Applying "Design by Contract"

Bertrand Meyer

Interactive Software Engineering

Reliability is even more important in object-oriented programming than elsewhere. This article shows how to reduce bugs by building software components on the basis of carefully designed contracts.

As object-oriented techniques steadily gain ground in the world of software development, users and prospective users of these techniques are clamoring more and more loudly for a "methodology" of object-oriented software construction — or at least for some methodological guidelines. This article presents such guidelines, whose main goal is to help improve the reliability of software systems. *Reliability* is here defined as the combination of correctness and robustness or, more prosaically, as the absence of bugs.

Everyone developing software systems, or just using them, knows how pressing this question of reliability is in the current state of software engineering. Yet the rapidly growing literature on object-oriented analysis, design, and programming includes remarkably few contributions on how to make object-oriented software more reliable. This is surprising and regrettable, since at least three reasons justify devoting particular attention to reliability in the context of object-oriented development:

- The cornerstone of object-oriented technology is reuse. For reusable components, which may be used in thousands of different applications, the potential consequences of incorrect behavior are even more serious than for application-specific developments.
- Proponents of object-oriented methods make strong claims about their beneficial effect on software quality. Reliability is certainly a central component of any reasonable definition of quality as applied to software.
- The object-oriented approach, based on the theory of abstract data types, provides a particularly appropriate framework for discussing and enforcing reliability.

The pragmatic techniques presented in this article, while certainly not providing infallible ways to guarantee reliability, may help considerably toward this goal. They rely on the theory of *design by contract*, which underlies the design of the Eiffel analysis, design, and programming language[1] and of the supporting libraries, from which a number of examples will be drawn.

The contributions of the work reported below include

- a coherent set of *methodological principles* helping to produce correct and robust software;
- a systematic approach to the delicate problem of how to deal with abnormal cases, leading to a simple and powerful *exception-handling* mechanism; and

0018-9162/92/1000-0040$03.00 © 1992 IEEE

- a better understanding of *inheritance* and of the associated techniques (redeclaration, polymorphism, and dynamic binding) through the notion of subcontract, allowing a systematic approach to using these powerful but sometimes dangerous mechanisms.

Most of the concepts presented here have appeared elsewhere. They were previewed in the book *Object-Oriented Software Construction*[2]; and a more complete exposition was presented in a recent book chapter,[3] from which this article has been adapted. More profoundly, this work finds its root in earlier work on systematic program development[4,5] and abstract data types.[6-8] This article focuses on the central ideas, introducing them concisely for direct application by developers.

Defensive programming revisited

Software engineering and programming methodology textbooks that discuss reliability often emphasize the technique known as *defensive programming*, which directs developers to protect every software module against the slings and arrows of outrageous fortune. In particular, this encourages programmers to include as many checks as possible, even if they are redundant with checks made by callers. Include them anyway, the advice goes; if they do not help, at least they will not harm.

This approach suggests that routines should be as general as possible. A partial routine (one that works only if the caller ensures certain restrictive conditions at the time of the call) is considered dangerous because it might produce unwanted consequences if a caller does not abide by the rules.

This technique, however, often defeats its own purposes. Adding possibly redundant code "just in case" only contributes to the software's complexity — the single worst obstacle to software quality in general, and to reliability in particular. The result of such blind checking is simply to introduce more software, hence more sources of things that could go wrong at execution time, hence the need for more checks, and so on ad infinitum. Such blind and often redundant checking causes much of the complexity and unwieldiness that often characterizes software.

Obtaining and guaranteeing reliability requires a more systematic approach. In particular, software elements should be considered as implementations meant to satisfy well-understood specifications, not as arbitrary executable texts. This is where the contract theory comes in.

The notion of contract

Assume you are writing some program unit implementing a task to be performed at runtime. Unless the task is trivial, it involves a number of subtasks. For example, it might appear as

```
my_task is
  do
    subtask1 ;
    subtask2 ;
    ...
    subtaskn ;
  end
```

a form that suffices for this discussion, although in many cases the control structure linking the various subtasks is less simple than the mere sequencing shown here.

For each of these subtasks, you may either write the corresponding solution in line as part of the body of *my_task*, or rely on a call to another unit. The decision is a typical design trade-off: Too much calling causes fragmentation of the software text; too little results in overcomplex individual units.

Assume you decide to use a routine call for one of the subtasks. This is similar to the situation encountered in everyday life when you decide to contract out for a certain (human) task rather than doing it yourself. For example, if you are in Paris and want an urgent letter or package delivered to another Paris address, you may decide to deliver it yourself, or you may contract out the task to a courier service.

Two major properties characterize human contracts involving two parties:

- Each party expects some benefits from the contract and is prepared to incur some obligations to obtain them.
- These benefits and obligations are documented in a contract document.

Table 1 shows an imaginary roster of obligations and benefits for the courier service of the example.

A contract document protects both sides:

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

As evidenced by this example, what is an obligation for one party is usually a benefit for the other.

This example also suggests a somewhat more subtle observation, which is important in the following discussion (and in studying the application of these ideas to concurrent computation). If the contract is exhaustive, every "obligation" entry also in a certain sense describes a "benefit" by stating that the constraints given are the only relevant ones. For example, the obligation entry for the client indicates that a client who satisfies all the constraints listed is *entitled* to the benefits shown in the next entry. This is the No Hidden Clauses rule: With a fully spelled out contract between honest parties, no requirement

Table 1. Example contract.

Party	Obligations	Benefits
Client	Provide letter or package of no more than 5 kgs, each dimension no more than 2 meters. Pay 100 francs.	Get package delivered to recipient in four hours or less.
Supplier	Deliver package to recipient in four hours or less.	No need to deal with deliveries too big, too heavy, or unpaid.

other than the contract's official obligations may be imposed on the client as a condition for obtaining the contract's official benefits.

The No Hidden Clauses principle does not prevent us from including references, implicit or explicit, to rules not physically part of the contract. For example, general rules such as the relevant laws and common business practices are implicitly considered to be part of every contract of a certain kind, even if not explicitly repeated in the text of each contract. They apply to both client and supplier and will lead below to the notion of class invariant.

```
routine_name (argument declarations) is
                -- Header comment
        require
                Precondition
        do
                Routine body, i.e. instructions
        ensure
                Postcondition
        end
```

Figure 1. A routine equipped with assertions.

```
put_child (new: NODE) is
        -- Add new to the children of current node
    require
        new /= Void
    do
        ... Insertion algorithm ...
    ensure
        new.parent = Current;
        child_count = old child_count + 1
    end -- put_child
```

Figure 2. Assertions for child insertion routine.

Assertions: Contracting for software

It is not difficult to see how the preceding ideas apply to software construction. If the execution of a certain task relies on a routine call to handle one of its subtasks, it is necessary to specify the relationship between the client (the caller) and the supplier (the called routine) as precisely as possible. The mechanisms for expressing such conditions are called assertions. Some assertions, called preconditions and postconditions, apply to individual routines. Others, the class invariants, constrain all the routines of a given class and will be discussed later.

It is important to include the preconditions and postconditions as part of routine declarations (see Figure 1).

In this Eiffel notation, the Require and Ensure clauses (as well as the header comment) are optional. They introduce assertions — respectively the precondition and the postcondition. Each assertion is a list of Boolean expressions, separated by semicolons; here a semicolon is equivalent to a Boolean "and" but allows individual identification of the assertion clauses.

The precondition expresses requirements that any call must satisfy if it is to be correct; the postcondition expresses properties that are ensured in return by the execution of the call.

A missing precondition clause is equivalent to the clause Require True, and a missing postcondition to the clause Ensure True. The assertion True is the least committing of all possible assertions. Any possible state of the computation will satisfy it.

Consider, for example, in a class *TREE* describing tree nodes, a routine *put_child* for adding a new child to a tree node *Current*. The child is accessible through a reference, which must be attached to an existing node object. Table 2 informally expresses the contract.

This is the contract enforced by *put_child* on any potential caller. It contains the most important information that can be given about the routine: what each party in the contract must guarantee for a correct call, and what each party is entitled to in return. Because this information is so crucial to the construction of reliable systems using such routines, it should be a formal part of the routine's text (see Figure 2).

A few more details about the rules of object-oriented programming as embodied in Eiffel should help make this example completely clear:

- A reference such as *new* is either void (not attached to any object) or attached to an object. In the first case, it equals the value *Void*. Here the precondition expresses that the reference *new* must not be void, as stated informally by the corresponding entry in Table 2.
- In accordance with Eiffel's object-oriented principles, the routine will appear in the text of a class describing trees, or tree nodes. This is why it does not need an argument representing the node to which the routine will add the reference *new* as a child; all routines of the class are relative to a typical tree node, the "current instance" of the class. In a specific call such as *some_node.put_child* (*x*), the value before the period, here *some_node*, serves as the current instance.
- In the text of the class, the predefined name *Current* serves, if necessary, to refer to the current instance. Here it is used in the postcondition.
- The notation Old *child_count*, appearing in the postcondition of *put_child*, denotes the value of *child_count* as captured on entry to a particular call. In

Table 2. The *put_child* contract.

Party	Obligations	Benefits
Client	Use as argument a reference, say *new*, to an existing node object.	Get updated tree where the Current node has one more child than before; *new* now has Current as its parent.
Supplier	Insert new node as required.	No need to do anything if the argument is not attached to an object.

other words, the second clause of the postcondition expresses that the routine must increase *child_count* by one. The construct Old may appear only in a routine postcondition.

```
if new = Void then
        ... Take care of special case ...
else
        ... Take care of standard case ...
end
```

Figure 3. Handling a special case.

The role of assertions

You may well be wondering what happens if one of these conditions fails to be satisfied during execution. This question will be answered by whether assertions are monitored at runtime, depending on programmer wishes. But this is not a crucial question at this point. The prime goal of this discussion is to find ways of writing reliable software — systems that work. The question of what happens when they do *not* work, although practically significant, comes only after achieving that more fundamental goal.

Another way of expressing this observation is to notice that assertions do *not* describe special but expected cases that call for special treatment. In other words, the above assertions are not a way to describe (for example) the handling of void arguments to *put_child*. If we wanted to treat void arguments as an acceptable (although special) case, we would handle it not through assertions but through standard conditional control structures (see Figure 3).

Further sources

One of the two primary sources of inspiration for this work is the research on program proving and systematic program construction pioneered by Floyd,[1] Hoare,[2] and Dijkstra.[3] Other well-known work on the application of proof methods to software construction includes contributions by Gries[4] and Mills.[5] The other major influence is the theory of abstract data types (see references in the body of the article).

The use of assertions in an object-oriented language and the approach to inheritance presented here (based on the notion of subcontracting) appear original to Eiffel. The exception-handling model and its implementation are also among Eiffel's contributions. These mechanisms, and the reasoning that led to them, are discussed in detail in references 1 and 2 of the main bibliography at the end of the article.

The notion of class invariant comes directly from Hoare's data invariants.[6] Invariants, as well as other assertions, also play an important role in the VDM software specification method, as described by Jones.[7] The transposition of data invariants to object-oriented software development, in the form of class invariants, appears to be new with Eiffel.

Nonobject-oriented research languages that support assertions have included Euclid[8] and Alphard[9]; see also the Ada-based specification language Anna.[10] CLU, cited in the text, includes nonformal assertions.

The view of programs as mechanisms to compute partial functions is central in the mentioned VDM method.

Another view of exceptions can be found in Cristian.[11] Eiffel's notion of a rescue clause bears some resemblance to Randell's recovery blocks,[12] but the spirit and aims are different. Recovery blocks as defined by Randell are alternate implementations of the original goal of a routine, to be used when the initial implementation fails to achieve this goal. In contrast, a rescue clause does not attempt to carry on the routine's official business; it simply patches things up by bringing the object to a stable state. Any retry attempt uses the original implementation again. Also, recovery blocks require that the initial system state be restored before an alternate implementation is tried after a failure. This appears impossible to implement in any practical environment for which efficiency is of any concern. Eiffel's rescue clauses do not require any such preservation of the state; the only rule is that the rescue clause must restore the class invariant and, if resumption is attempted, the routine precondition.

The rescue clause notion was actually derived from a corresponding formal notion of surrogate function, also called doppelgänger, which appeared in the specification method and language M,[13] a direct successor to Abrial's original Z language.[14] Like Z and unlike Eiffel, M was a formal specification language, not an executable language. Functions in an M specification may be partial. A surrogate is associated with a partial function and serves as a backup for arguments that do not belong to that function's domain.

References

1. R.W. Floyd, "Assigning Meanings to Programs," *Proc. Am. Math. Soc. Symp. in Applied Math.*, Vol. 19, J.T. Schwartz, ed., American Mathematical Society, Providence, R.I., 1967, pp. 19-31.
2. C.A.R. Hoare, "An Axiomatic Basis for Computer Programming," *Comm. ACM*, Vol. 12, No. 10, Oct. 1969, pp. 576-580, 583.
3. E.W. Dijkstra, A *Discipline of Programming*, Prentice Hall, Englewood Cliffs, N.J., 1976.
4. D. Gries, *The Science of Programming*, Springer-Verlag, Berlin and New York, 1981.
5. H.D. Mills et al., *Principles of Computer Programming: A Mathematical Approach*, Allyn and Bacon, Boston, 1987.
6. C.A.R. Hoare, "Proof of Correctness of Data Representations," *Acta Informatica*, Vol. 1, No. 4, 1972, pp. 271-281.
7. C.B. Jones, *Systematic Software Development Using VDM*, Prentice Hall, Englewood Cliffs, N.J., 1986.
8. B.W. Lampson et al., "Report on the Programming Language Euclid," *SIGPlan Notices*, Vol. 12, No. 2, Feb. 1977, pp. 1-79.
9. Mary Shaw et al., *Alphard: Form and Content*, Springer-Verlag, Berlin and New York, 1981.
10. D. Luckham and F.W. von Henke, "An Overview of Anna, a Specification Language for Ada," *IEEE Software*, Vol. 2, No. 2, Mar. 1985, pp. 9-22.
11. F. Cristian, "On Exceptions, Failures, and Errors," *Technology and Science of Informatics*, Vol. 4, No. 4, July-Aug. 1985.
12. B. Randell, "System Structure for Software Fault Tolerance," *IEEE Trans. Software Eng.*, Vol. SE-1, No. 2, June 1975, pp. 220-232.
13. B. Meyer, "M: A System Description Method," Tech. Report TRCS85-15, Computer Science Dept., Univ. of California, Santa Barbara, 1986.
14. J.-R. Abrial, S.A. Schuman, and B. Meyer, "A Specification Language," in *On the Construction of Programs*, R. McNaughten and R.C. McKeag, eds., Cambridge University Press, England, 1980.

Assertions (here the precondition) are something else: ways to describe the conditions on which software elements will work, and the conditions they will achieve in return. By putting the condition *new /= Void* in the precondition, we make it part of the routine's specification; the last form shown (with the If) would mean that we have changed that specification, broadening it to include the special case *new = Void* as acceptable.

As a consequence, any runtime violation of an assertion is not a special case but always the manifestation of a software bug. To be precise:

- A precondition violation indicates a bug in the client (caller). The caller did not observe the conditions imposed on correct calls.
- A postcondition violation is a bug in the supplier (routine). The routine failed to deliver on its promises.

Observations on software contracts

In Table 2, the bottom-right entry is particularly noteworthy. If the precondition is not satisfied, the routine is not bound to do anything, like a mail delivery company given a parcel that does not meet the specification. This means that the routine body should *not* be of the form mentioned above:

```
if new = Void then
        ...
else
        ...
end
```

Using such a construction would defeat the purpose of having a precondition (Require clause). This is an absolute rule: Either you have the condition in the Require, or you have it in an If instruction in the body of the routine, but never in both.

This principle is the exact opposite of the idea of defensive programming, since it directs programmers to avoid redundant tests. Such an approach is possible and fruitful because the use of assertions encourages writing software to spell out the consistency conditions that could go wrong at runtime. Then instead of checking blindly, as with defensive programming, you can use clearly defined contracts that assign the responsibility for each consistency condition to one of the parties. If the contract is precise and explicit, there is no need for redundant checks.

```
invariant
        left /= Void implies (left.parent = Current);
        right /= Void implies (right.parent = Current)
```

Figure 4. An invariant for binary trees.

The stronger the precondition, the heavier the burden on the client, and the easier for the supplier. The matter of who should deal with abnormal values is essentially a pragmatic decision about division of labor: The best solution is the one that achieves the simplest architecture. If every routine and caller checked for every possible call error, routines would never perform any useful work.

In many existing programs, one can hardly find the islands of useful processing in oceans of error-checking code. In the absence of assertions, defensive programming may be the only reasonable approach. But with techniques for defining precisely each party's responsibility, as provided by assertions, such redundancy (so harmful to the consistency and simplicity of the structure) is not needed.

Who should check?

The rejection of defensive programming means that the client and supplier are not both held responsible for a consistency condition. Either the condition is part of the precondition and must be guaranteed by the client, or it is not stated in the precondition and must be handled by the supplier.

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

The experience with Eiffel, in particular the design of the libraries, suggests that the systematic use of a demanding style can be quite successful. In this approach, every routine concentrates on doing a well-defined job so as to do it well, rather than attempting to handle every imaginable case. Client programmers do not expect miracles. As long as the conditions on the use of a routine make sense, and the routine's documentation states these conditions (the contract) explicitly, the programmers will be able to use the routine properly by observing their part of the deal.

One objection to this style is that it seems to force every client to make the same checks, corresponding to the precondition, and thus results in unnecessary and damaging repetitions. But this argument is not justified:

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

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

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

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

where the postcondition of *s* implies *p*.

- Assume that many clients will indeed need to check for the precondition. Then what matters is the "Special Treatment." It is either the same for all calls or specific to each call. If it is the same, causing undue repetition in various clients, this is simply the sign of a poor class interface design, using an overly demanding contract for *r*. The contract should be renegotiated and made broader (more tolerant) to include the standard Special Treatment as part of the routine's specification.
- If, however, the Special Treatment is different for various clients, then the need for each client to perform its own individual test for *p* is intrinsic and not

a consequence of the design method suggested here. These tests would have to be included anyway.

The last case corresponds to the frequent situation in which a supplier simply lacks the proper context to handle abnormal cases. For example, it is impossible for a general-purpose STACK module to know what to do when requested to pop an element from an empty stack. Only the client — a module from a compiler or other system that uses stacks — has the needed information.

Class invariants

Routine preconditions and postconditions may be used in non-object-oriented approaches, although they fit particularly well with the object-oriented method. Invariants, the next major use of assertions, are inconceivable outside of the object-oriented approach.

A class invariant is a property that applies to all instances of the class, transcending particular routines. For example, the invariant of a class describing nodes of a binary tree could be of the form shown in Figure 4, stating that the parent of both the left and right children of a node, if these children exist, is the node itself. (The Implies operator denotes implication. Eiffel operator precedence rules make the parentheses unnecessary here; they have been added for clarity.)

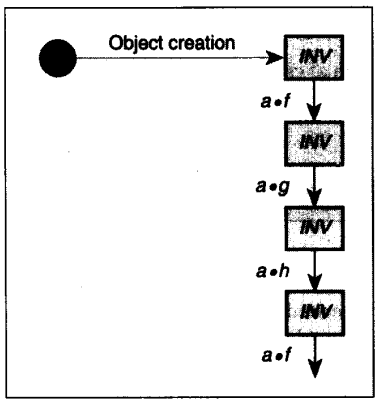

Figure 5. The invariant in an object's life cycle.

The optional class invariant clause appears at the end of a class text:

class *BINARY_TREE* [*T*] **feature**

... Attribute and routine declarations ...

invariant

... As shown above ...

end — class *TABLE*

Two properties characterize a class invariant:

- The invariant must be satisfied after the creation of every instance of the class (every binary tree in this example). This means that every creation procedure of the class must yield an object satisfying the invariant. (A class may have one or more creation procedures, which serve to initialize objects. The creation procedure to be called in any given case is specified in the creation instruction.)
- The invariant must be preserved by every exported routine of the class (that is to say, every routine available to clients). Any such routine must guarantee that the invariant is satisfied on exit if it was satisfied on entry.

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

Figure 5 illustrates these requirements by picturing the life cycle of any object as a sequence of transitions between "observable" states. Observable states, shown as shaded rectangles, are the states that immediately follow object creation, and any states subsequently reached after the execution of an exported routine of the object's generating class. The invariant is the consistency constraint on observable states. (It is not necessarily satisfied in between these states.)

The invariant corresponds to what

On the assertion language

This article includes many examples of typical assertions. But what exactly is permissible in an assertion?

Eiffel assertions are Boolean expressions, with a few extensions such as the old notation. Since the whole power of Boolean expressions is available, they may include function calls. Because the full power of the language is available to write these functions, the conditions they express can be quite sophisticated. For example, the invariant of a class *ACYCLIC_GRAPH* may contain a clause of the form

not *cyclic*

where *cyclic* is a Boolean-valued function that determines whether a graph contains any cycles by using the appropriate graph algorithm.

In some cases, one might want to use quantified expressions of the form "For all *x* of type *T*, *p* (*x*) holds" or "There exists *x* of type *T*, such that *p* (*x*) holds," where *p* is a certain Boolean property. Such expressions are not available in Eiffel. It is possible, however, to express the corresponding properties by using the same technique: calls to functions that rely on loops to emulate the quantifiers.

Although some thought has been given to extend the language to include a full-fledged formal specification language, with first-order predicate calculus, the need for such an extension does not seem crucial. In Eiffel, intended as a vehicle for industrial software development rather than just for research, the use of function calls in assertions seems to provide an acceptable trade-off between different design goals: reliability, the ability to generate efficient code, and overall simplicity of the language.

In fact, first-order predicate calculus would not necessarily be sufficient. Many practically important properties, such as the requirement that a graph be noncyclic, would require higher order calculus.

The use of functions — that is to say, computations — is not without its dangers. In software, a function is a case of a routine; it prescribes certain actions. This makes software functions *imperative*, whereas mathematical functions are said to be *applicative*. The major difference is that software functions can produce side effects (change the state of the computation). Introducing functions into assertions lets the imperative fox back into the applicative chicken coop.

In practice, this means that any function used in assertions must be of unimpeachable quality, avoiding any change to the current state and any operation that could result in abnormal situations.

```
put_child (new: NODE)
                -- Add new to the children of current node
        require
                new /= Void
        ensure
                new.parent = Current;
                child_count = old child_count + 1
```

Figure 6. The short form of a routine.

was called "general conditions" in the initial discussion of contracts: laws or regulations that apply to all contracts of a certain category, often through a clause of the form "all provisions of the XX code shall apply to this contract."

Documenting a software contract

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

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

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

The *short* command retains only the exported features of a class and, for an exported routine, drops the routine body and any other implementation-related details. However, pre- and postconditions are kept. (So is the header comment if present.) For example, Figure 6 shows what the *short* command yields for the *put* routine. It expresses simply and concisely the purpose of the routine, without reference to a particular implementation.

All documentation on the details of Eiffel classes (for example, the class specifications in the book on the basic libraries[9]) is produced automatically in this fashion. For classes that inherit from others, the *short* command must be combined with another tool, *flat*, which flattens out the class hierarchy by including inherited features at the same level as "immediate" ones (those declared in the class itself).

Monitoring assertions

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

In the development environment, the answer depends on a compilation option. For each class, you may choose from various levels of assertion monitoring: no assertion checking, preconditions only (the default), preconditions and postconditions, all of the above plus class invariants, or all assertions. (The difference between the last two follows from the existence of other assertions, such as loop invariants, not covered in the present discussion.)

For a class compiled under the "no assertion monitoring" option, assertions have no effect on system execution. The subsequent options cause evaluation of assertions at various stages: routine entry for preconditions, routine exit for postconditions, and both steps for invariants.

Under the monitoring options, the effect of an assertion violation is to raise an exception. The possible responses to an exception are discussed later.

Why monitor?

As noted, assertion violations are not special (but expected) cases; they result from bugs. The main application of run-time assertion monitoring, then, is debugging. Turning assertion checking on (at any of the levels previously listed) makes it possible to detect mistakes.

When writing software, developers make many assumptions about the properties that will hold at various stages of the software's execution, especially routine entry and return. In the usual approaches to software construction, these assumptions remain informal and implicit. Here the assertion mechanism enables developers to express them explicitly. Assertion monitoring, then, is a way to call the developer's bluff by checking what the software does against what its author thinks it does. This yields a productive approach to debugging, testing, and quality assurance, in which the search for errors is not blind but based on consistency conditions provided by the developers themselves.

Particularly interesting here is the use of *preconditions* in library classes. In the general approach to software construction suggested by the Eiffel method, developers build successive "clusters" of classes in a bottom-up order, from more general (reusable) to more specific (application-dependent). This is the "cluster model" of the software life cycle.[10] Deciding to release a library cluster *l* for general use normally implies a reasonable degree of confidence in its quality — the belief that no bugs remain in *l*. So it may be unnecessary to monitor the postconditions of routines in the classes of *l*. But the classes of an application cluster that is a client of *l* (see Figure 7) may still be "young" and contain bugs; such bugs may show up as erroneous arguments in calls to routines of the classes of *l*. Monitoring preconditions for classes of *l* helped to find them. This is one of the reasons why precondition checking is the default compilation option.

Introducing inheritance

One of the consequences of the contract theory is a better understanding and control of the fundamental object-oriented notion of inheritance and of the key associated techniques: redeclaration, polymorphism, and dynamic binding.

Through inheritance, you can define new classes by combining previous ones. A class that inherits from another has all the features (routines and attributes) defined in that class, plus its own. But it is not required to retain the exact form of inherited features: It may *redeclare* them to change their specification, their implementation, or both. This flexibility of the inheritance mechanism is central to the power of the object-oriented method.

For example, a binary tree class could provide a default representation and

the corresponding implementations for search and insertion operations. A descendant of that class may provide a representation that is specifically adapted to certain cases (such as almost full binary trees) and redeclare the routines accordingly.

Such a form of redeclaration is called a *redefinition*. It assumes that the inherited routine already had an implementation. The other form of redeclaration, called *effecting*, applies to features for which the inherited version, known as a deferred (or abstract) feature, had no implementation, but only a specification. The effecting then provides an implementation (making the feature effective, the reverse of deferred). The subsequent discussion applies to both forms of redeclaration, although for simplicity it concentrates on redefinition.

Redeclaration takes its full power thanks to polymorphism and dynamic binding. Polymorphism is type adaptation controlled by inheritance. More concretely, this means that if you have *b* of type *BINARY_TREE* and *sb* of type *SPECIAL_BINARY_TREE*, the latter class a descendant of the former, then the assignment

$b := sb$

is permitted, allowing *b* to become attached at runtime to instances of *SPECIAL_BINARY_TREE*, of a more specialized form than the declaration of *b* specifies. Of course, this is only possible if the inheritance relation holds between the two classes as indicated.

What happens then for a call of the form

t.insert (*v*)

which applies procedure *insert*, with argument *v*, to the object attached to *t*? Dynamic binding means that such a call always uses the appropriate version of the procedure — the original one if the object to which *t* is attached is an instance of *BINARY_TREE*, the redefined version if it is an instance of *SPECIAL_BINARY_TREE*. The reverse policy, static binding (using the declaration of *b* to make the choice), would be an absurdity: deliberately choosing the wrong version of an operation.

The combination of inheritance, redeclaration, polymorphism, and dynamic binding yields much of the power and flexibility that result from the use of the object-oriented approach.[2] Yet these techniques may also raise concerns of possible misuse: What is to prevent a redeclaration from producing an effect that is incompatible with the semantics of the original version — fooling clients in a particularly bad way, especially in the context of dynamic binding? Nothing, of course. No design technique is immune to misuse. But at least it is possible to help serious designers use the technique properly; here the contract theory provides the proper perspective.

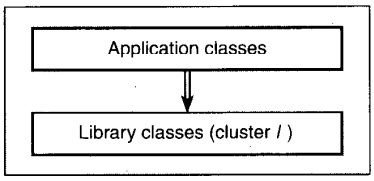

Figure 7. Library cluster and application cluster.

What redeclaration and dynamic binding mean is the ability to *subcontract* a task; preventing misuse then means guaranteeing that subcontractors honor the prime contractor's promises in the original contract.

Consider the situation described by Figure 8. *A* exports a routine *r* to its clients. (For simplicity, we ignore any arguments to *r*.) A client *X* executes a call

u.r

where *u* is declared of type *A*. Now *B*, a

The concurrency issue

The theory of design by contract raises important questions regarding the application of object-oriented ideas to concurrent computation. In discussing contracts, this article mentions that clients may view the precondition of a routine not just as an obligation but also in part as a benefit, since the contract implicitly indicates that a call satisfying the precondition will be serviced correctly. This is the No Hidden Clause rule. For example, if the insertion routine *put* for a *BOUNDED_QUEUE* class has the precondition

not *full*

to state that an insertion operation requires a queue that is not full, then a protected call of the form

```
q: BOUNDED_QUEUE [T];
x: T;
...
if not q.full then
        q.put (x)
end
```

will succeed, since the client executing this call has taken the trouble to check the precondition explicitly.

In parallel computation, however, things are not so nice. The bounded queue in this example may be used as a bounded buffer, accessible to several processors. The processor in charge of the client, which will carry out the above instructions, and the processor in charge of *q*, which will carry out the execution of *put*, could be different processors. Then, even if the test for *q.full* yields false, between the time the client executes this test and the time it executes the call *q.put* (*x*), quite a few events may have occurred. For example, another client may have made the queue full by executing its own call to *put*.

In other words, a different semantic interpretation may be necessary for preconditions in the context of parallel computation. This observation serves as the starting point for some of the current work on models for concurrent object-oriented programming.[1,2]

References

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

2. J. Potter and G. Jalloul, "Models for Concurrent Eiffel," in *TOOLS 6* (Technology of Object-Oriented Languages and Systems), Prentice Hall, Englewood Cliffs, N.J., 1991, pp. 183-192.

descendant of A, redeclares r. Through polymorphism, u may well become attached to an instance of B rather than A. Note that often there is no way to know this from the text of X alone; for example, the call just shown could be in a routine of X beginning with

some_routine (u: A) **is** ...

where the polymorphism only results from a call of the form

z.*some_routine* (v)

for which the actual argument v is of type B. If this last call is in a class other than X, the author of X does not even know that u may become attached to an instance of B. In fact, he may not even know about the existence of a class B.

But then the danger is clear. To ascertain the properties of the call $u.r$, the author of X can only look at the contract for r in A. Yet, because of dynamic binding, A may subcontract the execution of r to B, and it is B's contract that will be applied.

How do you avoid "fooling" X in the process? There are two ways B could violate its prime contractor's promises:

- B could make the precondition stronger, raising the risk that some calls that are correct from X's viewpoint (they satisfy the original client obligations) will not be handled properly.
- B could make the postcondition weaker, returning a result less favorable than what has been promised to X.

None of this, then, is permitted. But the reverse changes are of course legitimate. A redeclaration may weaken the original's precondition or it may strengthen the postcondition. Changes of either kind mean that the subcontractor does a *better* job than the original contractor — which there is no reason to prohibit.

These rules illuminate some of the fundamental properties of inheritance, redeclaration, polymorphism, and dynamic binding. Redeclaration, for all the power it brings to software development, is not a way to turn a routine into something completely different. The new version must remain compatible with the original specification, although it may improve on it. The noted rules express this precisely.

These rules must be enforced by the

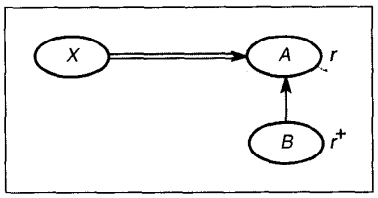

Figure 8. Redefinition of a routine under contract.

language. Eiffel uses a simple convention. In a redeclaration, it is not permitted to use the forms *require...* and *ensure...*. The absence of a precondition or postcondition clause means that the redeclared version retains the original version's assertion. Since this is the most frequent situation, the class author is not required to write anything special in this case. A class author who does want to adapt the assertion will use either or both of the forms

require else
    new_pre

ensure then
    new_post

which yield the following as new precondition and postcondition:

new_pre **or else** *original_precondition*

new_post **and then**
    original_postcondition

where Or Else and And Then are the noncommutative versions of the "or" and "and" operators (evaluating their second argument only if necessary). In this way, the new precondition is guaranteed to be weaker than or equal to the originals, and the new postcondition is guaranteed to be stronger than or equal to the originals.

Invariants and dynamic binding

In addition to the rules on preconditions and postconditions, another constraint ties assertions with inheritance: Invariants are always passed on to descendants.

This is a direct result of the view that inheritance is (among other things) classification. If we want to consider every instance of a class B as being also an instance of B's ancestors, we must accept that consistency constraints on a parent A also apply to instances of B.

For example, if the invariant for a class *TREE*, describing tree nodes, includes the clause

child.parent = *Current*

expressing that the parent of a node's currently active child is the node itself, this clause will automatically apply to instances of a class *BINARY_TREE*, which inherits from *TREE*. As a result, the language specification defines "the invariant of a class" as the assertion obtained by concatenating the assertion in the invariant clause of the class to the invariants of all parents (obtained recursively under this definition).[1]

As a result, the invariant of a class is always stronger than or equal to the invariants of each of its parents.

These rules lead to a better understanding of why static binding would be, as previously stated, such a disaster. Assume again the declaration and call

u: A;
...
$u.r$

where a descendant B of A redefines r. Call r_A and r_B the two implementations. Then r_A must preserve INV_A, the invariant of A, and r_B must preserve INV_B, the invariant of B, which is stronger than or equal to INV_A.

There is, of course, no requirement that r_A preserve INV_B. In fact, class A may have been written long before B, and the author of A does not need to know anything about eventual descendants of this class.

If u dynamically becomes attached to an instance of B, dynamic binding requires the execution of r_B for this call. Static binding would trigger r_A. Since this version of the routine is not required to preserve INV_B, the result would yield a catastrophic situation: an object of type B that does not satisfy the consistency constraint — the invariant — of its own class. In such cases, any attempt at understanding software texts or reasoning about their runtime behavior becomes futile.

A simple example will make the situation more concrete. Assume a class *ACCOUNT* describing bank accounts, with the attributes shown in Figure 9a and a procedure to record a new deposit shown in Figure 9b.

With this version of the class, obtaining an account's current balance requires a computation expressed by a function. Figure 10 shows how the *balance* function could appear, assuming the appropriate function *sum* in class *TRANSACTION_LIST*.

In a descendant class *ACCOUNT1*, it may be a better space-time trade-off to store the current balance with every account object. This can be achieved by redefining the function *balance* into an attribute (a process that is indeed supported by the language). Naturally, this attribute must be consistent with the others; this is expressed by the invariant of *ACCOUNT1*, shown in Figure 11.

For this to work, however, *B* must redefine any routine of *A* that modified deposits or withdrawals; the redefined version must also modify the *balance* field of the object accordingly, so as to maintain the invariant. This is the case, for example, with procedure *record_deposit*.

Now assume that we have the declaration and call

a: *ACCOUNT*;
...
a.record_deposit (*1_000_000*)

If in a certain execution, *a* happens to be attached to an object of type *ACCOUNT1* at the time of the call, static binding would mean applying the original, *ACCOUNT* version of *record_deposit* — which fails to update the *balance* field. The result would be an inconsistent *ACCOUNT1* object and certain disaster.

```
initial_deposit: INTEGER;
deposits, withdrawals: TRANSACTION_LIST
(a)

record_deposit (d: INTEGER) is
        do
                ... Update the deposits list ...
        end -- record_deposit
(b)
```

Figure 9. Features of a Bank Account class.

```
balance: INTEGER is
                -- Current balance
        do
                balance := initial_deposit +
                                deposits.sum -
                                withdrawals.sum

        end -- balance
```

Figure10. Computing the balance.

```
invariant
        balance = initial_deposit + deposits.sum
                        - withdrawals.sum
```

Figure 11. Invariant of the Account class.

Dealing with abnormal situations

The Design by Contract theory has one more immediate application to the practice of reliable software development: exception handling.

Exceptions — abnormal cases — have been the target of much study; and several programming languages, notably Ada, PL/I, and CLU, offer exception-handling mechanisms. Much of this work is disappointing, however, because it fails to define precisely what an abnormal case is. Then exception handling often becomes a kind of generalized, interroutine "goto" mechanism, with no clear guidelines for proper use.

To understand the issue better, I performed a study (reported elsewhere[3]) of Ada and CLU textbooks, looking for examples of exception handling and methodological principles. The results were disappointing, as the books showed many examples of *triggering* exceptions but few of how to *handle* them. Furthermore, some of the latter were hair-raising. For example, one textbook proposed an example of a square root routine which, when confronted with a negative argument, triggers an exception. The exception handler prints a message and then simply returns to the caller without notifying the caller that something wrong has occurred — fooling the caller, as it were, into believing that everything is going according to plan. Since a typical use for square roots in a typical Ada program is a missile trajectory computation, it is easy to foresee the probable consequences.

Beyond the bad taste of such individual examples, one may fault the design of the exception mechanism itself for failing to encourage, or even to define, a proper discipline for handling abnormal cases.

The contract theory provides a good starting point for a more rational solution. If a routine is seen not just as some "piece of code" but as the implementation of a certain specification — the contract — it is possible to define a notion of *failure*. Failure occurs when an execution of a routine cannot fulfill the routine's contract. Possible reasons for a failure include a hardware malfunction, a bug in the implementation, or some external unexpected event.

"Failure" is here the basic concept. "Exception" is a derived notion. An exception occurs when a certain strategy for fulfilling a routine's contract has not succeeded. This is not a failure, at least not yet, because the routine may have an alternative strategy.

The most obvious example of exception is the failure of a called routine: *r*'s strategy for fulfilling its contract involved a call to *s*; the call failed; clearly, this is an exception for *r*. Another example, previously mentioned, is a runtime assertion violation, if assertions are monitored. It is also convenient to treat as exceptions the signals sent by the operating system or the hardware: arithmetic overflow, memory exhaustion, and the like. They indeed correspond to failures of calls to basic facilities (arithmetic operations, memory allocation).

Equipped with this notion of failure and exception, we can define a coherent response to an exception. The exception occurs because the strategy used to achieve the routine's contract did not work. Only three possible responses then make sense:

(1) Perhaps an alternative strategy is available. We have lost a battle, but we have not lost the war. In this case the routine should put the objects back into a consistent state and make another attempt, using the new strategy. This is called *resumption*.

(2) Perhaps, however, we have lost the war altogether. No new strategy is

```
get_integer_from_user: INTEGER is
        -- Read an integer (allow user up to five attempts)

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

Figure 12. Reading an integer with an unsafe primitive.

available. Then the routine should put back the objects in a consistent state, give up on the contract, and report failure to the caller. This is called *organized panic*.

(3) A rare but possible third case is the *false alarm*. This may occur only for operating-system or hardware signals. On some multiwindowing systems, for example, a process receives a signal (transformed by the runtime into an exception) when its window is resized. In most cases, the process should be able to continue its execution, possibly after taking some corrective actions (such as registering the new window dimensions for use by editors and other tools).

The description of both resumption and organized panic mentions putting back the objects "in a consistent state." This is essential if further executions (after an eventual resumption) will use the objects again. The notion of consistent state should be clear from the preceding discussion: Any exception handling, whether for resumption or for organized panic, should restore the invariant.

A disciplined exception-handling mechanism

It is not hard to devise an exception mechanism that directly supports the preceding method for handling abnormal cases.

To specify how a routine should behave after an exception, the author of an Eiffel routine may include a "rescue" clause, which expresses the alternate behavior of the routine (and is similar to clauses that occur in human contracts, to allow for exceptional, unplanned circumstances). When a routine includes a rescue clause, any exception occurring during the routine's execution interrupts the execution of the body (the Do clause) and starts execution of the rescue clause. The clause contains zero or more instructions, one of which may be a Retry. The execution terminates in either of two ways:

- If the rescue clause terminates without executing a Retry, the routine fails. It reports failure to its caller by triggering a new exception. This is the organized panic case.
- If the rescue clause executes a Retry, the body of the routine (Do clause) is executed again.

As an example, here is a solution to a problem found in many Ada textbooks: Using a function *getint*, which reads an integer, prompt a user to enter an integer value; if the input is not an integer, ask again, unless the user cannot provide an integer after five attempts, in which case a failure occurs. It is assumed that *getint* is an external routine, perhaps written in C or assembly language, and we have no control over it. It triggers an exception when applied to input that is not an integer; the routine should catch that exception and prompt the user again. Figure 12 shows a solution.

The first five times the interactive user enters a wrong input, the routine starts again, thanks to the Retry. This is the direct implementation of resumption.

The local entity *failures* serves to record the number of failed calls to *getint*. Like any integer local entity, it is automatically initialized to zero on routine call. (The Eiffel language definition[1] specifies simple initialization values for every possible type.)

In this example, only one type of exception is possible. In some cases, the rescue clause might need to discriminate between possible types of exceptions and handle them differently. This is made possible through simple features of the kernel library class *EXCEPTIONS*, although it isn't necessary to look at the (straightforward) details here. This class also provides mechanisms for handling the false alarm case by specifying that for certain signals execution may be allowed to resume.

What happens after five successive failures of *getint*? The rescue clause terminates without executing a Retry and the routine execution fails (organized panic). The key rule in this case is that the caller of *get_integer* will get an exception, which it will have to handle by using the same policy, choosing between organized panic, resumption, and false alarm.

In a typical system, only a handful of routines have an explicit rescue clause. What if an exception occurs during the execution of a routine that has no such clause? The rule is simple: An absent clause is considered equivalent to an implicit clause of the form

```
rescue
        default_rescue
```

where *default_rescue* is a general-purpose procedure that, in its basic form, does nothing. Then an exception simply starts the rescue clause, which, executing the empty *default_rescue*, causes failure of the routine; this triggers the rescue clause, explicit or implicit. If exceptions are passed in this manner all the way back to the "root object" that started the execution, that execution halts after printing an exception history table that clearly documents the sequence of recorded abnormal events. But, of course, some routine in the call

chain may have a rescue clause, even one containing a Retry that will attempt a resumption.

Why define the default behavior as a call to *default_rescue* rather than just as an empty rescue clause? The reason comes from the methodological discussion. In the case of organized panic, it is essential to restore the invariant before conceding defeat and surrendering. A null action would not achieve this for a class with a nontrivial invariant.

The solution is provided once again by the coalesced forces of inheritance and assertions. Procedure *default_rescue*, in its default null form, appears as a procedure of the general-purpose class *ANY*. This library class, as defined by the language rules,[1] is automatically an ancestor of all possible developer-defined classes. So it is the responsibility of designers of a class *C*, if they are concerned about possible exceptions occurring in routines that do not have specific rescue clauses, to redefine *default_rescue* so that it will ensure the class invariant of *C*.

Often, one of the creation procedures may serve as a redefinition of *default_rescue*, since creation procedures are also required to ensure the invariant.

This illuminates the difference between the body (the Do clause) and the rescue clause:

- The body must implement the contract, or ensure the postcondition. For consistency, it must also abide by the general law of the land — preserve the invariant. Its job is made a bit easier by the assumption that the invariant will hold initially, guaranteeing that the routine will find objects in a consistent state.
- In contrast, the rescue clause may not make any such assumption; it has no precondition, since an exception may occur at any time. Its reward is a less-demanding task. All that it is required to do on exit is to restore the invariant. Ensuring the postcondition — the contract — is not its job.

A useful analogy is the contrast between the grandeur and servitude of two equally respectable professions — cook and fire fighter. A cook may assume that the restaurant is not burning (satisfies the invariant) when the workday begins. If the restaurant is indeed nonburning, the cook must prepare meals (ensure the postcondition). It is also a part of the cook's contract, although perhaps an implicit one, to avoid setting the restaurant on fire in the process (to maintain the invariant).

When the fire fighter is called for help, in contrast, the state of the restaurant is not guaranteed. It may be burning or (in the case of a wrong alert) not burning. But then the fire fighter's only duty is to return the restaurant to a nonburning state. Serving meals to the assembled customers is not part of the fire fighter's job description. ■

Status of Eiffel

The definition of the Eiffel language, used as the vehicle for this article, is in the public domain. The language evolution is under the control of an organization of users and developers of Eiffel technology: the Nonprofit International Consortium for Eiffel (NICE). Membership in NICE is open to any interested organization. The address is PO Box 6884, Santa Barbara, CA 93160.

Acknowledgments

I have been greatly influenced by the originators of the classical work on systematic software development, mentioned in the "Further sources" sidebar. With his usual thoroughness, Kim Walden read the text and pointed out errors and possible improvements. The anonymous referees made several useful comments.

Eiffel is a trademark of the Nonprofit International Consortium for Eiffel.

References

1. B. Meyer, *Eiffel: The Language*, Prentice Hall, Englewood Cliffs, N.J., 1991.
2. B. Meyer, *Object-Oriented Software Construction*, Prentice Hall, Englewood Cliffs, N.J., 1988.
3. B. Meyer, "Design by Contract," in *Advances in Object-Oriented Software Engineering*, D. Mandrioli and B. Meyer, eds., Prentice Hall, Englewood Cliffs, N.J., 1991, pp. 1-50.
4. C.A.R. Hoare, "An Axiomatic Basis for Computer Programming," *Comm. ACM*, Vol. 12, No. 10, Oct. 1969, pp. 576-580, 583.
5. E.W. Dijkstra, *A Discipline of Programming*, Prentice Hall, Englewood Cliffs, N.J., 1976.
6. J.A. Goguen, J.W. Thatcher, and E.G. Wagner, "An Initial Algebra Approach to the Specification, Correctness, and Implementation of Abstract Data Types," in *Current Trends in Programming Methodology*, Vol. 4, R.T. Yeh, ed., Prentice Hall, Englewood Cliffs, N.J., 1978, pp. 80-149.
7. J.V. Guttag, "Abstract Data Types and the Development of Data Structures," *Comm. ACM*, Vol. 20, No. 6, June 1977, pp. 396-404.
8. B. Meyer, "La Description des Structures de Données," *Bulletin de la Direction des Etudes et Recherches d'Electricité de France*, Série C (Informatique), No. 2, Paris, 1976.
9. B. Meyer, *Eiffel: The Libraries*, Prentice Hall, Englewood Cliffs, N.J., (to appear in 1993).
10. B. Meyer, "The New Culture of Software Development," *TOOLS 1* (Technology of Object-Oriented Languages and Systems), SOL, Paris, Nov. 1989, pp. 13-23. Slightly revised version in *Advances in Object-Oriented Software Engineering* (see reference 3).

Bertrand Meyer is president of Interactive Software Engineering Inc. and Société des Outils du Logiciel, Paris. His areas of interest include formal specification, design methods, programming languages, interactive systems, software development environments, and various aspects of object-oriented technology.

Meyer holds an engineering degree from Ecole Polytechnique, an MS from Stanford University, and a PhD from the University of Nancy. He is the author of a number of technical books and articles, editor of the Prentice Hall Object-Oriented Series, and chairman of the TOOLS (Technology of Object-Oriented Languages and Systems) conference series.

The author can be contacted at Interactive Software Engineering, 270 Storke Rd., Suite 7, Goleta, CA 93117.