DISCIPLINED EXCEPTIONS

Bertrand Meyer

Interactive Software Engineeering

Report TR-EI-13/EX

About the context of this report see "*A historical note about this paper*" on the last page.

1 Introduction

One of the most delicate problems of software construction is the handling of abnormal situations. How can a programmer write robust software components, which will take care of all possible cases, without impairing the elegance of the structure and the processing of normal cases?

The question is particularly important in object-oriented programming, which heavily relies on the notion of reusable software component. The quality requirements on reusable components are even higher than those on "regular" software; in particular, no imperfection can be tolerated with respect to correctness and robustness.

This paper presents the solution recently introduced into the Eiffel object-oriented language and environment for dealing with unexpected run-time situations. The Eiffel exception mechanism differs from existing exception mechanisms, particularly those of Ada and CLU in that it is based on a strong emphasis of program correctness, using the concept of **Design by Contract** [6, 7]. Exceptions are defined with respect with assertions, and exception handling is constrained to observe the semantics defined by the assertions.

2 Exceptions in existing languages

A number of languages have included exception handling facilities. The best known mechanisms are probably those of Ada [1] and CLU [5]. In both cases the exception mechanism is really a control structure.

The recommended CLU programming style, for example, commonly uses exceptions to deal with all results of an operation other than the most common one. A typical example [4] is a routine for determining the index of the first occurrence of a value in a list; an exception is triggered if there is no such occurrence.

It is hard to find arguments for this approach over the standard technique of returning a special value, say 0 if valid list indices lie between 1 and n, or other techniques that rely on standard control structures. In fact, the basic paper on CLU exceptions [5] states that although the presence of an exception mechanism should not affect the efficiency of algorithms in non-exceptional cases, efficiency concerns are less important for the handling of exceptional cases. This is justified if exceptions are confined to the handling of abnormal or erroneous situations, but seems hardly compatible with an approach that triggers an exception to signal an unsuccessful search – hardly an "exceptional" case.

In Ada, the reference manual uses the example of a stack module which raises an exception if the *pop* operation is applied to an empty stack. Although this example appears in numerous Ada textbooks, few give examples of how to **handle** such an exception. But this is the really difficult part: since a stack module lacks the proper context to deal with the exception, the calling modules (called **clients** in the sequel) must include a handler. But a client including such a handler will be significantly more complicated than the obvious solution using classical control structures, namely

```
if empty (s) then pop (s)
else ... "Deal with empty stack" ... end
```

With exceptions, a handler clause of the form

```
exception
    when Stack_underflow => ... deal with stack underflow ...
    ...
```

must be included in any client module; it is awkward to write since it is disjoint from the actual call and hence does not have the proper context to deal with the abnormal situation.

The use of exceptions in such examples appears as a vain attempt to do away with the necessity to deal with abnormal cases. Of course it is unpleasant to include many **if**... **then**... **else**... structures of the above form in a program; but this necessity is a fact of life. The need to specify the treatment of abnormal cases does not magically vanish through the **raise** incantation.

The only application for which exceptions may be justified in such cases is software fault tolerance. Assume the programmer has made every effort to ensure that all calls to *pop* in a given system are properly protected, but still wants to take into account the possibility of an error in the software by providing some response in the case of an erroneous call (one for which the stack is empty). This response can only be what we shall call "organized panic": try to bring the computation to a coherent state and report failure. We shall see below how to use a disciplined form of exceptions to describe such treatment.

The exception mechanism of Ada is particularly worrying because the handler of an exception may do any processing before returning control to the caller. It is quite possible in particular to catch an exception and return control to the calling routine without reporting failure. An extreme example, found in an Ada textbook [9], is a function for computing a real square root; when presented with a negative argument, this function raises an exception, which is immediately caught by a **when** clause that prints a message and then .. quietly returns to the caller!

In other examples, exceptions are used as a form of inter-procedural jump instructions. In our view, such applications of exceptions are abusive. The goto instruction was not banned from programming methodology in the late 1960s to be reintroduced at the interprocedural level into the languages of the 1980s.

Standard control structures may be used in most cases where CLU or Ada would use exceptions, using either the "a priori scheme" (test the applicability of an operation before attempting it) or the "a posteriori scheme" (attempt the operation, and then find out whether it has succeeded). These techniques are discussed in detail elsewhere [6, 7].

There remains, however, three cases in which classical techniques are not sufficient and exceptions may be needed. They are the following:

- The first case is one in which attempting the operation may cause a hardware or operating system signal if the operation was not applicable; the signal must be caught

to avoid catastrophes.

- In the second case, an abnormal situation must lead to immediate termination because physical danger may otherwise result, as in a robot manipulation system.
- The third case, mentioned above in the *pop* example, is software fault tolerance: guarding against the possibility of a remaining error.

The Eiffel exception mechanism is meant to deal with these cases. It is based on the Eiffel constructs for specifying and ensuring software correctness: assertions.

3 Assertions

To explain the Eiffel approach to exceptions, it is first necessary to recall the assertion mechanism offered by this language [6].

Eiffel assertions are used to specify the semantics of software elements (classes and routines). Syntactically, assertions are boolean expressions, plus a few extensions. An assertion may have several clauses separated by semicolons, which is semantically equivalent to an **and** but allows individual identification of the clauses. An example of an assertion is

```
index_large_enough: i >= 1;
index_small_enough: i <= nb_elements
```

As shown in this example, clauses may be labeled for individual identification (in particular for debugging purposes, when the optional monitoring mechanism, described below, is enabled). Labels will be dropped in subsequent examples.

Assertions have numerous uses. One of the most important applications is to characterize the semantics of routines through a precondition (introduced by the keyword **require**) and a postcondition (introduced by **ensure**). The precondition describes the condition that must be satisfied by the client for a call to be correct; the postcondition describes the condition that is guaranteed in return by the routine on exit. For example the *pop* routine in the *STACK* classes of the basic Eiffel library is written as

```
pop is
        -- Remove top element.
    require
        not empty
    do
        ... "Implementation of the pop operation" ...
    ensure
        not full;
        nb_elements = old nb_elements – 1
    end -- pop
```

The boolean functions *empty* and *full* express whether a stack is empty and full (respectively). The **old** notation, used only in postconditions, refers to the value of an attribute (here *nb_elements*, giving the number of elements of a stack) upon routine entry.

A further use of assertions is the class invariant, which states the properties that must be satisfied by all instances of a class in all "stable" states, that is to say after instance creation (obtained in Eiffel by executing the *Create* procedure of the class) and before and after the execution of every exported routine. For example, the invariant of the *ARRAY* class includes

```
size = upper – lower + 1;
size >= 0
```

A class invariant expresses the integrity constraints that must be satisfied by all instances of a class.

The invariant is implicitly added to both the precondition and the postcondition of every routine in the class (postcondition only for *Create*). However the invariant transcends individual routines and applies to the class as a whole; in particular, it constrains not only the routines that appear in the class at a given moment of its evolution, but any others that may be added later either through modification of the class or through inheritance. (Eiffel defines precise rules as to how assertions control the inheritance mechanism and particularly routine redefinition. This important topic falls beyond the scope of this discussion; see [6] and [7].)

Assertions in Eiffel have several roles. Two are conceptual in nature: assertions serve as powerful guiding constructs for the production of correct and robust software, and as an effective software documentation tool. The other two uses of assertions are debugging and exception handling. Both assume that assertions are monitored at run-time, which is possible in the Eiffel environment under the control of compilation options which, for each class, determine whether to check the validity of assertions. Various levels of checking may be enabled for each class: no checking at all, preconditions only, all assertions.

The notion of assertion is fundamental for ensuring software reliability. Without a technique for expressing the purpose of individual software elements independently of their implementations, there is no way to guarantee that they will serve any useful purpose. The underlying idea is "**Design by Contract**": every routine fulfils a precise job, defined by a specification that states precisely the obligations on the client, limiting the routine's responsibility (the precondition), and the obligations on the routine, guaranteeing the client a certain result (the postcondition). The class invariant states general integrity constraints that apply both to the client and the routine.

In this context, an exception is an abnormal event that prevents a routine from carrying out its task as initially planned; exception handlers should describe how to recover from such an event.

The deficiencies of existing exception mechanisms may be traced to the absence of any notion of contract. Without this notion and the associated assertion techniques, the intent of a routine is never stated precisely; then an exception handler may perform any action, including one that has no connection whatsoever with the routine's original purpose – or, as in the square root example cited above, one that defeats this very purpose.

4 Failures and exceptions

The following definitions will serve as the basis for the disciplined exception mechanism introduced below.

A **failure** is the inability of a routine to fulfil its contract as specified by the postcondition and the class invariant.

An **exception** is an abnormal event occurring at run-time during the execution of a routine.

In theory, there is only one type of exception: violation of an assertion during the execution of the routine (precondition or invariant violated on entry, postcondition of a called routine violated on return from that routine, etc.). However in practice more cases must be considered. A more complete list of possible exceptions in Eiffel is the following:

1 • Violation of an assertion, when monitored.

2 • Failure of a called routine.

3 • Access to a non-existent object, as in $x.f$ where x is a void reference.

4 • Signal sent by the hardware or operating system, indicating some abnormal event (numerical overflow, user interrupt, I/O error etc.) during the execution of the routine.

Cases 2 to 4 may be viewed conceptually as variants of case 1, where the violated assertion is implicit, for example the unstated assertion that, in the computation of $a + b$, the mathematical sum of a and b is small enough to be representable on the machine.

It is important to keep the notions of failure and exception distinct. Of course, they are connected: as noted in case 2, failure of a routine raises an exception in its caller; and, as will be seen below, occurrence of an exception in a routine leads to failure of the routine unless some special correcting action is taken.

5 Two principles of exception handling

Exception handling is controlled by the notion of contract. The following law expresses that the occurrence of an exception is not an excuse to violate the routine's contract:

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

Trivial as this law may seem, it is violated, for example, by the Ada exception mechanism, which makes it possible to write an exception handler that returns to the caller without correcting the cause of the exception (and without re-raising the exception). In such a case the routine has failed (since an exception prevented it from executing to its normal end), but returns control to its caller without signaling failure. It is like a "dishonest" contractor that has not performed its task but pretends to its client that it has.

The square root function mentioned above shows that such dangerous uses of the Ada exception mechanism not only are possible but have found their way into software engineering textbooks.

A corollary of the first law, which makes it clear that the Ada policy is too general, is:

> **Second Law of software contracting**: If a routine fails to fulfil its contract, the current execution of its caller also fails to fulfil its own contract.

What then should be done when an exception occurs? In view of the above principles, only two responses are reasonable.

- One response, **organized panic**, consists of admitting that the contract cannot be fulfilled: bring all affected objects to a coherent state, and report failure. Note that this will trigger an exception in the caller, which will recursively have to decide what to do in response to this exception, using the same two possible choices.
- The other response, **resumption**, consists in attempting to fix the reasons for the exception and trying the whole routine execution again.

These two responses are the only ones permitted by the Eiffel mechanism.

6 The rescue clause

The extension of Eiffel for exception handling is concise: two keywords and a library class.

First, a new clause, introduced by the keyword **rescue**, may be added to a routine to describe the treatment of exceptions. The general format of a routine becomes:

```
routine_name (optional_arguments): type is
        -- Header comment
    require
        precondition
    local
        local_variable_declarations
    do
        body
    ensure
        postcondition
    rescue
        rescue_clause
    end -- routine_name
```

(All clauses are optional, except for **do** *body* which may be replaced by **deferred** for a deferred routine. The *: type* part is only present for functions.)

The rescue clause is a sequence of instructions to be executed whenever an exception occurs during the execution of the routine.

A key property of the approach is that the rescue clause, if executed until the end, will cause failure of the routine, and thus an exception in the caller. (If there is no caller, that is to say at the root level, a message is printed and execution halts.) This is the organized panic mode: the rescue clause puts objects back into a stable state and signals failure. This policy is in accordance with the above laws: in contrast with the above square root function, an Eiffel routine should not "pretend" that it succeeded when it has not been able to correct the cause of an exception.

A routine which has no rescue clause is considered to have an empty one; this means that any exception will lead to failure of the routine. Also, a rescue clause may be given at the class level, and will then apply to any routine of the class which does not have its own explicit clause. This makes it possible to have a common treatment of exceptions in several routines.

7 Retrying

Organized panic, however, is only one possible response; the other is resumption. A rescue clause may terminate by executing the instruction

```
retry
```

which, as its name indicates, restarts the routine from the beginning. Clearly, the part of the rescue clause executed before the **retry** must have changed some of the context to ensure that the new execution of the routine tries some other road towards fulfilling the contract than the road initially followed. Examples will illustrate this.

It is important to note that, **retry** or not **retry**, the rescue clause never attempts to fulfil the routine's contract. This is solely the province of the body (**do**...). The aim of the rescue clause is to "patch things up" and either concede failure or retry. This will be expressed more formally below.

8 The *EXCEPTIONS* class

Sometimes it is useful to treat various types of exceptions differently. For this purpose, a library class *EXCEPTIONS* is provided. Any class needing its facilities can inherit from it; recall that Eiffel efficiently supports multiple inheritance, so that it is a standard technique in this language to package a number of constants or operations in a class; then any class needing these facilities can access them by inheriting from that class (on top of its "normal" parents). Class *EXCEPTION* includes a number of features, including an attribute

```
exception
```

which is set by the run-time system to the code of the last triggered exception. Exceptions have integer codes; codes for the most common exceptions are defined in the class as symbolic constants (constant attributes in Eiffel), such as *Overflow*, *No_more_memory* etc. So a common structure for a rescue clause is

```
if exception = Overflow then ...
elsif exception = No_more_memory then ...
elsif etc.
```

If there is an **else** clause, it should not end with a **retry**: this way, an unforeseen exception will result in failure, which is the appropriate effect.

Among other features of class *EXCEPTIONS* is a function that yields a new exception name. The equivalent of an explicit "raise" instruction is given by the following routine of this class:

```
raise (exception_code: INTEGER) is
        -- Raise an exception with the given code
    require
        false
    do
    end -- raise
```

Class *EXCEPTIONS* being compiled in such a mode as to monitor preconditions, any call to *raise* will indeed trigger the appropriate exception.

9 Formal requirements

The following methodological requirements apply to any rescue clause.

The rescue clause must admit **true** as precondition. This is because an exception may occur at any time, and the rescue clause should always be applicable.

Now consider a branch of the rescue clause not ending with **retry**. It will lead to failure, but must leave the object in a stable state, as noted above. This means that such a branch must admit the **class invariant** as postcondition. Note again that the branch is not constrained to ensure the routine's postcondition: this is the task of the body.

Finally any branch ending with **retry** must ensure the invariant and, since the routine will be restarted, the precondition. (If the precondition is not satisfied, of course, an exception will be triggered again immediately if precondition monitoring is on).

10 Examples

The mechanism turns out to yield a remarkably simple way of writing software to deal with exceptional conditions.

Consider for example a problem which is found under small variants in many Ada textbooks: get an integer from an interactive user; if the input is incorrect, the reading routine *getint* raises an exception; when this happens, ask the user again, but no more than 5 times. Note that a function such as *getint* producing side-effects and raising exceptions is anathema to the recommended Eiffel style, but we assume (for compatibility with the Ada examples) that such a low-level function is the only one available to read integers.

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

Like all integer entities, the local variable *failed* is initialized to zero on entry. The predefined entity *Result* denotes the result to be returned by the function.

Note how the task of carrying out the routine's contract is concentrated in the **do** clause; the **rescue** only patches things up when something is amiss.

Another example, adapted from [3], is that of a routine that returns $1/x$, or 0 if the division is impossible. This is typical of problems that are almost impossible to solve without an exception facility, because the only way to find out whether the operation is possible is to attempt it, but if it fails a hardware signal will be generated. (We assume this is the case whenever x is too small.) This may be written as:

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

The local variable *division_attempted* will be initialized to **false** upon routine execution, as would any boolean.

Note that a more robust version of the rescue clause should discriminate between exceptions by using the facilities of class *EXCEPTIONS*; only overflow should give rise to the above treatment (but not, for example, the user hitting the BREAK key during execution of the routine). Other cases should lead to failure.

The last example is taken from Saib [8]. It is an elementary case of "*n*-version programming" [2] – a method whereby better software reliability is sought, as is often done in hardware engineering, through fault-tolerance and redundancy; two or more teams are asked to implement an identically specified module, and both are used, each serving as standby if the other fails.

Although one may entertain reservations about this approach to software reliability, it is interesting to see how it may be programmed. Saib's version keeps alternating between the two versions as long as one fails; this is hardly realistic (one would normally stop after both attempts have failed), but we shall keep this hypothesis for the purpose of the comparison. The Ada version is the following:

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

The control structure necessary to achieve the result looks rather contorted: two blocks, two exception handlers, two exits from within a loop, and one goto that jumps out of two exception handlers, two blocks and a loop! This would be enough to bring "structured programming" back into fashion. A much simpler structure does not appear possible with the Ada exception mechanism. Compare with the Eiffel version:

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

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

11 Conclusion

It is remarkable to note that the mechanism described here could have been designed into Ada, although it fits particularly well within the object-oriented approach as promoted by Eiffel. (See in particular [6] for an explanation of inheritance and dynamic binding from the "programming as contracting" viewpoint.) What seems to have prevented the inclusion of such a mechanism in Ada is the lack of a notion of contract.

The emphasis on software reliability which was paramount in the design of Eiffel naturally led to the above facilities which, we believe, are at least at least as powerful for practical applications as the exception facilities built into previous languages, while being simpler to use and much safer.

References

[1] ANSI and AJPO, "Military Standard: Ada Programming Language (American National Standards Institute and US Government Department of Defense, Ada Joint Program Office)", ANSI/MIL-STD-1815A- 1983, February 17, 1983.

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

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

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

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

[6] Bertrand Meyer, "Programming as Contracting", Submitted for publication, 1988.

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

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

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

A HISTORICAL NOTE ABOUT THIS PAPER

The present paper was the first description of exception handling in the Design by Contract framework, as applied to Eiffel. Submitted to the ECOOP 1988 conference and rejected, it was never published (except as an ISE report). At ECOOP, copies were handed out to visitors of the ISE booth in the commercial exhibition, accompanied by the note excerpted below, which provides a vivid reminiscence of the early days when all attempts at publishing Eiffel-related ideas systematically failed, thwarted by competing interests.

Later on, partly thanks to the incontrovertible success of the book *Object-Oriented Software Construction*, the creation of independent publications like *JOOP* (SIGS's Journal of Object-Oriented Programming) with no allegiance to a particular school or group, and the emergence of new, fair-minded conferences, the lock was broken and it became possible to publish papers mentioning Eiffel.

If the note sounds bitter when read today, it's because at the time Eiffel very nearly got censored out of existence. Let's hope that by reminding us of a dark episode it will help protect future software innovations from having to face such systematic and ill-founded hostility.

A NOTE TO ECOOP ATTENDEES

We hope you have enjoyed reading this article about exception handling for object-oriented programming, describing the mechanism available in Eiffel.

Eiffel is an object-oriented language which you will not hear about in the program of ECOOP conferences.

The present article was rejected *without any single technical comment* by the ECOOP program committee. As if someone was trying to fend off suspicions of foul play, the rejection letter explained in long and contorted terms how fair the selection process had been; the letter stated both that papers were reviewed anonymously and that "*committee members left the* [*selection*] *meeting when papers related to them were discussed*". Asking the author to leave and then discussing his paper anonymously must have been an interesting exercise indeed.

The author's letter complaining about the unfairness of the review process did not elicit so much as a response from the program committee chairman.

Another major Eiffel contribution is its approach to multiple inheritance. We think the Eiffel solution is the cleanest available. At ECOOP 88, however, you will not be able to hear the arguments for it – nor, for that matter, any arguments against it. Requests to participate in the multiple inheritance panel remained unanswered. We don't think the conference gained much.

The ECOOP policy of excluding any mention of Eiffel is not new. In the 1987 conference, a general presentation of the language was also summarily rejected. In this case, there was *one* technical comment: "Eiffel has already been published." At that time, *not a single* general presentation of Eiffel was available in the open literature, precisely because the major article had been kept aside for submission at ECOOP.

Clearly, the systematic rejection of Eiffel by ECOOP will turn out to be a greater blow to ECOOP than to Eiffel. In the meantime, we thank ECOOP attendees for encouraging the conference to apply the standards of fairness, honesty and integrity that are normally accepted in the scientific community.