Writing Correct Software

Assertion and exception techniques can aid in class correctness

Bertrand Meyer

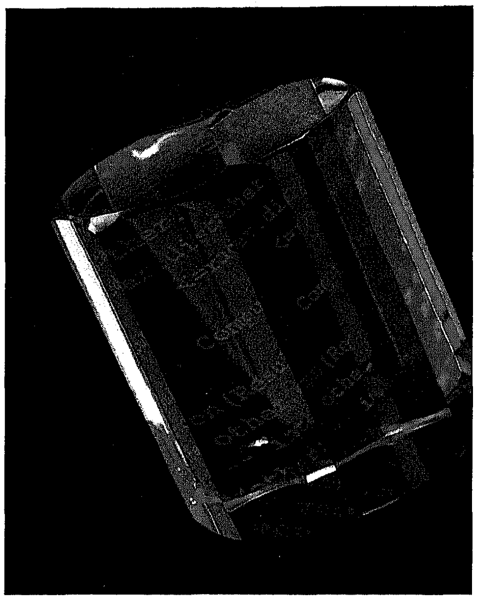
My aim in designing Eiffel was to produce a major programming language for the 1990s, catering to the needs of those software engineers willing to do what it takes to produce high-quality software. A key aspect of Eiffel, which makes it original in the world of object-oriented languages, and in the world of programming languages at large, is its strong emphasis on techniques that help produce highly reliable software.

Although, there are many more aspects to Eiffel (including those described in my book *Object-Oriented Software Construction*, Prentice-Hall, 1988) the reliability features deserve a presentation of their own. That is the focus of this article. I will show how it is possible to write software that programmers (and users) can place a much higher degree of confidence in than that written with traditional techniques. In particular, I will discuss the all important notion of assertion — the specification element included within the software itself. This will lead to a systematic view of exception handling, and a look at techniques (such as those offered by Ada) that I find somewhat unsafe.

Why All the Fuss?

The issue is simple. It is great to have flexible software that is easy to build and easy to maintain, but we also need to be concerned that the software does what it is supposed to do.

Bertrand is the president of Interactive Software Engineering and is the main designer of the Eiffel language. His book, Object-Oriented Software Construction, *was published by Prentice Hall in 1988. He can be reached at 805-685-1006, or through e-mail as Bertrand at Eiffel.com.*



From reading most of the object-oriented literature, one would think this is not a problem. Correctness concerns are hardly ever mentioned. Actually, it is unfair just to pick on object-oriented programming. Take any standard textbooks you have on programming, algorithms, data structures, and similar topics. See how many of them list "correctness," "reliability," "invariant," or "assertion" in their indexes. I have quite a few textbooks on my shelves, but could not find many that passed this simple test.

This apparent disregard for correctness issues cannot last forever. Even barring the occurrence of a major catastrophe resulting from faulty software, sooner or later someone will call the software engineers' bluff and ask them exactly why they think their systems will perform as announced. It is difficult to answer that question convincingly given the current state of the art.

Eiffel won't provide the magical key to the kingdom of software reliability. No existing method or tools will. I do believe, however, that the Eiffel techniques are an important step in the right direction.

If you are expecting a sermon telling you to improve your software's reliability by adding a lot of consistency checks, you are in for a few surprises. I suggest that one should usually check less. According to conventional software engineering wisdom, "defensive programming" is considered to be a programmer's best shot at reliability. I believe that defensive programming is a dangerous practice that defeats the very purpose it tries to achieve. To program defensively is one of the worst pieces of advice that can be given to a programmer.

That more checking can make software less reliable may seem foolish. Remember, though, that in science common sense is not always the best guide. If you have ever hit a wooden table with your fist, you probably found it hard to believe the physics professor who told you that matter is a set of tiny atoms with mostly nothing in-between.

Expressing the Specification

The ideas that help achieve correctness in Eiffel are much older than Eiffel it-

(continued on page 52)

(continued from page 48)
self. They come from work on program proving and formal specification. Oddly enough, research on these topics has remained estranged from most "real-world" software development. Part of the reason, at least in the United States, is the widespread view that formal specification and verification are specialized research topics whose application is mostly relevant to "mission-critical" software. Correctness, however, should be a universal concern. Eiffel looked at specification and verification work to see how much of it could be made part of a standard programming methodology.

Eiffel is a production language and environment. It is not a research vehicle. Eiffel relies on the technology of the last part of the twentieth century. It has to work now. This means that no miracles can be expected. In fact, the techniques are modest and almost naive. They are the result of an engineering trade-off between what is desirable in an ideal world and what can realistically be implemented today. But they make a big difference and I can't understand why no widespread language, other than Eiffel, has made any significant attempt in a similar direction.

The basic idea is rather trivial. Correctness is a relative notion. No software element is correct or incorrect per se; it is correct or incorrect only concerning a particular specification, or statement of its purpose. Correct elements cannot be written unless the time is taken to express all or part of this specification.

Writing the specification will not guarantee that it is met. But the presence of a specification, even one that is only partially spelled out, goes a surprisingly long way toward helping produce elements that satisfy their correctness requirements.

This idea was captured by the title of an article by Harlan Mills, then of IBM, published in 1975: "How to Write Correct Programs and Know Why." If you are a serious software engineer, you don't just want to hope that your programs are correct because you have been careful, and done a lot of testing, and so on. You need precise arguments that document the correctness of your software.

In Eiffel, such arguments are expressed as assertions — elements of formal specification that can be attached to software components, classes and their routines.

	Obligations	Benefits
Author	Bring in manuscript before March 1st	Get published text before May 1st
Publisher	Publish before May 1st	No need to do anything if no manuscript is received before March 1st

Figure 1: *Contract between a publisher and an author*

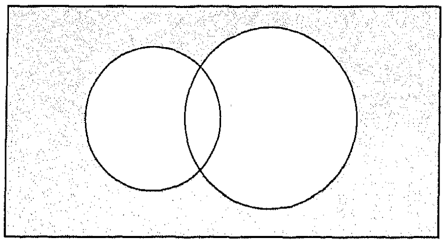


Figure 2: *Circle intersection*

	Obligations	Benefits
Client	Call the routine on two circles that intersect	Obtain as result of the function a point that is on both circles
Routine	Find a point that is on both circles	No need to return anything meaningful if the circles do not intersect

Figure 3: *A routine contract*

The Contract

Let's look at routines first. A routine is the description of some computation on the instances of a class, made available by that class to its clients (to other classes relying on its services). How do we specify the purpose of a routine?

The view I find most helpful is that a routine provides clients with a way to contract out for a certain task that the client's designer finds advantageous not to implement within the text of the client. This is the same way that we humans at times contract out for part or all of a task that we need to perform.

Human contracts have two important properties:

- Each party expects some benefits and is prepared to incur some obligations in return. What is an obligation for one party is a benefit for the other.
- The obligations and benefits are spelled out in a contract document.

Figure 1 illustrates an example of a contract between a publisher and an author. The author's obligation is to bring in a manuscript before March 1st. The benefit to the author is that the manuscript will be published before May 1st. The publisher's obligation is to publish the manuscript before the second date.

The publisher is not bound by any obligation if the author violates his part of the deal. In such a case the publisher may still publish the manuscript, but does not have to. The situation is outside of the contract's bounds.

Routine as Contract

Specifying a routine is based on the transposition of these observations to software. First, we need the equivalent of the contract document. It bewilders me that no such concept exists in standard approaches to software construction.

The specification consists of two parts:

- The precondition of a routine states the obligations of clients, which are also the benefits for the routine itself.
- The postcondition states the obligations of the routine, which are also the benefits for the clients.

The precondition is a set of initial conditions under which the routine operates. Ensuring the precondition at the time of any call to the routine is the clients' responsibility.

The postcondition is a set of final conditions the routine is expected to ensure. Ensuring the postcondition at return time (if the precondition was met on entry) is the routine's responsibility.

The concept of a contract is one of the most useful aids to understanding Eiffel programming. The role of contracts in Eiffel can be compared to what message passing represents in Smalltalk.

Figure 3 illustrates this idea. The function *intersect1* in a class *CIRCLE* (assumed to be part of some graphic package) returns one of the two intersecting points of two circles (see Figure 2). We will look at how to associate the precondition and the postcondition to the text of the function in the actual Eiffel class. In this example:

- The precondition is that the two rectangles should intersect.
- The postcondition is that the function result is a point that is on both circles.

Contract Variants

This is not the only possible specification. Programmers may feel uneasy about the just mentioned "demanding" form of the routine, which only works in some cases. Instead, a tolerant version implementing a different contract may be designed. For example:

- There is no precondition. More precisely, the precondition is true, and automatically satisfied by any client. Here, the routines will be applicable in all cases.
- The postcondition is more difficult

(continued from page 52)

to express in this case. Either the two circles intersect and the function result is a point on both circles; or the two circles do not intersect, the function result is an arbitrary point, and an error message has been displayed somewhere. The awkwardness of stating the postcondition in such a way is the first sign of why "demanding" versions are often better.

Expressing the Contract

Let's see how the preconditions and postconditions will be integrated. Listing One, page 125, shows what a class *CIRCLE* might look like. Assume the availability of a class *POINT* describing points, and a function *distance*, such that *p1.distance (p2)* is the distance between any two points (*p1* and *p2*).

Result is a predefined variable which, in a function, denotes the result of that function. *Create* is the initialization procedure. It is automatically exported.

The precondition of a routine, if any, is given by the require clause. The postcondition is given by the ensure clause. Preconditions and postconditions are assertions — logical constraints expressed as one or more Boolean expressions, separated by semicolons. They are essentially equivalent to Boolean ANDs, but allow assertion components to be identified individually. These components can be tagged for even better identification. For example, consider Listing Two, page 125.

Note that the first clause in this precondition (as well as clauses in the preconditions of *inside* and *outside*) express that the argument must be nonvoid. *Void* is a predefined language feature expressing whether there is an object associated with a certain reference.

Uses of Assertions

Along with invariants (discussed later), preconditions and postconditions play a fundamental role in the design of Eiffel classes. They show the purpose of routines and the constraints on their uses. A brief look at any well-designed set of Eiffel classes shows how wide their application is. The Basic Eiffel Library, which covers fundamental data structures and algorithms, is an example of a set of carefully designed classes that come fully loaded with expressive assertions.

The first application of assertions, perhaps the most powerful, is as a conceptual design aid for producing reliable software. In this role, preconditions and postconditions directly support the goal stated earlier: Writing correct software and knowing why it is correct. When a routine is written, its goal (contract) is expressed. If this goal cannot be expressed in a formal way, it should still be expressed as formally as possible.

Documentation is another key application of assertions. One of the most pervasive myths of software engineering literature is the idea that documenting software is a worthy goal. Instead, documentation should be viewed as an evil, made necessary by the insufficient abstraction level of current tools, techniques, and languages. It is an evil not just because documentation is tedious to produce, but also because it is almost impossible to maintain the consistency of a software system with its documentation throughout the system's evolution. Incorrect or out-of-date documentation is often worse than no documentation at all.

In an ideal world, software should be self-documenting, with no need for outside documentation. Failing this programmer's Eden, we should strive to have as little need for external documentation as possible. Documentation should be deduced from the software itself. "Self-documenting software" does not mean that the software is its own documentation. Instead, self-documenting software should contain part, or (ideally) all, of its documentation, corresponding to various levels of abstraction, which can be extracted by automatic tools.

Preconditions and postconditions play a key role because they document the essential properties of routines: What each routine expects and what each ensures in return. The Eiffel environment provides an automatic tool that yields the documentation of a class based on its assertion. This tool, the class abstracter, is implemented by a command called "short." Applying *short* to a class yields the description necessary to determine whether the class can be used in a certain situation, and, if so, how to use it effectively.

The result of *short* applied to class *CIRCLE* would be of the form shown in Listing Three, page 125.

As shown in this example, *short* keeps, as a complement to formal assertions, the natural language header comments of routines, if present, at a well-defined place. Only exported features are kept by *short*.

short provides documentation "for free" — it is extracted from the software. *short* is the major tool for documenting Eiffel classes. A companion tool, *good*, produces high-level system documentation in graphic form, showing the class structure with client and inheritance relationships. Remember, though, that *short* is meaningless without the presence of assertions in the language.

Invariants

Preconditions and postconditions can be used in a non-object-oriented context. Another use of assertions that is inseparable from the object-oriented approach is the class invariant. This is an optional clause of Eiffel classes. An invariant is a consistency constraint that applies to all instances of the class.

In the *CIRCLE* example, the invariant clause might state the following assertion:

```
radius >= 0;
inside (center)
```

In larger examples the invariants can be much more extensive.

Invariants can be viewed as general clauses that are implicitly added to all contracts of a certain class, without being expressly repeated for each of these contracts. The precise definition of the class invariant is that it is an assertion that:

- Must be ensured by the *Create* of the class
- Must be preserved by every exported routine of the class

In principle, we could do away with the invariant by adding its clauses to the precondition and postcondition of every exported routine, and to the postcondition of the *Create*. But, besides making these assertions unduly repetitive, this would be losing sight of the role of the invariant as a global integrity constraint on the class, independent of a particular routine.

The two properties used earlier to define the invariant imply that the invariant is satisfied in all observable states in the life of every instance of the class. Observable states are those immediately following the *Create*, and before and after application of exported routines. The life of a typical object is pictured in Figure 4, with observable states marked as square blocks. The idea of an observable state is important in the context of parallel programming.

In spite of its name, an invariant is not necessarily satisfied at all times. It may be temporarily violated during execution of exported routines, so long as it is restored for the next observable state.

An invariant captures the semantic properties of a class, independently of its current implementation, by a set of attributes and routines. These properties must be understood in a software

(continued from page 54)

engineering context in which software is always subject to change. Invariants can help bring some order to a constantly changing environment by expressing what does not change in a class — the basic semantics of the class.

Invariants can play a major part in establishing a scientific basis for software activities that currently rest on a rather shaky basis: Quality assurance, regression testing, and maintenance. Because an invariant expresses the essential semantics of a class that should be preserved through successive modification and extension, it provides a framework for making QA and associated activities more systematic.

Limitations of Assertions

The Eiffel assertion techniques are only partial. The assertion sublanguage is based on Boolean expressions with some extensions. Sometimes more is needed, such as first-order predicates. In the *CIRCLE* class it would be nice to have the invariant express that no point can be both inside and outside the circle, or that any such point must also be "on" the circle. The notation for this could be:

```
for p: POINT then
    inside (p) and outside (p)
        implies on (p)
end
```

This is not possible in current Eiffel, although properties involving quantifiers ("for all," "there exists") can sometimes be expressed through Boolean expressions involving function calls. These function calls require some care. Other limitations of assertions are due to the reference-based dynamic model used for objects.

The mechanism is the result of an engineering trade-off. Though limited, assertions are a tremendous asset in Eiffel programming.

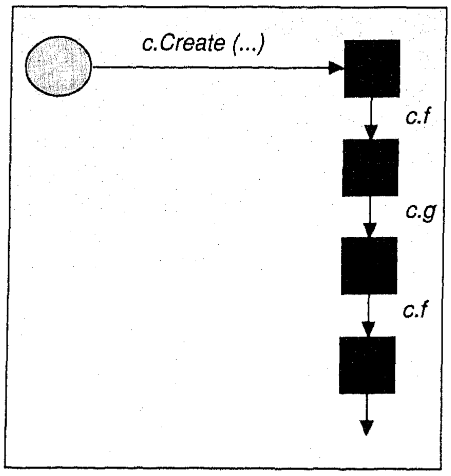


Figure 4: *The life of a typical object*

Assertions and Inheritance

Assertions also play an important role in the context of inheritance. Invariants are always inherited. When a routine is redefined, its precondition may be weakened, but not strengthened. Its postcondition may be strengthened but not weakened. To understand these rules, the contracting metaphor must be viewed in the context of inheritance, redefinition (subcontracting), and dynamic binding.

Monitoring Assertions

The question of what happens when an assertion is violated (such as if *intersect1* is called on two circles that do not intersect) is secondary. The main question is: How can we, as responsible software professionals, make sure that we produce software that is correct?

The tendency to reverse the priorities and ask the secondary question first is a sign of how insecure most of us in the software engineering profession feel about our techniques and tools. This article won't reverse this situation. Still, we must get our priorities straight.

The answer to what happens when an assertion is violated depends on how you have compiled your class. If you have made the effort of spelling out the mental hypotheses that underlie the correctness of your software, you could expect a theorem prover to check the software against these hypotheses. Unfortunately, this is beyond today's technology. The next best thing to static proof is run-time monitoring. If you compile a class under the *ALL_ASSERTIONS* mode, all assertions (preconditions, postconditions, invariants) are checked at the appropriate times during execution. If one is found to be violated, an exception is triggered. Unless you have made explicit provisions to handle it, the exception will result in program termination with a clear message identifying the context of the failure.

There is never a good reason to compile a class under any option other than *ALL_ASSERTIONS*, except performance. If you are sure your software is correct and do not want to incur the overhead of checking, use the *NO_ASSERTION_CHECK* mode. If a bug does remain, though, you are on your own. The default is an intermediate mode, which generates code that checks preconditions only. Switching modes may be needed a number of times during development. This switch is easy. Only the last stage of compilation is repeated for the corresponding class.

Run-time monitoring of assertions provides a powerful debugging mechanism. Assertions are a way to make explicit the otherwise implicit mental assumptions that lie behind our software. It is typical for a bug to cause one of these assumptions to be violated. When this occurs, run-time monitoring will catch the violation. This debugging technique takes on its full meaning in the object-oriented context. I used it when using the Algol W compiler in the seventies. Its superiority over usual debugging methods is hard to imagine until you have actually applied it.

Defensive is Offensive

If a routine has a precondition *p*, defensive programming would mean that the text of the routine should test again for *p*, in case the client forgot. For instance, consider Listing Four, page 125.

The form as shown in Listing Four is never acceptable. It is a sloppy style of programming in which responsibility for ensuring various consistency conditions (contract clauses) have not been clearly assigned. Because the contract is unclear, the scared programmer includes redundant checks "just in case." This is a self-defeating policy. Complexity is the single, worst enemy of software reliability. The more redundant checks, the more complex the software becomes, and the greater the risk of introducing new errors.

Reliability is not obtained by cowardly adding even more checks, but by precisely delineating whose responsibility it is to ensure each consistency requirement. A party in a contract may fail to meet the requirement imposed on it. This is precisely what a bug is. The solution, however, is not to make the software structure more complex by introducing redundant checking, which only makes matters worse. For fault-tolerant design, you should be able to rely on a general-purpose run-time checking mechanism. In Eiffel, this mechanism is the monitoring of assertions as described above.

With redundant checking being unacceptable, we still face a choice between the "demanding" (strong precondition) style and the "tolerant" (no precondition) style, with the intermediate spectrum. Mathematically, tolerant routines represent total functions and demanding routines represent partial functions. Which one to use depends on the circumstances. The closer a routine is to uncontrolled "end users," the more tolerant it should be. But even with general-purpose library routines, there is a strong case for demanding routines.

With a strong precondition, a routine can concentrate on doing a well-defined job and doing it well, rather

(continued from page 56)

being concerned with other things. The *intersect1* routine becomes a mess if it isn't assumed that the circles do intersect. Tolerant routines must address user interface concerns for which the routines do not have the proper context. The *intersect1* routine must address problems of geometrical algorithmics (computing the intersection of two intersecting circles in the best possible way). It is difficult to reconcile these two aspects in a single routine. The solution that will ensure reliability more certainly than blindly checking all constraints all the time, is to separate the checking and the computation.

Conventional wisdom, which says "never assume anything, anywhere," is wrong and dangerous. Its pervasiveness can only be explained by the absence of any notion of contract in standard approaches to programming. If clients have no precise specification of the conditions they are supposed to observe, they can't be trusted to observe these conditions and there is no choice but to include as many consistency checks as possible. In a systematic approach to software construction, however, the contract is clearly and adequately expressed, independently of its implementation, through assertions. By using the short command to let client designers see this contract, you can concentrate on doing your job rather than checking theirs.

Considered in the perspective of other engineering disciplines, the often recommended ban on "partial" routines seems absurd. If you ask an electrical engineer to design an amplifier that will work for any input voltage, or a mechanical engineer to build a bridge that will hold any load, they will laugh at you. Any engineering device has preconditions. There really is no good reason why software routines should be required to be total.

The reference to electronic components is not coincidental. One of the most exciting advantages of object-oriented techniques is the ability to work from libraries of standardized, off-the-shelf, reusable components. These components are similar to hardware components used in electrical engineering. These libraries cannot be successful unless the components are specified in a precise and standardized way. Trying to sell a class without its invariant, preconditions and postconditions is like trying to sell an amplifier without its engineering specs.

Programming by Prayer

Assertions are not a way to program the handling of special cases. An exception violation is not an expected situation that you want to handle separately from the others — it is the manifestation of a bug. To handle special cases, there is not much substitute for what you learned on day two of Introduction To Programming 100 — the if . . . then . . . else construct.

There seems to be another pervasive myth in the industry that one can forget about special cases through a form of faith healing. This can be called "programming by prayer." In Ada, the sacred word is raise. Whenever you encounter a situation that threatens to disrupt the spiritual harmony of your program, kneel down and say, raise *some_exception* and a saint or angel will come and take your worries away.

It doesn't work this way. The "angel" has to be programmed, and usually by you. Postponing a problem does not solve it.

In Ada, after a raise, a chain of calls that led to the exception is explored, in reverse order, until a block is found that includes an exception clause of the form:

```
exception
   when
      some_exception=>some_action;
when
   other_exception=>other_action;
      . . .
```

One of the *when* branches names the current exception. Then the code *some_action* is executed and control returns to the handling block's caller.

If your aim was to make your software simpler by separating the processing of "normal" and "special" cases, you will be disappointed. Special cases will not go away through the raise attempt at absolution. Such as old sins, they will come back to haunt you in your exception clauses. In the program text, such clauses are far away from the source of the exception. They usually lack the proper context to deal with the exception.

There are two cases of exception handling. One is when the exception must be handled identically for all calls of the routine. This type of exception is much better handled by an if . . . then . . . else . . . clause in the routine itself. In other words, the routine should be made more tolerant.

The second is when the handling of the special case is different for each client. This can be achieved by protecting each call with an if . . . then . . . else. The routine itself remains demanding. In either case no special control structure is needed.

Exceptions

Once the naive faith in exceptions as exorcism has been dispelled, there is still room for an exception mechanism. Exceptions should not be used as control structures. They have no advantage over standard control structures, and have many drawbacks. Some mechanism is needed however, to deal with an operation that might fail in such a way that it is difficult or impossible to check for with a standard control structure. Following are three main examples:

1. Bugs. By definition, a bug is unexpected. If you were able to test for its occurrence, you would correct the bug in your software, not handle it at run time. If, in spite of your best efforts, a bug does occur, you still want the ability to recover from it somehow at run time, even if only to terminate the execution gracefully.
2. Uncheckable consistency conditions. Some preconditions may be impossible to check as part of an if . . . then . . . else, either because they are too complex to express formally, or because the applicability of an operation can only be ascertained by attempting the operation and seeing if it fails. For example, a write to disk operation may fail, but it is not useful to ask first and then write. The only way to know if you can write is to attempt to write. Then, if something goes wrong, you must be able to recover. Another example, in an interactive system, is the implicit precondition that the user will not hit the BREAK key. Obviously, you cannot test for the occurrence of such events.
3. Impractical to check before each call. These are operations for which expressible preconditions exist in principle, but for which it is impractical to check before each call. For example, few programmers want to protect every addition by a test for non-overflow, or every object allocation (*Create*) by a check that enough memory remains. As in the previous case, but for practical, rather than theoretical reasons, you want to be able to attempt the operation, proceed as if everything went all right, but recover if something goes wrong.

These three cases are ones for which exceptions are needed. They are not "special" or expected algorithmic cases, but abnormal situations that cannot be properly handled by standard algorithmic techniques.

In Eiffel, an exception occurs in the following situations:

- Assertion violations (if monitored). The violation of an assertion is always a bug. A violated precondition reflects a bug in the client; a violated

(continued from page 58)

postcondition reflects a bug in the routine.

- Hardware or operating system signals, such as arithmetic overflow, memory exhaustion, and so forth.
- An attempt to apply an operation to a non-existent object (Void reference).
- Failure of a called routine.

The range of such exceptions is much less extensive in Eiffel because of the disciplined nature of the language. In particular, the static typing mechanism of Eiffel implies that for a correctly compiled system there is no exception for a "feature applied to an object that cannot handle it (a message sent to an object that cannot process it)."

Dealing with Exceptions

What happens when an exception occurs? The Ada answer is dangerous. Because you can do essentially anything you like in a *when* clause, there is no guarantee that you will achieve anything remotely resembling the original purpose of the routine that failed.

To obtain a satisfactory solution, it is necessary to think in terms of the contract that a routine is meant to ensure. The routine initially tries to satisfy its contract by following a certain strategy, implemented by the routine's body (the *do* clause). An exception occurs when this strategy fails. In the disciplined approach, only two courses of action make sense:

- The routine (contractor) may have a substitute strategy. If so, it should bring the target object back to a stable state and use this strategy. This is the resumption case.
- If no substitute strategy is available, the routine should bring the target object back to a stable state, concede failure, and pass the exception to its client. This is the failure case.

In the exception history table shown in Figure 5, some exceptions are dealt with in each of these two modes. The table, shown as it is printed at run time, is divided into periods, separated by double lines. Each period, except the last, ended with a retry.

The absence of a clear-cut choice between resumption and retry is what makes the Ada mechanism too general, and hence dangerous. Some Ada examples show cases in which a routine reacts to an exception, fails to correct the cause of the exception, and returns to its caller without signalling the exception. This is extremely dangerous.

Eiffel enforces the choice between resumption and retry. The key idea is that of routine failure — a routine may succeed or fail. If it fails to achieve its contract, it may either try again or give up. It should not conceal the failure from its caller.

This explains the fourth case in the earlier list of Eiffel exceptions. The failure of a routine automatically triggers an exception in its caller. This is implemented by the optional routine clause *rescue*. If present, the rescue clause is executed whenever an exception occurs during the routine's execution.

If a rescue clause is executed to the end, the routine terminates by failing. As noted, this automatically raises an exception in the caller, whose own rescue clause should handle it. If a routine has no rescue clause, as will typically be the case with most routines, then it is considered to have an empty rescue clause — any exception occurring during the execution of the routine leads to immediate failure and an exception in the caller. If no routine in the call chain has a rescue clause, the entire execution fails and an appropriate message, recording the history of recent exceptions in reverse order, is printed. Note the use of assertion tags, when present, in the messages shown in Figure 5.

Object	*Class*	*Routine*	*Name of exception*	*Effect*
2FB44	*INTERFACE*	*m_creation*	*Feature "quasi_inverse": Applied to void reference*	*Retry*
2F188	*MATH*	*quasi_inverse (from BASIC_MATH)*	*"positive_or_null": Precondition violated*	*Fail*
2F188	*MATH*	*raise (from EXCEPTIONS)*	*"Negative_value": Programmer exception*	*Fail*
2F188	*MATH*	*filter*	*"Negative_value": Programmer exception*	*Fail*
2F321	*MATH*	*new_matrix (from BASIC_MATH)*	*"square_matrix": Invariant violated*	*Fail*
2FB44	*INTERFACE*	*create*	*Routine failure*	*Fail*

Figure 5: *An exception history table*

Not all exceptions cause failure. A rescue clause may execute a *retry* instruction, in which case the body (*do* clause) of the routine must be tried again, presumably because a substitute strategy is available. This is the *resumption* case.

For example, consider the routine in Listing Five, page 125, for attempting to write to disk, from a generic class *C*.

Here it is assumed that the actual *write* is performed by a lower-level external routine attempt-to-write, written in another language, over which we have no control. If this routine fails, it triggers an exception, which is caught by the *rescue* clause. This results in a *retry*. Local routine variables are initialized on routine entry. An integer variable such as *attempts*, is initialized to 0.

The routine *write* never fails. Its contract says, "write if you can, otherwise record your inability to do so by setting the value of attribute *write_successful* to false, so that the client can determine what happened." It is always possible to satisfy such a contract.

The version of *write* shown in Listing Six, page 125 is a variant of the class that does not include attribute *write_successful*. It may succeed or fail.

In this version, after five attempts, the routine terminates through the bottom of its rescue clause. This means the routine fails, triggering an exception in the caller. This contract is more restrictive than the one shown in Listing Three. It requires that the routine be able to write. If this contract cannot be fulfilled, the only exit is through failure.

Formal Requirements

The deeper meaning of the rescue clause can be understood in the object-oriented context, and with reference to the contract of a routine, as expressed by assertions.

The following expresses the requirements on a contractor that implements software element *e*:

$$\{P\}\ e\ \{Q\}$$

This means the contractor must write *e* in such a way that, whenever *P* is satisfied on entry, *Q* will be satisfied on exit. The stronger *P* is, the easier the contractor's job (more can be assumed); the stronger *Q* is, the harder the contractor's job is (more must be produced).

Consider routine *r* with body *do*, precondition *pre*, and postcondition *post*, in a class with invariant *INV*. The

(continued on page 63)

(continued from page 60)
requirement on the author of the *do* clause is:

{pre and INV} do {post and INV}

In other words, the invariant and the precondition can be assumed, the invariant must be preserved, and the postcondition must be ensured. Now, consider a branch *rescue*, of the rescue clause, not ending with a retry. The requirement here is:

{true} do {INV}

The input condition is the weakest possible (hardest from the contractor's viewpoint), because an exception may occur in any state. The rescue clause must be prepared to work under any condition, but the output condition only includes the invariant. Ensuring the invariant brings the object back to a stable state. Integrity constraints play a similar role in data base systems. The *rescue* clause is not, however, constrained to ensure the entire postcondition. This is the sole responsibility of the *do* clause. If the contractor satisfies the routine's contract, there is no need for the *rescue* clause.

This shows the clear separation of concerns between the *do* clause and the *rescue* clause. The former is responsible for achieving the contract when possible. The latter takes over in case the *do* clause falters. The rescue clause restarts the *do* clause under improved conditions, or closes the store after putting things in order. The requirements on the *rescue* clause are both harder (a weaker precondition) and easier (a weaker postcondition).

Fine-Tuning the Mechanism

Those are the basics of Eiffel exception handling. In practice, some fine-tuning may be needed for particular applications. This is done not through the language itself, but through the library class *EXCEPTIONS*. Classes needing the corresponding facilities should inherit this class.

It is sometimes necessary to treat various exceptions differently. Attribute exception in class *EXCEPTIONS* has the value of the code of the last exception that occurred. Exception codes are integer symbolic constants (attributes) defined in that class. Examples include *Precondition* (precondition violated) and other assertion-related exceptions, *No_object*, *No_more_memory*, operating system signals (*Sighup* and so on.) and others. A *rescue* clause may contain a test of the form:

```
if exception=No_more_ memory
then . . . elsif  and so on.
```

Generally, it is wise to resist the temptation to attach too much meaning to the precise nature of an exception. An exception usually points to a symptom, rather than a cause.

For programmers who want to define and raise their own exceptions, the routine raise is available in class *EXCEPTIONS*. The default handling of certain exceptions, especially operating system signals, can be changed by redefining certain routines from class *EXCEPTIONS*. By using class *EXCEPTIONS*, application software can access information about the last exception. This information includes the exception type, its meaning expressed as a plain English string, and so on. This is particularly useful for printing informative error messages.

Why Not Make It Right?

Reliability is a primary concern in any serious view of software construction. In the object-oriented approach, it is even more essential. Reusability of software is meaningless unless the reusable components are correct and robust. Static typing is an important aspect of Eiffel's contribution to this goal (see the article "You Can Write, but Can You Type?" in the March 1989 issue of the *Journal of Object-Oriented Programming* for more on this subject).

The assertion and exception techniques described in this article provide the complement to static typing. They don't absolutely guarantee that your classes will be correct and robust, but they sure can help.

Availability

All source code is available on a single disk and online. To order the disk, send $14.95 (Calif. residents add sales tax) to *Dr. Dobb's Journal*, 501 Galveston Dr., Redwood City, CA 94063, or call 800-356-2002 (from inside Calif.) or 800-533-4372 (from outside Calif.). Please specify the issue number and format (MS-DOS, Macintosh, Kaypro). Source code is also available online through the *DDJ* Forum on CompuServe (type GO DDJ). The *DDJ* Listing Service (603-882-1599) supports 300/1200/2400 baud, 8-data bits, no parity, 1-stop bit. Press SPACEBAR when the system answers, type: listings (lowercase) at the log-in prompt.

DDJ

(Listings begin on page 125.)

Vote for your favorite feature/article.
Circle Reader Service **No. 4.**

Learn in 3 Intensive Days How Object-Oriented Approaches Will Change the World of Software Development in the 1990s.

Larry L. Constantine
Chairperson

THE OBJECT-ORIENTED SYSTEMS SYMPOSIUM

Speakers Include:
- Bertrand Meyer
- David W. Embley
- Robert Howard & Marie Lenzi
- Burt Rubenstein
- Stephen Schur
- Paul Ward
- Rebecca Wirfs-Brock

Product Education Sessions By:
- AICorp, Inc.
- Aion Corporation
- Complete Computer Corporation
- Interactive Development Environments, Inc.
- Object Design, Inc.
- Softlab, Inc.
- The Stepstone Corp.
- Symbolics, Inc.
- The Whitewater Group

Chicago
December 5-7, 1989

Tutorial Session:
December 4, 1989

Digital Consulting, Inc.

dci

For detailed 8-page brochure:
Call (508) 470-3880

1A29DW

CIRCLE NO. 70 ON READER SERVICE CARD