescue*) may be redefined in any class to yield specific behavior. The procedure has an integer argument *code* to which the exception handling mechanism, when calling *continue_action* as a response to an exception for which "continue" status has been prescribed, will attach the code of that exception.

The false alarm policy would not make sense for exceptions which cause irrecoverable damage to the current routine execution. For example, an assertion violation indicates a breach of some consistency condition, making it impossible to continue normal execution. For this reason, *continue* has the precondition *is_signal*(*code*). No such precondition has been imposed on *ignore* in deference to the potential needs of developers of low-level systems software; except in very special cases, however, *ignore* must only be applied to signals.

To restore the default behavior after a call to *ignore* or *continue*, use the call

```
catch (code)
```

To know the behavior specified for *code*, use

```
status (code)
```

whose result, an integer, is given by one of the Unique constants *Caught* (the default), *Continued* and *Ignored*.

As a final comment, it is useful to note once again that the best exception handling is simple and modest. The facilities of class *EXCEPTIONS* are there to give you full access to the context of exceptions if you need it; remember, however, that if you are trying to do something complicated in a Rescue clause, you are probably misusing the mechanism.

Non-trivial algorithms belong in the Routine_body; the Rescue clause is there to recover in a simple and non-committing way from abnormal situations which it was absolutely impossible to avoid.