Publication reference: Bertrand Meyer, *From Structured Programming to Object-Oriented Design: The Road to Eiffel*, in *Structured Programming*, vol. 10, no. 1, January 1989, pages 19-39.

TO OBJECT-ORIENTED DESIGN:

THE ROAD TO EIFFEL

Bertrand Meyer

Interactive Software Engineering Inc.
356 Storke Road, Suite 7 Goleta, CA 93117 USA
(805) 685-1006 — http://www.eiffel.com

ABSTRACT

An exploration of how object-oriented design, as implemented in the Eiffel language and environment, integrates and extends the seminal concepts of structured programming.

1 INTRODUCTION

"Structured programming" has two meanings. One refers to a movement of ideas which approximately spanned the decade starting in 1968, a year notable both for Dijkstra's anti-goto letter [12] and for the first software engineering symposium [8]. The other denotes an enduring view of software construction as a valuable scientific pursuit.

"Object-oriented design" is a more recent buzzword. To be fair, the underlying ideas, or at least the most important among them, are as old as the principles of structured programming; but it is only in the nineteen-eighties that object-oriented concepts have reached the limelight, with an éclat that seems like a revenge for the years when Simula 67 [9], the advance ship of the object-oriented fleet, was struggling alone against the mainstream. Not surprisingly for such a fashionable term, "object-oriented" means different things to different people. This discussion will use for object-oriented design a definition similar to one already given elsewhere [31]:

> A method for building the architecture of software systems by combination of basic units called classes, where each class is a possibly partial implementation of some abstract data type, and may be connected to other classes by two relations: *client*, enabling the implementation of a class to rely on the facilities provided by another through its official interface, and *multiple inheritance*, where a class is defined as extension or specialization of one or more others.

The design of the Eiffel language and environment [27, 28, 31, 33], which is the main focus of this article, results from a conscious effort to combine structured programming with object-oriented design. Not that any significant effort is needed to reconcile the two: in spite of a few differences, to be discussed shortly, I will discuss object-oriented techniques and Eiffel not just as "the next thing" after structured programming, but rather as its consequence; paraphrasing Clausewitz, as structured programming continued by other means.

In keeping with the spirit defined by the editor of *Structured Programming*, this article is a free-ranging discussion of a number of issues of software design, programming methodology and programming languages. The reader should view it as a promenade on the border road between structured programming and object-oriented design, with stops whenever we find a spot where one of these regions encroaches on the other.

One non-technical feature that unites structured programming and object-oriented design is their common status as buzzwords, and the resulting trivialization of the concepts. Structured programming is the older and has suffered the worst consequences: widespread casual approval of the ideas at the vaguest level (what programmer openly claims to apply *un*structured techniques?); exaggerated focus on visible but superficial issues (the famous goto); ignorance of the most far-reaching and challenging concepts (most importantly, the systematic use of rigorous program construction techniques rooted in mathematics). A 1984 study [44] showed how small a subset of the structured programming ideas has actually found its way, despite all the posturing, into practical development environments. The same fate obviously threatens object-oriented concepts; already we can hear people saying that they have always used them in their designs (we paid a lot of attention to the data, didn't we?), and any interactive program that includes so much as a menu is triumphantly advertized as the latest in object-oriented technology . Obviously the "vulgar" views of both structured and object-oriented methods are of little relevance to the rest of this discussion.

Although I have tried to acknowledge some of my major debts, I have made no attempt to hide or soften my personal views, as one would in a more formal publication. The purpose of critiquing previous efforts is not to find fault with their authors – with enough hindsight, this exercise is as easy as it is vain – but to go further. Although obvious, these comments are necessary in a discussion that touches on programming methodology and programming languages, two topics on which professionals tend to be rather opinionated. I am no exception; much as this discussion aims to generate light rather than heat, it probably won't be able to avoid offending some of the readers some of the time.

2 IDEAS NOT RETAINED

We should not pretend that *every* precept made popular by the "structured revolution" must be retained in the object-oriented world. The harmony is not perfect, and before we play the object-oriented cadenza to the structured allegro it is appropriate to sound a few dissonances.

2.1 Top-down design

One difference involves the notion of top-down functional design, which after the work of Wirth [42] and Mills [37] has become almost uniformly identified with good software practices. There are some arguments for this approach: it is a rational, systematic, teachable method, which may work for problems that can be characterized by a purely functional specification, known and frozen at the outset. The main flaw of top-down functional techniques, however, is that they neglect fundamental aspects of software construction: the need for change, and the need for reuse. In top-down design, each module is produced in response to a precise subspecification, with no provision for future evolution, and no incentive to make the module any more general than it needs to be for its immediate purpose.

In contrast, although "bottom-up" is about as popular an expression in software circles as "government subsidies" at a convention of the US Republican party, the bottom-up approach is, I believe, the real engineer's view. The term is often taken to mean that designers should start from the lowest possible level – the "bottom". Bottom-up design is actually the opposite: you start from available components and build on them. The important word is *up*: rather than attempting to produce

the best possible solution for the current problem, you try to produce a good solution, minimizing the effort by building on previous achievements, and striving for the highest possible degree of generality to facilitate future developments. Rather than local optimization of the software effort, over a particular state in the life of a particular product, you seek global optimization, over the life of your organization.

A top-down object-oriented method may perhaps be conceived, although I find it hard to imagine, so close is bottom-up design to the very idea of reusability, one of the tenets of the object-oriented approach. The Eiffel view is resolutely bottom-up; the aim of the game is to produce components that may be combined in various ways, not privileging any of these ways during development.

To pick just one example, consider the classes recently added to the Basic Eiffel Library for building compiling tools. These classes do not define a compiler, not even a compiler generator, but rather a set of compiling facilities. You may decide to use these facilities for many different applications: parsing (in the sense of just checking for syntax errors), compiling (transforming from source to object), checking the grammar (for left-recursion or other properties), static analysis, syntax-directed editing etc.

The particular choice of a set of facilities and of their sequencing is the least committing decision of system design; because it is bound to be the first to change, it should be made last. This is reflected in the absence of a main program concept in Eiffel: an Eiffel *system* is simply an assembly of classes, one of which is designated as *root*, or seed for the execution. Changing root is perhaps the easiest operation in the whole Eiffel environment.

2.2 Testing

Another of the ideas and attitudes that emerged from the initial structured programming wave needs to be reconsidered: the general hostility toward testing. Tests were frowned upon in the structured programming literature, following Dijkstra's often quoted remark [10] that "testing can never be used to show the absence of bugs, only to show their presence", with the understanding that only proofs will succeed in achieving the former goal.

This warning against undue reliance on testing was a healthy reaction to haphazard techniques used by most of the industry to try to get software products to work more or less correctly (here the situation is not appreciably better in 1988). No one would, however, seriously contend that software should not be tested before release. There are a number of convincing arguments in favor of testing:

- Proving techniques, in spite of all the effort expended over the past 20 years, cannot be applied to fully prove many practical programs, especially when they involve such features as complex data structures and floating-point arithmetic.
- Proofs require completely formal specifications, which are seldom feasible.
- Even when theoretically feasible in the case of formally specified systems using only provable constructs, proofs may be incorrect, as demonstrated by the serious errors detected by Gerhart, Goodenough and Yelowitz [15, 17] in published programs that had supposedly been proven correct.
- Producing unimpeachable proofs is a tedious process requiring software support, which is not available in a form usable by mainstream software developers.
- Finally tests have other uses than finding deviations from a specification. They can uncover aspects of the system for which the specification was inappropriate (although of course we would hope that these would be uncovered much earlier, perhaps through prototyping), or aspects that had been forgotten in the specification (such as the system's behavior in certain special cases). In this respect they are particularly useful for improving such system features as its user interface and its performance before it is delivered to its users.

Of course the creators of structured programming did not really advocate full formal proofs but rather a general method of software production which, as described by Dijkstra [13] and Gries [18] associates partly formal correctness arguments with the programs as they are being built. Following the work of Floyd [14] and Hoare [20], these arguments are expressed as assertions – expressions describing the state of program entities and used to formalize static properties of operational constructs such as instructions. Assertions play a major role in Eiffel, as will be discussed below.

Perhaps because of the initial taboo on tests in the structured world, most of the subsequent work on testing was performed separately from research on formal specification and verification. This is regrettable. Not only are proving and testing obviously complementary tools in the search for software reliability; they can both benefit from the same fundamental tools: assertions. This will be explored in more detail below.

2.3 Structured design

Structured programming was followed in the nineteen-seventies by a new buzzword, "Structured Design". (Another somewhat related method also has the word "structured" in its acronym: SADT, or Structured Analysis and Design Technique, not to be confused with "Structured Analysis", which is a continuation of the Structured Design work.)

One of the contributions of Constantine's and Yourdon's advocacy of Structured Design [43] is their analysis of modular structures and of techniques that improve the the modularity of a software system (by increasing "cohesion" and decreasing "coupling"). They were able to convince the world that software products have an architecture, independent of the contents of each module. Of course this idea was already present in DeRemer's and Kron description of programming-in-the-large versus programming-in-the-small [11], and earlier in Parnas's seminal work on information hiding [39] (actually an even older paper by Wilkes [41] already contained some important ideas about distinguishing structure from contents).

Beyond this, it is hard to assess the contribution of Structured Design to the overall goal of software engineering – software quality. Whatever its authors may have intended, the main practical effect of Structured Design will have been to convince a notable segment of the computing industry that:

1 • As opposed to "programming", a despicable occupation fit for the toiling masses, design is a noble craft, the only one worthy of consideration by the data processing elite.

2 • This craft is carried out by drawing circles and connecting them with arrows.

In the nineteen-eighties, with the advent of powerful personal computers and workstations with graphics facilities, it became possible to draw the circles and arrows using a computer. The corresponding tools have been labeled "CASE" (Computer-Aided Software Engineering, yet another buzzword) and a whole industry was born around them.

What effect will all this ultimately have? It was certainly important to persuade a large audience that the design process and software architectures are issues worthy of consideration, and that software can benefit just as much as other engineering artefacts from the use of computerized design aids. What we may regret, however, is the trivialization of design, and the total loss of concern for the central issue of structured programming – correctness. Advocates of "CASE tools" repeat the old cliché: a picture is worth a thousand words. Who will recall the difficulty of finding the seven errors in the picture?

To summarize, the advocates of structured design asked the right questions, but failed to come up with useful answers.

Interestingly, the answers were latent in earlier work: Parnas's papers, and the abstract data types introduced by Liskov and Zilles [24].

3 OBJECT-ORIENTED PRINCIPLES AND TECHNIQUES

I believe, of course, that to get the full answers to the modularity issues raised by the proponents of Structured Design you have to turn to object-oriented techniques. This section will review the main architectural concepts of object-oriented design.

3.1 The overall structure

The main contribution of object-oriented techniques is indeed to tackle head-on the key issues of modular design.

This observation is important because in-the-small issues often play an important role in discussions of object-oriented programming. This is inevitable in, for example, presentations of Smalltalk or Lisp-based object-oriented languages, whose in-the-small components depart noticeably from those of classical languages. The situation is quite different in Eiffel, where the in-the-small aspects are fairly conventional (with the exception of the run-time model, relying on dynamic storage allocation for all objects, and reference semantics for objects of non-simple types). In such a framework it becomes possible to emphasize what is really revolutionary: not the way you deal with (say) conditional instructions or local variables, but the high-level mechanisms for describing system architectures.

Architectures obtained with Eiffel are a radical departure from traditional ones. As already noted, there is no main program. Also, there is no nesting of program texts. Systems are built as combinations of autonomous software elements – classes –, each of which is organized around a data abstraction. For example, an air traffic control system might include classes such as *AIRPORT*, *RUNWAY*, *RADAR*, *AIRPLANE*, *COMMERCIAL_AIRPLANE*, *MILITARY_PLANE* etc. Each such class describes potential run-time objects (to be allocated dynamically as needed); it is defined by the set of "features" applicable to these objects. A feature is either an operation on objects (such as *take_off* for *AIRPLANE*) or a primitive accessing properties of the objects (such as *passenger_capacity*).

The main structuring facilities used to connect classes are the client and heir (inheritance) relations. A class is a client of another when it includes a component of the other's types; it is an heir of another when it represents a special case of that other. (The word "descendant" is used to denote a direct or indirect heir.) Inheritance may thus be viewed as "is" and client as "has". For example *COMMERCIAL_AIRPLANE* will inherit from *AIRPLANE* and *AIRPORT* will be a client of *RUNWAY*. In practice, inheritance must be multiple whenever needed: *AIRPLANE* might also inherit from *INVENTORY_ITEM*.

These structuring mechanisms are complemented by genericity, allowing classes to be parameterized by types. Genericity provides a "horizontal" direction for extending a class, as opposed to the "vertical" variation permitted by inheritance. This mechanism is also essential if static type safety is to be ensured in the object-oriented context. Genericity constructs were first introduced by formal specification languages such as CLEAR [7] and Z [1].

3.2 Controlling inheritance

Also crucial in the Eiffel approach to building good software architectures is the detail of the techniques used to make inheritance practical. The most important are explicit redefinition, polymorphism, dynamic binding and renaming.

Explicit redefinition reconciles the desirability of relying on general mechanisms with the need to override these mechanisms if circumstances warrant. For example class *AIRPLANE* might define a general landing routine, applicable in most cases; a descendant class such as *MILITARY_AIRPLANE*

might, however, redefine this routine to account for special circumstances. To avoid any oversight, redefinition only occurs in Eiffel when explicitly requested by the programmer through a special clause.

Such a facility may be described as "dignified hacking" and is essential to make reusability practical. The usual, undignified form of hacking arises from a legitimate concern: the need to deal with occasional exceptions to patterns that work in most cases. Efficiency is often the immediate motive: a general purpose algorithm may be too slow or bulky in some cases because it fails to take advantage of special circumstances. With traditional techniques, such exceptions must be implemented in the form of patches to the original code; this is one of the major sources of pollution of software structures. With redefinition, no structure is disrupted: you leave the original routine undisturbed, and simply override it with a more specific version in the descendant class. You win on all counts: generality (a general-purpose algorithm is available), architecture (the overall modular decomposition remains clean) and performance (the general algorithm is overridden when circumstances warrant). This is a key factor behind the elegance of object-oriented architectures.

Polymorphism ensures the appropriate degree of flexibility by allowing variables that may take more than one form at run-time. For example a variable of type *AIRPLANE* may refer to a value which is an instance of any of the descendants of that class such as *COMMERCIAL_PLANE*. Because new descendants may be added to any class at any time in its evolution without any disruption of the class, polymorphism is an open mechanism; this is in contrast with the Pascal-Ada technique of using records with variants, which freezes the list of choices.

Polymorphism in Eiffel is safe, as it is controlled by the inheritance mechanism (you can only assign from a descendant to an ancestor type), which forms the basis of a powerful and flexible type system.

Dynamic binding complements redefinition by ensuring that when a routine is applied to a polymorphic variable, and the routine has one or more redefinitions, a run-time mechanism selects the appropriate version, on the basis of the actual type of the variable's value. For example, applying the *land* routine to a variable declared of type *AIRPLANE* will trigger the appropriate implementation depending on the actual type of plane referred to by the variable.

The combination of static typing and dynamic binding in Eiffel is noteworthy. Static typing guarantees that *at least one* version of a routine will be applicable in all possible cases (who wants to wait until run-time to realize that no mechanism is available for landing a particular plane?); dynamic binding then ensures that *the best one* will be selected if there is more than one. I believe this is the right combination.

Finally **renaming** serves to remove name clashes in multiple inheritance and, even more importantly, to provide adequate interfaces for classes obtained by inheritance [29].

Again, these are architectural techniques; they are used to define the bones of a system's structure and would still be applicable if the meat of the language were of a totally different kind. By changing the in-the-small aspects of Eiffel we could obtain functional (Eiflisp?) or logic (Eiflog?) object-oriented languages. The same concepts can also be applied to non-programming-language contexts where the same structuring problems arise: for example an Object Management System as needed in integrated programming environments (PCTEiffel?), where the objects are files, processes, users and the like, could be organized around the same techniques of multiple inheritance, genericity, polymorphism, typing and renaming.

3.3 Deferred classes

One inheritance technique deserves a special mention: deferred classes. A class was defined at the beginning of this article as a *possibly partial* implementation of an abstract data type; if not fully implemented, a class is said to be deferred. This concept is fundamental to obtain the full reusability benefits of object-oriented design.

A deferred class is one that contains one or more routines themselves declared as deferred, that is to say specified but not implemented. Various actual implementations may be given in descendant classes.

Deferred classes address what is perhaps the most difficult issue of software reuse, to which I know of no other solution: how to produce a reusable software component which is general enough to cover a number of different variants (not all of which are necessarily known when the reusable component is designed), but specific enough to capture what is common to all of these variants.

A typical example (already sketched in [31]) applies to a set of classes describing a general notion of "table". A subset covers tables managed sequentially. This can be described by a class which, because there are so many possible implementations of sequential tables (by an array, a linked list, a file etc.), can only be deferred. However some routines, such as *present* which tests whether an element appears in a sequential table, can be expressed in non-deferred form because their implementation is the same in all cases.

A partial listing of the corresponding class will provide our first example of an Eiffel text. Particularly noteworthy are the **assertions** associated with the class and its routines: the class invariant at the end; routine preconditions (**require** clauses) describing the conditions under which a routine is applicable; and routine postconditions (**ensure**) describing constraints on the results. Preconditions and postconditions may be associated with deferred as well as non-deferred routines; this explains the above observation that a deferred routine may be specified even though it is not implemented.

The given class is generic; *T* represents the type of table elements and may be instantiated (in any practical use of *SEQUENTIAL_TABLE*) to any Eiffel type. *SEQUENTIAL_TABLE* inherits from *TABLE*, which describes an even more general notion of table and, clearly, must also be deferred.

```
deferred class SEQUENTIAL_TABLE [T] export
        empty, present, insert, ...
inherit
        TABLE [T]

feature
        present (x: T): BOOLEAN is
                        -- Does x appear in the table?
                do
                        from
                                if not empty then start end
                        until
                                over or else found
                        loop
                                move
                        end;
                        Result := found
                end; -- present

        start is
                        -- Move search cursor to first position
                require
                        not empty
                deferred
                ensure
                        position = 1
                end; -- start
```

```
        move is
                -- Advance search cursor by one position
            require
                not over
            deferred
            ensure
                position = old position + 1
            end; -- move

        position: INTEGER;

        ... Declarations for empty, found, over, ...

    invariant

        empty implies over;
        position >= 0; position <= size + 1

    end -- class SEQUENTIAL_TABLE
```

(As seen in *present*, the Eiffel loop syntax includes loop initialization in the **from** clause and gives the exit condition in the **until** clause.)

The searching mechanism used by function *present* relies on lower level mechanisms for moving a search cursor (*start*, *move*), testing whether it is off the right edge (*over*) and comparing the current value to the search key (*found*). These are expressed as deferred routines; they can only be implemented in descendant classes such as *ARRAY_TABLE* or *FILE_TABLE*; however their formal properties may be expressed in the deferred class through preconditions and postconditions.

What is particularly important here is the ability to write a non-deferred routine such as *present* which calls deferred routines such as *start* and the like. A partially deferred class such as *SEQUENTIAL_TABLE* captures the common behavior of a number of variants; non-deferred routines such as *present* embody the commonality, whereas deferred routines such as *move* embody the differences between variants.

Many of the most elegant examples of object-oriented design, solving problems that would be extremely difficult to address properly with traditional techniques, use such a combination of deferred and non-deferred routines. A typical example, described elsewhere (see [31], chapter 12) is a general-purpose, multi-level UNDO-REDO mechanism for interactive systems.

Deferred classes illustrate what may be called the **reusability game**. This is a board game played with inheritance diagrams such as given on figure 1; you gain points by moving operations up in such diagrams. Every point gained represents a net increase in generality, and a potential long-term improvement for any organization that has implemented a reusability policy.

3.4 Design versus implementation

The view of design as conceptually separate from implementation extends well beyond Structured Design circles. It underlies job classifications (“systems analyst”, “designer”, “programmer”) and design-specific formalisms (“Program Design Languages”).

Introducing such an artificial gap is detrimental to software quality. Design and programming are the same activity, raising the same issues – structure, modularity, correctness, balancing between function and data aspects etc. – and amenable to the same intellectual mechanisms. Any method that considers them to be separate introduces interface problems between the designers and the implementers, a difficulty which is compounded in many cases by the use of different formalisms: a PDL (program design language) for design and a programming language for implementation. Worse

yet, perhaps, is the maintenance issue: how do you guarantee that when the system evolves, design and code will remain consistent? There is no satisfactory solution.

"Design" and "implementation" should instead be viewed as instances of the same general activity, which may be called simply programming. The only difference is in the abstraction levels of the virtual machines being programmed. The standard dichotomy artificially reduces the number of levels to two. A more realistic software construction process involves a continuum from the most abstract view of a system down to the lowest level components.

Eiffel is meant to support this process and may be viewed a PDL as well as an implementation language. In its role as a PDL, it is actually of a higher level than many existing design languages, in particular thanks to the presence of deferred classes: a deferred class may be used to express incompletely implemented concepts, which may still be semantically specified using assertions. Because a deferred class may contain non-deferred as well as deferred routines, the entire spectrum is covered: from fully abstract modules to fully implemented ones.

People used to Structured Design or similar methods often have the impression that object-oriented techniques only cover implementation, and ask whether their use can be preceded by Structured Design techniques. This is of course meaningless: the conceptual assumptions of the two approaches are incompatible (Structured Design is heavily functional in its view of the world). What looks as program text in a formalism such as Eiffel is often of a higher abstraction level than what passes for design in more traditional approaches.

Part of the confusion is due to the popularity of graphical design descriptions, which give the impression of being very high-level. The object-oriented method can indeed benefit from graphical representations to show system structures; in the Eiffel environment, the GOOD tool is used for this purpose. GOOD ("Graphics for Object-Oriented Design") [30] is a high-level system browser; as illustrated in figure 2, GOOD graphically shows the architecture of a system, represented by its classes and the relations (inheritance and client) that connect them. GOOD may be used to explore existing structures and also to design new ones (with automatic generation of the corresponding class skeletons).

Thanks to GOOD, then, object-oriented has its own circles and arrows, which serve to visualize software designs and help designers get a good grip on the architecture. Useful as they may be, however, such tools should not lead us to forget the key lesson of structured programming: that software design is a challenging intellectual problem, and one that is unlikely to be solved by graphical gimmicks.

3.5 Management issues

This article is not the appropriate place to discuss detailed issues of software project management. One observation, however, can be made to relate the previous discussion to one of the ideas often associated with structured programming: chief programmer teams.

As introduced by Baker and Mills [2, 38], the chief programmer team concept was closely connected to the top-down approach, which we do not retain for this presentation. One of their precepts, however, remains very current: the rule that the project leader should not just be a technical administrator of a kind, setting directions and checking other people's work, but should write code himself; in fact, he should be responsible for the most critical code (which in top-down design is the functional "top"). This is in line with the argument developed by Brooks [6], using examples from other technically critical endeavors such as space exploration: the senior technical leaders, not the administrators, should ultimately be in command.

Stripped of their top-down components, these ideas combine well with object-oriented design and with the need to remove any artificial barrier between design and implementation, as advocated above. Here the critical code that will be under the direct responsibility of the lead designer might be a cluster of basic classes defining the key reusable components, a role that the Basic Eiffel Library fulfilled in

the development of Eiffel.

3.6 Generalization

Traditional approaches to design, based as they are on top-down development and the myth that the aim of a software project is to implement a supposedly frozen set of requirements, have completely missed an activity which is essential to quality design. This activity (which, to my knowledge, was first described convincingly in a recent report by Gindre and Sada of Thomson-CSF about practical experiences with object-oriented design in Eiffel [16]) is only meaningful if you take a long-term, company- or department-wide view of software quality, as opposed to the short-term, project-wide view which is taken for granted in most of the software engineering literature. The activity may be called **generalization** and involves working on a component *after* its initial (and perhaps satisfactory) version has been released; the aim is to make the component amenable for reuse in contexts other than the initial project for which it was developed.

In Eiffel generalization may involve factoring out common elements in different classes by making them descendants of a common ancestor, cleaning up overly specific implementation choices by moving up the implementation-independent features of a class to a deferred ancestor, or making some classes generic; it also includes such ex-post-facto improvements of a class as better assertions, better comments or a specific test driver.

In standard discussions of software engineering, there is no room for this activity: if "software engineering economics" [3, 4] applies to single projects only, then generalization is even detrimental: why would someone take a component that fulfils its purpose and devote extra work to it? Indeed, Gindre and Sada report that the net immediate effect of this step is to *decrease* productivity as measured by the usual criteria of lines of working code divided by effort: you devote extra effort to components that *worked*, and because of the factoring out you often end up *decreasing* the measurable output! This is enough to drive a cost-conscious manager crazy.

Because of this apparent paradox, generalization is only meaningful in an organization that has a long-term management commitment to software quality and, in particular, an official reuse plan.

If this is the case, generalization can bring remarkable results. In my experience and that of other Eiffel users it was found to be a key component of the object-oriented design lifecycle. In a way, generalization is the poor man's approach to reusability: rather than setting out to build a library of reusable components, you derive increasingly general components from perhaps not-so-reusable ones, built initially to satisfy the immediate requirements of some customer or boss who (as most customers and bosses in this world) was initially more concerned with immediate results than with the demands of posterity.

3.7 Object-oriented software lifecycle: the cluster model

The well-known waterfall model [3], of which a variant appears as figure 3, has been repeatedly criticized. Yet no satisfactory replacement has gained widespread acceptance. It is fair to ask what kind of lifecycle is appropriate to object-oriented design.

The main ingredients of a possible answer to this question have already been introduced:

- The merging of the design and implementation activities, traditionally considered to be different phases of the lifecycle.
- The general bottom-up approach, which de-emphasizes the immediate requirements of the current project in favor of a long-term view of software production, and suggests that general-purpose utility modules should be built first, specific ones last.

- The new lifecycle phase just described: generalization, which is profitably merged with the more usual phase of component validation.

One more concept is needed to complete the picture: the cluster concept. A cluster is a group of classes which relate to a common aim; for example a system could contain a basic cluster (the Basic Eiffel Library), a graphics cluster (the Eiffel Graphics Library or another set of graphics classes), a simulation cluster, a synchronization cluster etc.

In Eiffel there is no need to define "cluster" as a language construct because the notion of directory, available on all modern operating systems, provides the ideal basis. Eiffel classes are stored in files (one class per file); quite naturally, the files containing a set of logically related classes will be maintained in the same directory. The notion of cluster has also been integrated with the Eiffel automatic recompilation mechanism: once compiled, the classes of a cluster are linked together, so that no intermediate relinking is necessary if nothing has changed in the cluster.

With this notion in mind we can take a fresh look at the waterfall model. The continued success of this model in the software engineering literature, in spite of its known deficiencies, should perhaps be credited to two of its properties, already noted by Boehm [3]: the lifecycle steps (requirements, specification, design, implementation, validation, distribution) reflect meaningful and necessary activities of software construction, although, as we have seen, it may be appropriate to merge some adjacent pairs; and it is hard to imagine of a theoretically more satisfying order than the one given: who would seriously advocate distributing before specifying?

We may realize, however, that nothing really forces us to apply this sequence of steps *to the system as a whole*. This would be keeping the negative legacy of top-down design: the all-or-nothing approach which considers a system as a monolithic entity fulfilling a frozen specification. The notion of cluster provides the appropriate unit to which each sub-lifecycle should be applied. As shown on Figure 4, these sub-lifecycles may overlap in time, and I believe they should.

The other ideas developed so far help further define this new lifecycle model, which we may call the **cluster model** of software development:

- The best order for starting cluster development is bottom-up: from the most general clusters, providing utility functions, to the most application-specific ones. Of course, some of the lower-level clusters will be available from the start as part of the standard delivery (in Eiffel, the Basic and Graphical Libraries); and as the method is applied to repeated projects within an organization, other reusable clusters will become readily available.
- A possible sequence to apply to each sub-lifecycle includes the following three steps: specification (labeled SPEC on Figure 4); design and implementation (DESIMPL); validation and generalization (VALGEN).
- Each cluster may be a client of lower-level ones. The client relation enables the "DESIMPL" of the classes in a cluster to rely on the specification of classes in another. In contrast with hierarchical abstract machine methods, we should not require that each cluster only be a client of the immediately lower one, or impose any similarly artificial restriction.

These lifecycle techniques for Eiffel design are discussed in more detail in a recent article stemming from a collaboration between designers and users of the language [36]. We have found them to yield a software development process which is smoother and more effective than traditional approaches because it integrates at its very basis the concern for change and the concern for reuse.

4 CORRECTNESS: ASSERTIONS

At the basis of structured programming lies the idea that programs should be built so as to satisfy a precise specification, by a process which derives the program from the specification. To justify the individual constructs entering into a program, the programmer associates with them static properties derived from parts of the specification, or assertions. These techniques were refined in work on abstract data types [19, 24, 25], making it possible to describe data by the applicable operations and properties of these operations. In simplistic presentations, abstract data types are taken to mean just information hiding. There is much more to this notion: an abstractly defined type is characterized by formal axioms, which again may be expressed as assertions. Not surprisingly, assertions are also present in formal specification languages [1, 7, 23, 26], aimed at providing the rigorous basis for writing provably correct software.

Eiffel, influenced as it was by structured programming, data abstraction and formal specification languages, includes its own assertion mechanism. This is a toned-down version, appropriate for use in a practical production environment (as opposed to a research vehicle), but still holds the promise of great benefits in reliability. The next sections describe this mechanism and then assess its significance.

4.1 The assertion mechanism

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

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

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

One of the most important uses of assertions is to characterize the semantics of routines through a precondition, introduced by the keyword **require**, and a postcondition, introduced by **ensure**. The postcondition describes the requirements that must be satisfied by a client (caller) for a call to be correct; the postcondition describes the result that the routine ensures in return to its client.

An example was given in class *SEQUENTIAL_TABLE*: procedure *move* can only be applied to a table which is not in the "over" state, and will result in *position* being increased by one.

```
move is
        -- Advance search cursor by one position
    require
        not over
    ... routine body (deferred or not) ...
    ensure
        position = old position + 1;
    end; -- move
```

The **old** notation, used only in postconditions, makes it possible to refer to the value an attribute had on routine entry.

A further use of assertions is the class invariant, which states the properties that must be satisfied by all instances of a class in all "stable" states, that is to say after instance creation (obtained in Eiffel by

executing the *Create* procedure of the class) and before and after the execution of every exported routine. For example, the invariant of *SEQUENTIAL_TABLE* includes:

empty **implies** *over*;
position >= 0; *position* <= *size* + 1

A class invariant expresses the integrity constraints that must be satisfied by all instances of a class. The invariant is implicitly added to both the precondition and the postcondition of every routine in the class (postcondition only for *Create*). It transcends, however, the individual routines, since it applies to the class as a whole. In particular, the invariant constrains not only the routines that appear in the class at a given moment of its evolution, but any others that may be added later either through modification of the class or through inheritance.

4.2 Uses of assertions

The primary use of assertions is as a conceptual tool for improving software reliability. Without a technique for expressing the purpose of individual software elements independently of their implementations, there is no guarantee that they will do any useful job. The underlying idea is "**programming by contract**": every routine is charged with a precise task, defined by a specification that states precisely the obligations on the client, limiting the routine's responsibility (the precondition), and the obligations on the routine, guaranteeing the client a certain result (the postcondition). The class invariant states general constraints that apply both to the client and the routine.

The notion of programming by contract, a direct development of structured programming ideas, has been developed elsewhere [34].

Assertions are also useful as a documentation tool: by considering the assertions associated with a class, you obtain an abstract view of the class, free of implementation details. This view is produced in the Eiffel environment by a **class abstracter**, the "short" command. This command is used to produce documentation on classes from the standard Eiffel libraries [35] and other software written in Eiffel.

Tools such as "short" reflect an approach to documentation that, again, differs from the views which are popular in most of the software engineering literature. It is generally agreed that software engineers should produce documentation as an artefact separate from the software itself. To me the need to write documentation is not a desirable goal per se, but a reflection on the inadequacy of our current notations for writing programs. In an ideal world, the documentation would be executable and hence there would be no need for distinguishing program from documentation. The arguments for this are similar to those for avoiding the design-programming gap: ensuring consistency. In a less ideal but still acceptable world, programs have to include conceptually irrelevant implementation details and hence contain more than the documentation, but contain *all* their documentation; in other words, the documentation is a more abstract view of the program, which can be extracted from it by automatic tools. (Different tools might produce different views, at various levels of abstraction.) Although progress remains to be done to reach even these goals, tools such as "short" and, at the system level, the GOOD graphical system described previously, are (I believe) steps in the right direction.

The preceding two applications of assertions are purely conceptual and do not require that assertions be actually evaluated. My experience of teaching these concepts to software audiences provides as good a symptom as any of the failure of structured programming to reach the mainstream of the industry. Whenever you talk about assertions and their benefit in writing reliable, well-documented software, nobody will pay any attention until you have answered the question "What happens at run-time if an assertion is not satisfied?". It seems many software engineers have their priorities reversed; instead of focusing first on techniques for making their programs correct, and then worrying about what will happen if a mistake remains, they are apparently prepared to live with buggy programs and only want to know what the consequences will be.

Once you get your priorities straight, you will of course want to know the behavior of a buggy program. Assertions only have an effect if the programmer so decides. A compiler option determines, for each class, whether to check the validity of assertions. Three levels of run-time assertion monitoring are possible: no checking at all, preconditions only, all assertions.

The amount of monitoring that you choose is the result of a tradeoff between performance requirements and the degree of trust you put in the correctness of your software. If possible, maximum monitoring should always be in effect, although performance requirements will usually lead implementers to remove monitoring in production runs. (Tony Hoare [21] noted the paradox of this common attitude, which he compared to wearing your lifejackets for practice outings and leaving them at home when you go on a real cruise.)

In an ideal world, the consistency of a system (set of classes) with respect to its assertions would be statically checked by a class prover, integrated perhaps with the compiler, so that there would be no need for run-time monitoring. The appropriate proof technology is, unfortunately, not available (although I have a fairly clear if unimplemented idea of how Eiffel could be made practically provable). Run-time monitoring is the next best thing to static verification.

As anyone who has understood this discussion will by now realize, assertions are not meant to serve as a technique for handling special cases – situations that must be handled in a particular way but, nevertheless, are expected to occur in some program executions as part of the accepted course of events. For these, the usual if-then-else instruction is appropriate. The violation of an assertion is always the result of a programming error. If the assertion is a precondition, the error is in the client; if it is a postcondition, the error is in the class itself. (Combined with the general emphasis on bottom-up development and validation, these observations explain why the default mode is to check the precondition only. Once the classes of a cluster have been validated and you move up to the next cluster, the routines of the lower-level cluster will always ensure their postconditions if the validation process has been well done; but their clients in the higher-level cluster may contain errors that result in violating the lower-level preconditions.)

4.3 Limitations

The Eiffel assertion mechanism is the result of an engineering tradeoff and is not meant to be entirely formal; Eiffel is not a formal specification language but a practical tool for building efficient production software. The following limitations apply to the assertion language:

- Assertions are boolean expressions, with some extensions (such as the **old** notation for taking a "snapshot" of a variable's value on routine entry).
- No quantifiers are supported.
- Since full boolean expressions are acceptable, function calls may be included. This compensates for the previous limitation (an existential or universal quantifier may be simulated by a loop) but, of course, brings in an operational element to a mechanism that is supposed to be purely applicative. In practice, any function that is called in an assertion should be of a "higher quality"; if it contains an error, there is no guarantee as to what may happen. (For obvious reasons, the assertion checking mechanism is disabled when a routine is executed as part of the evaluation of an assertion.)

4.4 Debugging, testing and quality assurance

Run-time monitoring of assertions provides a powerful debugging tool. Assertions serve to express the many assumptions we rely on when we write a program; in traditional approaches, these assumptions remain implicit. Often, when we make an error, one of them will be violated. Once the assertions are made explicit, the run-time checking mechanism will detect a number of bugs that might otherwise have been difficult to track. This has turned out to be one of the major pleasures of building software in this environment.

This use of assertions extends to testing, maintenance and quality assurance. As commonly practiced, these activities lack any systematic basis. Assertions can provide this basis by describing what we are testing for.

Of particular importance here are class invariants. As noted, an invariant captures the integrity constraints that apply to all instances of a class; they express the deep semantic properties of the class. This provides a good potential basis for the QA process. These ideas (which deserve much further exploration) are especially relevant to regression testing: the invariant expresses the essential semantics of a class, which should be preserved throughout successive modifications and extensions. I believe that the notion of invariant holds the key to a powerful theory of testing, maintenance and quality analysis.

4.5 Partial functions

As soon as we associate preconditions with routines, we accept that the underlying mathematical functions may be partial and that the routines themselves are not necessarily prepared to deal with all possible inputs. In particular the body of a routine should *never* test for the precondition (if you are going to test for a special case, it becomes part of the routine's specification and the ceases to be part of the precondition's business).

This principle is contrary to conventional wisdom in software engineering. Virtually all the discussions I have read on this topic conclude that routines should be made as general and robust as possible. This "common sense" approach is naive and in fact detrimental to software quality since it leads to undue increases in software complexity. The detailed argument for focusing on routines that state precisely their conditions of applicability, rather than attempting to chase after all possible cases, has been made elsewhere [34] (see also [31], section 7.3); here I have probably said enough to make some readers think these ideas are crazy ("How can you make software *more* reliable by writing routines that check *less*?").

Beyond referring these readers to the more extensive discussions, all I can do is to urge them to take an in-the-large, global view of software reliability. Reliability is not obtained by accumulating blind run-time checks for special cases; the responsible method is to assign to each software module a precise specification, and then to verify that each module is both a faithful supplier and a faithful client, in other words that it satisfies its specification (invariant, postcondition) and calls other modules in a manner that satisfies *their* specification (precondition, invariant).

5 EXCEPTIONS

As will be clear from the preceding discussion, there is little sympathy in the Eiffel approach for the way many recent languages, notably Ada and CLU, encourage programmers to deal with special cases: writing algorithms that concentrate on the most standard cases, raising a special signal (an exception) when an non-standard situation is detected, and handling these in special "exception" clauses.

This distrust of exception mechanisms was motivated by a deep conviction that traditional if-then-else structures were appropriate in most cases and observation of the deficiencies of existing exception mechanisms. Hoare's scathing indictment of Ada exception techniques in his Turing lecture [22] exerted a strong influence here. Accordingly, Eiffel initially did not have any exception mechanism.

After further analysis, however, I came to realize that if-then-elses did not solve all problems and that a case could be made for a disciplined exception mechanism, now part of all released versions of Eiffel. This mechanism is that it is a direct result of the assertion-based approach to software reliability, as described in the previous section.

The design goal for Eiffel exceptions can be stated quite simply: to come up with a facility that a card-carrying devotee of structured programming, even after reading Hoare's paper, would not disavow. The reader will judge whether this has been achieved. (This discussion is only an overview; more detailed presentations may be found elsewhere [31, 34].)

5.1 Exceptions in existing languages

Before we present the Eiffel design, it is useful to briefly consider the mechanisms built in existing languages. We will use Ada as a basis; the discussion transposes to CLU with some variations.

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

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

With exceptions, a handler clause of the form

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

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

The use of exceptions in such examples appears as a vain attempt to do away with the necessity to deal with abnormal cases. Of course it is unpleasant to include many if-then-else structures of the above form in a program; but this necessity is a fact of life. Often, of course, the presence of many special cases simply reflects an inadequate specification; many systems can be made more simple and regular through some extra work on the specification. But special cases will remain, if only because software

systems interact with the external world, which is not necessarily regular. Then if your specification calls for special cases, your code must include branches that deal with them.

The need to specify the treatment of abnormal cases does not magically vanish through the **raise** incantation.

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

The exception mechanism of Ada is particularly worrying because the handler of an exception may do any processing before returning control to the caller. It is quite possible in particular to catch an exception and return control to the calling routine without reporting failure. An extreme example, found in an Ada textbook and analyzed elsewhere [34], is a function for computing a real square root; when presented with a negative argument, this function raises an exception, which is immediately caught by a **when** clause that prints a message and then .. quietly returns to the caller!

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

5.2 Cases for exceptions

Other references (quoted above) discuss how standard control structures may be used in most cases where CLU or Ada would use exceptions, using either the "a priori scheme" (test the applicability of an operation before attempting it) or the "a posteriori scheme" (attempt the operation, and then find out whether it has succeeded). There remains, however, three cases in which classical techniques are not sufficient and exceptions may be needed.

- The first case is one in which attempting the operation may cause a hardware or operating system signal if the operation was not applicable; the signal must be caught to avoid catastrophes.
- In the second case, an abnormal situation must lead to immediate termination because physical danger may otherwise result, as in a robot manipulation system.
- The third case, mentioned above (*pop*) is software fault tolerance: guarding against the possibility that an error remains.

The Eiffel exception mechanism is meant to deal with these cases. It is a direct consequence of the assertion techniques, which make it possible to express the specification of a software element, and hence to define precisely what is "normal" and what is a "failure" or an "exception".

5.3 Failures and exceptions

The following definitions serve as the basis for the disciplined exception mechanism to be introduced next.

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

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

The major types of exceptions include the following:

1 • Violation of an assertion, when monitored.

2 • Failure of a called routine.

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

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

Cases 2 to 4 may be viewed conceptually as variants of case 1, where the violated assertion cannot be properly expressed in the assertion language, for example the unstated assertion that, in the computation of $a + b$ the mathematical sum of a and b is small enough to be representable on the machine.

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

5.4 Two principles of exception handling

The following law expresses that the occurrence of an exception is not an excuse to violate the routine's contract:

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

Trivial as this law may seem, it is violated, for example, by the Ada exception mechanism, which makes it possible to write an exception handler that returns to the caller without correcting the cause of the exception (and without re-raising the exception). In such a case the routine has failed (since an exception prevented it from executing to its normal end), but returns control to its caller without signaling failure. It is like a "dishonest" contractor that has not performed its task but pretends to its client that it has. The square root function mentioned above shows that such dangerous uses of the Ada exception mechanism not only are possible but have found their way into software engineering textbooks.

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

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

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

One response, **organized panic**, consists of admitting that the contract cannot be fulfilled: bring all affected objects to a coherent state, and report failure. Note that this will trigger an exception in the caller, which will recursively have to decide what to do in response to this exception, using the same two possible choices.

The other response, **resumption**, consists in attempting to fix the reasons for the exception and trying the whole routine execution again.

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

5.5 The rescue clause

The support for exception handling is concise: two keywords and a library class.

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

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

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

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

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

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

5.6 Retrying

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

retry

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

It is important to note that, **retry** or not **retry**, the rescue clause never attempts to fulfil the routine's contract. This is solely the province of the body (**do**...). (This is perhaps the major difference beteen the Eiffel exception mechanism and Randell's recovery block mechanism [40]). The aim of the rescue clause is to "patch things up" and either concede failure or retry. This property will be characterized more formally below in reference to assertions.

5.7 The *EXCEPTIONS* class

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

exception

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

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

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

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

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

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

Here I must admit to a certain weakness. Hoare clearly warned against attaching too much significance to the apparent cause of an exception. Quoting from his Turing lecture:

> *The danger of exception handling is that an "exception" is too often a symptom of some entirely unrelated problem. For example, a floating-point overflow may be the result of an incorrect pointer pointer use some 43 seconds before; and that was due perhaps to programmer oversight, transient hardware fault, or even a subtle compiler bug... The right solution is to treat all exceptions in the same way as a symptom of disaster.*

The facilities from the *EXCEPTIONS* class were introduced in defiance of this warning; I don't think the Eiffel users would have forgiven me if I had decided otherwise. To satisfy the ideologically pure, however, these facilities were kept out of the language proper and included in a special library class that you have to bring in explicitly.

5.8 Formal requirements

To understand more deeply the Eiffel exception mechanism and its relation to the general assertion-based approach to software reliability, as rooted in structured programming concepts, it is essential to understand the properties which in principle constrain all rescue clauses.

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

Now consider a branch of the rescue clause not ending with **retry**. It will lead to failure, but must leave the object in a stable state, as noted above. This means that such a branch must admit the **class invariant** as postcondition. Note that the branch is by no means constrained to ensure the routine's postcondition: this is the task of the body. This is similar to the requirements on transactions in a database system: some transactions will fail, but all must leave the database in a state that satisfies the integrity constraints. We do not require, however, that the initial state be restored; this would be too demanding, if only because some actions such as output may be irreversible. The final state is simply required to be consistent.

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

5.9 Examples

The mechanism turns out to yield a remarkably simple way of writing software to deal with exceptional conditions. Only two examples will be given.

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

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

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

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

Another example, adapted from [5], is that of a routine that returns $1/x$, or 0 if the division is impossible. This is typical of problems that are almost impossible to solve without an exception facility, because the only way to find out whether the operation is possible is to attempt it, but if it fails a hardware signal will be generated. (We assume this is the case whenever x is too small.) This may be written as:

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

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

5.10 Further comments

Interestingly enough, the mechanism described here could have been designed into Ada, although it fits particularly well within the object-oriented approach as promoted by Eiffel. What seems to have prevented the inclusion of such a mechanism in Ada is the lack of a notion of contract.

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

6 ON LANGUAGE DESIGN

On the last stop of this tour it is appropriate to comment on some of the thinking that should go into the design of a language.

6.1 Simplicity

The milestone reference on programming language design is a 1973 article, again by Hoare [21]. The key design quality stressed in this article is simplicity:

> *It ... seems especially necessary in the design of a new programming language, intended to attract programmers away from their current high-level language, to pursue the goal of simplicity to an extreme, so that a programmer can readily learn and remember all its features, can select the best facility for each of his purposes, can fully understand the effects and consequences of each decision, and can then concentrate the major part of his intellectual effort on understanding his problem and his programs rather than his tool.*

As an excuse for overly complex designs, the argument is often made that a programmer only needs to master parts of the language. This is a dangerous approach. The programming language is the programmer's primary tool; we cannot hope to produce quality software unless programmers are in complete control of this tool. Quoting from the same article again:

> *If to the complexity of the language is added the complexity of its implementation, the complexity of its operating environment, and even the complexity of institutional standards for the use of the language, it is not surprising that when faced with a complex programming task so many programmers are overwhelmed.*

No doubt Hoare had in mind such counter-examples as PL/I which, starting from a well-meaning attempt to include something for everyone, became far too complex to be mastered by anyone. It is unfortunate that his article, although frequently referenced, has had so little real impact on later languages; consider for example Ada which (although certainly better organized than, for example, PL/I) offers numerous features that most programmers will never comprehend.

One of the worst excuses for violating the tenet of design simplicity is compatibility with previous designs. Backwards compatibility is the reason why the still elusive Fortran 8X – a language supposedly meant for programming the most powerful supercomputers of tomorrow – continues to carry instructions inherited from the structure of the IBM 701, such as the arithmetic IF. The search for compatibility at any cost is also the reason behind the centaurs sporting an object-oriented head on top of a C body, such as C++. Imagine this: on the one hand, inheritance; on the other hand, pointer arithmetic! One can only think of Liszt's disastrous transcription of Schubert's Wanderer Fantasia: add a full orchestra, and *keep the piano*. (In this case, of course, the original was perfect anyway.)

Complexity, whether or not justified by the compatibility excuse, is the one irredeemable sin in language design. This is best seen if we compare with *software* design. Simplicity is an important goal in the design of a software system. If, however, a certain system is overly complex, you may be able to live with it if the complex elements are isolated from the rest of the system, work properly, do not need to be modified too much, and do not need to be rewritten for porting to other machines. To be sure, these conditions are harsh, but one can imagine cases in which they are satisfied; at least they may be satisfied by part of the structure, and you will only have to rewrite the rest. In language design, no such hope is possible. Language complexity can only increase. Once you have introduced a poor feature in a language, and users have started to rely on it, it will be there forever. To describe the sad state of certain programs, computer hackers sometimes use the expression *patched being repair*. When a programming language starts getting patched, it is always beyond repair.

6.2 What's not in Eiffel

Having seen language complexity exposed to such abuse, the reader will expect me to claim that Eiffel is simple. Since simplicity is a subjective criterion, it is best to stick to facts and let the reader decide. The guiding principle was an attempt to maximize the "signal-to-noise" ratio both by increasing the signal part (features with such power of expression as multiple inheritance, dynamic binding, genericity, assertions and the like) and by minimizing the noise; the latter goal meant pruning the language of anything that was felt not absolutely necessary. When there is one good way to express something in Eiffel, there often are not two. For example, I really don't see why there should be a "for" loop when a "while" loop is available; I'd rather have users code the *i* := *i*+1 themselves if this kind of stinginess on the designer's part means that all users will know the all of the language, rather than most users knowing only a subset. (Since a responsible language designer should also be concerned with the quality of implementations, non-redundancy also means reducing the probable density of bugs and easing the maintenance task, two altogether non-negligible concerns.)

The "small is beautiful" ideology has its limits, of course. Eiffel has been beaten in the austerity race by at least one recent design, Wirth's Oberon, which somewhat reminded me of Samuel Beckett's admirable last plays (just one person, then only a head, then only a mouth). Still, after having sketched what Eiffel has, it is not inappropriate, as we come to the conclusion of this tour, to list what it has not. This is a dangerous exercise, as one or two at least of the reader's favorite features are likely to be on this list. Actually Eiffel excludes a number of language features that I like. But this is precisely the point: any design is a choice, and is defined by what it has excluded just as much as by what it has

retained.

- **Global variables**: The scope of entities declared in an Eiffel class is local to that class. When there is a justified need for shared parameters or objects, the effect is obtained through multiple inheritance and the "once" mechanism [32].
- **Nesting**. A software system written in Eiffel is made of a number of classes. A class is made of a number of routines (and other features). There is no further nesting. My experience with Simula convinced me that in an object-oriented language nesting brings an unnecessary degree of complexity and tends to conflict with inheritance.
- **Packages**. There is no higher-level structuring mechanism than the class. Clusters, as defined previously, are adequately supported by the directory structure of modern operating systems.
- **Gotos**. It is hard to understand that, twenty years after 1968, a single letter about the goto instruction should trigger an apparently endless stream of responses to the *Communications of the ACM*, many of them advocating the use of gotos. Why not Roman numerals?
- **Gotos in disguise**. No "exit", "return" or "continue loop". Eiffel control structures are members of the most fundamentalist sect of structured programming: one-entry-one-exit, no exception, no excuses.
- **Loop varieties**. Using the Henry Ford quote, you can have it any color provided it's black. No **repeat... until...** or **for**. The Eiffel loop (written **from... until...** with its own initialization, and room for a loop invariant and a variant) is similar to a **while** loop; any variation is programmed.
- **Case instruction**. This freezes a set of choices and is one of the worst enemies of extendibility of object-oriented software. The appropriate object-oriented technique is to redefine a function in a number of ways and apply dynamic binding. With this approach a new case may be added at any time, with minimal disruption of existing software.
- **Enumerated types, union types**. These are the hideous accomplices of case instructions and should be indicted for the same crime. (Oberon has recognized this too.)
- **Pointer arithmetic**. This one is better left without comment.
- **Side-effects in arithmetic expressions**. In C you can write *y = x++*, which changes *x* as well as *y*. In Eiffel, you are condemned to two instructions: *y := x; x := x+1*. You must type 9 more characters including blanks. Sorry.
- **Routines as routine parameters**. It is well-known that these are difficult to handle in a typed environment. There is no need for such a facility in an object-oriented language with deferred routines and dynamic binding.
- **Array types**. *ARRAY* is a class of the basic library, not a predefined language construct. It is treated like normal classes (although internally the compiler "cheats" by recognizing their special role for efficiency reasons). As a consequence, any class can inherit from *ARRAY*; this is the basis for many fixed-size implementations of basic data structures in the library.
- **Type casts**. When you declare a variable as being of a certain type, you must stick to using it accordingly. Of course, this is possible because of the flexibility of the type system (for variables representing class instances, you can assign from a descendant type to an ancestor type). The presence of type casts or coercions in a language simply reflects an inadequate type system.

6.3 CONCLUSION

In this tour of object-oriented design, I have tried to show how deeply the object-oriented approach is rooted in the revolution that structured programming brought to our view of software construction.

As noted, both approaches enjoy buzzword status. Such a situation brings visibility and excitement to a field; it can also be a curse. This presentation will have achieved its goal if it plays a small part in enabling object-oriented techniques, which if taken seriously hold the potential to dramatically improve the state of software technology, to avoid the trivialization that has plagued their structured predecessors.

[1] Jean-Raymond Abrial, "The Specification Language Z: Syntax and "Semantics"", *Oxford University Computing Laboratory, Programming Research Group*, Oxford, April 1980.

[2] F. Terry Baker, "Chief Programmer Team Management of Production Programming", *IBM Systems Journal*, vol. 11, no. 1, pp. 56-73, 1972.

[3] Barry W. Boehm, *Software Engineering Economics*, Prentice Hall, Englewood Cliffs (NJ), 1981.

[4] Barry W. Boehm, "Software Engineering Economics", *IEEE Transactions on Software Engineering*, vol. SE-10, no. 1, pp. 4-21, January 1984.

[5] Grady Booch, *Software Engineering with Ada*, Benjamin/Cummings Publishing Co., Menlo Park (Calif.), 1983 (new edition, 1986).

[6] Fred P. Brooks, *The Mythical Man-Month*, Addison-Wesley, Reading (Mass.), 1974.

[7] Rod M. Burstall and Joseph A. Goguen, "An Informal Introduction to Specifications using Clear", in *The Correctness Problem in Computer Science*, ed. R. S. Boyer and J. S. Moore, pp. 185-213, Academic Press, London, 1981.

[8] J. M. Buxton, P. Naur and B. Randell, *Software Engineering Concepts and Techniques (Proceeedings of 1968 NATO Conference on Software Engineering)*, Van Nostrand Reinhold, 1976.

[9] Ole-Johan Dahl, Bjørn Myrhaug and Kristen Nygaard, "(Simula 67) Common Base Language", Publication N. S-22, Norsk Regnesentral (Norwegian Computing Center), Oslo, October 1970. (Revised version, February 1984.)

[10] Ole-Johan Dahl, Edsger W. Dijkstra and C.A.R. Hoare, *Structured Programming*, Academic Press, London, 1972.

[11] Frank DeRemer and Hans H. Kron, "Programming-in-the-Large Versus Programming-in-the-Small", *IEEE Transactions on Software Engineering*, vol. SE-2, no. 2, pp. 80-86, June 1976.

[12] Edsger W. Dijkstra, "Go To Statement Considered Harmful", *Communications of the ACM*, vol. 11, no. 3, pp. 147-148, March 1968.

[13] Edsger W. Dijkstra, *A Discipline of Programming*, Prentice Hall, Englewood Cliffs (N.J.), 1976.

[14] Robert W. Floyd, "Assigning Meanings to Programs", in *Proceedings American Mathematical Society Symposium in Applied Mathematics*, vol. 19, pp. 19-31, 1967.

[15] Susan I. Gerhart and L. Yelowitz, "Observations of Fallibility in Applications of Modern Programming Methodologies", *IEEE Transactions on Software Engineering*, vol. September 1976, pp. 195-207.

[16] Cyrille Gindre and Frédérique Sada, "A Development in Eiffel: Design and Implementation of a Network Simulator", *Journal of Object-Oriented Programming*, vol. 2, no. 2, pp. 27-33, May 1989. Revised version in D. Mandrioli and B. Meyer (eds.), *Advances in Object-Oriented Software Engineering*, Prentice Hall International, Hemel Hempstead, 1992, pp. 199-214.

[17] John B. Goodenough and Susan Gerhart, "Towards a Theory of Testing: Data Selection Criteria", in *Current Trends in Programming Methodology, Vol. 2*, ed. Raymond T. Yeh, pp. 44-79, Prentice Hall, Englewood Cliffs (N.J.), 1977.

[18] David Gries, *The Science of Programming*, Springer-Verlag, Berlin-New York, 1981.

[19] John V. Guttag, "Abstract Data Types and the Development of Data Structures", *Communications of the ACM*, vol. 20, no. 6, pp. 396-404, June 1977.

[20] C.A.R. Hoare, "An Axiomatic Basis for Computer Programming", *Communications of the ACM*, vol. 12, no. 10, pp. 576-580, 583, October 1969. Reprinted in C.A.R. Hoare and C. B. Jones (ed.), *Essays in Computing Science*, Prentice Hall International, Hemel Hempstead, 1989, pages 45-58.

[21] C.A.R. Hoare, "Hints on Programming Language Design", in *ACM SIGACT/SIGPLAN Symposium on Principles of Programming Languages*, October 1973. Reprinted in E. Horowitz (ed.), *Programming Languages: A Grand Tour*, Computer Science Press, 1983, pp. 31-40 and in C.A.R. Hoare and C. B. Jones (ed.), *Essays in Computing Science*, Prentice Hall International, Hemel Hempstead, 1989, pages 193-216.

[22] C.A.R. Hoare, "The Emperor's Old Clothes", *Communications of the ACM*, vol. 21, no. 8, pp. 75-83, February 1981. Reprinted in C.A.R. Hoare and C. B. Jones (ed.), *Essays in Computing Science*, Prentice Hall International, Hemel Hempstead, 1989, pages 1-18.

[23] Cliff B. Jones, *Systematic Software Development Using VDM*, Prentice Hall International, Hemel Hempstead, 1986.

[24] Barbara H. Liskov and Stephen N. Zilles, "Programming with Abstract Data Types", Computation Structures Group, Memo no. 99, MIT, Project MAC, Cambridge (Mass.), 1974. (See also SIGPLAN Notices, 9, 4, pp. 50-59, April 1974.)

[25] Bertrand Meyer, "La Description des Structures de Données (The Description of Data Structures)", *Bulletin de la Direction des Etudes et Recherches d'Electricité de France, Série C (Informatique)*, no. 2, Clamart, 1976.

[26] Bertrand Meyer, "M: A System Description Method", Technical Report TRCS85-15, University of California, Santa Barbara, Computer Science Department, 1985.

[27] Bertrand Meyer, Jean-Marc Nerson and Masanobu Matsuo, "Eiffel: Object-Oriented Design for Software Engineering", in *Proceedings of ESEC 87 (First European Software Engineering Conference), Strasbourg, 8-11 September 1987*, Springer-Verlag, Berlin-New York, 1987.

[28] Bertrand Meyer, "Eiffel: Programming for Reusability and Extendibility", *ACM Sigplan Notices*, vol. 22, no. 2, pp. 85-94, February 1987.

[29] Bertrand Meyer, "Mastering Multiple Inheritance", *Journal of Object-Oriented Programming*, vol. 1, no. 6, November 1988.

[30] Bertrand Meyer, "The Eiffel Environment", *Unix Review*, vol. 6, no. 8, pp. 44-55, August 1988.

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

[32] Bertrand Meyer, "Bidding Farewell to Globals", *Journal of Object-Oriented Programming*, vol. 1, no. 5, September 1988.

[33] Bertrand Meyer, "Eiffel: A Language and Environment for Software Engineering", *The Journal of Systems and Software*, 1988.

[34] Bertrand Meyer, "Design by Contract", Technical Report TR-EI-12/CO, Interactive Software Engineering, Santa Barbara (Calif.), 1988.. Revised version in D. Mandrioli and B. Meyer (eds.), *Advances in Object-Oriented Software Engineering*, Prentice-Hall, 1991, pp. 1-50.

[35] Bertrand Meyer, "Eiffel: The Libraries", Technical Report TR-EI-7/LI, Interactive Software Engineering Inc., Santa Barbara (Calif.), October 1986 (version 2.2, August 1989).

[36] Bertrand Meyer, "The New Culture of Software Development: Reflections on the Practice of Object-Oriented Design", in *TOOLS 89 (Technology of Object-Oriented Languages and Systems)*, pp. 13-23, Angkor/SOL, Paris, November 1989.

[37] Harlan D. Mills, "Top-Down Programming in Large Systems", in *Debugging Techniques in Large Systems*, ed. R. Rustin, pp. 41-55, Prentice Hall, Englewood Cliffs (NJ), 1971.

[38] Harlan D. Mills and F. Terry Baker, "Chief Programmer Teams", *Datamation*, vol. 19, no. 2, pp. 58-61, December 1973.

[39] David Lorge Parnas, "On the Criteria to Be Used in Decomposing Systems into Modules", *Communications of the ACM*, vol. 15, no. 12, pp. 1053-1058, December 1972.

[40] Brian Randell, "System Structure for Software Fault Tolerance", *IEEE Transactions on Software Engineering*, vol. SE-1, no. 2, pp. 220-232, June 1975.

[41] Maurice V. Wilkes, "The Inner and Outer Syntax of a Programming Language", *Communications of the ACM*, 1962.

[42] Niklaus Wirth, "Program Development by Stepwise Refinement", *Communications of the ACM*, vol. 14, no. 4, pp. 221-227, 1971.

[43] Edward Nash Yourdon and Larry L. Constantine, *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design*, Prentice Hall, Englewood Cliffs (N.J.), 1979.

[44] Marvin V. Zelkowitz, Raymond T. Yeh, Richard G. Hamlet, John D. Gannon and Victor D. Basili, "Software Engineering Practices in the US and Japan", *IEEE Computer*, vol. 17, no. 6, pp. 57-66, June 1984.